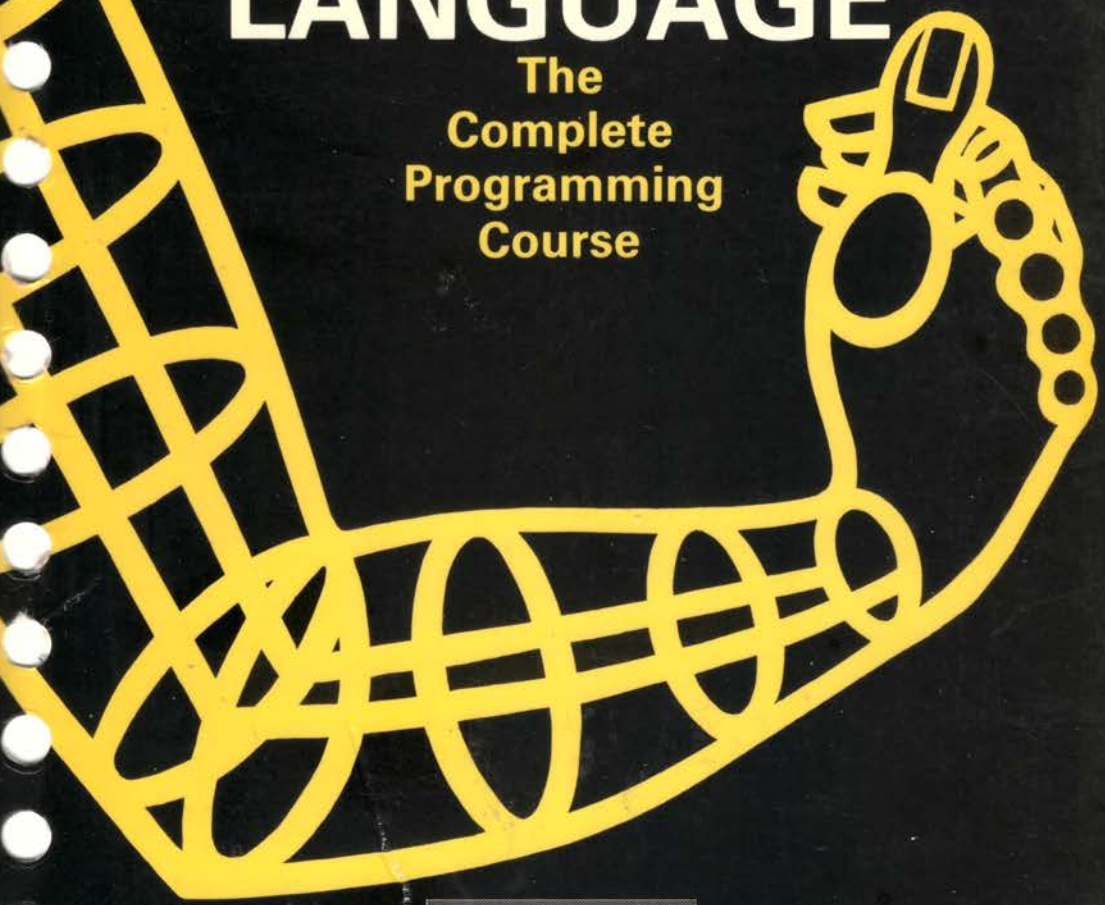
A Dabhand Guide

MIKE GINNS

ARCHIMEDES ASSEMBLY LANGUAGE

The Complete Programming Course



DABS PRESS

Archimedes Assembly Language

A Dabhand Guide

Mike Ginns

Archimedes Assembly Language: A Dabhand Guide

ISBN 1-870336-20-8. First edition May 1988

Editor: Shona McIsaac
Typesetting: Bruce Smith
Cover: Paul Holmes/Clare Atherton
Illustrations: David Price/David Atherton

Acornsoft is a trade mark of Acorn Computers Ltd, 645 Newmarket Road, Cambridge, CB5 8PB. MacAuthor is published by Icon Technology Ltd, Leicester, England. The Apple Macintosh and Laserwriter are produced by Apple Computer Inc.

Within this book the letters BBC refer to the British Broadcasting Corporation. The terms BBC micro, Master 128, Master Compact and Archimedes refer to the computers manufactured by Acorn Computers Ltd under licence from the BBC. InterWord is published by Computer Concepts.

Disclaimer: Because neither Dabs Press nor the author have any control over the way the material in this book and accompanying programs disc is used, no warranty is given or should be implied as to the suitability of the advice or programs for any given application. No liability can be accepted for any consequential loss or damage, however caused, arising as a result of using the programs or advice printed in this book/programs disc.

Published by Dabs Press, 76 Gardner Road, Prestwich, Manchester M25 7HU, UK. Tel. 061-773 2413 Telecom Gold 72:MAG11596, Prestel 942876210.

Typeset in 10 on 11pt Palatino by Dabs Press using the Acornsoft VIEW wordprocessor, MacAuthor, Apple Macintosh SE and LaserWriter II.

Printed and bound in the UK by A. Wheaton & Co., Ltd, Exeter.

Contents

1 : Introduction 13

The Archimedes and the ARM 13
RISC Design 13
Notation 15

2 : An Overview of the ARM 17

A Typical Computer System Model 17
Input/Output 18
Memory 18
Communication Buses 18
The Data Bus 19
Words 19
The Address Bus 20
Byte and Word Accessed Memory 21
Word-aligned Addresses 22
Virtual Memory 23
Executing Machine Code Instructions 24
Pipelining 25

3 : Internal Architecture 27

The Arithmetic Logic Unit 27
The Barrel Shifter 28
Processor Registers 29
Registers on the ARM 30
Uncommitted Registers 31
Special Purpose Registers 31
R14 : The Link Register 32
R15 : Program Counter and Status Register 32
The Program Counter 32
The Status Flags 33
Setting the Flags 34
Mode Flags 34
User Mode 35

Supervisor Mode 35
Interrupt Modes 35
Registers – Different Processor Modes 35
ARM Instructions 37
The RISC Concept 37
RISC verses CISC 37
Instruction Length 39
Conditional Execution 39
Data Shifts 40

4 : The BASIC Assembler 41

General Format of ARM Instructions 42
The Assembler 43
Entering the Assembler 43
The Assembler Location Counter - P% 44
Reserving Memory 45
Assembler Listings 47
Executing Machine Code Programs 48
Returning to BASIC 48
Comments in Assembly Language 49
Assembler Labels 50
The ADR Directive 51
BASIC from the Assembler 52
Passing Data 53
Returning Values 53

5 : The ARM Instruction Set 55

Conditional Execution 56
Condition Codes and the Assembler 56
Conditional Execution After Comparisons 63
Controlling the Status Flags 66
Mixing Conditional and S Suffixes 66
Instruction Groups 68

6 : Data Processing – Format 69

Opcode Mnemonic 69
Destination 70
Operand Two : A Simple Register 70
Operand Two : An Immediate Constant 71
Range of Immediate Constants 71
Operand Two : A Shifted Register Operand 74

7 : Shift Instructions 77

Data Processing Instructions 77
Logical Shift Left : LSR 77
Logical Shift Right : LSR 79
Arithmetic Shift Right : ASR 81
Rotate Right : ROR 83
Rotate Right With Extend (One Bit Only) : RRX 84

8 : Processing Instructions 85

ADD : Addition 85
ADC : Add with Carry 87
SUB : Subtract 89
SBC : Subtract with Carry 90
RSB : Reverse subtract 91
RSC : Reverse subtract with Carry 92
MOV : Move data 93
MVN : Move Inverted Data 94
CMP : Compare 95
CMN : Compare negative 99
AND : Logical AND 100
ORR : Logical OR 101
EOR : Logical Exclusive-OR 102
BIC : Bit Clear 103
TST : Test Bits 104
TEQ : Test Equivalence 106
MUL : Multiplication 107
MLA : Multiplication with Accumulator 109

9 : Register R15 110

Register with Data Processing List 110
Register R15 as Operand One 110
Register R15 as Operand Two 110
The Program Counter and Pipelining 111
Register 15 as the Destination Register 113

10 : Data Transfer 115

Between Memory and Registers 115
Accessing Memory 115
Addressing Modes 115
Indirect Addressing 116

Pre-indexed Addressing 117
Simple Register 118
An Immediate Constant 119
Shifted Register 120
Using Write Back 121
Post-indexed Addressing 122
PC Relative Addressing 124
Byte and Word Addressing 125
Multiple Register Transfers 125
STM 126
Direction of Storage 127
Pre or Post-address Modification 127
Write Back 129
Applications of STM, LDM 130

11 : Branches and SWI 131

Simple Branch (B) 131
Conditional Branches 132
Branches and Conditional Instructions 133
Branch with Link : BL 134
Preserving the Link Register 137
Software Interrupt : SWI 138

12 : Stacks and LDM/STM 140

Computer Stacks 141
Types of Stack 142
Implementing Stacks using LDM and STM 142
Stack Applications 146

13 : The BASIC Assembler 2 147

OPT Settings 147
Error Control 148
Offset Assembly 151
Storing Data in Assembly Programs 152
The ALIGN Directive 154
CALL Parameters 155
The Operating System from BASIC 158

14 : Techniques & Debugging 161

Macro Assembly 161
Conditional Assembly 164
Mixing Macros and Conditional Assembly 166
Debugging Machine Code Programs 167
The Debugger 167
Using the Debugger 168
Breakpoints 168
Examining Memory and Registers 169

15 : Interrupts and Events 172

Interrupts on the Archimedes 173
Disabling Interrupts 173
Interrupt Processing 174
Returning From Interrupts 175
Writing Interrupt Routines 176
Events 177

16 : Vectors 179

ARM Hardware Vectors 180
Software Vectors 181
Intercepting Vectors 181
Claiming Vectors 182
Releasing Vectors 182
Writing Vector Intercept Routines 182
The Operating System Vectors 183
Main Line System Vectors 184

17 : OS SWI Routines 190

Input/Output Facilities 191
Character Input/Output 191
String Input/Output 192
Conversion Routines 195
Other Conversion Routines 196
System Calls 200
Interrupt Driven Routines 205

18 : The WIMP Environment 207

Controlling the WIMP Environment 207
Accessing the Mouse 208

Initialising the WIMP 210
WIMP Windows 210
Creating Windows 212
Icons 214
Defining Icons 215
Opening Windows 216
Polling the WIMP 218
Simple Window Program Example 221

19 : Managing Fonts 226

The Character Fonts 226
Initialising a Font 227
Painting Text in Different Fonts 228
Anti-aliasing 230
Setting Up the Anti-aliasing Colour Palette 232
The Anti-aliasing Transfer Function 234
Changing the Painting Colour 236
Losing Fonts 238

20 : Templates and Input/Output 239

Input/Output 240
String Manipulation 240
Miscellaneous Statements 241
Control Constructs 241
Graphics 241
Template Format 242
Register Use 242

21 : Manipulating Strings 252

Representing Strings 252
String Manipulation Routines 253
String Assignment 253
String Concatenation 255
String Comparison 256

22 : Functions, Operators... 270

SGN 270
ABS 270
DIV and MOD 271
Logical Operators : AND, OR, EOR 273

Logical Operators : NOT 273
Arrays 274
Dimensioning Arrays 274
Array Access 275
SOUND 277

23 : Control Statements and Loops 279

IF...THEN...ELSE...ENDIF 279
Multi-condition IF..THEN..ELSE Statements 282
Non-numeric Comparisons 284
REPEAT...UNTIL 285
WHILE..ENDWHILE 285
FOR...NEXT 286
CASE Statement 289
Procedures 291
Local Variables 292
Parameter Passing 293
Example of Recursive Procedures 293

24 : Graphics Templates 297

VDU, PLOT 298
SWI PLOT 302
MOVE , POINT 303
DRAW, BY 304
LINE 305
CIRCLE 307
Filled Circles 309
RECTANGLE 310
Outline Rectangle 311
FILL 312
ORIGIN, MODE, CLS, CLG 313
COLOUR 314
GCOL 316
POINT 317
ON, OFF 317
WAIT 318

Appendices 319

A : Representing Numbers 320
B : Computer Arithmetic 330
C : Logic Operations 335
D : Instruction Set Format 339
E : OS SWI Routines 343
F : OSBYTE Routines 346
G : OSWORD Routines 350
H : VDU Control Codes 351
I : Plot Codes 352
J : Programs Disc 354
K : Dabhand Guides Guide 356

Index 361

Program Listings

2.1 Words, bytes and word-aligned addresses 22
4.1 Entering the assembler from BASIC 43
4.2 Simple moving character 46
4.3 Fully commented version of listing 4.3 49
4.4 A simple loop using labelled addresses 51
4.5 Passing data to and from machine code 54
5.1 Letter print 57
8.1 Simple two-word addition 85
8.2 Simple two-word subtraction 89
8.3 A demo of comparison and condition codes 96
8.4 Case conversion using the ORR instruction 101
8.5 Toggling data using the EOR instruction 102
8.6 Printing binary 104
8.7 Multiplying two numbers together 107
9.1 The effect of pipelining the program counter 111
9.2 Skipping instructions 112
10.1 Demo of post-indexed indirect addressing 119
10.2 Accessing tables with indirect addressing 120
11.1 Branches and loops 132
11.2 Subroutines using branch with Link 120
12.1 Example of machine code stacks 145
13.1 Forward references 149
13.2 Forward references using two-pass assembly 150
13.3 Using the EQU directives 153
13.4 The ALIGN directive 155

13.5	Passing strings to machine code	158
14.1	The 'Beep' macro	162
14.2	Conditional assembly – a demonstration	165
17.1	Converting a number to a hexadecimal string	197
17.2	Converting numbers to binary	199
17.3	Manipulating characters using OSWORD 10	201
17.4	Using OSCLI to catalogue a disc	203
18.1	A simple sketch pad using the mouse	209
18.2	Example of creating windows	222
19.1	Painting text in the 'Trinity' font	229
19.2	Demonstration of anti-aliasing shading	233
19.3	Painting text in different colours	236
20.1	INPUT template	244
20.2	Demo of INKEY template from machine code	246
20.3	SPC(n) template	248
20.4	TAB(n) template	249
20.5	TAB(x,y) template	250
21.1	String assignment	254
21.2	String concatenation	255
21.3	String comparison	256
21.4	String length (LEN)	258
21.5	LEFT$ template	259
21.6	RIGHT$ template	260
21.7	MID$ template	262
21.8	INSTR template	263
21.9	STRING$ template	265
21.10	VAL template	267
21.11	STR$ template	268
22.1	Template to perform DIV and MOD	271
22.2	Array access in machine code	275
22.3	Simple sound effects	278
23.1	Example of using the FOR...NEXT template	281
23.2	A FOR...NEXT loop in assembly code	287
23.3	Example of the CASE template	290
23.4	An example of recursive procedures	294
24.1	Example of a PLOT command from machine code	300
24.2	Example of a LINE template	305
24.3	Example of a CIRCLE template	308
24.4	Printing coloured stars	314
A.1	Binary patterns	322

This Book and You!

This book was written using the InterWord wordprocessor on a Master 128 micro. The author's files were edited after transferring them to VIEW. The finished manuscript was transferred serially to an Apple Macintosh where it was typeset in MacAuthor. Final camera-ready copy was produced on an Apple LaserWriter IINT from which the book was printed by A. Wheaton & Co.

Correspondence with Dabs Press, or the author, should be sent to the address on page 2 or via electronic mail on Telecom Gold (72:MAG11596) or Prestel (942876210). An answer to your letter or mailbox cannot be guaranteed, but we will try our best.

All corresponents will be advised of future publications, unless we receive a request otherwise. Personal details held will be provided on request, in accordance with the Data Protection Act. Catalogues detailing the full range of Dabs Press books and software are available on request.

1 : Introduction

The Archimedes and the ARM

The Archimedes is the revolutionary new micro from Acorn Computers. It follows a long line of famous predecessors including the successful BBC micro, the BBC B+ and the current Master series. The Archimedes, however, is unlike anything which has gone before – it is a totally new machine. While remaining as compatible as possible with earlier models, it represents a major new departure for Acorn and an exciting leap forward for Acorn enthusiasts.

The Archimedes is unique in many ways – it has stunning multi-coloured graphics, a stereo sound system and supports a Window and mouse environment, to mention a few. However, perhaps the most startling innovation is the totally new microprocessor used in the system.

Acorn has moved away from the familiar 6502 used in earlier machines. For the Archimedes, Acorn developed its own processor using the most up-to-date ideas and technology.

Called the Acorn RISC Machine – or simply the ARM – it is the power of this remarkable chip which provides the advanced facilities of the machine. The ARM out-performs not only the 6502, but also most comparable processors, including the much-used MC68000.

RISC Design

The ARM represents a totally new philosphy in microprocessor design. It is an example of a Reduced Instruction Set Computer (RISC). It is called this because the designers have dispensed with many of the unnecessary and inefficient instructions found on many processors. The RISC chip is equipped with relatively few instructions, but these few are flexible, powerful and optimised so that they can be processed exceedingly fast. This gives the ARM unprecedented power which, until now, was only available on larger and more expensive machines.

To be able to program the ARM processor directly, we must be able to communicate with it in its own language – ARM machine code. This is very different to the high-level languages, such as BBC BASIC, which most people are familiar with. Machine code programs are simple sequences of numbers and data, held in the computer's memory, which have some significance to the ARM processor.

Faced with the task of writing machine code programs in this numeric form – most of us would probably give up! However, to help us in our task, the Archimedes provides a superb ARM machine code assembler. This allows us to write machine code programs in a more understandable form, using assembler statements which are then translated into actual machine code data. It is not difficult to program in assembly language, it just requires the use of certain special techniques.

This book aims to provide a complete tutorial course in writing ARM assembly code programs on the Archimedes computer. It explains the special elements which make up the ARM processor, and how these elements are used to execute machine code programs.

For the complete beginner, there is a section containing a comprehensive guide on fundamental topics such as number bases, binary and machine arithmetic, and logic. This enables readers with only a general knowledge of BASIC programming, to learn the concepts and ideas used in the book in a step-by-step way.

The book describes each of the machine code instructions provided by the ARM, together with explanations and examples of how they are used to construct machine code programs. Various assembly programming techniques are covered, such as memory allocation, access, data structures, control constructs and so on.

The powerful Arthur operating system used in the Archimedes is covered, with details of how to access its many facilities from the machine code level. How graphics, sound, windows, the font painter and the mouse work from within machine code programs are covered.

To make the transition from BASIC to machine code as painless as possible, the book contains a section on implementing BASIC statements in machine code. All of the most useful BASIC statements are covered and for each an assembler 'template' routine is developed which will mimic the statement's function in machine code.

Throughout the book, you will be encouraged to put theory into practice by trying out example programs on your own machine. To save typing these into the computer, the accompanying programs disc contains all the programs used in the text. The disc also includes some extra utilities not covered in the book such as a complete memory editor, disassembler and other utilities. Full details can be found in Appendix J.

Notation

A standard notation has been adopted throughout the book. The symbols below have the following special meanings:

The & symbol	This signifies that the following number is in hexadecimal. For example, &12CA
The % symbol	This signifies that the following number is in binary. For example, %10100111010101010100011100110100
>> and <<	These are BASIC shift operations and are described in the Archimedes BASIC manual
< > brackets	Brackets are used extensively when giving the syntax of various instructions and commands. The two angled brackets mean that the word between should not be taken literally. It simply refers to what sort of object should be used with the command. For example: <Register> means that a register name should be used in the brackets.
{ } brackets	Unless otherwise stated, the curly brackets mean that the object contained in them is optional and can be omitted. For example: ADD {S} means that the 'S' argument is optional to the instruction

Acknowlegements

Thanks are due to Siobhan FitzPatrick for reading my manuscript and pointing out, in as tactful a way as possible, my many mistakes! Thank you to Tony Goodhew for all the support, ideas and practical advice given on

this and other projects. Thanks are also due to Charlie, Mike, Andy and Robert for putting up with me while I wrote the book. James Knight gave invaluable help with the demonstration programs. Thankyou to Mark Gould-Coates for his understanding, help and general comments on all aspects of the book. Special thanks are also due to Jeff Fidler for his encouragement and support, and for providing some very welcome distractions during the writing of difficult parts of the book! Finally, a big thank-you to Bruce and David for publishing the book – and coming up with the idea in the first place!

Dedication

This book is dedicated to my parents and family for all the help, support and encouragement they have given me over the years.

2 : An Overview of the ARM

Before embarking on a detailed examination of the ARM chip and how it is programmed in assembly language, it's important to understand some fundamental computer principles.

In this chapter, we shall consider a general model of a computer and see how the ARM chip fits into this. We shall also examine how the ARM communicates with other parts of the computer and with the outside world. Finally, the way in which the ARM executes machine code instructions will be investigated.

A Typical Computer System Model

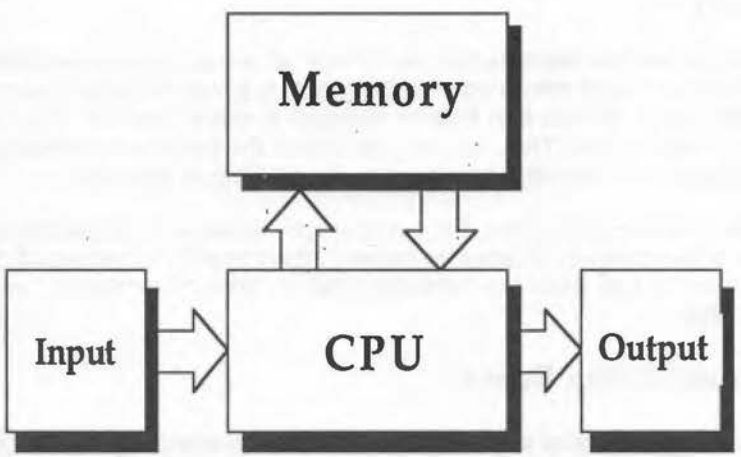


Figure 2.1. A model of a typical computer system.

At the simplest level, most computers can be represented by the model in figure 2.1. Data is obtained from the input to the system. It is then worked on by the central processing unit (CPU) and the results produced are sent to

the output. The main memory of the computer is used during the processing as workspace. It holds the program being executed, the data being processed and intermediate results produced. Any computer system must, therefore, resolve the problems of how to connect these separate elements so that data can pass efficiently between them.

Input/Output

In the majority of computer systems, including the Archimedes, the input/output is handled in the same way as the memory itself. This is known as device memory mapping. Physical input/output devices, eg, the disc controller, keyboard interface, video chip and so on, are made to appear as normal memory locations in the memory map to the processor. When the processor accesses these locations, it in fact accesses the hardware registers of the corresponding device. Data can thus be passed to and from devices simply by reading and writing the associated memory locations. This scheme provides a uniform way for the CPU to communicate with the outside world. The remaining problem is connecting the CPU and memory so that any arbitrary location can be accessed.

Memory

Memory on the Archimedes can be viewed as a long sequential series of bytes (there are eight bits in a byte). Each byte is given an identifying number starting at '0'. So the first byte of memory is called location '0', the next location '1' and so on. Thus we can talk about the processor accessing the data in location 'n', which means we use the nth byte of memory.

To access memory, therefore, the CPU requires some way of specifying the number of the memory location to be used. It also needs a method of transferring data to and from the memory. This is done by using the address and data bus.

Communication Buses

A bus is simply a series of electrical signal lines connecting the CPU to the other elements in the computer system. Each physical line in the bus can represent a single binary bit, ie:

+5 volts = Logic 1
0 volts = Logic 0

By placing combinations of +5 and 0 volts on the separate lines in the bus, binary numbers can be represented and transferred around the computer system. The number of signal lines, or bits, in a bus is called the bus width. Thus, we can talk about buses which are eight, 16 or 32 bits wide.

The Data Bus

The data bus, as the name suggests, is used by the CPU to pass data to and from the computer's memory. It is called a bi-directional bus because data can flow in either direction. In data storage operations, the processor puts the data on to the data bus and the memory reads it. In load operations, the processor requests the memory to put data on to the data bus, which it then reads.

The ARM processor is a 32-bit machine. This means that its data bus is 32 bits wide. This has far-reaching consequences on the performance of the Archimedes and explains, at least in part, why the computer is so powerful. The provision of a 32-bit data bus means that larger pieces of data can be processed in single operations.

An example should illustrate the point. Supposing we wanted to add together the contents of two integer variables. Each of these are 32-bits long. The 6502 processor on the BBC micro, has an 8-bit data bus and would therefore have to process the numbers in four, single-byte chunks. It would have to perform four load operations, four additions and four stores. On the ARM processor the two numbers could be loaded in their entirety and added together in a single operation. This gives a huge speed advantage over 8-bit machines as the number of memory accesses and processor operations is drastically reduced.

Words

A very useful, if slightly vague, concept often quoted when referring to memory is 'word'. A 'word' of memory is a logical unit defined as the number of bits manipulated in parallel by the processor in single operations.

Unfortunately, the definition of a memory word is not universally accepted and tends to vary from computer to computer. For example, the BBC micro's 6502 was an 8-bit machine and clearly manipulated eight bits of data at a time. It should therefore be talked about as having an 8-bit word length. However, in most applications, 16-bit quantities were more often

needed. It was very common, therefore, to talk about words when actually meaning these 16-bit quantities.

This is strictly incorrect, as the 6502 cannot handle 16 bits of data at a time. Sixteen-bit quantities actually had to be accessed by the 6502 in two separate chunks of eight bits. Nevertheless, the terminology persisted and this can be confusing.

The ARM manipulates data of 32 bits in length. Words on the Archimedes are therefore defined as being 32-bit quantities. It is important to appreciate and understand this difference between words on the BBC micro and on the Archimedes.

To avoid any confusion, in this book, we shall *always* refer to words as being 32 bits of memory unless otherwise stated.

The Address Bus

Obviously, the data bus does not provide a complete memory access system. An address bus is also needed so that the CPU can specify which location in memory is to be accessed. The CPU places the address of the required location on the address bus in binary. The memory decoder then reads this and sends control signals to the memory. These cause the relevant memory locations to respond and take part in the transferral of data over the data bus.

The width of the address bus specifies the size of the memory which can be accessed by the CPU. For example, on the BBC micro machines, the 6502 CPU had a 16 bit address bus. This means that 2^{16} different numbers can be represented on it and thus, 2^{16} different memory cells can be addressed. The maximum amount of memory available on these machines, ignoring paging techniques such as sideways memory, is therefore 2^{16} bytes = 65535 bytes = 64 kilobytes.

On the ARM processor, the address bus is 26-bits wide. This allows the Archimedes to have up to 67108864 bytes of memory (64 megabytes). On production machines, 0.5 megabytes, one megabyte, or four megabytes of writable memory are actually provided. This is still very large and will seem massive to anyone who is used to managing with the 32k provided on the standard BBC B computer.

In general, the size of memory which can be accessed via the address bus is called the address space. Thus the Archimedes has an address space of 64 megabytes even though, in paractice, not all of this memory is provided.

Byte and Word Accessed Memory

We have already noted that the memory on the Archimedes is byte-organised. That is, each byte of memory has its own unique address. However, we have also seen that the ARM processor has a 32-bit data bus and accesses memory in 32-bit chunks (four bytes). This apparent discrepancy occurs because the ARM can access memory in two ways.

In most cases it will be convenient to use the full power of the 32-bit data bus and access memory as complete 32-bit words. However, in some cases, for example when manipulating 8-bit quantities, it will be more convenient to access bytes individually from anywhere within the memory map. The ARM processor supports both methods, and it is to allow for byte access that each byte of memory has a unique address.

When accessing complete words of data (32 bits in length), the memory can be regarded as being split into separate chunks of four bytes (32 bits) in length. This is illustrated in figure 2.2.

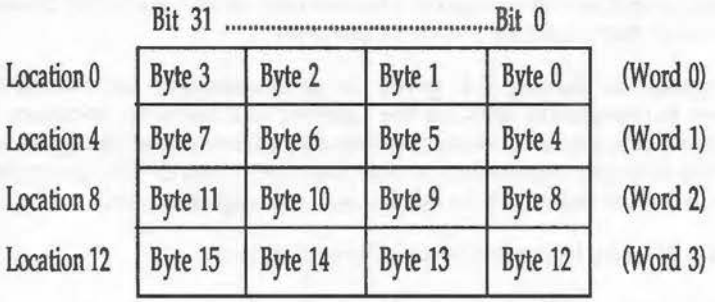


Figure 2.2. Byte and word-organised memory.

Word '0' starts at location 0 and includes bytes 0, 1, 2 and 3, word '1' starts at location 4 and includes bytes 4, 5, 6 and 7 and so on. Any complete word can be accessed by the ARM in a single operation.

When specifying which word we want to access in memory, we give the address of the location at which it starts. So the address of the first word is 0, that of the second word is 4, the third is 8 and so on.

Word-aligned Addresses

A memory address which corresponds to the start of a word is called a word boundary and is said to be word-aligned, ie, it is divisible by four. The following addresses are all word-aligned:

&00000000
&00000004
&00000008
&0000000C
&00000010

Word-aligned addresses are especially significant to the ARM. When accessing a word of memory, the address given must be word-aligned. For example, we could not access a word consisting of bytes 2, 3, 4 and 5, by specifying location &00000002 as the word address. This is because &00000002 is not a word-aligned address and so the required bytes are in fact split over two separate words of memory.

The program in listing 2.1 gives a demonstration of word-aligned addresses. It repeatedly asks for the address of a memory location. It then prints out which memory word contains the address, and the byte number which the address represents within the given word. The program also tells you whether the entered address is word-aligned or not.

Listing 2.1. Words, bytes and word-aligned address.

```
10  REM Word-aligned Addresses
20  REM (c) Michael Ginns 1987
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50  REPEAT
60  MODE 3
70  INPUT "Enter the address: " address
80  PRINT
90  IF address MOD 4 = 0 THEN
```

```
  100  PRINT "Word-aligned"
  110  ELSE PRINT "Not word-aligned"
  120  ENDIF
  130  PRINT "Word containing this address is: " address DIV 4
  140  PRINT "Within this word, address is byte number: "; address
MOD 4
  150  PRINT ' "Enter another address ? (y/n) : ";
  160  UNTIL GET$ ="n"
```

The significance of word-aligned addresses will crop up again when we consider ARM machine code instructions, as each of these must start on a word boundary. They are described in detail in a later chapter.

Virtual Memory

Before leaving the subject of how the ARM processor organises its memory, it is useful to look at how the physical memory is spread over the available address space.

We have seen that the Archimedes address bus supports a maximum memory size of 64Mb. Currently, however, a maximum of only 4Mb of writeable memory is provided. How then is this physical memory distributed over the much larger 64Mb address space?

The simplest scheme, assuming a 4Mb system, would be to make addresses 0-4Mb correspond to the available memory, and to make addresses higher than this invalid. However, things are not as straightforward as this! The allocation of physical memory is in fact controlled by a highly-sophisticated memory management chip called MEMC. This chip can be programmed to make blocks of real memory appear at any address in the system. Thus, the 4Mb of memory would not appear as one contiguous area, but would be split into blocks which could exist anywhere in the memory map.

The next question is: what happens if we try to access a memory location at which no 'real' memory exists? The answer is that the MEMC chip complains and sends an abort signal to the ARM processor. This normally causes an error message to appear on screen. However, it is possible to trap this event and use it to implement what is called virtual memory.

In a virtual memory system, the computer's main memory is supplemented by some form of secondary or backing store - usually a hard disc. The secondary memory is typically much larger than the main memory, but will have a slower access time.

The program running in the machine assumes that memory is provided over the whole address space (64Mb in the Archimedes). In reality, however, this memory is actually held on the hard disc.

As long as the program accesses locations at which real memory exists, then everything operates normally. However, if an area of non-existent memory is accessed, then the abort error occurs. This is trapped and a special software routine is called. This routine determines which area of memory the user was attempting to access. It then loads the corresponding block from the hard disc into main memory, replacing a previously loaded memory block. The user routine can then access the required data as if it had been present all the time! The only difference being that there is a slight time delay introduced by disc activity. In this way the computer's main memory is used as a 'buffer' into which chunks of the larger hard disc memory are loaded as they are needed.

Virtual memory is not currently implemented on the Archimedes, but the hardware to support it does exist. It could, therefore, be added to it as an expansion in the future.

Executing Machine Code Instructions

To complete our overview of the ARM system, we will look at how machine code instructions are obeyed by the ARM.

Machine code instructions are binary numbers which have some significance to the processor. Typically, a group of bits in the instruction will define the operation which the processor is to perform. Another group will then tell the processor where to get the data. Further bits may control the use of special options to the instruction and so on. For example, the following 32-bit binary pattern below is the ARM machine code instruction to add two numbers together:

%11100000100000010000000000000010

Instructions can therefore be held in memory, like any other piece of data, and moved into the processor using the address and data buses. The ARM processor works by continually repeating a simple sequence of operations. This is commonly known as the fetch-execute cycle and consists of three main parts as follows:

1) Fetch instruction

2) Decode instruction

3) Execute instruction

In the first part, the address of the instruction to be obeyed is placed in the address bus. The complete instruction, which is always 32-bits long, is then fetched from memory, over the data bus, to the ARM.

In the second part, the previously fetched instruction is decoded. This involves looking at the bit pattern making up the instruction, and deciding which of the possible operations in the ARM's instruction set it represents.

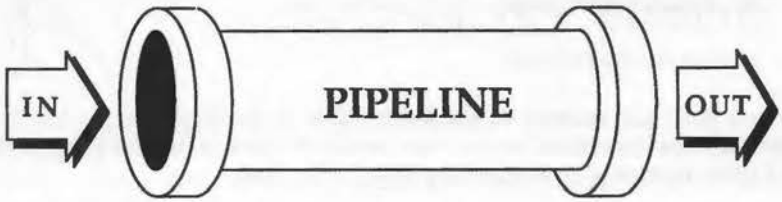
In the final part, the previously decoded instruction is executed. That is, the operation which the instruction specifies is carried out by the hardware elements of the CPU.

Pipelining

A special feature of the ARM processor is that the three parts or phases just mentioned are independent, and are performed by separate sections of the processor. They can, therefore, be overlapped. Obviously we can't overlap the fetching, decoding and executing of the same instruction! However, when an instruction has been fetched, there is no reason why the ARM cannot begin fetching the next one while the first is being decoded. Similarly, while the first instruction is being executed, the second can move on to be decoded and a third instruction can be fetched and so on. This obviously makes the machine very fast!

The ARM exploits this idea by overlapping all three phases of the cycle. Thus, at a given time, the ARM could hold three different instructions. The first having just been fetched, the second in the process of being decoded and the third being executed.

Internally, the ARM holds the three instructions in a hardware element called the 'instruction pipeline'. Instructions move along the pipeline through each of the three phases in turn. New instructions are fetched in at one end of the pipeline and the completed, executed instructions appear at the other. This scheme is illustrated in figure 2.3.



Cycle 1	Instruction 1 Fetched	<Empty>	<Empty>
Cycle 2	Instruction 2 Fetched	Instruction 1 Decoded	<Empty>
Cycle 3	Instruction 3 Fetched	Instruction 2 Decoded	Instruction 1 Executed
Cycle 4	Instruction 4 Fetched	Instruction 3 Decoded	Instruction 2 Executed

Figure 2.3. Pipelined execution of instruction.

As you can see from figure 2.3, it takes three cycles to fill the pipeline in the first instance. However, from this point onwards the overlap of the phases means that the ARM is, in effect, executing one instruction for every cycle. (There are circumstances when the pipeline has to be 'flushed' and we have to start again from the empty state.)

The pipeline system allows the ARM to perform at least a degree of parallel processing of instructions. It attempts to ensure that all parts of the processor are fully utilised at all times. This highly efficient way of operating helps to explain some of the amazing speed of the ARM processor.

3 : Internal Architecture

We have looked at how the ARM communicates with the outside world, at how it organises memory access and, in general terms, how it processes instructions. Now we can probe a little deeper and examine the hardware elements which perform the operations specified in the processor's instruction set.

The Arithmetic Logic Unit

When the ARM obeys an instruction, the exact course which the execution phase follows depends on which instruction is being performed. At the simplest level it may involve moving data around the processor, or perhaps initiating further ARM-to-memory transfers. However, there is also a series of instructions which perform operations on pieces of data, transforming them and producing a result. Some of these operations are familiar arithmetic ones, eg, addition, subtraction, multiplication and so on. Some are based on logic, eg, ANDing, ORing, EORing and so forth. If you are unfamiliar with such operations, they are fully explained in Appendix C.

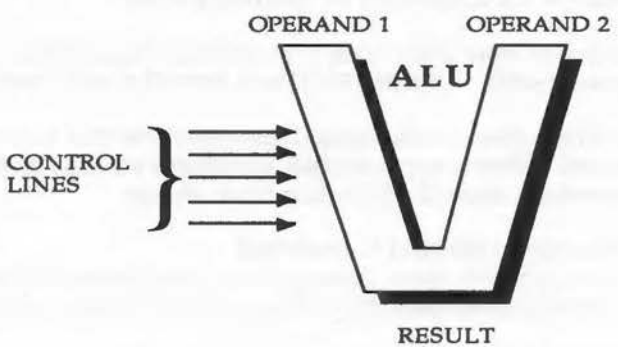


Figure 3.1. The arithmetic logic unit (ALU).

To carry out these operations, the ARM must have a special element of hardware. It must be capable of taking two data words as operands and process them, to produce the required result as output. This element is called the arithmetic logic unit (ALU), and is fundamental to the operation of the entire processor.

The ALU has, in addition to two data inputs and one result output, several control connections to the rest of the ARM (figure 3.1). The ALU can perform several different operations. These control lines tell the ALU which of these operations is to be performed on the data presented. The appropriate control signals are selected using the results obtained by previously decoding the instruction.

The Barrel Shifter

This oddly-named device operates together with the ALU to increase the overall processing power of the ARM.

Before the ALU performs any operation, it first must obtain two operands on which to work. The second operand is passed through a special element of the ARM on its way to the ALU. This is the barrel shifter, and it is used to apply one of several types of shift to the operand before it is used by the ALU.

By shifting an operand, all the bits in the operand data are moved a number of places (binary positions) to the left or right. For example, a shift of three places to the left could have the following effect:

```
Data before shift : %01101001100110011110010110101111
Data after shift  : %01001100110011110010110101111000
```

Shifts of one to 32 places can be carried out directly by the barrel shifter. In addition, several different types of shift operations are supported. The barrel shifter, therefore, takes the following three inputs:

1) The original operand to be shifted
2) The number of places through which the operand shifts
3) Control signals specifying the type of shift to be performed

The barrel shifter is shown diagramatically in figure 3.2.

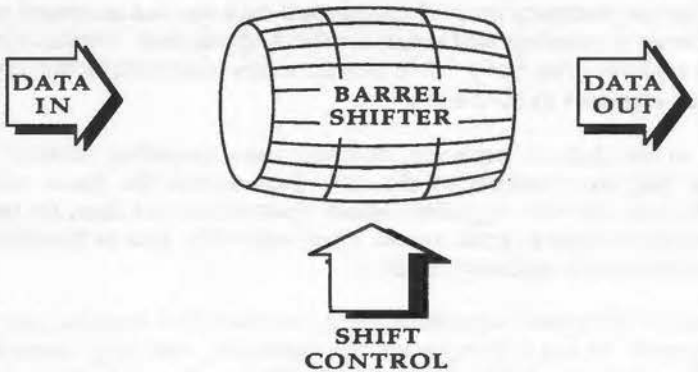


Figure 3.2. The barrel shifter.

Many central processing units, including the 6502, provide instructions to perform data shifts. However, they usually use the ALU to perform the shift operation, and are hence much slower than the ARM. Also, restrictions are often made on the types of shift available, or on the maximum number of places that the data can be shifted by. For example, the 6502 only allows shifting by one place at a time. However, the ARM can perform a single 26-bit shift operation – the 6502 must execute 26 single position shift instructions making it much slower.

The barrel shifter in the ARM implements shift operations in a general way. It allows shifts to be automatically applied to an operand used by any appropriate instruction. As it is a separate hardware element, no time penalty is incurred for using shifted operands - no matter how many places the data is shifted by.

Processor Registers

A register is a hardware element inside the CPU, and it can store data. Registers are used by the ARM to hold the operands needed by instructions.

Registers are somewhat analogous with memory locations, in that the bank of registers in the CPU can be thought of as being the private memory of the processor. The CPU can store and manipulate data in the registers, and can transfer data between external memory and the registers.

Remember that the registers are distinct from main memory. They do not appear in the memory map of the system and are not accessed using the normal way of placing addresses on the address bus. Instructions which refer to register data have fields within them which explicitly specify, by name, the registers to be used.

Access to the data in registers, is faster than accessing memory. This is because they are internal to the CPU. Just access the main memory to transfer data into the registers. Many operations can then be performed on the register data at great speed. Only when the data is finished with, is it returned to main memory again.

The number of registers provided by a processor has a major effect on processor power. If too few registers are provided, the programmer quickly runs out of them and data has to be shuffled back and forth between the registers and the slower main memory.

Registers on the ARM

If you are used to writing BBC micro machine code, you will know how restrictive the 6502 processor is as regards registers. There are only three registers available to the programmer (A, X and Y). There are rigid rules governing which registers can be used with different instructions, and how they must be used. For example, the accumulator register must be cited in all arithmetic logical operations.

On the ARM, things are very different. Acorn has provided a large number of registers. The use of these registers has been made as general and uniform as possible.

You have access to 16 registers. Each of these registers is 32-bits (one word) wide. They are referred to as registers R0, R1, R2 and so on, up to R15. Only two of these registers are used for special purposes by the ARM, the others are uncommitted. The register bank, as seen by the programmer, is summarised in figure 3.3.

R0	Uncommitted
R1	Uncommitted
R2	Uncommitted
R3	Uncommitted
R4	Uncommitted
R5	Uncommitted
R6	Uncommitted
R7	Uncommitted
R8	Uncommitted
R9	Uncommitted
R10	Uncommitted
R11	Uncommitted
R12	Uncommitted
R13	Uncommitted
R14	Link Register
R15	Program Counter

Figure 3.3. The ARM's register bank.

Uncommitted Registers

Registers R0 to R13 are totally uncommitted. Any one of them can be used in instructions which make references to a register. For example, supposing we want to carry out the following steps:

1) Load register R1 from memory
2) Add the contents of R1 and R2, placing result in R3
3) Multiply R3 by 10
4) Subtract R2 from R3, storing the result in R12
5) Shift the contents of R12 21 places left, add the contents of R6 and store the result in R0.

Each of these steps could be carried out by single ARM instructions, and the choice of which register to use is left to the programmer.

Special Purpose Registers

Of the remaining two registers (R14 and R15), one is permanently used for a special purpose by the ARM, while the other only occasionally takes on a special function.

R14 : The Link Register

The link register is register R14. It is used to provide a degree of support for implementing subroutines in machine code. The way this is done is covered in Chapter 11 when the BL instruction is described. Briefly, however, this instruction allows you to jump to another part of the program, execute a section of code and then return to the start point again. This is like the GOSUB instruction in BASIC. Register R14 is used to store the address. It must then be returned to after the subroutine has been executed.

So R14 is used by the ARM each time the branch with link instruction (BL) is carried out. At other times it is unused and is free for you to use in whatever way you want.

R15 : Program Counter and Status Register

Register R15 is a special purpose register dedicated to maintaining the ARM's program counter, status flags and mode flags. Figure 3.4 shows how R15 is organised.

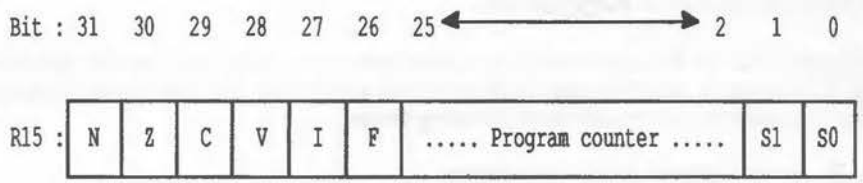


Figure 3.4. Register 15 - The program counter and status flags.

The Program Counter

Bits two to 25 of R15 contain the program counter. This is how the ARM keeps track of which instruction to fetch and carry out next. The program counter always contains the memory location address from which the next instruction is to be fetched.

The more observant among you may have noticed that the program counter only occupies 24 bits of register R15. Surely though, as the address bus of the ARM is 26 bits wide, we should also have a 26-bit program counter!

This would be true if it weren't for the fact that all ARM instructions must be word-aligned in memory. (The concept of word-aligned addresses was covered in Chapter 2.) Each ARM instruction must be stored in a word of memory whose address is divisible by four. As the lower two bits of such addresses are always '0', there is no need to store them in the program counter. In effect therefore, the program counter holds the word number of the next instruction.

When the ARM fetches an instruction, it places the contents of the program counter on bits two to 25 of the address bus and simply zeros bits zero and one before fetching the required instruction.

As we shall see, all ARM instructions are the same length (one word). After fetching an instruction, therefore, the ARM simply increments the program counter so that it points to the following word of memory. It is then ready to fetch the next instruction.

There are other ways of explicitly changing the contents of the program counter, for example, implementing branches. These will be described along with their relevant instructions in Chapter 11.

The Status Flags

The second use of R15 is to store the various ARM status flags. The first group of these (bits 28 to 31), make up what is called the status register. These status flags reflect the results of previous operations performed by the ARM.

For example, if two numbers are subtracted and the result is negative, the ARM could set the negative flag (N) to indicate this. Figure 3.5 shows the purpose of each of the status register flags. Later on, we shall see that there are instructions which explicitly test the state of the flags and will take different actions, depending on whether a flag is set or clear. Thus we can make the results of part of a program affect the execution of other parts and therefore create conditional statements – an essential requirement in most programs.

Bits 26 and 27 of register 15 contain two more flags which reflect the state of the interrupt system on the ARM. (Described in Chapter 15.)

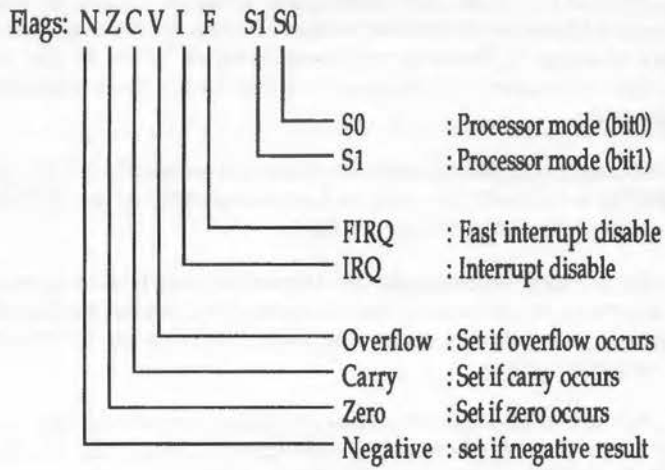


Figure 3.5. The ARM's status flags.

Setting the Flags

In some cases we need to set or clear flags explicitly. Processors, like the 6502, have their status flags stored in a special register which is not directly accessible. Therefore, they have to provide pairs of dedicated instructions to set and clear the flags. The ARM processor, on the other hand, implements its status flags in R15 – a normal user register. This can be accessed by the programmer like any other register. Thus, by storing appropriate data in it, any combination of flags can be set or cleared. This is a good example of how the RISC philosophy gives the processor extra power while reducing its instruction set.

Mode Flags

Bits zero and one of register R15 form two processor mode flags. The ARM can execute instructions in four distinct modes, and the current operating mode is always reflected in the mode flags. Figure 3.6 lists the four modes and shows the corresponding state for each flag.

S1	S0	Processor mode
0	0	User mode
0	1	Fast Interrupt mode (FIRQ)
1	0	Interrupt mode (IRQ)
1	1	Supervisor mode (SVC)

Figure 3.6. The mode flags and ARM processor modes.

User Mode

The normal mode in which our programs execute is called user mode. Unless we are involved in writing very specialised systems routines, this is effectively the only mode which will concern us. However, for the more adventurous - read on!

Supervisor Mode

The alternative to user mode is supervisor mode (SVC). All of the routines in the ARTHUR operating system work in this mode. When we ask it to perform some task for us, for example, reading mouse co-ordinates, there is an implicit change to supervisor mode. After the task is completed, control is returned to user mode.

Interrupt Modes

Interrupt mode and fast interrupt mode are entered in response to some external device in the computer system which interrupts the normal processing of the ARM, and demands immediate attention. Before executing appropriate code to deal with these situations, the ARM will automatically switch to the appropriate interrupt mode. The concept of interrupts is dealt with in Chapter 15.

Registers Available in Different Processor Modes

The register set available to the programmer varies according to which mode the processor is in. When executing in user mode, the normal set of registers, R0 to R15, are available. However, when the processor switches to one of the other modes, this changes.

For example, in supervisor mode, registers R13 and R14 effectively disappear from view. These are replaced by two new registers which we will call R13-SVC and R14-SVC. This means that instructions which would have accessed registers R13 and R14 in user mode, will now access the contents of registers R13-SVC and R14-SVC in supervisor mode.

The idea behind this system is that each processor mode has some private registers which it can use without affecting the values of the normal registers. This makes it unnecessary for the programmer to save the contents of all user registers when a special mode is entered. The private registers can be used freely without corrupting the data in the corresponding user mode registers.

Register	**Processor register accessed when in:**			
Name	**User mode**	**FIRQ mode**	**IRQ mode**	**SVC mode**
R0	R0	R0	R0	R0
R1	R1	R1	R1	R1
R7	R7	R7	R7	R7
R8	R8	R8_FIRQ	R8	R8
R9	R9	R9_FIRQ	R9	R9
R10	R10	R10_FIRQ	R10	R10
R11	R11	R11_FIRQ	R11	R11
R12	R12	R12_FIRQ	R12	R12
R13	R13	R13_FIRQ	R13_IRQ	R13_SVC
R14	R14	R14_FIRQ	R14_IRQ	R14_SVC
R15	R15	R15	R15	R15

Figure 3.7. The register bank in different processor modes

Alternative registers are made to replace the normal ones in each of the other processor modes. The ARM contains 25 programming registers of which 16 are visible in any given mode. Figure 3.7 illustrates this. For each mode it gives the physical register used when one of the registers R0 to R15 is accessed.

ARM Instructions

A major innovation of the ARM is its special RISC architecture. We cannot, therefore, leave our examination of the processor without mentioning some of the special features of the instruction set.

The RISC Concept

We have seen that the designers of the ARM have tried to make the processor architecture as general and flexible as possible. It should come as no surprise that the instruction set also follows this design philosophy.

The ARM is a Reduced Instruction Set Computer (RISC). This means that compared to other processors, it supports relatively few instructions. However, each instruction is designed to be as general and flexible as possible. Thus, a given instruction can be used in many different ways – each of which would have required a separate dedicated instruction in conventional architectures. This allows the ARM to perform similar operations to other processors but using a fraction of the number of instructions.

The advantage of this approach is twofold: First, the small number of instructions supported can be optimised to work as efficiently and quickly as possible. Second, the programmer is less constrained when writing machine code programs. The ARM's instruction set does not place needless restrictions on the programmer. This allows programs to be problem-orientated rather than implementation-orientated. In other words, the programmer can write machine code which matches the logical algorithm of a problem solution, rather than trying to program around the peculiar quirks of the processor's instruction set.

The 6502 in the BBC micros is an example of a Complicated Instruction Set Computer (CISC). A few example comparisons between this and the ARM processor should help to make the RISC advantage clear.

RISC Versus CISC

Instructions on CISC processors, especially the 6502, tend to be tailored to very specific purposes. They have a great many associated restrictions that exactly define which situations they can or cannot be used in. This results in a bewildering array of instructions which are very inflexible.

For example, the 6502 provides no fewer than four separate instructions to transfer data between its three programming registers (A, X and Y).

TAX	Transfer contents of A to X
TXA	Transfer contents of X to A
TAY	Transfer contents of A to Y
TYA	Transfer contents of Y to A

Figure 3.8. 6502 instructions for inter-register data transfer.

Even with these instructions there is still a restriction. If we want to move data directly between the registers X and Y, we are out of luck! Wouldn't it be better if there were a single generalised instruction which could move data between any two named registers? There is, because this is the approach that the ARM takes by providing the single move instruction.

This may be a slightly trivial example, but it does help to illustrate how a reduced number of instructions can provide added power.

Another example concerns the way instructions are allowed to reference their operands. It would be possible for the operands of all instructions to be held in memory and accessed directly by the ALU. Alternatively, some CPUs, including the 6502, hold one operand in memory and the other one in a register.

Both of these systems, however, inevitably result in added complexity in the instruction set. Each of the instructions has to have a host of variants. Each of these variants is a separate instruction, which performs the same operation, but which obtains the operands by accessing memory in a different way.

As you might expect, the ARM does things in a very different way. No data processing instruction accesses its operands from memory. Instead they simply reference the data held in the processor registers. A few instructions are then provided, which do have different addressing modes, to transfer data to and from the registers and memory in the first place.

Instruction Length

Those of you have programmed the 6502 processor with the BBC micros will know that its instructions can be one, two or three bytes long (eight, 16 or 24 bits) Being an eight-bit machine, however, it can fetch only one byte at a time over the data bus. The fetch/execute cycle of the 6502 is therefore:

1) Fetch byte one of instruction
2) Partially decode the instruction
3) If a complete instruction has not been obtained then fetch another byte and repeat this step
4) Fully decode instruction
5) Execute instruction

Up to three separate memory accesses may be required to simply fetch an instruction - let alone execute it!

The ARM has a 32-bit data bus and so can use a more uniform scheme. All ARM instructions are one word (32 bits) long. This allows a complete instruction to fetched over the data bus in one go.

In order for this to happen, however, all ARM instructions must be stored at word-aligned addresses in memory, that is, at addresses which are divisible by four. (See Chapter Two).

This does not present any problems in practice. If the first instruction in a piece of code is word-aligned then, as each instruction is one word (four bytes) long, instructions consecutively following it will also be word-aligned. As we shall see later, the Archimedes BASIC assembler provides a facility for controlling this for us.

Conditional Execution

A very special feature of the ARM is that the execution of any ARM instruction can be made to be conditional on the current settings of the status flags. This means that the instruction will only be executed if the status flags are in a specific pre-defined state. If this is not the case, then the ARM will ignore the instruction completely. The only effect of this being the small time delay introduced.

Most processors have branching or jumping instructions which work on the status flags. The ARM, however, generalises this idea to cover all instructions. A detailed account of the conditional execution facility and its use is given in Chapter Five.

Data Shifts

All data processing instructions supported by the ARM can have a shift operation applied to one of the operands. This was mentioned earlier when we looked at the ARM's barrel shifter.

Being able to apply shifts to the data used in any instruction, is a great source of power for the programmer. It allows some quite sophisticated effects to be achieved in very few instructions. We will look at the details of the system, including the types of shifts available and their usage in Chapter Seven.

4 : The BASIC Assembler

This chapter gives a brief introduction to the Archimedes BASIC assembler. It is intended to give just enough information to allow very simple machine code programs to be assembled on the Archimedes. This will make it possible for you to try out some of the machine code instructions covered in the next chapter, which deals with the ARM's instruction set. We will return to the subject of the BASIC assembler in Chapter 13, where some of the more complex aspects of the assembler will be described.

An assembler allows us to write our machine code programs in terms of symbols, mnemonic names and labels. This is called the assembler source code and cannot be executed directly by the processor. The assembler is used to translate this source code into machine code which the processor can obey. Each individual assembler statement is converted into the corresponding machine code instruction. This is illustrated in figure 4.1.

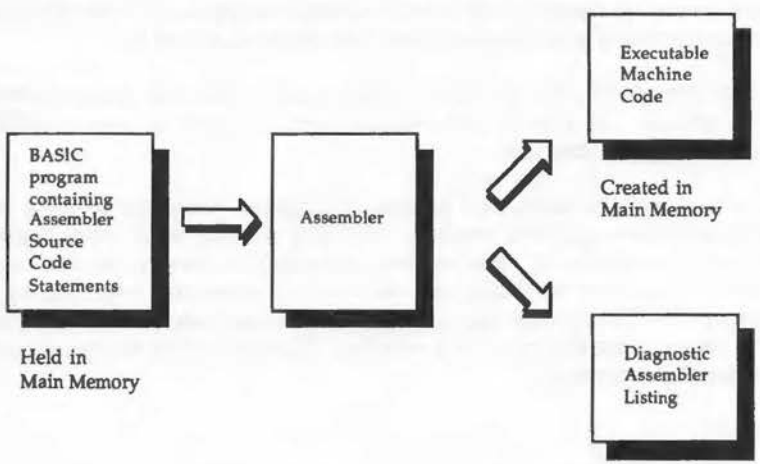


Figure 4.1. Assembling source code into machine code.

Without an assembler, the process of writing machine code would be very laborious and error-prone. The binary pattern representing each machine instruction would have to be remembered or looked up. Also the addresses and operands, used in the various instructions, would all have to be calculated by hand and given in numeric form.

Some typical machine code instructions are given in figure 4.2 in binary and hexadecimal format. Opposite these are the same instructions in assembly language. Don't worry about what the instructions do at this stage, they are purely to illustrate the advantage of using an assembler.

Binary	Hexadecimal	Assembler
%11100001101000000001000000000010	&E1A01002	MOV R1,R2
%11100000100000110001000000000101	&E0831005	ADD R1,R3,R5
%11100001010100010000000000000101	&E1510005	CMP R1,R5

Figure 4.2. Instructions in binary, hexadecimal and assembler.

General Format of ARM Assembler Instructions

All ARM instructions have a similar format under the assembler. A mnemonic name is used to specify which instruction is being used. This is followed by various operands which specify the data to be operated on.

The exact syntax of the operands varies with different instructions. A range of special characters and suffixes can be used to select different options with each instruction.

When an instruction refers to a processor register, there are several ways of specifying the register's number. We can simply write the register's number in the statement. This can be confusing, however, and so we can also write the number as R<n>, where <n> is the register number, eg, R10 for register 10. Finally, we can also quote a BASIC variable, the value of which is taken to be the register's number. The following examples are all legal under the assembler:

```
MOV  0,3
MOV  R0,R3
MOV  invader_status,destroyed
MOV  PC,R14
```

Note that the program counter, register R15, may also be referred to as 'PC' without setting up a corresponding variable.

The Assembler

The assembler provided on the Archimedes forms an integral part of the BASIC interpreter. This has the great advantage of always being available from within BASIC programs. BASIC and assembly code can be mixed freely. This results in hybrid programs which are part BASIC, part assembler. Also the full power of the BASIC interpreter is available from within the assembler allowing some very sophisticated facilities to be used.

The assembler source code is written as a series of numbered program lines, just like a BASIC program. These are delimited by special characters to inform BASIC that assembly code is being used. When a program of this sort is run, BASIC's assembler is called and the assembler statements are converted into machine code instructions which are stored in the computer's memory.

Entering the Assembler

The assembler is entered from BASIC by using the square brackets, []. These can be included, like any ordinary statement, anywhere in a BASIC program. BASIC will expect the program lines between these two brackets to contain ARM assembler statements.

When the first square bracket is encountered, BASIC's assembler will start to work its way through the following assembler statements, converting each to the equivalent machine code instruction and storing it in memory. This continues until the final square bracket is reached. This is the signal that the assembler code section is over and normal BASIC statements are to be executed again.

Listing 4.1. Entering the assembler from BASIC.

```
10  REM Entering The BASIC Assembler
20  REM (c) Michael Ginns 1987
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50  PRINT "This is BASIC"
60  [
70  ; THIS IS NOW THE ASSEMBLER
80  ; AND SO IS THIS
```

```
 90  ; THESE LINES ARE ONLY ASSEMBLER COMMENTS
100  ]
110  PRINT "BACK IN BASIC AGAIN"
```

Type in listing 4.1 to show how assembly code and BASIC can be mixed. When run, the program should produce the following output, although some of the numbers may be different.

```
>RUN
THIS IS BASIC
00000000
00000000            ; THIS IS NOW THE ASSEMBLER
00000000            ; AND SO IS THIS
00000000            ; THESE LINES ARE ONLY ASSEMBLER COMMENTS
BACK IN BASIC AGAIN
```

The Assembler Location Counter - P%

In the previous example we said that the machine code, produced by the assembler, was stored in memory. But where in memory? We must have some system for telling the assembler the address at which we want the machine code program to start. This is done on the Archimedes by borrowing one of BASIC's integer variables, P%.

P% has a special significance to the assembler. The number which it contains is taken as the start address of the memory area which is to contain the assembled machine code. If P% contains the number &8000, for example, then it will store its first assembled machine code instruction at location &8000 in memory.

Obviously, if we use the address stored in P% again to store the next assembled instruction, then it will overwrite the first. To prevent this happening, the number of bytes used to store an instruction is automatically added to P% after the instruction has been assembled. ARM instructions are *always* four bytes (one word) long, and so P% will be incremented by four each time.

Thus machine code instructions produced by the assembler are stored consecutively in memory, starting at the address originally contained in P%. At any given time, P% always holds the address in memory where the next instruction will be assembled to.

We shall see several examples of P% in action later on, but first we must examine how we chose the the initial P% address. In other words, how do

we select the area of the computer's memory in which to store our machine code programs?

Reserving Memory

The BASIC assembler makes no checks on the value of P% to determine whether or not the memory which it points to is 'safe' to use. Machine code can be assembled which will overwrite our assembler text program, operating system workspace, or produce some equally disastrous result. It is, therefore, most important that a suitable area is found to hold our machine code program.

On BBC micros, memory was in very short supply. Consequently, a whole series of 'tricks' were developed for cramming machine code programs into every conceivable space. On the Archimedes, memory is more plentiful so these practices are not necessary.

The simplest way to reserve a safe area of memory for our machine code, is to use a special form of the DIM statement. We are used to seeing DIM when declaring arrays, but it can also take the form:

```
DIM <VAR> <number>
```

Where <VAR> is any numeric variable and <number> is the number of bytes of memory to be reserved.

For example:

```
DIM code 1024
```

This will instruct BASIC to reserve 1024 bytes of memory, and will set the variable 'code' to the address of the first of these bytes. The address returned is also guaranteed to be on a word boundary, so no further correction is required. It is vital that the space we reserve is sufficient to hold the machine code program produced by the assembler. If in doubt, always reserve too much, rather than too little, memory.

We can now tell the assembler to use the reserved area of memory, for storing machine code in, by simply saying:

```
P% = code
```

A typical Archimedes assembler program will, therefore, take the following form:

```
10 DIM code 1024
20 P% = code
30 [
40
50 ; Lines containing the
60 ; assembler code program
80
90 ]
```

Having found somewhere to store our machine code programs, let's move on and actually write one! Type the program in listing 4.2. Don't worry at this stage that the instructions are unfamiliar to you, we will have an in depth look at the ARM instruction set later on. The program is only intended to illustrate general features common to most assembler programs.

Listing 4.2. Simple moving character.

```
 10  REM Simple Moving Character Program
 20  REM (c) Michael Ginns 1987
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 60  VDU 23,240,&3C3C;&FFDB;&1818;&E77E;12
 70  OFF
 80
 90  vdu = 256
100
110  DIM start 100
120  P%=start
130  [
140  .loop
150  MOV R0,#19
160  SWI 6
170  SWI vdu+8
180  SWI vdu+32
190  SWI vdu+240
200  B loop
210  ]
220
230  PRINT "PROGRAM ASSEMBLED AT : &"; ~ start
240  PRINT "PROGRAM SIZE IS : "; P%-start ;" Bytes"
```

Line Meaning

60-70 These perform some setting up operations for the program: redefining a character and turning the cursor off. This is an example of the hybrid machine code and assembler programs we came

across earlier. Parts of the program which do not need to be written in machine code can be left in BASIC.

90 Sets up the variable VDU to contain the value 256. This is used later in the assembler program. The number 256 could have been used directly in the assembler code, but the use of a named variable makes the program more readable. Also, if we need to change the number at some time, modifying the value in line 90 is all that is required.

110 These contain the familiar commands to reserve some memory for the machine code, and setting P% to the beginning of it.

130 At line 130 we leave BASIC and enter the assembler. The instructions on lines 140 to 2000 are assembler mnemonics for ARM machine code instructions.

210 We return back to BASIC again.

Assembler Listings

When the previous program was run, you should have been presented with an assembler listing like the one given in figure 4.3. Again some of the addresses may vary. The assembler produces a listing by default which shows what has been assembled and at which address.

```
000088D4
000088D4                .loop
000088D4 E3A00013  MOV R0,#19
000088D8 EF000006  SWI 6
000088DC EF000108  SWI vdu+8
000088E0 EF000120  SWI vdu+32
000088E4 EF0001F0  SWI vdu+240
000088E8 EAFFFFF9  B loop

PROGRAM ASSEMBLED AT : &88D4
PROGRAM SIZE IS : 24 Bytes
```

Figure 4.3. Listing produced for the character move program.

The first column of the listing is the address of each machine code instruction assembled. This is the value of P%, printed out after each instruction has been assembled. Remember that we said it was incremented by four bytes each time?

The next column gives the actual machine code instruction stored in memory. This is in hexadecimal.

The third column contains the assembler text which produces the instruction. This is the mnemonic form of the instruction. Even though you may not know what each of these instructions does, I am sure you will agree that they are much more readable than their equivalent hexadecimal machine code instructions!

Executing Machine Code Programs

By running listing 4.2, we have converted the mnemonic assembler instructions into machine code instructions. However, as yet we have not executed the machine code itself. To make the ARM processor to execute the machine code, we use BASIC's CALL command.

The CALL statement is followed by the address, or a variable containing the address, of the machine code program which we wish to execute. In the case of listing 4.2, we assembled the machine code to the area of memory pointed to by the variable 'start'. Thus to execute our machine code program, we type:

```
CALL start
```

This may be issued from within a program, or from command mode, but remember that the source program must have been assembled first! Try running the program again, then type 'CALL start' in command mode. You will see a little man-shaped character moving across the screen - which is all that listing 4.2 does!

Returning to BASIC

We will often want to go back to BASIC after executing machine code. We may be executing the machine code routine from inside a BASIC program, or may just want to return to BASIC's command mode.

When a machine code routine is called from within BASIC, a special address is placed automatically in register R14. If we make the ARM jump to this address, after our routine has been completed, then BASIC will be returned to at the point immediately after the original CALL statement.

We can accomplish this simply by moving the contents of R14 back into R15 – the program counter. This will cause the ARM to break off its normal sequential execution of instructions, and start executing them from the new address transfered into the program counter from R14.

The instruction to move data between two registers is described in detail in Chapter Eight. However, for this specific purpose we always use the following instruction:

```
MOV PC,R14
```

This can be regarded as a 'return to BASIC' instruction and should always be used at the end of our machine code routines.

Comments in Assembly Language

In BASIC we often add comments to our programs using the REM statement. REM stands for 'reminder' and these statements help to explain parts of the program, making it more understandable.

In assembler, it is even more important to add comments to our programs. The low-level nature of machine code makes assembler programs very unreadable at the best of times. Imagine coming back to modify one of your programs several months after it was written. Without explanatory comments it would be virtually impossible.

Comments are introduced into assembler programs using either a semicolon (;), a backslash symbol (\), or by a REM statement. Note, however, that when teletext mode 7 is used the backslash is displayed as a $^{1}/_{2}$ character. Any text following these characters, up to a new line or a colon (:), is ignored by the assembler, but is displayed in assembler listings.

The machine code instruction mnemonics in listing 4.2, are made clearer if they are commented. A fully-commented version of listing 4.2 is given in listing 4.3.

Listing 4.3. Fully commented version of listing 4.2.

```
10  REM Simple Moving Character Program
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
60  VDU 23,240,&3C3C;&FFDB;&1818;&E77E;12
70  OFF
```

```
 80
 90 vdu = 256
100
110 DIM start 100
120 P%=start
130 [
140 .loop
150 MOV R0,#19      ; Wait for 1/50th of a second
160 SWI 6           ; ( Reduces screen flicker )
170 SWI vdu+8       ; VDU 8
180 SWI vdu+32      ; VDU 32
190 SWI vdu+240     ; VDU 240
200 B loop          ; Jump back to beginning of program
210 ]
220
230 PRINT "PROGRAM ASSEMBLED AT : &"; ~ start
240 PRINT "PROGRAM SIZE IS : "; P%-start ;" Bytes"
```

Assembler Labels

When programming in assembler, we frequently want to refer to other parts of the program – perhaps to jump to another section of the code, or to access data stored elsewhere in memory.

We could do this by quoting the relevant address in a suitable ARM instruction. However, we frequently do not know the absolute address at the time of writing the assembler code. Also, if we make changes to the assembler program, it is quite likely that the addresses of given instructions within it will be different. To get around this problem, most assemblers (including the Archimedes) allow us to define labels within the assembler program.

A label is simply a name which is used to mark a given place within a section of code. When the assembler encounters the label definition, it will associate the name of the label with the current value of P%. Thus, the label is made to point to the address at which it was defined within the assembler program.

Subsequently, the label can be referred to and the assembler will look up then substitute the appropriate address. Changes to the program no longer cause problems as re-assembling will automatically re-calculate the addresses associated with all the labels.

On the Archimedes, labels are defined simply by writing their name prefixed with a dot character (.). Examples of valid labels are:

```
.explosion
.output
.loop2
.create_picture
```

The program in listing 4.4 contains a loop which repeatedly outputs '*' characters to the screen. The address in the program, which the ARM must loop back to, is marked using a label. The program can be RUN to assemble it and then executed by typing 'CALL star'.

Listing 4.4. A simple loop using labelled addresses.

```
 10  REM Using a Simple Loop to Print Stars
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM star 256       : REM Reserve space for machine code
 70  P%=star            : REM Set P% to start of reserved space
 80  [
 90  .beginning_of_loop \ Mark begining of program with a label
100  MOV R0,#ASC("*")   \ Move ASCII code for '*' into reg R0
110  SWI "OS_WriteC"    \ Output character in R0 to the screen
120  B beginning_of_loop\ Branch back to label at program start
130  ]
```

It is good practice to make labels 'meaningful', so that their name reflects their purpose. The names are constructed following the same rules used for choosing BASIC variables. In fact, defining a label in the assembler simply sets up a variable of that name, the value of which is the address of the label. Incidentally, this means that all variables used in a program can be listed out by using BASIC's LVAR command *after* it has been assembled. Try typing 'LVAR' after running listing 4.3 but *before* entering the CALL statement to execute it.

The ADR Directive

It is often very useful to be able to get the actual address associated with a label into one of the processor registers. For example, the label could mark the beginning of a data table in a program. For this we would need to have this address in a register to access entries in the table. The assembler provides the ADR directive for this purpose. The syntax of ADR is:

```
ADR <register>, <address>
```

<Register> is the name of the register which is to contain the address and <Address> is usually a label, the address of which is stored in the register.

Despite its appearance, ADR is *not* an ARM instruction. Neither does it simply move the absolute address of the label directly into the register. ADR is an assembler command (directive). When encountered the assembler will calculate the difference (offset) between the specified label's address and the current instruction address contained in P%. It will then assemble an appropriate ARM instruction (either an add or subtract instruction). This instruction, when executed, will use the offset in conjunction with the program counter to reconstruct the original address of the label, and store it in the given register.

BASIC from the Assembler

We mentioned earlier that the assembler on the Archimedes was part of the BASIC language. An added advantage of this arrangement is that many of the functions provided in BASIC are also available in assembler.

Almost any BASIC function which returns a numeric value can be used where a constant would normally be required in assembler. For example, the following instruction moves the ASCII value of a 'C' (67) into a processor register.

```
MOV R0,#67
```

However, to avoid looking up the value, we could write:

```
MOV R0,ASC("C")
```

This is a trivial example, but it should illustrate the principle. We can use any expression required – as long as the final result yields a number which is acceptable to the assembler. Some more examples of what is possible may help. Again, don't worry about what the actual machine instructions do, just look at the way in which their arguments can be given in terms of BASIC functions.

```
MOV R0,fred
ADD R0,R0,X*2+5
AND R0,R0,#%10100101
MOV R0,&FFEE
MOV R0,# (start MOD 256)
MOV R1,# (start DIV 256)
MOV R2,#INT(SIN((DEG(60))*100)
```

It is important to remember that all BASIC functions are evaluated at assembly time, not when the machine code is executed. The values returned are simply written as constants into the machine code instructions.

Passing Data: BASIC to Machine Code Routines

We will often need to pass data from a BASIC program to a machine code routine. The CALL statement has some advanced extensions for this purpose. It is described in detail in Chapter 13. If, however, we only want to pass a few integer values to our machine code routine, then we can do this using BASIC's resident integer variables A% to H%.

Just before transferring control to a machine code routine, BASIC copies the values of the integer variables A% to H% into the processor registers R0 to R7. Thus, up to eight, 32-bit integers can be passed from BASIC to our machine code routine very easily indeed.

Returning Values: Machine Code Routines to BASIC

If we want to pass an integer value back from a machine code routine to BASIC, then we can use the USR statement. It has the following syntax:

```
<var> = USR( <address> )
```

This is similar to CALL because it causes BASIC to execute a machine code routine at a specified address. However, when BASIC is returned to, USR returns the contents of register R0 as a value. Thus, by storing a result in R0, just before our machine code routine terminates, we can pass the result back to BASIC.

Listing 4.5 illustrates parameter passing and result returning. When executed, the routine passes two integer values using variables A% and B%. These specify a text character position on the screen. The routine moves

the cursor to this position, uses OSBYTE 135 to read the ASCII value of the character , and returns it to BASIC via the USR statement.

As a demonstration of the routine, a message is printed at the top of the screen. The routine is then used to read the characters from the top line of the screen and re-print the message at the bottom of the screen.

Listing 4.5. Passing data to and from machine code routines.

```
 10 REM Passing Integers to Machine Code Routines using A%-Z%
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 vdu = 256
 70 move_cursor = 31
 80
 90 DIM char_read 256
100 P% = char_read
110 [
120 \ co-ordinates of the character to be read are passed
130 \ using A% and B% into registers R0 and R1 respectivly
140 \ The character at this position is returned from reg R0
150 \ using the USR function
160
170 SWI vdu+move_cursor     \ Perform VDU 30
180 SWI "OS_WriteC"         \ Output x co-ord from Register R0
190 MOV R0,R1               \ Move y co-ord from Reg R1 to R0
200 SWI "OS_WriteC"         \ Output y co-ord from Register R0
210 MOV R0,#135             \ Move 135 into R0
220 SWI "OS_Byte"           \ Issue *FX 135 to read the character
230 MOV R0,R1               \ Move read character into R0
240 MOV PC,R14              \ Return to BASIC
250 ]
260
270 MODE 1
280 PRINT "The Archimedes Micro Computer System"
290 FOR n = 0 TO 39
300 A% = n
310 B% = 0
320 ascii = USR(char_read) : REM Read char at position(A%,B%)
330
340 delay = INKEY(20)
350 PRINT TAB(n,20) ;CHR$(ascii);
360
370 NEXT
380 PRINT
```

5 : The ARM Instruction Set

Much of the power of a processor depends on how it can be programmed and the range of operations it can perform. In this, and the following chapters, we shall look at one of the most important aspects of the ARM – its instruction set.

First, we shall cover some general features of ARM instructions. We shall then move on to describe fully the function of each of the instructions and how they may be used.

Conditional Execution

We have said that every ARM instruction is 32-bits long. These are divided up into groups of bits, called *fields*. One of these fields is used to store the instruction's condition code. (For a full description of the internal binary format of ARM instructions, see Appendix D.

The condition code field is four bits wide and can therefore be used to specify one of 16 conditions. The condition associated with an instruction must be TRUE when the ARM attempts to execute the instruction. If the condition is not met, then the ARM will not execute the instruction – it will effectively be skipped.

The condition code works by specifying which flags in the ARM must be set and which must be clear for the instruction to execute. Remember that the flags in the status register reflect the result of previous instructions. In particular, there is a comparison instruction which compares two operands and records the result in the status flags. This result can then be acted upon using conditional executed instructions. For example, the condition code:

```
%0100
```

means that the status register's negative flag (N) must be set for the instruction to execute, that is, a previous ARM operation must have produced a *negative* result.

The previous case was a very simple example of a condition which involved only one flag – the negative flag. Other condition codes specify more complex relationships between the status flags. For example, a condition code of:

%1011

requires one of the following to be true for the instruction to execute:

Either:	N flag	=	SET	and	V flag = CLEAR
or:	N flag	=	CLEAR	and	V flag = SET
or:	Z flag	=	SET		

This may seem a somewhat arbitrary relationship! However, if used after an instruction which compares two operands, it produces the result that the instruction is only executed if it was found that operand one was less than or equal to operand two.

Each of the 16 possible condition codes specifies a potentially very useful condition on which the execution of any instruction can depend.

Condition Codes and the Assembler

An instruction is specified as being conditional in the BASIC assembler by adding a two-letter suffix to the instruction's opcode mnemonic. There are 16 different suffixes available, one for each of the 16 possible condition codes. These are shown in figure 5.1.

EQ : Equal
NE: Not equal

VS: Overflow set
VC: Overflow clear

AL: Always
NV : Never

HI : Higher
LS : Lower than or same

PL : Plus
MI: Minus

CS : Carry set
CC : Carry clear

GE : Greater than or equal
LT : Less than

GT : Greater than
LE : Less than or equal

Figure 5.1. The assembler's condition code suffixes.

As an example, we could write:

```
SUBPL R0,R1,R2
```

The SUB mnemonic means that the ARM subtraction instruction is being used. (This is described in detail, along with the other instructions, in Chapter Eight.)

The PL suffix means that the subtraction instruction is only to be executed if the status register's negative flag is *clear*, that is, the result of a previous operation gave a positive result.

Each of the available suffixes will now be listed together with a description of the condition that they represent.

EQ: Equal

Condition : Z flag = Set

Instructions using this conditional suffix will be executed only if the zero (Z) flag is currently set. This will be the case if a previous operation gave a zero result. For example, subtracting two numbers of the same value can set the Z flag. If used after a comparison (CMP) instruction, it indicates that the two operands used in the comparison were the same.

Examples:

```
MOVS   R0, R1    Move data from register R1 to R0
MOVEQ  R0, #1    IF zero was moved into R0 then move one
                 into it

CMP    R5,R10    Compare contents of registers R5 and R10
ADDEQ  R5,R5,#2  IF they were equal then add two to R5
```

NE: Not Equal

Condition : Z flag = Clear

Instructions using this conditional suffix will be executed only if the zero (Z) flag is clear. This is the reverse case of the EQ suffix. Used after a CMP instruction, it indicates that the two operands used in the comparison were *not* the same.

Example:

```
CMP   R2,R0     Compare contents of registers R2 and R0
SUBNE R2,R2,R0  If not the same,then subtract them
```

VS: Overflow Set

Condition : V flag = Set

Instructions using this conditional suffix will be executed only if the overflow (V) flag is set. This flag is set as a result of an arithmetic operation producing a result which cannot be represented in the 32-bit destination register, that is, an overflow situation. In cases like these. the data placed in the destination register may not be valid and thus require special corrective action to retrieve the correct result.

VC: Overflow Clear

Condition : V flag = Clear

Instructions using this conditional suffix will be executed only if the overflow (V) flag is currently clear. This is the reverse case of the VS suffix. It indicates that no overflow has been detected.

MI: Minus

Condition : N flag = Set

Instructions using this conditional suffix will be executed only if the negative (N) flag is set. This flag is set as a result of an arithmetic operation producing a result which is less than zero. This could be the case if we subtract a number from a smaller one. Also logical operations, which cause bit 31 of the destination register to be set, may also set the negative flag.

Example:

```
SUBS  R0,R0,#5   Subtract five from the contents of R0
ADDMI R0,R0,#5   If it gave a negative result,
                 Add five again
```

PL: Plus

Condition : N flag = Clear

Instructions using this conditional suffix will be executed only if the Z flag is clear. This is the reverse case of the MI suffix. It indicates that an arithmetic operation produced a positive result, that is, one which is greater than *or* equal to zero. Logical operations which clear bit 31 of the destination register will give a positive result.

CS: Carry Set

Condition : C flag = Set

Instructions using this conditional suffix will be executed only if the carry (C) flag is set. This flag is set if an arithmetic operation produces a carry from bit 31 of the destination register. If this occurs, then it indicates that the result of the operation could not be represented in 32 bits. The carry can be thought of as the 33rd bit of the result, that is, bit number 32.

The carry flag can also be set or cleared by shifting data into it using one of the ARM's various shift operations. Full details of these will be given in Chapter Seven.

Example:

```
ADDS  R1,R1,#1024 Add 1024 to the contents of R1
ADDCS R2,R2,#1    If carry set, add one to register R2
```

CC: Carry Clear

Condition : C flag = Clear

Instructions using this conditional suffix will be executed only if the carry (C) flag is clear. This will be the case if a previous operation didn't produce a result which had a carry from bit 31. As we said previously, the carry is also affected by various ARM shift operations.

AL: Always

Condition : ALWAYS

There will be many cases when we do not want to use conditionally executed instructions. Instructions with this suffix, therefore, always execute, and do not depend on the settings of any flags. As the majority of instructions will have this suffix, it is taken to be the default by the assembler. If no suffix is specified with an instruction, then the assembler uses the AL suffix.

Examples:

```
ANDAL R0,R1,R2     ALWAYS perform R0 = R1 AND R2

ADD R1,R1,#2       ALWAYS add two to R1 (default assumed)
```

NV: Never

Condition : NEVER

This is not a very useful suffix, as it means that the instruction with which it is used is NEVER executed. It is included for completeness, as it is the inverse of the AL suffix.

Example:

```
MULNV R1,R2,R3     Never perform the multiplication
```

Conditional Execution After Comparisons

The next group of condition codes are based on the states of several flags. They are most often used after a CMP or CPN instruction to determine the result of the comparison. A program is presented in Chapter Eight (listing 8.3.) which illustrates the use of the comparison instruction. This will also be of use in understanding the operation of the various condition codes.

HI: Higher (Unsigned)

Condition : C flag = Set AND Z flag = Clear

Instructions using this conditional suffix will be executed if, as the result of a previous comparison instruction between two numbers, it was found that operand one was greater than operand two. It is important to note that the condition assumes that the two numbers compared were unsigned, tha is, all their 32 bits represent the number's magnitude and none are given o ver to representing their sign in two's complement for

Example:

```
CMP   R11,R6   Compare registers R11 and R6
MOVHI R11,#0   IF R11 > R6 then set R11 to zero
```

LS: Lower Than or the Same (Unsigned)

Condition : C flag = Clear or Z flag = Set

This is the reverse condition to the previous one. Instructions using this suffix will be executed if, as the result of a previous comparison instruction between two numbers, it was found that operand one was lower than or the same as operand two. Again, it is important to note that the condition assumes that the two numbers compared are unsigned.

Example:

```
CMP   R4,R2      Compare registers R4 and R2
ADDLS R4,R4,#1   IF R4 <= R2 then Add one to R4
```

GE: Greater Than or Equal (Signed)

Condition : N flag = Set AND V flag = Set
or: N flag = Clear AND V flag = Clear

Instructions using this conditional suffix will be executed if, as the result of a previous comparison instruction between two numbers, it was found that operand one was greater than, or equal to, operand two.

This time the condition is tested using the assumption that the two numbers compared are signed quantities. That is they are represented in two's complement form.

Example:

```
CMP   R5,R2     Compare registers R5 and R2
SUBGE R5,R5,#2 IF R5 >= R2 then subtract two from R5
```

LT: Less Than (Signed)

Condition : N flag = Set AND V flag = Clear
or: N flag = Clear AND V flag = Set

This is the reverse condition to the previous one. Instructions using this suffix will be executed if, as the result of a previous comparison instruction between two numbers, it was found that operand one was less than operand two. Again, the condition assumes that the two numbers compared are signed quantities represented in two's complement form.

Example:

```
CMP    R1,#0      Compare register R1 with zero
RSBLT  R1,R1,#0   IF R1<0 then R1=0-R1, ie, make positive
```

GT: Greater Than (Signed)

Condition : N flag = Set AND V flag = Set
or : N flag = Clear AND V flag = Clear
and : Z flag = Clear

Instructions using this conditional suffix will be executed if, as the result of a previous comparison instruction between two numbers, it is found that operand one is greater than operand two. Once more, the condition is tested using the assumption that the two numbers being compared are signed quantities.

Example:

```
CMP   R8,R9          Compare R8 with R9
SWIGT 256+ASC(">") IF R8 > R9 then print a > character
```

LE: Less Than or Equal To (Signed)

Condition : N flag = Set AND V flag = Clear
or: N flag = Clear AND V flag = Set
or: Z flag = set

Instructions using this conditional suffix will be executed if, as the result of a previous comparison instruction between two numbers, it is found that operand one is less than or equal to operand two. Once more, the condition is tested using the assumption that the two numbers being compared are signed quantities.

Example:

```
CMP   R13,#100      Compare register R13 with 100
SUBLE R13,R13,#10   IF R13 <= 100 subtract 10 from R13
```

Controlling the Status Flags

We said earlier that the status register flags reflect the result of previous ARM instructions. A useful feature of the ARM is that the programmer can define whether or not a given instruction is to be allowed to reflect the results of its execution in the status flags.

This allows the results obtained by executing one instruction to be preserved while several other instructions are executed. This is particularly useful when several instructions are to be conditional on the same setting of the status flags. By *not* allowing the instructions to modify the status flags when they execute, we ensure that the original state of the flags is preserved and can be tested by each instruction in the chain.

This feature is controlled from the assembler by using an S suffix to the instruction's opcode mnemonic. If the S suffix is present then the instruction is allowed to affect the status flags. If it is absent, then the flags will be unaffected by the execution of the instruction. (There are a few obvious exceptions to this rule and these will be described when the instructions are covered later.)

A very common mistake made when writing ARM assembly code, is to forget to add the S suffix to instructions. 6502 programmers, in particular, get used to almost every instruction automatically affecting the status flags. On the ARM this will not happen unless the S option is selected.

Example:

```
ADD  R0,R3,R5   Doesn't affect status flags when executed
ADDS R0,R3,R5   Does affect status flags when executed
```

Mixing Conditional and S Suffixes

We can use both the S option and a conditional suffix in the same instruction. In this case the two character condition suffix is written first, followed by the S character. For example:

```
ADDCCS R0,R1,R2
```

This add instruction will only execute if the CC (carry clear) condition is true. If it does execute, then the result of the operation will be reflected in the status flags – because the S option has been used.

Listing 5.1 provides a real example of the use of both conditionally executed instructions and the S suffix. When executed, it repeatedly prints a letter of the alphabet to the screen. The number of letters printed and the ASCII code of the character used are both prompted for before the program is assembled.

The program contains conditional instructions to check that the ASCII code entered is in the correct range, that is, 65-90. If this is not the case, then the program bleeps and a star (*) character is used. The number of characters to be output is also validated. If a negative number has been entered, then this is converted to a positive value before continuing.

Listing 5.1. Letter print.

```
 10 REM Printing Letters - A demo of conditional execution
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 REM Define names for the registers used in the program
 70 char     = 0
 80 quantity = 1
 90 count    = 2
100
110 REM Define constants for SWI routine and ASCII characters
120 vdu = 256
130 star = 42
140 beep = 7
150
160 DIM letters 256
170 P% = letters
180
190 [
200 CMP   char,#ASC("Z")    ; Compare character with "Z"
210 MOVGT char,#star        ; IF greater  THEN - char = "*"
220 SWIGT vdu+beep          ;   - issue 'beep'
230 CMP   char,#ASC("A")    ; Compare character with "A"
240 MOVLT char,#star        ; IF less THEN - char = "*"
250 SWILT vdu+beep          ;   - issue 'beep'
260 MOVS  count,quantity    ; Move no. of chars into 'count'
270 RSBMI count,count,#0    ; IF count<0  THEN count = 0-count
280
290 .print_loop
300 SWI  "OS_WriteC"        ; Print the character to the screen
310 SUBS count,count,#1     ; Decrement the value of 'count'
```

```
320 BNE  print_loop        ; If NOT zero then repeat loop
330 MOV  PC,R14            ; RETURN to BASIC
340 ]
350
360 REPEAT
370 PRINT
380 INPUT "Enter ASCII code of letter to be used :",A%
390 INPUT "Enter number of letters to be printed :",B%
400 CALL letters
410 UNTIL FALSE
```

Instruction Groups

The ARM processor actually supports 25 different instructions. Each instruction may be modified by using condition codes, S suffixes, shifted operands and so on, but there are still only 25 fundamental operations which can be carried out. These can be conveniently grouped as follows:

1) Data processing instructions
2) Transfers between processor and memory
3) Multiple transfers between processor and memory
4) Branches
5) Software interrupts

The following chapters describe the instructions in each of these groups.

6 : Data Processing – Format

This is by far the largest group of instructions. It contains instructions which manipulate or transform data in some way. There are 18 data processing instructions listed below in figure 6.1.

ADD	ADC
SUB	SBC
RSB	RSC
MOV	MVN
CMP	CPN
AND	ORR
EOR	
BIC	TST
TEQ	
MUL	MLA

Figure 6.1. Data processing instructions.

Apart from a few exceptions, all instructions in this group have the same assembler format. This can be summarised as:

```
<OPCODE Mnemonic> <Destination> <Operand 1> <Operand 2>
```

Opcode Mnemonic

The opcode mnemonic is the name of the instruction to be used. It is one of those given in figure 6.1. The various option suffixes can be added to this to modify the operation of the instruction.

Destination

The destination is simply the name of a register, that is, R0 to R15. This specifies the register into which the result of the instruction will be placed. The destination register may be the same as one of the registers containing the operands.

Operand One

Operands one and two specify the two pieces of data which are to be operated on by the instruction to produce the result.

Operand one must be the name of one of the registers R0 to R15. It is the data contained in this register which will eventually be used as operand one by the instruction.

Operand Two

Operand two can be specified in three different ways:

1) As a simple register
2) As an immediate constant
3) As a shifted register operand

Before looking at the specific instructions in the data processing group, we must examine these three ways of specifying the second operand. This will become a little involved as there are a large number of different options and formats. However, be patient – we will look at some 'real instructions' very soon!

Operand Two: A Simple Register

At its simplest level, operand two may also be the name of the register which contains the second operand for the instruction. Using this format, some typical instructions would be:

```
ADD R1,R2,R3      R1  = R2 + R3
AND R2,R10,R6     R2  = R10 AND R6
EOR R12,R12,R0    R12 = R12 EOR R0
```

Don't worry if the actual instructions are unfamiliar to you – they will be described later on. The important thing to note is, however, the format of the data processing instructions and their operands.

Operand Two: An Immediate Constant

The second form of operand two is to use it as an immediate constant. This means that the value of operand two is given directly in the assembler instruction. This is then encoded into the machine code equivalent of the instruction at assembly time. From this point on, the data used as operand two is fixed and doesn't depend on the contents of any registers.

Using immediate constants with data processing instructions from the assembler is very easy. Instead of writing a register name for operand two, we simply write '#n'. Where the '#' informs the assembler that an immediate operand follows, and 'n' is the value to be used as the immediate constant. Examples of data processing instructions using immediate constant operands are:

```
MOV R0,#100      Move 100 into register R0
ADD R5,R3,#1024 Add 1024 to R3 and store result in R5
AND R0,R4,#%101 R0 becomes R4 logically ANDed with %101
```

Range of Immediate Constants

There is a very important restriction imposed on the use of immediate operands. To understand this, we must look at how the immediate constant is encoded within an ARM instruction.

We have seen that the 32 bits comprising an instruction are split up into fields. One such field is used to store a binary representation of the immediate constant used with the instruction. Obviously, the number of bits allocated to this field will determine the range of numbers which can be represented in it. In practice, 12 bits are allocated for this purpose.

If all 12 bits of the field were used to simply store the binary representation of the immediate constant, then numbers in the range zero to 4096 could be used. Compared with what is possible using the 8-bit 6502 processor, this may seem very good. Remember, however, that the ARM is a 32-bit machine and, as such, we are used to manipulating 32-bit data.

The problem is that, without allocating extra bits, we cannot increase the number of values which can be represented in the immediate operand field. However, we can widen the range over which numbers can be represented, providing we accept that not every single individual number in the new range can be represented. This is the approach that the designers of the ARM decided to follow.

The 12-bit immediate operand field is split to create two fields of eight and four-bits, (see figure 6.2). The eight-bit data field is used to represent the numeric constant in binary. The four-bit field specifies one of 16 different positions in a 32-bit word at which the data in the eight-bit field should be placed. The scheme is summarised in figure 6.3.

```
<---   12-bit immediate operand field   --->
bit 11 .....bit 8          bit 7..... bit 0
 <Position field>          <Data field>
```

Figure 6.2. The split immediate operand field.

Bit 31 Bit 0	Position
.....76543210	0
10.....765432	1
3210.....7654	2
543210.....76	3
76543210.....	4
..76543210.....	5
....76543210.....	6
.....76543210.....	7
.....76543210.....	8
.....76543210.....	9
.....76543210.....	10
.....76543210.....	11
.....76543210.....	12
.....76543210.....	13
.....76543210.....	14
.....76543210..	15

Figure 6.3. The position system used in immediate operands.

An example should help clarify this somewhat confusing system! Suppose we wanted to represent the number 173. In pure binary this is:

%00000000000000000000000010101101

This can be represented as a data field of 173 (%10101101) and requires no position shift. It would be represented as:

Immediate operand: %0000 10101101

Data = 173 = %10101101
Position = 0 = %0000

However, suppose now that we wanted to represent the number 19968. In binary this number is:

%00000000000000000100111000000000

This corresponds to the data value 78 (%01001110) together with a shift number of 12, so that the data appears in the correct place in a 32-bit word. It would therefore be represented as:

Immediate operand: %1100 01001110

Data = 78 = %01001110
Position = 12 = %1100

Using this system we can represent values over the entire 32-bit range, although not every value in this range is allowed. As numbers get larger, we loose more and more 'low-order' bits from their representation. For example, all the numbers in the range zero to 255 can be represented. However, numbers in the range 256 to 1023 require a shift number of 15 to bring them into the correct position. This makes the two bottom bits of the 32-bit word unusable. In this range, therefore, only numbers which are divisible by four can be represented. Similarly, in the range 1024 to 4095, only numbers divisible by 16 can be stored and so on.

When we use immediate operands, we simply quote the number required. The assembler then tries to generate appropriate corresponding data and shift numbers. If this is not possible, with the number 257 for example, then an error is produced at assembly time.

Operand Two: A Shifted Register Operand

The third format of operand two in data processing instructions, is the shifted register operand. We have just seen that the ARM has the ability to apply bit shifts to data.

To specify a shifted operand, we use the normal syntax for a data processing instruction, but give operand two in the following form:

```
<Register>,<Shift>
```

The actual value of operand two is the contents of <Register>, after the shift operation specified in <Shift> has been applied to it. Note that the actual contents of the named register are not altered. It is just the value used by the instruction which is shifted. An example of an instruction using a shifted register operand is given below. Don't worry at this stage what the actual shift operation does!

```
ADD       R0           R1          R4           LSR#2
Opcode    Destination  Operand 1   <Register> <Shift>
Mnemonic                                Operand 2
```

<Shift> specifies the type of shift which is to be applied to the contents of the register. It also defines how many places the data shifts by. Each shift type has a mnemonic name (like the instruction opcode mnemonic) which is used to select it. A complete list of the shift types available is given in figure 6.4 on the next page.

Following the name of the shift operation is a field which defines the number of places to shift the data by. As the register to be shifted is 32-bits wide, shifting the contents by anything greater than 32 places is pointless.

A fixed number of shift positions can be specified by giving an immediate number in this field. This is done by writing a '#' followed by the number of places to shift by. For example, to shift by 23 places use '#23'.

Alternatively, a register name can be given. In this case the contents of the named register's least significant byte (at the time the instruction is executed) defines the number of places to shift by. That is, if the register contained 14, when the instruction was executed, then an appropriate shift of 14 places would be performed.

Shift mnemonic	Shift operation notes
LSL	Logical shift left
ASL	Arithmetic shift left (identical to LSL)
LSR	Logical shift right
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right with extent (one position only)

Figure 6.4. Shift operations supported by the ARM.

A couple of examples of typical shifted operand instructions should make the syntax clear:

Example 1:

```
ADD R0, R1, R3, LSL #3
```

This instruction performs the following:

1) Take the contents of register R3
2) Perform the LSL shift on this data, shifting it by three places
3) Add this modified value to the contents of register R1
4) Store the final result in register R0

Example 2:

```
ADD R0, R1, R3, LSL R10
```

This instruction performs the following:

1) Take the contents of register R3
2) Perform the LSL shift on this data. The number of places to shift by is defined by the contents of the low byte of register R10. For example, if R10 contained 27 then a shift of 27 places would take place

3) Add this modified value to the contents of register R1

4) Store the final result in register R0

The next chapter contains detailed descriptions of each of the available shift operations.

7 : Shift Instructions

Data Processing Instructions

We have seen how shifts can be used with instructions. We can now look at each of the different types of shift operation supported by the ARM. Listing 14.2 in Chapter 14 illustrates the use of conditional assembly. The program, however, also provides a pictorial demonstration of various types of shift operation. This program should be used to try out some of the theory presented in the following sections.

Logical Shift Left: LSL

Syntax:

```
LSL #n
LSL Rx
```

Where: n is an immediate number and,
Rx is a register (R0 to R15)

A logical shift left operation of 'n' places moves all the bits 'n' positions to the left. An extra zero bit is shifted into bit zero of the data on the right-hand side. Bit 31 of the data, lost from the left-hand end, is shifted into the carry flag. For example:

```
                LSL #1

Before:  X  <-  b31 b30 b29 b28 b27 ..... b4 b3 b2 b1 b0 <- 0

After:   b31    b30 b29 b28 b27 b26 ..... b3 b2 b1 b0  0

         Carry  Data word
```

Example:

Before:	X	10110011001100011100110101011101
After:	1	01100110011000111001101010111010
	Carry	Data word

The LSL operation has the effect of multiplying the data by two for each place it is shifted. That is, a shift left of five places would multiply the data by 2*2*2*2*2 = 32. The previous example showed a shift of one place, ie, a multiplication of two. In general, a shift of 'n' places left, has the effect of multiplying the data by two to the power of 'n'.

This assumes that no significant bits are lost from the left-hand side of the data. The new number must be small enough to fit into 32 bits.

The shift operation treats the data as a series of 32 arbitrary bits. If we interpret the bits as forming a binary number, then multiplication occurs. However, if we try to extend this to shifting data which represents two's compliment negative numbers, then the multiplication rule can break down and errors occur.

This happens because inappropriate bits may be shifted into the sign bit (bit 31) from bit 30. This can change the sign of the data. For example, the following shift changes the original negative number into a positive one:

Before:	X	10110011001100011100110101011101 (negative number)
After:	1	01100110011000111001101010111010 (positive number)
	Carry	Data word

Notes: The mnemonic ASL (arithmetic shift left) may be used in place of LSL. This is simply another name for the same shift operation and has exactly the same effect.

Logical Shift Right: LSR

Syntax:

```
LSR #n
LSR Rx
```

Where: n is an immediate number and,
Rx is a register (R0 to R15)

A logical shift right operation of 'n' places moves all the bits in the data 'n' positions to the right. An extra zero bit is shifted into bit 31 of the data on the left-hand side. Bit zero of the data, lost from the right-hand end, is shifted into the carry flag. For example:

```
                LSR #1
Before:  0 ->   b31 b30 b29 b28 b27 ..... b4 b3 b2 b1 b0 ->  X
After:          0   b31 b30 b29 b28 ..... b5 b4 b3 b2 b1    b0
                          Data word                         Carry
```

Example:

```
Before:   10110011001100011100110101011101          X
After:    01011001100110001110011010101110          1
                          Data word                 Carry
```

The LSR operation has the effect of dividing the data by two for each place it is shifted. The previous example showed a shift of one place right, ie, a division of two. Obviously only integer division is performed. The effect is the same as if the BASIC DIV operator was being used. In general, a shift of 'n' places right will divide a number by two to the power of 'n'.

Once again, the shift only produces the division operation for unsigned numbers. If, for example, the data is a negative number, stored in two's

compliment form, then when a zero is shifted into bit 31 on the left-hand side, the sign will be changed. For example:

Before:	X	10110011001100011100110101011101	(negative)
After:	1	01100110011000111001101010111010	(positive)
	Carry	Data word	

Arithmetic Shift Right: ASR

Syntax:

```
ASR #n
ASR Rx
```

Where: n is an immediate number and,
Rx is a register (R0 to R15)

An arithmetic shift right operation of 'n' places moves all the bits in the data 'n' positions to the right. The original contents of bit 31 are shifted back into the data on the left-hand side. Bit zero of the data, lost from the right-hand end, is shifted into the carry flag.

The shift is called an 'arithmetic' shift because it preserves the original arithmetic sign of the number. If the number is negative then bit 31 will be a one. In this case an extra one will be shifted into the word on the left-hand side – maintaining the negative representation.

Similarly, if the number is positive, bit 31 will be zero. In this case an extra zero will be shifted into the left-hand side of the word, again preserving the original sign.

```
ASR #1
```

Before:	b31 ->	b31 b30 b29 b28 b27 ..... b4 b3 b2 b1 b0 ->	X
After:		b31 b31 b30 b29 b28 ..... b5 b4 b3 b2 b1	b0
		Data word	Carry

Example:

Before:	10110011001100011100110101011101	X
After:	11011001100110001110011010101110	1
	Data word	Carry

The ASR operation, like LSR, divides the data by a factor of two for each position shifted. Once again integer division is performed.

This time, however, the shift takes into account the fact that the data may be representing a two's complement negative number. It extends the original sign of the number from bit 31 into bit 30. This ensures that the shift performs division correctly for both positive and negative numbers.

Rotate Right: ROR

Syntax:

```
ROR #n
ROR Rx
```

Where: n is an immediate number and,
Rx is a register (R0 to R15)

A rotate right operation of 'n' places moves all the bits in the data 'n' positions to the right. Unlike the shift operations, bits lost from one end of the data word reappear at the other end. Thus the bits are rotated, rather than shifted, in a cyclical manner.

The value of bit zero, lost from the right-hand end, is shifted back into bit 31 at the left-hand side. A copy of the original contents of bit zero are also shifted into the carry flag. For example:

```
ROR #1
```

Before:	b31 b30 b29 b28 b27 ..... b4 b3 b2 b1 b0	X
After:	b0 b31 b30 b29 b28 ..... b5 b4 b3 b2 b1	b0
	Data word	Carry

Example:

Before:	00110011001100011100110101011101	X
After:	10011001100110001110011010101110	1
	Data word	Carry

Rotational operations do not have any arithmetical significance. They are used simply to manipulate bit patterns.

Rotate Right With Extend (One Bit Only): RRX

Syntax:

RRX

This shift operation is unique in that it is not possible to specify the number of places for it to shift data by! The RRX operation always rotates the data right by one position.

The operation of RRX is similar to that of ROR except that the carry flag acts as a 'bit 32' in the rotation. The value of bit zero, lost from the right-hand end, is shifted into the carry flag. The value of the carry flag is shifted into bit 31 on the left-hand side. For example:

RRX

Before:	b31 b30 b29 b28 b27 ..... b4 b3 b2 b1 b0	X
After:	X b31 b30 b29 b28 ..... b5 b4 b3 b2 b1	b0
	Data word	Carry

Example:

Before:	00110011001100011100110101011101	X
After:	X0011001100110001110011010101110	1
	Data word	Carry

This shift operation effectivly allows 33-bit rotation to be performed by including the carry flag as an extra bit. Remember, however, that only single position rotations may be performed at once.

You will be relieved to know that we have now completed our general look at data processing instructions and associated operands. We can now move on to the next chapter to look at the operation of the 18 data processing instructions themselves.

8 : Processing Instructions

In this chapter we will look at the function and use of each of the ARM's data processing instructions. For each instruction the assembler syntax is given. Within this, the phrase {<suffix>} means that the conditional suffixes and/or the S suffix may be used if required. The names of any status flags affected are also listed.

ADD: Addition

Syntax:

```
ADD {<suffix>}   <destination>, <operand1>, <operand2>
```

Operation: destination = operand one + operand two

Flags: N, Z, C, V

The ADD instruction performs the arithmetic addition of its two operands, and stores the result in the destination register. The result is valid if unsigned numbers or signed, two's compliment, numbers are added. The result may always be interpreted in the same way as the operands.

Examples:

```
ADD     R0,R3,R4         R0 = R3 + R4
ADDS    R0,R3,#2         R0 = R3 + 2 (Setting status flags)
ADDMI   R0,R0,#1         If minus flag set Increment R0
ADD     R0,R0,R0,LSL#1   R0 = R0 + 2*R0   (R0 = 3*R0)
```

Listing 8.1 demonstrates the operation of the ADD instruction. From BASIC, two numbers are entered. A machine code routine is then called to add them together. The result is stored back, via USR ready for BASIC to print .

Listing 8.1. Simple two-word addition.

```
10  REM Simple 32-bit addition using ADD
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
```

```
 40  REM
 50
 60  DIM add 256
 70  P% = add
 80  [
 90  \ Two 32-bit numbers to be added are passed from A% and B%
100  \ into registers R0 and R1 when the routine is called
110  \ The result, stored in R0, is passed back to BASIC by  USR
120
130  ADD   R0,R0,R1
140  MOV   PC,R14
150  ]
160
170  REPEAT
180  INPUT "Number 1 : " A%
190  INPUT "Number 2 : " B%
200  PRINT "Result of Addition is : " ; USR(add)
210  UNTIL FALSE
```

ADC: Add with Carry

Syntax:

```
ADC {<suffix>}   <destination>, <operand1>, <operand2>
```

Operation: destination = operand one + operand two + carry

Flags: N, Z, C, V

The ADC instruction is almost the same as the previous ADD instruction. It performs the arithmetic addition of its two operands, but also adds in the carry flag. If the carry flag is set, then it is treated as a one in the addition. Otherwise, it is treated as a zero. The result is again stored in the destination register.

The ADC instruction allows numbers to be added together which require more than a single word to represent them. For example, we may want to perform 64-bit addition which involves two pairs of 32-bit words. This operation can be summarised as follows:

```
    Upper 32-bit word                    Lower 32-bit word

01101010101000111001110011001100   10110101000110001100011100011110
01010111000110100100011001100011   10101010101010011110011010011001

11000001101111011110001100110000(1)01011111110000101010110110110111
                                   ^^
                          Carry from bit 31 of low word
                          into bit zero of high word
```

To perform the addition, we add together the lower two words of each operand using the normal ADDS instruction. This may set the carry flag to indicate that a carry has been produced from bit 31. This will then need to be added in when the addition of the upper two words is performed. To do this we simply add the two high words using the ADCS instruction. This automatically takes care of any carry digit which may have been produced.

Example 1:

```
ADDS result_low,low1,low2       Add low words
ADCS result_high,high1,high2    Add high words + carry
```

This system can be extended to add together operands which require any number of words to represent them. We simply repeat the ADCS instruction as many times as required.

Example 2:

```
ADDS    R1,R4,R7        Add low words
ADCS    R2,R5,R8        Add middle words + carry
ADCS    R3,R6,R9        Add high words + carry
```

This will add together the two 96-bit numbers represented in registers R4, R5, R6 and R7, R8, R9. The 96-bit result is produced in registers R1, R2, R3. When programming the 6502, we frequently have to concatenate addition in this way as only eight-bit quantities can be processed at one time. On the ARM, however, 32-bit numbers can be processed directly and so the technique is used less often.

Note: It is vital that the S suffix is used with the instructions. If this is not done, then the carry flag setting will not be affected and so won't be carried forward into the next addition.

SUB: Subtract

Syntax:

```
SUB {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one – operand two

Flags: N, Z, C, V

The SUB instruction performs the arithmetic subtraction of its second operand from its first operand. The result of the operation is stored in the destination register. The result is valid if unsigned numbers or signed, two's compliment, numbers are added. The result may always be interpreted in the same way as the operands.

Examples:

```
SUB   R10,R2,R4          R10 = R2 - R4
SUBMI R1,R3,#1024        If neg flag set R1 = R3 - 1024
SUB   R0,R0,R0,LSL#1     R0 = R0 - 2*R0 (R0 = -R0)
```

Listing 8.2 demonstrates the operation of the SUB instruction using BASIC and machine code. The result is passed back for BASIC to print.

Listing 8.2. Simple two-word subtraction.

```
 10  REM Simple 32-bit subtraction using SUB
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM subtract 256
 70  P% = subtract
 80  [
 90  \ Two 32-bit numbers for subtraction passed from A% and B%
100  \ into registers R0 and R1 when the routine is called
110  \ The result, stored in R0, passed back to BASIC by USR
120  SUB   R0,R0,R1
130  MOV   PC,R14
140  ]
150  REPEAT
160  INPUT "Number 1 : " A%
170  INPUT "Number 2 : " B%
180  PRINT "Result of Subtraction is : " ; USR(subtract)
190  UNTIL FALSE
```

SBC: Subtract with Carry

Syntax:

```
SBC {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one – operand two – not (carry)

Flags: N, Z, C, V

The SBC operation allows multi-word subtraction to be performed in the same way that ADC allows multi-word addition. This time the carry flag is used to indicate that a 'borrow' occurred when subtracting two words, and that this borrow should be taken into account when subtracting the next two words.

The subtract operations, SUB and SBC, affect the carry flag in one of two ways as follows:

If a borrow is generated, then the carry is clear (0)
If a borrow isn't generated, then the carry is set (1)

When we perform multi-word subtraction, a borrow from one word means that we want to subtract an extra one from the next word. However, as we have just seen, a borrow results in the carry flag being zero, not one as we would have liked.

To compensate for this, the ARM actually inverts the carry flag before using it in the SBC operation. The SBC operation therefore, performs the following operation:

```
destination = operand one - operand two - not (carry)
```

This system can be extended to subtract operands which require any number of words to represent them. We simply repeat the SBCS instruction as many times as required.

Example:

```
SUBS result_low,low1,low2     Subtract low words
SBCS result_high,high1,high2 Subtract high words + carry
```

Again, it is vital that the S suffix is used if instructions are to be able to affect the status flags.

RSB: Reverse subtract

Syntax:

```
RSB {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand two – operand one

Flags: N, Z, C, V

This instruction is similar to the SUB instruction in that it also performs the subtraction of its operands. However, this time the subtraction is reversed, ie, operand one is subtracted from operand two.

This may seem a waste of an instruction. However, remember that operand two can be specified in several different formats, and it is thus much more flexible than operand one. By providing the RSB instruction, we ensure that either of the operands in the subtraction operation can be specified using the flexible format allowed by operand two.

Example:

```
RSB R0,R0,#0          R0 = 0 - R0   (R0 = -R0)
RSB R6,R3,R7,LSL#2    R6 = (R7*4) - R3
```

RSC: Reverse subtract with Carry

Syntax:

```
RSC {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand two – operand one – not(carry)

Flags: N, Z, C, V

The RSC instruction performs a reverse subtract operation while taking account of a previous borrow in the carry flag. It corresponds to the SBC instruction in the same way that RSB corresponds to SUB.

It allows reversed subtraction to be performed on multi-word operands.

Example:

```
RSBS result_low,low1,low2      Reverse subtract low words
RSCS result_high,high1,high2   Reverse subtract high words
                                and carry
```

MOV: Move data

Syntax:

```
MOV {<suffix>}  <destination>, <operand2>
```

Operation: Destination = operand two

Flags: N, Z, (C)

The MOV operation is different to normal data processing instructions in that it does not have an operand one. It is used to move data into the destination register.

The source of the data to be moved is given in operand two. Like any operand two, this can be specified as a register, an immediate operand or as a shifted register. Thus, immediate constants can be moved into registers or data can be moved between two registers.

The normal shift operations can be used to modify the data moved to the destination register.

When using shifts, it is frequently useful to specify both source and destination registers as being the same. This has the effect that the specified shift is applied to the contents of the register and the results written back to the same register. Thus, we can achieve the same results as dedicated shift instructions on other processors.

If a number is moved into R15, then the program counter and/or the status flags can be modified directly. A frequent use of this is to move the return address of a subroutine from the link register (R14) back into the program counter (R15). See Chapter Nine for a full description of using R15 in data processing instructions.

Examples:

```
MOV     R12,R0          Move the contents of R0 into R12
MOV     R6,R6,ASL#2     R6 = R6 * 4
MOV     R0,R2,ASL R5    R0 = R2 * (2^R5)
MOVEQS  R0,R4           If Z flag set THEN R0=R4
                        (setting flags)
MOV     R15,R14         Return from subroutine
```

MVN: Move Inverted Data

Syntax:

```
MVN {<suffix>}  <destination>, <operand2>
```

Operation: Destination = not (operand two)

Flags: N, Z, (C)

This instruction performs an identical function to MOV, except that the ARM automatically inverts all of the bits moved from the source register. This is done to allow negative immediate numbers to be moved into registers. An example will show why this could be a problem without the MVN instruction. Consider the two's compliment binary representation of minus one:

```
%11111111111111111111111111111111
```

Bearing in mind the scheme for representing immediate operands on the ARM, this number could not be used. Similarly, most negative numbers are not directly representable as immediate operands. However, by using MVN we can use an appropriate positive operand, in this case zero, and the ARM will invert it to obtain the desired value, ie, minus one.

Under the two's complement scheme, the number –n is represented as:

```
NOT (n) + 1
```

Thus to make the MVN use a value of –n we in fact specify n–1. So to move a value of –10 into a register, the immediate operand used with the MVN instruction is 10–1 = 9.

Examples:

```
MVN R0,#0          Move minus one into register R0
MVN R1,#9          Move -10 into register R1
MVN R3,R5           R3 = not(R5), ie, R5 with bits inverted
MVN R6,R7,LSR #1   R6 = not(R7 div 2)
```

CMP: Compare

Syntax:

```
CMP {<suffix>}  <operand1>, <operand2>
```

Operation: Reflect result of operand one – operand two

Flags: N, Z, C, V

This is a very important instruction connected with conditional instruction execution. It is an exception to the normal data processing instructions in that it does not have a destination register.

The instruction is used to compare two operands, and to reflect the result of the comparison in the status flags. This result can then be acted upon using the conditional execution system which is available with all instructions.

The CMP instruction performs the following 'notional' subtraction:

```
Operand one - operand two
```

The subtraction is notional because the result of the operation isn't retained anywhere. This explains why there is no destination field. The instruction merely conditions the status flags appropriately, and then discards the actual result.

As far as the programmer is concerned, the subtraction which CMP performs is not important. It is enough to know that the instruction is used before conditional statements to compare two operands. This makes statements execute conditionally on the result of the comparison.

The only thing to remember is that the various condition codes refer to operand one compared with operand two. Thus, the LT (less than) suffix will execute if operand one is less than operand two.

Since the purpose of the CMP instruction is to affect the status flags, the S suffix does not have to be used. The instruction will modify the status flags whether S is present or not.

You can investigate the operation of CMP, in conjunction with conditional statements, by typing in listing 8.3. When run, the 16 conditions supported

by the ARM are displayed. A pair of numbers are then prompted for. When these have been entered, a machine code routine is called. This compares the two numbers, then attempts to execute a series of 16 instructions which print a tick on the screen. Each of these instructions is executed on one of the 16 condition codes. The effect of this is that any condition which is satisfied has a tick printed next to it on the screen.

By varying the two numbers entered you can see how each of the conditional suffixes works after a CMP instruction. Try comparing minus one with one to show the difference between signed and unsigned condition codes.

Listing 8.3. A demonstration of comparisons and condition codes.

```
 10 REM Demonstration of CMP and conditional suffices
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 REM Define character 255 as a small 'tick' shape
 70 VDU 23,255,0,0,1,3,6,108,56,16
 80
 90 REM Set up constants
100 vdu  = 256
110 tick = 255
120
130 DIM compare 512
140 P% = compare
150 [
160 \ The two numbers to be compared are passed
170 \ into registers R0 and R1 from A% and B% when
180 \ the routine is called.
190
200 CMP    R0,R1   ; Compare the two numbers
210
220 \ There now follows one pair of instructions for each
230 \ condition code. These test the condition and if it
240 \ succeeds, performs VDU 255 ie, outputs a tick
250 \ A SWI command to start a new line is also called
260
270 SWIEQ  vdu+tick
280 SWI    "OS_NewLine"
290
300 SWINE  vdu+tick
310 SWI    "OS_NewLine"
320
330 SWIVS  vdu+tick
340 SWI    "OS_NewLine"
350
```

```
360  SWIVC  vdu+tick
370  SWI    "OS_NewLine"
380
390  SWIPL  vdu+tick
400  SWI    "OS_NewLine"
410
420  SWIMI  vdu+tick
430  SWI    "OS_NewLine"
440
450  SWICS  vdu+tick
460  SWI    "OS_NewLine"
470
480  SWICC  vdu+tick
490  SWI    "OS_NewLine"
500
510  SWIAL  vdu+tick
520  SWI    "OS_NewLine"
530
540  SWINV  vdu+tick
550  SWI    "OS_NewLine"
560
570  SWIHI  vdu+tick
580  SWI    "OS_NewLine"
590
600  SWILS  vdu+tick
610  SWI    "OS_NewLine"
620
630  SWIGE  vdu+tick
640  SWI    "OS_NewLine"
650
660  SWILT  vdu+tick
670  SWI    "OS_NewLine"
680
690  SWIGT  vdu+tick
700  SWI    "OS_NewLine"
710
720  SWILE  vdu+tick
730  SWI    "OS_NewLine"
740
750  MOV    PC,R14
760  ]
770
780  MODE 3
790  PRINT
800  REM Read in names of conditions and print them
810  FOR condition = 0 TO 15
820  READ name$
830  PRINT name$
840  NEXT
850
860  REM Keep getting two numbers and calling compare to show
870  REM the result of the comparison
880
```

```
 890 VDU 28,34,23,79,0
 900 REPEAT
 910 INPUT TAB(0,18) "Enter first number : " A%
 920 INPUT TAB(0,19) "Enter second number : " B%
 930 CLS
 940 PRINT "Comparing '";A%;"' with '";B%;"'"
 950 CALL compare
 960 UNTIL FALSE
 970
 980 REM Names of all 16 conditions
 990 DATA "Equal                          (EQ)"
1000 DATA "Not Equal                      (NE)"
1010 DATA "Overflow Set                   (VS)"
1020 DATA "Overflow Clear                 (VC)"
1030 DATA "Plus                           (PL)"
1040 DATA "Minus                          (MI)"
1050 DATA "Carry Set                      (CS)"
1060 DATA "Carry Clear                    (CC)"
1070 DATA "Always                         (AL)"
1080 DATA "Never                          (NV)"
1090 DATA "Higher           -Unsigned-    (HI)"
1100 DATA "Lower OR Same    -Unsigned-    (LS)"
1110 DATA "Greater OR Equal   -Signed-    (GE)"
1120 DATA "Less Than          -Signed-    (LT)"
1130 DATA "Greater Than       -Signed-    (GT)"
1140 DATA "Less OR Equal      -Signed-    (LE)"
```

CMN: Compare negative

Syntax:

```
CMN {<suffix>}  <operand1>, <operand2>
```

Operation: Reflect result of operand one – (– operand two)

Flags: N, Z, C, V

CMN performs an exactly equivalent operation to CMP, except that it compares operand one with the negative of operand two.

The idea behind this is the same as that of the MVN instruction. It allows comparisons to be made with small negative immediate constants which could not be represented otherwise.

An important point to be wary of is that in MVN the logical NOT of operand two is taken. In CMN it is the negative of operand two which is used. Thus, to compare register R0 with minus three we would write:

```
CMN R0,#3
```

The ARM will automatically form the negative of operand two and then make the comparison.

Since the purpose of the CMN instruction is to affect the status flags, the S suffix does not have to be used. The instruction will modify the status flags whether S is present or not.

Examples:

```
CMN  R5,R7            Compare R5 with -R7
CMN  R6,#1            Compare R6 with minus one
CMN  R3,R0,LSL#1      Compare R3 with -R0*2
```

AND: Logical AND

Syntax:

```
AND {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one AND operand two

Flags: N, Z, (C)

This instruction performs a logical bitwise AND operation between its two operands. The result of the operation is placed in the destination register. The S suffix can be used with the instruction in the normal way so that the results of the AND are allowed to affect the status flags.

Example:

```
AND   R0,R1,R2          R0 = R1 AND R2
AND   R5,R5,#%1111      R5 = R5 AND %1111
                        (clear all but low four bits)
ANDS     R4,R4,#1       R4 = R4 AND 1
                        (setting flags on result)
```

The AND operation and its uses are covered in Appendix C.

ORR: Logical OR

Syntax:

```
ORR {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one OR operand two

Flags: N, Z, (C)

This instruction performs a logical bitwise OR operation between its two operands. The result of the operation is placed in the destination register.

Note: The OR operation is particularly useful for forcing certain bits to be set in a data word.

```
ORR  R0,R11,R2      R0=R11 OR R2
ORR  R7,R7,#%1100   R7=R7 OR %1100 (set bits 2 & 3)
ORRS R5,R5,#2       R5=R5 OR #2 (setting flags on result)
```

The OR operation and its uses are covered in Appendix C.

Listing 8.4 illustrates a use of the ORR instruction. It reads a character from the keyboard, forces bit five in its ASCII code to be set, and prints the modified character to the screen. This has the effect of forcing all characters entered to be displayed in lower case on the screen.

Listing 8.4. Case conversion using the ORR instruction.

```
 10 REM Using the ORR instruction to perform case conversion
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM convert 256
 70 P%=convert
 80 [
 90 SWI "OS_ReadC"      ; SWI routine to read character into R0
100 ORR R0,R0,#%100000 ; Set bit 5 of the characters ASCII code
110 SWI "OS_WriteC"     ; Use SWI to output modified char from R0
120 B convert           ; Branch back to beginning of the routine
130 ]
140 PRINT
150 PRINT "Entered characters will be converted into lower case"
160 CALL convert        : REM Call the routine
```

EOR: Logical Exclusive OR

Syntax:

```
EOR {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one EOR operand two

Flags: N, Z, (C)

This instruction performs a logical bitwise EOR operation between its two operands. The result of the operation is placed in the destination register.

Examples:

```
EOR  R7,R5,R2     R7 = R5 EOR R2
EOR  R7,R7,#1     R7 = R7 EOR 1    (invert bit zero in R7)
EORS R3,R8,#12    R3 = R8 EOR #12  (set flags on result)
```

The EOR operation and its uses are covered in Appendix C. EOR is very useful in 'toggling' data between two pre-defined values. Listing 8.5 shows this in practice. It toggles the register R0 between 65 and 90 by EORing its contents with 27. The character whose ASCII code is in R0 is printed each time, printing alternate As and Zs on the screen.

Listing 8.5. Toggling data using the EOR instruction.

```
 10  REM Use EOR instruction to toggle between two characters
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM toggle 256
 70  P%=toggle
 80  [
 90  MOV R0,#ASC("A")
100  .loop              ; Mark beginning of loop with a label
110  EOR R0,R0,#27      ; EOR the ASCII code in R0 with 27
120  SWI "OS_WriteC"    ; Output char whose ASCII code is in R0
130  B loop             ; Branch back to beginning of loop
140  ]
150  CALL toggle        : REM Call the routine
```

BIC: Bit Clear

Syntax:

```
BIC {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one AND (NOT (operand two))

Flags: N, Z, (C)

The BIC instruction provides a useful way of clearing (forcing to zero) certain bits within a data word, while leaving the others unchanged. Operand one in the instruction is the data word to be modified.

Operand two is a 32-bit word called the bit mask. A set bit (one) in the bit mask will force the corresponding bit in the data word to be reset when the instruction is executed. A zero bit in the bit mask will leave the corresponding bit in the data word in its original state. The modified data word is placed in the destination register.

Example of BIC operation:

```
Original:   %10101000111001010011001110111011
Bit mask:   %10000000000000000000000000000111
Result:     %00101000111001010011001110111000
```

Examples:

```
BIC  R0,R0,#%1111   Clear low four bits of R0
BIC  R1,R1,R2       Clear bits in R1 which were set in R2
BIC  R6,R6,R6       Clear bits which were set in R6(R6=0)
```

TST: Test Bits

Syntax:

```
TST {<suffix>}  <operand1>, <operand2>
```

Operation: Reflect result of operand one AND operand two

Flags: N, Z, (C)

The TST instruction, like CMP, has no destination field to it. It performs the logical bitwise ANDing of operands one and two, but does not store the result anywhere. The status flags are set, however, to show the result of the operation and this can then be acted upon.

TST can be used to see if a particular bit in a data word is set or clear. The data word forms one operand. A bit mask, in which the appropriate bit is set, forms the other operand. After the TST operation the Z flag will be set if the bit is set in the data word, but clear if it is not.

As the purpose of the TST instruction is to always affect the status flags, the S suffix does not have to be used. The instruction will modify the status flags whether S is present or not.

Examples:

```
TST R1,#%1000    Test to see if bit three is set in R1
TST R3,R4        Test if any bits set in both R3 and R4
```

An obvious application for TST is to print a number in binary. This is implemented in listing 8.6. The program tests each bit in register R1 in turn starting at bit 31. If the bit is set then a one is printed, otherwise a zero is printed instead.

Listing 8.6. Printing binary.

```
10  REM Printing Binary using the bit test (TST) instruction
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  REM declare registers names for those used
70  number  = 0
80  mask    = 1
90
```

```
 100  DIM Binary 256
 110  P%=Binary
 120  [
 130  ; The number to be printed in binary is passed
 140  ; from A% into R0 when the routine is called
 150
 160  MOV mask,#1 << 31  ; Move %10000000000000000000000000000000
into mask
 170  .bits              ;Start of loop to print binary digits
 180  TST number,mask    ; See if current bit is set in the number
 190  SWIEQ 256+ASC"0"   ; IF not set then VDU 48 ie. print a '0'
 200  SWINE 256+ASC"1"   ; IF set then VDU 49 ie. print a '1'
 210  MOVS mask,mask,LSR#1 ; Move 'current bit' right 1 place in
mask
 220  BNE bits           ; If all bits not looked at branch back
 230  SWI "OS_NewLine"   ; Output new line using SWI call
 240  MOV PC,R14         ; Return back to BASIC
 250  ]
 260
 270  REPEAT
 280  INPUT A%
 290  CALL Binary
 300  UNTIL FALSE
```

TEQ: Test Equivalence

Syntax:

```
TEQ {<suffix>} <operand1>, <operand2>
```

Operation: Reflect result of operand one EOR operand two

Flags: N, Z, (C)

TEQ is very similar to TST. The only difference is that it performs a notional EOR operation between its operands, instead of an AND. The TEQ instruction can be used to see if the bits in two data words are the same or not. This would normally be done using CMP. However, with TEQ the carry flag is unaffected. This can be useful if the equality of two operands has to be tested while preserving the setting of the carry flag.

Since the purpose of the TEQ instruction is to affect the status flags, the S suffix does not have to be used. The instruction will modify the status flags whether S is present or not.

Examples:

```
TEQ R1,#5       Test to see if R1 contains five
TEQ R3,R4       Test to see if R3 and R4 are the same
```

MUL: Multiplication

Syntax:

```
MUL {<suffix>} <destination>, <operand1>, <operand2>
```

Operation: destination = operand one * operand two

Flags:

N, Z	reflect result
V	is not changed by the instruction
C	is undefined after this operation

This instruction performs 32-bit multiplication. Operand one and operand two are multiplied together and the result stored in the destination register. If the two operands are interpreted as being signed two's compliment numbers, then the result may also be treated as being signedo.

MUL is different to the previous data processing instructions in that certain restrictions exist about how its operands may be specified. The destination, operand one and operand two must all be given as simple registers. No immediate or shifted operands may be given as operand two. Also, there is the restriction that the destination and operand one must be different registers. Finally, register R15 may not be used as the destination register.

Example:

```
MUL  R0,R1,R3          R0 = R1 * R3
```

Listing 8.7 shows the MUL instruction working. Two numbers are entered and passed to a machine code routine which multiplies them. The result is then passed back for BASIC to print.

Listing 8.7. Multiplying two numbers together.

```
 10  REM Multiplying two 32-bit numbers using MUL
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50  DIM multiply 256
 60  P% = multiply
 70  [
 80  ; The two number to be multiplied are passed into registers
 90  ; R0 and R1, from A% and B% when the routine is called.
100  ; The result is passed back to BASIC from register R0
110  ; by the USR statement
```

```
120  MUL   R2,R0,R1 ; Multiply the numbers in R0 and R1 together
130  MOV   R0,R2    ; Move result from R2 into R0 return with USR
140  MOV   R15,R14  ; Return to BASIC
150  ]
160  REPEAT
170  PRINT
180  INPUT "Number 1 : " A%
190  INPUT "Number 2 : " B%
200  PRINT "Result of multiplication is : "; USR(multiply)
210  UNTIL FALSE
```

MLA: Multiplication with Accumulate

Syntax:

```
MLA {<suffix>}<destination>,<operand 1>,<operand 2>,<sum>
```

Operation: destination = (operand one * operand two) + sum

Flags:

N, Z	reflect result
V	is not changed by the instruction
C	is undefined after this operation

This instruction performs a similar operation to the MUL instruction. The difference is that the contents of the register given in the sum field are added into the result of the multiplication before storing it in the destination register. Like MUL, all data fields of the instruction can only bes simple registers, and must observe the same restrictions.

The MLA instruction is used to keeping a running total of a series of multiplications. If the sum register is specified as being the same as the destination, then the result of each multiplication will be accumulated in the destination register.

Example:

```
MLA R0,R1,R2,R3          R0 =  (R1 * R2) + R3
MLA R0,R1,R2,R0          R0 =  (R1 * R2) + R0
```

9 : Register R15

Register R15 with Data Processing Instructions

In the previous description of data processing instructions, we have generally indicated that if a register can be used with an instruction, then it may be any one the processor registers R0 to R15. This is perfectly true. However, if register R15 is used, then we would expect some special results to occur since this is also the program counter and status flag register. The effects of using R15 in instructions depends on whether it is being used as operand one, operand two or the destination register.

Register R15 as Operand One

When register R15 is used as source operand one, only the program counter part of it is accessible. Thus, the data used by the instruction as operand one, are bits two to 25 of R15. All of the other bits are assumed to be zero. This is done so that the value of the program counter can be used in operations without the settings of the status flags having any effect. For example, if we wanted to add 1024 to the program counter, and store the result in register R0, we could write:

```
ADD     R0,R15,#1024
```

Register R15 as Operand Two

If R15 is used as operand two in an instruction, then all 32 bits are accessible. The value used in the instruction will therefore be made up from the program counter in bits two to 25, the flags in bits zero to one and bits 26 to 31. This is useful if we want to access the state of any of the ARMs processor flags.

The program fragment in figure 9.1, for example, accesses the processor mode flags in the lower two bits of R15. The values of all the other bits are

masked out. The value in R3 can then be used to determine which mode the processor is executing in.

```
MOV   R3,#%11       Put bit mask into register R3
AND   R3,R3,R15     AND R15 with bit mask to get bits
                    zero and one
CMP   R3,#%00       Is it user mode?
BEQ   user_mode
CMP   R3,#%01       Is it FIRQ mode?
BEQ   FIRQ_mode
CMP   R3,#%10       Is it IRQ mode?
BEQ   IRQ_mode
CMP   R3,#%11       Is it supervisor mode?
BEQ   SVC_mode
```

Figure 9.1. Testing the mode flags.

The Program Counter and Pipelining

Previously, we have said that the value of the program counter can be accessed by specifying R15 as a source operand in an instruction. We would expect that the value of the program counter used would be the address of the instruction, as this is the one currently being executed. However, typing in and running listing 9.1, will show that this is not the case.

Listing 9.1. The effect of pipelining on the program counter.

```
  10  REM A demonstration of the effects of pipelining
  20  REM (c) Michael Ginns 1988
  30  REM Dabs Press : Archimedes Assembly Language
  40  REM
  50
  60  DIM test 256
  70  P%=test
  80  [
  90  ; The value of the program counter when the MOV instruction
 100  ; is executed is passed back to BASIC using USR
 110
 120  .inst_address  ; Label the address of the instruction
 130  MOV  R0,R15    ; Move the current value of PC into R0
 140  MOV  R15,R14   ; Return back to BASIC
 150  ]
 160
 170  PRINT
 180  PRINT "Addr of the 'MOV R0,R15' instruction="; ~inst_address
 190  PRINT "Addr of PC when instruction executed = "; ~USR(test)
AND &3FFFFFF
```

The program simply stores the contents of the program counter in register R0 for BASIC to print out. This allows the address of the MOV instruction to be compared with the contents of the program counter when the instruction is executed. Note that the value of the PC is eight bytes greater than the address of the MOV instruction.

The reason for this is that the ARM uses pipelining when processing instructions. Pipelining was fully explained in Chapter Two. It means that at the time an instruction is executed by the ARM, a second one is being decoded and a third is being fetched. When an instruction is executed, the program counter is already pointing two instructions further on. The address it contains is, therefore, two words (eight bytes) more than the address of the executing instruction.

The effect of pipelining must be taken into account, otherwise some peculiar things can happen! An example of this is illustrated in the listing 9.2. At first sight, it seems that the MOV instruction will have no effect, and all the program does is produce a 'beep'. However, it doesn't even do that!

MOV causes the next instruction to be skipped. This is because the address accessed from the PC is eight bytes more than the address of the MOV instruction. When written back into the PC, therefore, execution resumes eight bytes further on, thereby skipping the next instruction.

Listing 9.2. Skipping instructions.

```
 10 REM Skipping instructions due to pipelining
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50 REM Declare constants
 60 vdu  = 256
 70 beep = 7
 80 DIM test 256
 90 P% = test
100 [
110 MOV R15,R15    ; Move contents of R15 into R15
120 SWI vdu + beep ; Make a 'Beep'  (VDU 7)
130 MOV R15,R14    ; Return to BASIC
140 ]
150 REM Calling the routine should make a 'beep'
160 REM But it won't bacause pipelining has caused
170 REM the instruction to be skipped
180 CALL test
```

Always remember that when the PC is accessed, the address it contains is always eight bytes more than the address of the instruction currently being executed by the processor.

Register 15 as the Destination Register

When R15 is named as being the destination register in an instruction, only the program counter normally is affected by the new data. Bits 26 to 31 of the data written into R15 are not allowed to modify the status bits.

If we want to change the settings of the status flags, we must add the usual S suffix on to the instruction. We are then free to set or clear any flag we want. Bits 26 to 31 of the data being written into R15, define the new states of the flags. Obviously, we can only modify flags which are accessible from the current processor mode. We could not, for example, change the interrupt flags from user mode.

If we need to change the settings of the status flags without altering the program counter, things are more complex. We could try to use an instruction like:

```
EORS R15,R15,#1<<31
```

This should invert the status register's negative flag, held in bit 31 of R15, without changing anything else. However, as we have just seen, pipelining will cause the following two instructions to be skipped. The value of the program counter, read from R15, will be eight bytes (two words) greater than the address of the instruction. When it is written back into R15, therefore, causing the ARM to execute the instruction two words further on.

To allow for this, the assembler provides us with the P suffix. For our purposes, we use this suffix with the TEQ instruction. You will remember that this instruction performs a notional Exclusive OR with its operands. The operation is notional because the value produced is not stored anywhere. Instead, the result of the operation is reflected in the status flags.

When the P suffix is used, however, bits 26 to 31 of the EOR result are written directly to bits 26 to 31 of R15. The status flags are therefore changed while leaving the program counter unaffected. We can now write statements of the form:

```
TEQP R15,mask
```

Since R15 is given as operand one, bits 26 to 31 of it (the status flags) are seen as zeros. However, anything EORed with zero is left unchanged. Thus, when the notional EOR operation is performed by TEQ, bits 26 to 31 of the result will be a direct copy of the corresponding bits in operand two, the mask. Finally, because we have used the P suffix, bits 26 to 31 of this result will be written to bits 26 to 31 of R15, the status flags. The effect of all this forces the status flag to take on the settings of bits 26 to 31 in the mask, while leaving the program counter unchanged.

By choosing appropriate masks, we can set or clear any accessible status flag. An example should make this clear. We want to set the negative flag. The first thing we do, is place a copy of R15 in another register and set bit 31 (the negative flag) in it:

```
ORR R0,R15,#1<<31
```

Next, we write the modified copy of bits 26 to 31 back into R15 using the TEQP instruction:

```
TEQP R15,R0
```

This will set bit 31 of R15 (the negative flag) while leaving the other flags and the program counter unchanged.

10 : Data Transfer

Between Memory and Registers

All of the data processing instructions discussed previously, accessed their operands from the processor's internal registers. Obviously, we must also have access to some method of transferring data between the registers and main memory.

The two instructions load register (LDR) and store register (STR) are provided by the ARM for this purpose. LDR transfers data from memory into one of the processor registers. STR performs the reverse operation, transferring data from a processor register to memory.

Accessing Memory

Instructions which transfer data between processor and memory must have two things specified within them. First, we must specify the register which is to be used as the source or destination of the data. This can be done simply by quoting the register's name. This is equivalent to the way that we gave the destination register in data processing instructions.

The second thing which we need to do, is to give the address of the memory location which is to be used in the transfer. This could be done in a number of ways. The method by which the ARM obtains the address is called the addressing mode.

Addressing Modes

This simplest scheme for specifying the address would be to give the location as an absolute address number. To be able to specify the full range of ARM addresses, we would need a 26-bit field in the instruction. After allocating bits for the instruction opcode, the condition flags, the register number and so on, this size of field is simply not available.

Indirect Addressing

An alternative scheme is to specify the source location address indirectly. In the instruction we give the name of a processor register called the addressing register. When the instruction is executed, the processor will look at the contents of the addressing register. The number contained in this is then taken as the address of the location in memory to be accessed.

For example, suppose we have an LDR instruction and quote register R3 as being the address register. If, when the instruction is executed, R3 contained the number 1000, then the data would be loaded from location 1000 of memory. This scheme is summarised in figure 10.1.

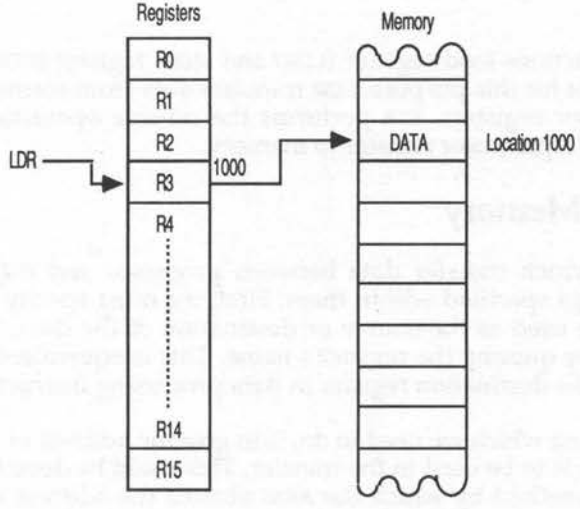


Figure 10.1. Summary of the indirect addressing scheme.

Indirectly addressing memory also has another advantage. The address of the location accessed is not fixed in the instruction. It is defined by the contents of a register and can be changed dynamically as the program executes. This provides a very flexible memory access system, which can be used to support high-level data structures such as arrays, tables, lists and so on.

The ARM supports two forms of enhanced indirect addressing called pre-indexed and post-indexed addressing. We will now look at these using the LDR instruction as the example. All comments about the two addressing modes equally apply to the STR instruction. The way in which addressing mode calculations are made is the same for both LDR and STR, the only difference is the 'direction' in which data is transferred. That is, from memory to registers, or from registers to the memory. Remember that conditional suffixes can be used with both LDR and STR, although these have been left out for clarity in the following descriptions.

Pre-indexed Addressing

An LDR instruction using pre-indexed addressing has the following syntax in assembler:

```
LDR <destination>, [<base>{,<offset>}]
```

The destination field is the register into which the data is to be transferred.

The contents of the base and offset fields together specify the memory word to be accessed by the instruction.

If the optional offset field is not present, then the contents of the base register alone are taken to be the memory address. If the offset is given, however, then the contents of it are added to the contents of the base field. The resulting number is then taken to be the required address.

Base is always given as a simple register. It is intended to contain the start or base address of the section of memory which is going to be accessed. Offset is more flexible and is intended to contain an offset from the address stored in base to the address of the required location.

Offset is specified in a similar way to that used in operand two of the data processing instructions. For example:

A simple register
An immediate constant
A shifted register

Simple Register

In this form, the address of the memory location accessed by the instruction is made by adding the contents of the base and offset registers. An example is as follows:

```
LDR R0,[R1,R2]    Load R0 from the address R1+R2
```

This would add the contents of registers R1 and R2. The result would be taken by the ARM to be an address in memory. The data word at this location would then be loaded into register R0 as illustrated in figure 10.2.

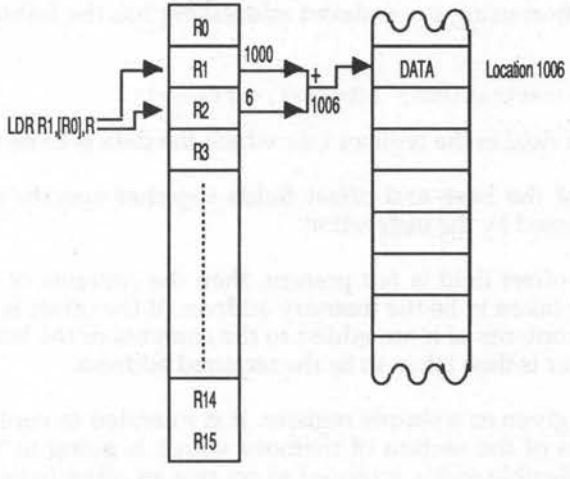


Figure 10.2. Pre-indexed addressing using a register offset.

Note that a minus sign (–) may be included before the offset register name. This instructs the ARM to treat the offset in the register as being negative, ie, it will subtract it from the base address.

Listing 10.1 shows this pre-indexed addressing in use. The program repeatedly stores pseudo-random data words into screen memory. The top of screen memory is always at address &2000000, so this becomes the contents of our base register. The offset register starts at one and is incremented in a loop up to a value of &14000 (80k). The minus sign in front of the offset register specifies that the offset is to be subtracted from the base. We

thus repeatedly store the data words in the 80k of screen memory beneath the base address. Incidentally, the program also shows how fast the ARM is. Remember that over 80000 bytes of memory are being written to several times a second.

Listing 10.1. Demonstration of pre–indexed indirect addressing.

```
 10 REM Storing random words in the screen memory using
 20 REM the ARM's STR instruction with pre-indexed addressing
 30 REM (c) Michael Ginns 1988
 40 REM Dabs Press : Archimedes Assembly Language
 50 REM
 60
 70 MODE 15
 80 REM Give names to the registers used
 90 base   = 0
100 offset = 1
110 data   = 2
120
130 DIM screen 512
140 P%=screen
150 [
160 ; The end of screen memory is placed into register R1
170 ; Random data words are then stored in the 80k of memory
180 ; below this address
190
200 .store_screen      ; Label loop - keep filling scrn memory
210 MOV base,#&2000000 ; Move end screen memory addr into base
220 MOV offset,#1      ; Set offset of store instruction to 1
230 .store_words       ; Loop - store data in each word of 80k
240 ADD data,data,data,ROR#1   ; Get new pseudo-random  word
250 STR data,[base,-offset]    ; Store word at 'base+(-offset)'
260 ADD offset,offset,#4       ; Inc offset by 1 word (4 bytes)
270 CMP offset,#&14000 ; See if 80k of memory has been filled
280 BCC store_words    ; If not then branch back
290 B store_screen     ; Do all again by branching to the start
300 ]
310
320 C%=1
330 CALL screen
```

An Immediate Constant

This format allows the offset to the address held in base to be given as an immediate constant. The constant, unlike those used in the data processing instructions, must be in the range –4069 to 4069. An example is:

```
LDR R0,[R1,#-4]  Load R0 from the address R1-4
```

This would load data into register R0 from an address which is four bytes (one word) lower than that contained in register R1.

Shifted Register

The offset can also be given as the contents of a register to which a shift operation has been applied. The shift operations are the same as those used in the data processing instructions. An added restriction, however, is that the number of places to shift by must be specified as an immediate constant. With data processing instructions we were allowed to specify this as the contents of yet another register, however, this is no longer possible.

This form of the instruction is of particular use when accessing data from an array, or table, using an index. Suppose that each entry of the table or array occupied four bytes of memory. To access the nth entry, we could use the following instruction:

```
LDR R0,[base,index,LSL#2]
```

Where base and index are two registers containing the base table address and the index of the required entry within it. The instruction will take the value of index, multiply it by four (using the shift operation), add it to the contents of base and then use the result as the address from which a data word is to be loaded in to R0.

Listing 10.2 illustrates this application. A table of cosine and sine values are created in BASIC. A machine code routine then accesses entries in these tables to draw a circle on the screen.

Listing 10.2. Accessing tables using indirect addressing.

```
 10  REM Drawing circles using indexed addressing to access
 20  REM a table of SIN and COS values
 30  REM (c) Michael Ginns 1988
 40  REM Dabs Press : Archimedes Assembly Language
 50  REM
 60
 70  REM Create COS and SIN tables
 80  REM COS and SIN values for angles 0-360 are calculated
 90  REM the value stored is multiplied by 400 and has 600
100  REM added to it. This ensures correct ranges for the screen.
110  REM Note each value in the table takes two bytes
120
130  DIM cosine 720
140  DIM sine   720
```

```
150 FOR angle = 1 TO 360
160 cosine!(angle*2)=(COS(RAD(angle))*400)+600
170 sine!(angle*2)=(SIN(RAD(angle))*400)+600
180 NEXT
190
200 DIM circle 512
210
220 REM Set up names for all the registers used
230 index = 2      : REM Register to index the TRIG tables
240 cos_base = 3   : REM base address of the cosine table
250 sin_base = 4   : REM base address of the sine table
260
270 REM Define constants
280 plot = 25      : REM Plot is performed by VDU 25
290 dot =69        : REM dots are drawn by PLOT command 69
300 vdu = 256      : REM Start of SWI block to perform VDU n
310
320 P%=circle
330 [
340 ADR cos_base,cosine; Get start addr of COS table in base reg
350 ADR sin_base,sine  ; Get start addr of SIN table in base reg
360 MOV index,#360     ; Index pointer=360 and decrements
370 .draw              ; loop to draw points in circle
380 SWI vdu+plot       ; VDU 25 ie, PLOT
390 SWI vdu+dot        ; VDU 69 ie, code to PLOT a dot
400 LDR R0,[sin_base,index,LSL#1]  ; Access SIN_table(index)
410 SWI "OS_WriteC"    ; Send high and low bytes to VDU driver
420 MOV R0,R0,LSR#8    ; to specify the x co-ord for the plot
430 SWI "OS_WriteC"
440 LDR R0,[cos_base,index,LSL#1]  ; Access COS_table(index)
450 SWI "OS_WriteC"    ; Send high and low bytes to VDU driver
460 MOV R0,R0,LSR#8    ; to specify the y co-ord for the plot
470 SWI "OS_WriteC"
480 SUBS index,index,#1            ; Decrement the index
490 BNE draw           ; If index not at '0' then repeat loop
500 MOV PC,R14         ; Return back to BASIC
510 ]
520
530 MODE 0
540 PRINT" PLOTTING CIRCLE !"
550 CALL circle
```

Using Write Back

In calculating which word of memory is to be accessed, the ARM adds together the contents of the base and offset registers. It is sometimes useful to retain this newly calculated address for future use. In pre-indexed addressing this is done by using the ARM's 'write back' facility.

Write back is an extension to the data transfer instruction. We specify that we want write back to occur by including a '!' suffix on the instruction. An example is illustrated below:

```
LDR <destination>, [<base>{,<offset>}] !
```

Examples:

```
LDR R0,[R1,R2]!        Load R0 from addr R1+R2: R1=R1+R2
LDR R3,[R5,#10]!       Load R3 from addr R5+10: R5=R5+10
LDR R7,[R3,R8,LSL#2]! Load R7 from addr R3+R8*4:R3=R3+R8*4
```

When the ARM executes the instruction, it will perform the usual addition of the base and offset fields. It will then access the data at the resulting address. Finally, as write back is selected, it will store the newly-calculated address back into the base register. Write back is available with both the LDR and STR instructions.

Write back is particularly useful when accessing a sequence of memory locations. For example, to access consecutive memory words, we can use the following:

```
LDR R0,[base,#4]!
```

When executed for the first time, this will access location base+4. This calculated address will then be written back automatically into the base register. The next time the instruction is executed, therefore, the location accessed will be at base+4+4 = base +8. Again, the base register will be updated from this address. In this way addresses base+4, base+8, base+12 and so on can be accessed by simply looping back to the instruction.

This could be useful, for example, when summing the contents of an array. A program to do this is presented when we consider implementing arrays in machine code.

Post-indexed Addressing

Post-indexed addressing is the other way in which the ARM can access memory. In assembler it has the following form:

```
LDR <destination>, [<base>],<offset>
```

The three fields can be given in exactly the same forms as used with pre-indexed addressing. Note, however, that the offset field isn't optional and must be included.

Examples of LDR instructions using post-indexed addressing are:

```
LDR R1,[R0],R7          Load R1 from addr R0: R0=R0+R7
LDR R6,[R7],#4          Load R6 from addr R7: R7=R7+4
LDR R8,[R2],R5,LSL#4    Load R8 from addr R2: R2=R2+R5*16
LDR R0,[R0],#20         Load R0 from addr R0: R0=R0+20
```

When post-indexed addressing is used, the contents of the base register alone are taken as the address of the memory word to be accessed. Only after this word has been accessed, are the contents of the offset field added to the base register and the result stored back in the base register. Obviously, this implies that write back always occurs, so we do not need to specify it.

In the first of the examples, the contents of register R0 would be taken as being the address to be accessed. The word of memory at this address would then be transferred into register R1. Finally, the contents of R7 will be added to R0 and the result written back to R0. This example is illustrated in figure 10.3.

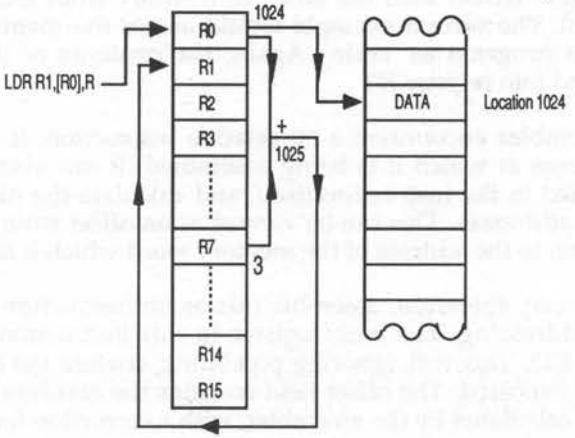


Figure 10.3. Post-indexed addressing using a register offset.

PC Relative Addressing

We have said that the ARM processor supports two distinct forms of addressing: post-indexed and pre-indexed. However, the BASIC assembler on the Archimedes also allows another form, PC relative addressing.

This is really a pseudo-addressing mode as it is not a distinct addressing mode supported by the ARM. Instead, instructions using PC relative addressing are accepted by the assembler, but are converted into an appropriate pre-indexed instruction.

The general form of instructions using PC relative addressing is as follows:

```
LDR <destination>,address
```

The destination is the same as before, ie, a register into which the data is to be transferred from memory. The address is simply an absolute number, or assembler label (which is the address in memory from which the data is to be accessed). For example, we could write:

```
LDR R0,&1000
LDR R0,table
```

The first example would load the word of memory from location &1000 into register R0. The second example would access the memory location labelled in the program as 'table'. Again, the contents of this location would be loaded into register R0.

When the assembler encounters a PC relative instruction, it will always know the address at which it is being assembled. It can also look at the address specified in the instruction itself, and calculate the difference between the two addresses. This can be viewed as an offset from the address of the instruction to the address of the memory word which it accesses.

The assembler can, therefore, assemble this as an instruction which uses pre-indexed addressing. The base register in this instruction is the program counter, R15. This will, ignoring pipelining, contain the instruction's ad-dress when executed. The offset field contains the absolute offset number previously calculated by the assembler, with a correction for pipelining.

When the ARM performs the pre-index calculation and adds together the contents of the PC and the offset, the address of the data originally given in the instruction, is obtained.

An important point to remember is that the range of the offset in pre-indexed addressing is –4096 to 4096. When using PC relative addressing the difference between the address of the instruction and that of the memory location to be accessed, must be within this range. If this is not the case, then the assembler will not be able to produce a legal pre-indexed equivalent to the instruction and an error will be given.

Byte and Word Addressing

In the previous sections, we have looked at the different ways in which the ARM can access complete words of memory (four bytes). In some situations, however, it is more convenient to access single bytes of memory. For example, when manipulating character strings, each character will only require a single byte to store it. In cases like this we need to use some form of single byte access.

All of the previously described addressing modes can still be used when we access single bytes. The syntax of each instruction is virtually the same. The only difference is we tell the ARM that when it accesses data at a given address, it is only to transfer a single byte rather than a complete word.

In assembler, we specify that we are accessing bytes instead of words by using a B suffix to the instruction mnemonic. This is placed after any condition codes which may be present. A few examples should make the syntax a litle clearer:

```
LDRB    R0,[R2,R4]
STRB    R0,[R5,#4]
LDRB    R0,[R6,R5,LSR#6]!
LDRNEB  R0,[R1],R3
LDRB    R0,table
```

When we access complete words of data, the ARM requires the final address to be word aligned. As we are now dealing with single bytes of data, this requirement does not apply. The final address, derived after performing any addressing mode calculations required, can be anywhere in the memory map.

Multiple Register Transfers

In the previous section we saw how individual words and bytes of data can be transferred between registers and memory. Often, however, we will need to transfer data between several different registers and memory. It

would be extremely tedious and inefficient to repeatedly write LDR and STR instructions for each of these transfers. For this reason, the ARM provides us with two instructions which load and store the contents of several registers at a time. These instructions are LDM and STM, the multiple load and store instructions respectively.

STM

The syntax of the STM instruction is:

```
STM <options> <base> {!}, <register_list>
```

The register_list is the series of register names, separated by commas, the contents of which we want to store in memory. The order of the registers in the list is of no significance and any number of registers can be given up to the maximum of 16. The assembler will allow a range of registers to be specified by using a '–' character. The following are all legal ways of specifying the same list of registers:

```
R0,R1,R2,R3,R9,R13
R0-R3,R9,R13
R9,R0-R3,R13
```

The base field in the instruction must be given as a simple register.

The contents of this are taken to be the start address in memory from which the registers are to be saved.

The options field is a two-character code which defines how the instruction should be executed. The options available will be described later.

As the ARM executes the instruction, it will store the contents of each of the registers, named in 'register_list', in consecutive memory words. A copy of the address in the base register is used and modified by the ARM as each register is stored. The actual contents of the base are not changed, unless we request this using the write back option.

After storing each register, the address being used will be modified so that the next register is stored in the next consecutive location. We can specify whether we want the address to be incremented or decremented after each register store. Thus, we can define the direction in which registers are stored in memory.

Direction of Storage

The storage direction used by the instruction is controlled by the first character in the option field. This may be either of the following:

I Increment address after storing each register
D Decrement address after storing each register

If an incrementing address is specified, then registers will be stored in locations: 'base', 'base + 4', 'base + 8' and so on. If a decrementing address is specified then registers will be stored in locations: 'base', 'base – 4', 'base – 8' and so on.

Pre or Post-address Modification

The second letter in the option field specifies whether the address is to be modified before or after each register is stored. The following options can be used:

A Modify address after storing each register
B Modify address before storing each register

If the address is modified after storing each register, then the first register will be stored at the address in base, and the second at (base + 4) or (base – 4), depending on the increment/decrement option.

If the address is modified before storing each register, then the first register will be stored at the address in (base +4) or (base – 4), again depending on the increment/decrement option. The second register will be stored at (base + 8), or (base – 8), and so on.

Examples of instructions using all four option codes are given in figure 10.4 on the next page.

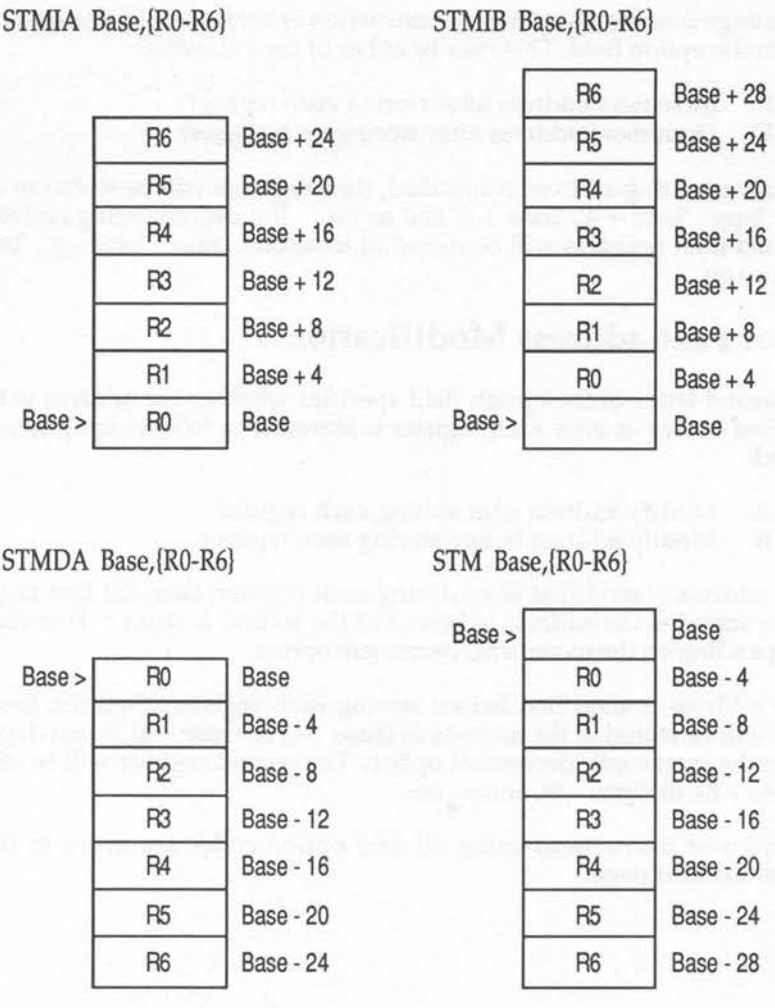


Figure 10.4. Examples of LDM instructions using various option codes.

Write Back

We have said that as the ARM stores registers, it modifies the address being used. This ensures that the next register processed is stored in a consecutive word of memory, and does not overwrite the previous one.

If we specify that we want write back, then the final address, obtained after storing all of the registers in the list, will be written back into the base register.

Write back is selected, as before, by including a '!' character. Thus, the following instructions all have write back selected:

```
STMIA   R0!,{R1,R2}
STMDA   base!,{R4,R5-R9}
LDMIB   R6!,{R12,R11,R10}
```

After each instruction, the ARM performs one of the following depending on the direction of storage used:

```
base = base + 4*n    (increment)
```

or alternatively:

```
base = base - 4*n    (decrement)
```

Where 'n' is the number of registers stored by the instruction. Figure 10.5 shows the the effects of write back in some example cases. Write back is provided so as to support the creation of stacks using the LDM/STM instructions. This is covered in Chapter 12.

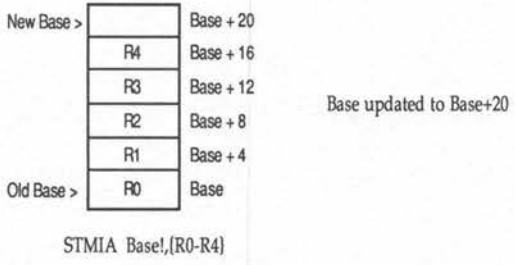


Figure 10.5. The effects of using write back.

Applications of STM, LDM

Taken at its simplest level, STM can be used to preserve the contents of a register group in an arbitrary block of memory. The original contents can then be restored at a later time using LDM with the same memory address and register list. For example, to preserve the contents of registers R1 to R14 we could use the following, assuming that register R0 contains the address of a free block of memory:

```
STMIA  R0,{R1-R14}
```

The register contents could then be restored by using the following instruction, assuming that R0 contains the address of the same block of memory:

```
LDMIA  R0,{R1-R14}
```

The programmer is left to decide which options are to be used with the instructions. However, they must be used consistently in both instructions otherwise the registers will be reloaded from the wrong address.

The major use of LDM and STM is in the support of data structures known as stacks. The implementation and use of these is described in Chapter 12.

11 : Branches and SWI

In this chapter the final two ARM instructions will be described. These are the branch and software interrupt (SWI) instructions.

There are two variants of the branch instruction supported by the ARM:

Branch (B)
Branch with Link (BL)

Simple Branch (B)

This is the simplest form of the branch instruction and is analogous to the BASIC GOTO statement. It is used to make the processor break off from its normal sequential execution of instructions and jump to a new instruction at a designated location. Ignoring the usual conditional suffixes, which can be used with any instruction, the syntax of branch is:

```
B <address>
```

The address is the address of the instruction which the ARM must branch, or jump, to. This may be done by specifying the absolute address to be branched to, or an assembler label which will be evaluated to get the branch address.

Although in the assembler we give the actual address to be branched to, this is not what is encoded into the branch instruction.

Assembler calculates the difference, or offset, between the branch instruction address and the location to be branched to. It is this offset which is encoded into the instruction. When the ARM executes the branch, it treats the offset as being relative to the current contents of the program counter, R15. The result of adding the offset to the program counter derives the original, absolute address which is to be branched to.

This is done to aid making machine code programs relocatable. A relocatable program is one which will operate correctly irrespective of its absolute

address in memory. If branches used absolute addresses, then each and every one of them would have to be modified if the program is moved.

Using offsets to implement relative branching eliminates this problem. As long as the target address to be branched to doesn't move relative to the branch instruction, then the absolute position of the program in memory does not matter.

Conditional Branches

Many processors, including the 6502, have a complete set of distinct branch instructions. Each of these instructions causes a branch to occur if a specified condition is TRUE, eg, if the carry flag is set. These branches are used to implement conditional sections in programs. By using an appropriate branch instruction, pieces of code can be skipped over, or executed, depending on the result of a previous operation.

The ARM only has one branch instruction. However, it allows any instruction to execute conditionally upon any one of 16 conditions. We do not, therefore, need separate instructions to implement conditional branching. We simply use the fundamental B instruction, then add the appropriate suffix to make the branch conditional. For example, if we want a piece of code to be branched to where a previous operation gave a negative result, we write the following:

```
BMI routine
```

A major use of branch instructions is to create program loops. Using branches, we can repeatedly execute a section of code, as long as a certain condition is met. Listing 11.1 uses this technique to implement two nested loops. The inner loop accesses and prints the first 'n' characters in a string. The outer loop increases 'n' from '0' until all of the string is outputted.

Listing 11.1. Branches and loops

```
10  REM An example of branches and loops
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  REM Set up string to be printed
70  DIM string_buff 32
80  $string_buff = "Acorn RISC Machine"
90  length = 18
```

```
100
110 REM Define names for register used
120 index     = 1
130 base      = 2
140 num_chars = 3
150
160 DIM loops 512
170 P%=loops
180 [
190 ADR base,string_buff   ; Get addr of string in base register
200 MOV num_chars,#1       ; First output 1 character in string
210 :
220 .outer_loop            ; Inc 'n' loop - no. chars to print
230 MOV index,#0           ; Initialise string index pointer
240 .inner_loop            ; Output first 'n' characters loop
250 LDRB R0,[base,index]   ; Get character pointed to by index
260 SWI "OS_WriteC"        ; Print character on the screen
270 ADD index,index,#1     ; Inc index to the next character
280 CMP index,num_chars    ; Have first 'n' chars been printed?
290 BLT inner_loop         ; If not, branch to start of printing
300 SWI "OS_NewLine"       ; Output a newline to the screen
310 ADD num_chars,num_chars,#1 ; Increment 'n'
320 CMP num_chars,#length  ; Has 'n' reached full string length?
330 BLE outer_loop         ; If not branch and print again
340 MOV PC,R14 ; Return to BASIC
350 ]
360
370 PRINT
380 PRINT "Demo of loops and branches"
390 PRINT
400 CALL loops
```

Chapter 23 contains details of how to implement various high-level machine code looping constructs using the branch instruction.

Branches and Conditional Instructions

Readers who are familiar with programming the 6502 will know how often branches are used to skip one or two instructions. For example, the following type of code frequently crops up:

```
[
    SUBS R0,R0,#1
    BPL  not_negative
    MOV  R0,#10
.not_negative
]
```

This decreases the value of register R0 by one, and reloads it with the number 10 should it becomes negative. A branch instruction skips the re-load instruction if it isn't needed. While there is nothing wrong with this code, the ARM processor offers facilities to write it more efficiently:

```
[
        SUBS    R0,R0,#1
        MOVMI   R0,#10
]
```

In this version, the ARM's generalised conditional execution facility is used to completely remove the need for the branch. This type of instruction crops up a great deal. Thus, branches in ARM programs are not used quite as often as they are with other processors.

Branch with Link: BL

The second form of the branch instruction is Branch with Link (BL). The ARM provides this as a primitive to implement subroutine mechanisms in machine code.

The instruction has the same format as the simple branch instruction:

```
BL <address>
```

However, BL copies the contents of register R15 into R14 immediately before the ARM branches to the new address.

This preserves a copy of the program counter and status flags in register R14. When copying the program counter, the effects of pipelining are automatically corrected. The address stored in bits two to 26 of R14, therefore, is really the instruction immediately following the branch instruction.

By using the address in R14, we can effectively return to the original section of code immediately after the branch. This is achieved by moving the contents of R14 back into R15. Execution will then resume from the statement following the branch instruction.

The analogies between the BL instruction and subroutines in BASIC are clear. It allows us to call self-contained sections of code from anywhere in a program, and return to the original position after the subroutine has been executed.

R14 is called the link register because it contains the address at which we can re-link back into the program which called the subroutine.

The general outline of how a subroutine is implemented using BL is outlined in figure 11.1.

```
.main_program
    .
    .
BL subroutine
    .
    .
.end_program
.subroutine
    .
<body of subroutine>
    .
MOV R15,R14
```

Figure 11.1. Subroutine outline using the BL instruction.

A specific example should make this clear. Listing 11.2 contains a small sub-routine which implements the BASIC command PLOT k,x,y. The k,x,y parameters are passed to the subroutine in registers R0, R1 and R2 respectively. The main program simply calls the subroutine a few times to draw a triangle and circle.

Listing 11.2. Sub-routines using Branch with Link.

```
 10  REM A general PLOT subroutine using the BL instruction
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  REM Program uses 2 pass assembly. This is described in
 70  REM Chapter 13 of the book
 80
 90  REM Define constants for the program
100  vdu  = 256     : REM Start of SWI block to perform VDU n
110  plot = 25      : REM PLOT is implemented as VDU 25
120  move = 4       : REM Move is PLOT command 4
130  triangle = 85  : REM Triangle is plot code 85
140  circle = 157   : REM Circle is plot code 157
150
160  REM Define names for the registers used in the program
```

```
170  k = 0          : REM  Passes PLOT option code 'k'
180  x = 1          : REM  Passes PLOT x co-ordinate
190  y = 2          : REM  Passes PLOT y co-ordinate
200
210  DIM shapes 512
220
230  REM TWO pass Assembly
240  FOR pass = 0 TO 3 STEP 3
250  P%=shapes
260
270  [
280  OPT pass       ; Select assembly option
290
300  MOV R10,R14    ; Preserve R14 contains BASIC return address
310
320  MOV k,#move    ; Set regs k,x,y for calling PLOT subroutine
330  MOV x,#640     ; Using the subroutine to perform
340  MOV y,#512     ; MOVE 640,512
350  BL plot_it     ; Call the subroutine
360
370  MOV k,#circle  ; Set up registers k,x,y again
380  MOV x,#640     ; This time for PLOT 157,640,256
390  MOV y,#256
400  BL plot_it     ; Call the subroutine
410
420  MOV k,#move    ; Set up registers k,x,y again
430  MOV x,#420     ; This time for MOVE 420,640
440  MOV y,#64 0
450  BL plot_it     ; Call the subroutine
460
470  MOV k,#triangle    ; Set up registers k,x,y again
480  MOV x,#860     ; This time for PLOT 85,860,640
490  MOV y,#640
500  BL plot_it     ; Call the subroutine
510
520  MOV PC,R10     ; Return to BASIC - Addr moved to R10
530
540  .plot_it       ; Start of PLOT subroutine
550  SWI vdu+plot   ; Issue VDU 25 ie. PLOT
560  SWI "OS_WriteC"    ; Output the PLOT option code 'k'
570  MOV R0,x       ; Move x co-ord of the point into R0
580  SWI "OS_WriteC"    ; Output low byte x co-ord to VDU driver
590  MOV R0,x,LSR#8 ; Get high byte of x co-ord in low byte of R0
600  SWI "OS_WriteC"    ; Output high byte x co-ord to VDU driver
610  MOV R0,y   ; Move the y co-ordinate of the point into R0
620  SWI "OS_WriteC"    ; Output low byte y co-ord to VDU driver
630  MOV R0,y,LSR#8 ; Get high byte of y co-ord in low byte of R0
640  SWI "OS_WriteC"    ; Output high byte y co-ord to VDU driver
650  MOV PC,R14 ; Return from subroutine to main program
660
670  ]
680  NEXT
690
```

```
700  MODE 0
710  GCOL 3,1
720  CALL shapes
```

Note that a MOV instruction is used to move the return address from register R14 back into the program counter. This will have no effect on the status flags. The original settings of the flags, will not be restored. This is useful for the subroutine to communicate some results to the calling routine by conditioning the flags.

You may, however, want the status flags to remain unaffected by the call to the sub-routine. In this case the subroutine should return using the following instruction:

```
MOVS R15,R14
```

This will restore the value of the program counter and the original settings of the status flags.

Preserving the Link Register

It is important to remember that every time a BL instruction is executed the contents of R15 are copied into R14. This means that if we are already in a subroutine, when a second one is called, the original re-link address of the first subroutine will be over-written by the second one.

For this reason, we must save the contents of R14 when another subroutine is called. We also do this if we want to return to BASIC from our routine, as the BASIC return address is passed in register R14.

You can simply move R14 into another register to preserve it. This is shown in listing 11.2. However, there is a problem. When the second subroutine calls a third one, which calls a fourth and so on, we have to prese rve R14 each time. If the depth of these sub-routine calls is too great, we qui kly run out of registers. Also, in dynamic problems which use recursive subroutine calls, we don't know beforehand the depth of the

A more general solution is to store R14 on a stack every time a procedure is called. This is described in the next chapter.

Software Interrupt: SWI

The SWI instruction is one of the simplest, yet most important, of the ARM's instruction set. SWI stands for software interrupt. An SWI instruction is used when we want the operating system to perform some task on our behalf. For example, controlling the mouse, creating screen windows, reading keys, loading disc files, making sound effects and so on. The syntax of the instruction is as follows:

```
SWI <argument>
```

When executed, this instruction causes the processor to break off from the current program. The ARM then switches into supervisor mode and jumps to a pre-defined address in the operating system. The argument field is then examined to determine which of the many operating system facilities has been requested. When the appropriate routine has been completed, the ARM resumes the execution of the user program where it left off.

The argument field of the SWI is a 26-bit quantity which defines the number of the operating system routine required. We can, therefore, write statements like:

```
SWI 0
```

This will call operating system routine number zero, which writes a character to the screen. In practice, however, it is difficult to remember which routine has which number. However, assembler allows us to specify the name of the routine we want. It will then look up the corresponding number in an internal table and construct an appropriate SWI instruction. In assembler, we can write statements like:

```
SWI "OS_Mouse"
```

or like:

```
SWI "Wimp_CreateWindow"
```

The number of each of the named routines will be looked up and substituted in the SWI instruction. Note that when specifying the names of the routines, the quotes are compulsory as we are really giving a string argument. The name of the routine must exactly match up with that recognised by the assembler including the case of each character. For example, if we write:

```
SWI "OS_mouse"
```

an error would be produced because we have not used a capital M. This is a common mistake!

Most of the examples given previously have used SWI calls in some way or other. A complete list of the operating system routines accessed through SWIs is given in Appendix E. For detailed descriptions of many of the more useful routines see Chapters 17 to 19.

12 : Stacks and LDM/STM

A stack is a widely used data structure. The standard stack analogy compares the stack to a pile of plates. When new plates are added to the pile they are always placed on the top of existing plates. Similarly, when a plate is removed, it is always the plate on top of the stack which is taken off first.

The most important part of the analogy to remember is that the plate on top of the stack is always the last one added. This is also the first one to be removed. The stack is therefore called a 'last in, first out' structure (LIFO). When an element is added, we say that it has been 'pushed' onto the stack. When we remove an element, we say that it has been 'pulled' from the stack. Figure 12.1 illustrates these two operations. Note that a series of items pulled from a stack are always obtained in the reverse order to when they were pushed on to it.

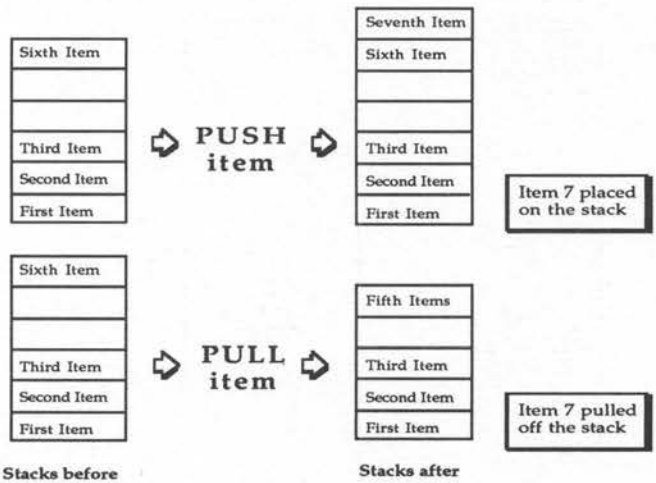


Figure 12.1. Simple model of a stack.

Computer Stacks

Computer stacks are implemented using exactly the same principles as the pile of plates. A series of contiguous memory locations are set aside to hold the data in the stack. We also need some sort of pointer to record where the top of the stack is. When we add an item to the stack, we store it in the memory word pointed to by the stack pointer. We then increment the pointer. When we remove an item from the stack, we first decrement the stack pointer and then access the memory word which it points to. An example of this is shown in figure 12.2. Entries in the stack are complete words of memory (four bytes) and so the stack pointer is incremented and decremented in units of four each time.

The disc which accompanies this book contains a program modelling the operation of a stack. This allows us to view the stack structure as data is pushed onto it and pulled off it.

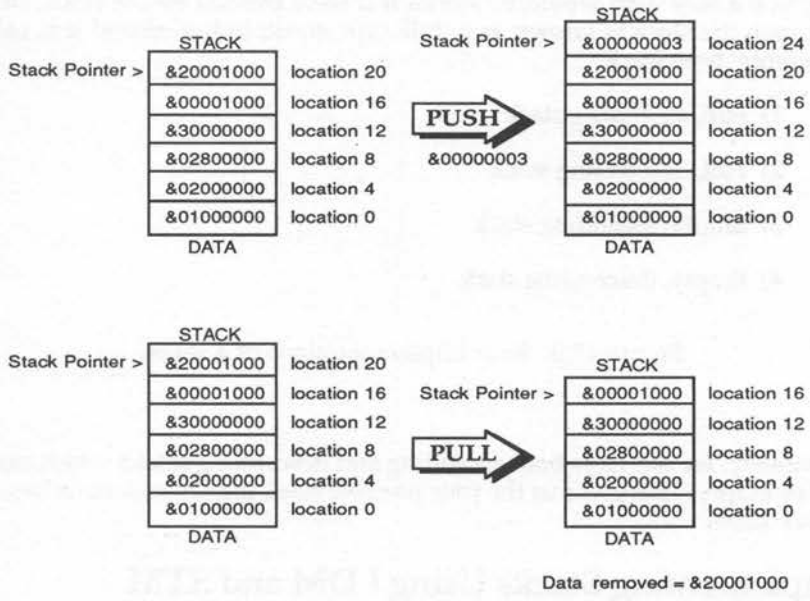


Figure 12.2. A computer stack and stack pointer.

Types of Stack

The stack we just looked at was only one example of the way in which a stack can be created. There are four possible stack structures. These are made of combinations of two variants. The first of these variants determines which direction the stack grows in.

We can create stacks that grow upwards in memory as extra items are pushed on, and contract downwards as items are pulled off. This type is called an ascending stack. Similarly, we could implement a stack which grows downwards in memory as items are added, and contracts back up again as they are pulled off. This is called a descending stack.

When you implement a stack, you must decide exactly what the stack pointer should point to. It could point to the top entry on the stack, ie, the one most recently pushed on the stack. Alternatively, it may point to the next available space in the stack's memory area. This would be the address at which a new item would be stored if it were pushed on the stack. In the first case the stack is known as a 'full' type stack, in the second it is called an 'empty' type stack.

1) Full, ascending stack

2) Full, descending stack

3) Empty, ascending stack

4) Empty, descending stack

Figure 12.3. Four implementations of a stack.

Obviously, we can have both ascending and descending stacks which can be full or empty. This gives us the four possible stack implementations listed in figure 12.3.

Implementing Stacks Using LDM and STM

The LDM and STM instructions provide the facilities to implement stacks in machine code. The elements pushed to, and pulled from, the stack are the

register contents specified in the instruction's register list. This means that we can push or pull several items in a single instruction.

The stack pointer is implemented using the instruction's base register. This always points to the address in memory where the instruction will store or load data. From now we will refer to this register as the stack pointer. Register R13 is used most often for this purpose although any register could be used.

Initially, the stack pointer will be set to contain the base address of the stack's memory area. Write back can then be used with the LDM and STM instructions to make the ARM automatically update the stack pointer each time registers are pushed to the stack or pulled from it. Remember that in this type of stack, the stack pointer always points to the next free space on the stack, ie, immediately after the last item pushed.

When we push registers on this stack, we obviously want the registers to be stored ascendingly in memory. Also, as the stack is empty, we want the address used to be incremented after storing each register. This will ensure that the first register is stored at the address contained in the stack pointer. Similarly, after all the registers have been stored, the current address will be the location after the last item in the stack. The appropriate multiple store instruction is as follows:

```
STMIA  stack_pointer!,{register_list}
```

The use of write back is vital. Without it the stack pointer will never be updated and the stack will be corrupted.

When we pull registers off the stack, we need to use an LDM instruction with a decrementing address. We do this because the items to be pulled are located immediately before the address contained in the stack pointer. The address in the stack pointer is the location immediately after the top item on the stack. We must, therefore, decrement the address used before loading each register. The corresponding LDM instruction for this is as follows:

```
LDMDB  stack_pointer!,{register_list}
```

To summarise, for an empty ascending stack, we use the following instructions. Note that the options for the STM instruction are always reversed in the case of the LDM instruction:

Push registers:

```
STMIA  stack_pointer!,{register_list}
```

Pull registers:

```
LDMDB  stack_pointer!,{register_list}
```

It can be a bit confusing to have to translate the stack type used into LDM and STM instructions with appropriate increment/decrement and before/after options! For this reason assembler provides an easier way.

We simply can specify the type of stack being used, which will be the same for both instructions. The assembler will look at the stack type, whether the instruction is LDM or STM and choose appropriate options for the instruction. The stack type is specified using a different set of option codes. These are given in figure 12.4.

FA	Full, ascending stack
FD	Full, descending stack
EA	Empty, ascending stack
ED	Empty, descending stack

Figure 12.4. Option codes for specifying stack types.

The empty ascending stack would thus be implemented as follows:

Push registers:

```
STMEA    stack_pointer!,{register_list}
```

Pull registers:

```
LDMEA    stack_pointer!,{register_list}
```

This clearer notation is used whenever the LDM/STM instructions are performing stack operations. The other codes, which reflect what is being done by the instructions, are used when registers are being dumped and reloaded from memory.

Listing 12.1 shows an implementation of a real stack. The stack is a full ascending one. The program accepts characters from the keyboard until you press RETURN. As each character is typed, its ASCII code is pushed onto a stack. When you press RETURN, characters are pulled back off the stack

and printed until the stack is empty again. The characters will be outputted in the reverse order to when they were input. This shows the LIFO nature of the stack.

Listing 12.1. Example of machine code stacks.

```
 10 REM An Example of a stack using LDM and STM instructions
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40
 50 REM The stack is a full-ascending type
 60 REM Reserve space for the stack
 70 DIM stack 256
 80
 90 return = 13    : REM Character constant
100 count  = 2     : REM Register counting characters entered
110
120 DIM code 512
130 P%=code
140 [
150 ADR R7,stack        ; Point R7 to bottom of stack
160 mov count,#0        ; Set character counter to '0'
170 .get_chars          ; Loop to get and push characters
180 SWI "OS_ReadC"      ; Read character in
190 SWI "OS_WriteC"     ; Echo it to the screen
200 STMFA R7!,{R0}      ; Push the character onto the stack
210 ADD count,count,#1 ; Increment the character count
220 CMP R0,#return      ; See if the last character was return
230 BNE get_chars       ; If not, carry on getting characters
240 SWI "OS_NewLine"    ; Print a new line
250
260 .pull_chars         ; Loop to pull chars from stack
270 LDMFA R7!,{R0}      ; Pull next char from stack
280 SWI "OS_WriteC"     ; Print the character
290 SUBS count,count,#1; Dec count of chars on stack
300 BNE pull_chars      ; If some remain repeat loop
310 SWI "OS_NewLine"    ; Print a new line
320
330 MOV R15,R14         ; Return to BASIC
340 ]
350 PRINT
360 PRINT "Enter string now!"
370 REPEAT
380 PRINT
390 CALL code
400 UNTIL FALSE
```

Stack Application

Stacks are used extensively when data needs to be preserved in a general way. This could be dumping the register contents to a stack on entry to a routine and reloading them later. Several such routines could use the stack and we would not, therefore, need to allocate separate memory areas for each.

The beauty of the stack, however, is that its LIFO nature makes it ideal in cases where nested constructions are being used. For example, when implementing subroutines, the return address of each routine could be saved on a stack when it is called. This would allow sub-routine calls to be nested arbitrarily. The return address of the most recently called subroutine would always be the top item on the stack. Both the operating system and BASIC make use of stacks in this way.

13 : The BASIC Assembler 2

In this chapter we return to the subject of the BASIC assembler. This was covered briefly in Chapter Four, but we will now look at some of the more advanced facilities.

OPT Settings

In Chapter Four we said that a listing was produced by the assembler by default, but could be suppressed if required. This, and several other functions, are controlled by a special assembler command called OPT.

OPT is an example of an assembler directive or pseudo opcode. These appear as source code in assembler in exactly the same way as an ARM instruction mnemonic. However, they do not produce any machine code when assembled. Instead, they direct the assembler to perform a special function.

OPT is almost always the first instruction in an assembler program. It is followed by a single three-bit number, ie, in the range zero to seven, (or a variable containing a number in the appropriate range). For example:

```
OPT 0
OPT p*3
OPT pass
```

Each bit in the number selects, or de-selects, a specific assembler function. The functions controlled are: producing assembler listings, the reporting of errors and offset assembly. The function of each bit is summarised in figure 13.1. All possible OPT settings (zero to seven) are listed in figure 13.2 on the next page.

Bit 0 Assembler listing:	0 = No listing produced 1 = Listing produced
Bit 1 Error control:	0 = No errors reported 1 = Errors reported
Bit 3 Offset assembly:	0 = No offset assembly 1 = Offset assembly performed

Figure 13.1. Functions controlled by OPT bits.

	Offset Assembly	Errors Reported	Listing Produced
OPT 0	No	No	No
OPT 1	No	No	Yes
OPT 2	No	Yes	No
OPT 3	No	Yes	Yes
OPT 4	Yes	No	No
OPT 5	Yes	No	Yes
OPT 6	Yes	Yes	No
OPT 7	Yes	Yes	Yes

Figure 13.2. All possible OPT settings.

If no OPT directive is used, then the default value of three is selected. This corresponds to:

Listing produced
Errors reported
No offset assembly

Error Control

It may seem strange that the OPT directive can be used to prevent the error reporting. After all, if there is an error in our code, then surely we would want to know about it!

The reason for this is clear when we look closely at assembler's system of defining and referencing labels. This is best illustrated by examining a real program. Type in listing 13.1. When called from BASIC the routine out-

puts the character passed into register R0 via the integer variable A%. If this character's ASCII code is less than 32, the print instruction is skipped using a forward branch.

Listing 13.1. Forward references.

```
 10 REM The problems of forward references
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 REM Program will not work because of the forward reference
 70
 80 DIM output 256
 90 P%= output
100 [
110 CMP R0,#32          ; Compare character's ASCII value with 32
120 BLT finish          ; IF < 32 then skip print instruction
130 SWI "OS_WriteC:"    ; Print the character
140 .finish             ; Branch destination of skip print inst.
150 MOV PC,R14 ; Return to BASIC
160 ]
170 REPEAT
180 A%=GET
190 CALL output
200 UNTIL FALSE
```

So everything looks OK. The branch instruction will jump over the print instruction to the 'finish' label, if R0 is less than 32. However, run the program and see what happens. Figure 13.3 shows the output produced.

```
>RUN
0000879C
0000879C E3500020  CMP R0,#32

Unknown or missing variable at line 120
```

Figure 13.3. The output produced by listing 13.1.

The assembler tries to assemble the branch instruction and finds a reference to the label. However, at this stage it does not know the label address and so it can't complete the assembly. An 'unknown or missing variable' error message is displayed. It makes no difference if the label is defined later on in the program. This is an example of a reference before a label definition and occurs when we want to make forward references in our code.

The solution is to make the computer assemble the source code twice! The first time we know errors will be produced because of forward references, so we suppress them using the OPT directive. After this phase is completed, assembler will have gone through the entire source program once. All the label definitions in the program will, therefore, have been encountered, so all labels used should now be defined.

Thus, the second time the program is assembled, all instructions which reference a label can be correctly assembled. This system is known as a two-pass assembler.

So much for the theory, but how do we implement two-pass assembler in practice? Once more the fact that assembler is part of BASIC helps us. We simply enclose the entire assembler section in a FOR...NEXT loop which repeats the assembly twice. If we modify the previous program to incorporate two-pass assembly, we arrive at listing 13.2.

Listing 13.2. Forward references using two-pass assembly.

```
 10 REM Forward references using 2 pass assembly
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 :
 70 :
 80 DIM output 256
 90
100 REM Use a FOR .. NEXT loop to implement 2 pass assembly
110
120 FOR pass = 0 TO 3 STEP 3
130 P%= output
140 [
150 OPT pass            ; Select current pass option (0 or 3)
160 CMP R0,#32          ; Compare character's ASCII value with 32
170 BLT finish          ; IF < 32 then skip print instruction
180 SWI "OS_WriteC"     ; Print the character
190 .finish             ; Branch destination of skip print inst.
200 MOV PC,R14          ; Return to BASIC
210 ]
220
230 NEXT pass           : REM do the next pass
240
250 REPEAT
260 A%=GET
270 CALL output
280 UNTIL FALSE
```

Note that the control variable of the loop (pass) is used to select an appropriate OPT setting on each pass. On the first pass we want to suppress all errors. Also the assembler listing is unlikely to be very helpful, so we suppress it. Thus, we use an OPT 0 for the first pass.

On the second pass, any errors which occur are due to real mistakes in the program, so we certainly want these to be reported. It is also useful to have a listing at this stage. In pass two therefore, OPT 3 is used.

Offset Assembly

The need for offset assembly arises when we want to make the assembler store the assembled machine code at an address other than the one where it will ultimately execute.

This often happened on the BBC and Master series computers when sideways ROMs were being written. Such programs were designed to execute starting at address &8000, however, on a standard BBC micro the BASIC ROM occupied this memory area. Programs, therefore, had to be assembled to a different area of the computer's memory. Offset assembly allows us to do exactly this.

The address at which the machine code will start when it is executed is still placed in P% as normal. However, a second address (the address where the code is to be stored by the assembler) is placed in the variable O%. The assembler assembles code exactly as if it were storing it in the address contained in P%. However, it physically writes the machine code to the address contained in O%.

To select offset assembly, we simply use our normal OPT values but with bit two also set. In most cases this gives values of four and seven for the two assembler passes, rather than the values zero and three when the offset assembly is not being used.

On the Archimedes, the ARM processor instruction set allows programs to be written so that they are totally independent of the address where they are executed. Such programs are called relocatable and can be loaded in at any address and executed, no matter what original address they were assembled at.

Writing relocatable programs makes offset assembly almost redundant and is a much better practice to get into. There are still uses for offset assembly, but in most cases it will not be required.

Storing Data in Assembly Programs

In all but the simplest machine code programs, we have to make use of some memory as workspace. This may be needed to store text strings, data tables, variables and so on, or perhaps just to act as scratch areas for various routines to use. The assembler has a series of directives which allow us to reserve given amounts of memory for purposes such as these. It also allows us to define the contents of the memory reserved.

The directives provided are shown in figure 13.4 together with the number of bytes that each reserves. When one of these directives is assembled, the argument following the directive is evaluated and the result is stored in memory. The value of P% is then incremented by the appropriate amount. For example:

```
EQUB 20
```

would store the number 20 in the next memory location and increment P% by one byte. Similarly:

```
EQUD &12345678
```

would store the number &12345678 as a four-byte number and increment P% by four.

Directive's name	Alternative name	Function
EQUB	DCB	Reserve one byte
EQUW	DCW	Reserve one 'word' (two bytes)
EQUD	DCD	Reserve a double 'word' (four bytes)
EQUS	-	Reserve string (zero to 255 characters)

Figure 13.4. Data defining and space reserving directives.

Note, that in the context of the EQU directives a 'word' refers to 16 bits (two bytes) of memory rather than the 32 bits (four bytes) which we would expect on the ARM. Thus, EQUW reserves two bytes of memory and EQUD

(Equate double word) reserves four bytes. This discrepancy is due to the desire to keep the commands compatible with those available on the BBC micro where the terminology is slightly different.

The EQUS directive is different from the others in that it takes a string as its argument. This string is copied character by character into memory. P% is then incremented by the length of the string so that it points to the location immediately after the string.

Listing 13.3 shows some examples of using the EQU directives. In the program a series of strings are stored using the EQUS directive. Following each string is a zero byte which marks the end of the string. The operating system SWI routine (OS_Write0) is used to print the strings.

Listing 13.3. Using the EQU directives.

```
 10 REM Printing strings created by EQU directives
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM print_strings 256
 70
 80 REM Two Pass assembly again
 90 FOR pass = 0 TO 3 STEP 3
100 P%= print_strings
110 [
120
130 OPT pass
140
150 ADR R0,string1      ; Get address of first string into R0
160 SWI "OS_Write0"     ; Use SWI to print the string
170 SWI "OS_NewLine"    ; Output a Newline
180
190 ADR R0,string2      ; Get address of second string into R0
200 SWI "OS_Write0"     ; Use SWI to print the string
210 SWI "OS_NewLine"    ; Output a Newline
220
230 ADR R0,string3      ; Get address of third string into R0
240 SWI "OS_Write0"     ; Use SWI to print the string
250 SWI "OS_NewLine"    ; Output a Newline
260
270 ADR R0,string4      ; Get address of fourth string into R0
280 SWI "OS_Write0"     ; Use SWI to print the string
290 SWI "OS_NewLine"    ; Output a Newline
300
310 MOV PC,R14          ; Return to BASIC
320
330 ; Store strings using EQUS. Terminate each string with
```

```
340  ; a zero byte using EQUB
350
360  .string1
370  EQUS "Hello... this is the first string"
380  EQUB 0
390  .string2
400  EQUS "That was easy - here is the next string"
410  EQUB 0
420  .string3
430  EQUS "Printing strings is easy using SWI calls"
440  EQUB 0
450  .string4
460  EQUS "Good old ARTHUR!!"
470  EQUB 0
480  ]
490  NEXT pass
500
510  CALL print_strings
```

Note that two-pass assembly is used because forward references are made to the string starting labels. Note also, the use of the ADR directive to place the start addresses of the strings into a register.

The ALIGN Directive

We have already said that the ARM processor requires all its instructions to be stored on word boundaries, ie, at addresses which are divisible by four. This immediately raises a problem with the use of the EQU directives. It is possible, using EQUB, EQUW or EQUS to end up with a value of P% which is not word-aligned. For example, consider the directive:

```
EQUS "This string contains 35 characters!"
```

This directive will store 35 characters in memory and add 35 to P%. If P% is word-aligned to start with, the address it contains after the directive will not be on a word boundary. If we now want to continue assembling instructions, then we should correct the value of P% to make it word aligned again. This can be done by using assembler's ALIGN directive.

When ALIGN is used in a program, the assembler checks the value of P% to ensure that the address it contains is word-aligned. If it is not, an appropriate number of bytes are added to the address in P% to correct it. Listing 13.4 gives an example of ALIGN. Run the listing and look at the addresses printed in the left-hand column of the assembler listing. Verify that after

the EQUS directive, the address has become non-aligned but that after ALIGN it is corrected.

Listing 13.4. Demonstration of the ALIGN directive.

```
 10  REM THE ALIGN Directive
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40
 50  DIM test 256
 60  P% = test
 70  [
 80  EQUS "This string contains 35 characters!"
 90  ; P% not word-aligned here
100  ALIGN
110  ; P% corrected - word-aligned again!
120  ]
```

CALL Parameters

The full syntax of the CALL statement is:

```
CALL <address> {,<parameters>}
```

The optional parameters are a list of comma, separated BASIC variables, the values of which are to be made available to the machine code routine. CALL provides ways of passing information about these parameters to the machine code routine. In addition to setting up registers R0 to R8 from the integer variables, CALL also sets up the following two registers:

R9: Pointer to parameters descriptor block
R10: Number of parameters passed using CALL

When the routine is entered, register R10 always contains the number of parameters given in the CALLing statement. In the following example, therefore, R10 would contain four as four parameters are being passed:

```
CALL address, A,B$,table(),?&FFEE
```

Register R9 points to the memory block where a parameter list has been created. This list has a two-word entry in it for each variable in the parameter list. An important point to note is that the entries in the parameter block are set up in reverse order, ie, the entry for the last parameter in the list appears first in the parameter block.

The first word of each entry is a pointer to the address where the variable itself is stored. The second word contains a number which represents the type of the variable passed. Figure 13.5 lists all the different type numbers. It also gives examples of BASIC variables of each type and lists what the associated address of each type points to.

Type number	Object address points to	Examples of possible BASIC variables
0	Single byte number	?var
4	Four-byte integer value	!var, var%, var%(n)
5	Real number (five bytes)	var, var, var(n)
128	String information block	var$, var$(n)
129	Terminated character string	$var
256+4	Integer array block	var%()
256+5	Real array block	var()
256+128	String array block	var$()

Figure 13.5. Parameter types set up by CALL.

Note, the values of variables are not guaranteed to be stored at word-aligned addresses.

In the case of parameter types 4, 5 and 129 their address points to the variable value. In other cases the address points to another information block about the corresponding variable.

For BASIC string variables, type 128, the address in the parameter block points to a string information block (SIB). This block is guaranteed to exist at a word-aligned address and has the following format:

Bytes zero to four	Pointer to the characters contained in the string
Byte five	Current number of characters in the string

Variables of type 256+ refer to arrays and require a more complicated system and will not be described here. The section of the Archimedes User Guide which deals with CALL contains full details.

An example should help to clarify the layout of the parameter block. Consider the following statements:

```
table% = &1234
B$ = "Dabs Press"
freddy = 69
CALL code,B$,table%,freddy
```

The 'code' routine will be entered with the following set up:

```
R9  = address of parameter block
R10 = 3 (as three parameters are being passed)
```

The corresponding parameter block created at the address in R9, is shown in figure 13.6 below:

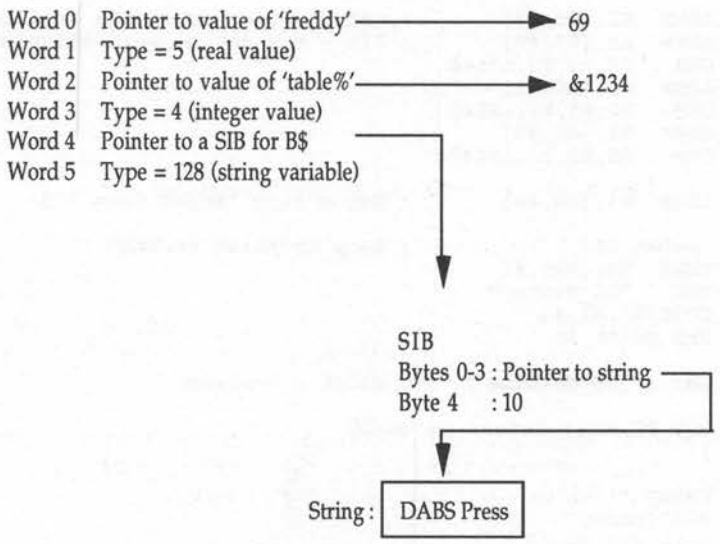


Figure 13.6. Example of parameter block set up by CALL

A simple example of passing string variables to machine code is given in listing 13.5. It prints the string passed to it from BASIC. Note that the method of accessing the word in the SIB (which contains the string contents address) is more complicated than might be expected. We can't load the word in a single instruction, however, as it is not guaranteed to be on a word boundary.

Listing 13.5. Passing strings to machine code.

```
 10 REM Passing strings using the CALL statement
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM string 256
 70 P%=string
 80 [
 90 ; On entry CALL has set up a parameter block
100 ; and made register R9 point to it
110
120 LDR   R0,[R9]          ; Get Addr of SIB from Param block
130
140 LDRB  R2,[R0,#3]       ; Get the address of the string from
150 LDRB  R1,[R0,#2]       ; SIB - may not be word-aligned!
160 ORR   R2,R1,R2,LSL#8
170 LDRB  R1,[R0,#1]
180 ORR   R2,R1,R2,LSL#8
190 LDRB  R1,[R0,#0]
200 ORR   R2,R1,R2,LSL#8
210
220 LDRB  R1,[R0,#4]       ; Get string length from SIB
230
240 .print_it              ; Loop to print string
250 LDRB  R0,[R2],#1
260 SWI   "OS_WriteC"
270 SUBS R1,R1,#1
280 BNE print_it
290
300 SWI   "OS_NewLine"     ; Print a newline
310
320 MOV PC,R14 ; Return to BASIC
330 ]
340
350 PRINT '''
360 A$="freddy"
370 CALL string,A$
380 mac$ = "Archimedes RISC Machine"
390 CALL string,mac$
400 b$="this is a complicated way to print strings!!"
410 CALL string,b$
```

The Operating System from BASIC

As well as accessing our own machine code routines from BASIC, we will frequently need to make use of the various operating system routines. BASIC makes provision for this using the SYS statement. SYS gives access to

the full range of operating system routines. These routines are usually accessed by a SWI instruction in machine code. The syntax of the SYS command is as follows:

```
SYS <routine>,<expression list> TO <variable list>;<flags>
```

'Routine' identifies which operating system routine we require. As with SWIs, this may be given as the routine's number, or as a string containing its name.

'Expression list' is a list of up to eight expressions or variables which are used to pass information to the calling routine via the processor registers. When evaluated, they must yield either a number or a string. If a number is used, it is converted into an integer and stored in the appropriate ARM register R0 to R7. If a string is produced, a copy of it is placed on BASIC's stack, and the corresponding register is set up as a pointer to it.

Variable list is a list of variables used to receive data from the called routine. Again a list of up to eight variables can be given. Numeric variables simply receive the value of the corresponding processor register R0 to R7. If a string variable is given then the corresponding register is assumed to point to a string in memory, terminated by ASCII 0, 10 or 13. This string is then copied into the given string variable.

The final part of the SYS statement is a variable which will have the contents of the status register copied into it on exit from the routine.

All parts of the SYS statement (except for routine) are optional and any combination of the various parts is acceptable. An example of a SYS statement will be given shortly. It is slightly artificial because it doesn't correspond to any real operating system routine. However, it does show examples of all the various possibilities within a SYS command:

```
SYS n,1,freddy,A$,,3*G TO size,,name$,file%,vec(1);status
```

CALL the operating system command 'n'

On entry:

Register R0 = 1
Register R1 = Value of 'freddy'
Register R2 = Pointer to a copy of A$

Register R3 = Undefined
Register R4 = 3* value of G

On exit:

size = Contents of register R0
names$= Copy of string pointed to by R2
file% = Contents of register R3
vec(1)= Contents of register R4
status= The processor status register

The SYS command allows us to access any operating system routine from BASIC. Some examples are:

SYS "OS_Byte",0,0,0	Perform *FX 0,0,0
SYS "OS_CLI","help"	Perform *HELP
SYS "OS_RemoveCursors"	Turn cursors off
SYS "OS_ReadC" TO char	Read a character into char

14 : Techniques & Debugging 

As it stands, Archimedes assembler is relatively simple when compared to other dedicated assembler systems. However, because of the way that it is integrated into BASIC, we can use the power of BASIC to enhance the assembler's facilities. In this section, two of these enhancements are described:

1) Macro assembly
2) Conditional assembly

In addition we will also examine the assembler's in-built debugger.

Macro Assembly

An assembler macro is a section of assembler code which has been given an identifying name. When the name is quoted in the main assembler program, the assembler will locate the corresponding macro and assemble the instructions associated with it. After the macro has been assembled, the main program continues.

As an example, we could write a piece of code which makes a 'beep' sound. This could then be defined as a macro, having the name 'beep'. Whenever we need a 'beep' in the main program, we simply quote the name of the macro, and assembler will find and assemble the required instruction.

It is important not to confuse macros with subroutines. Subroutines are triggered at execution time, whereas macros are processed at assembly time. In the previous example, we do not create a 'beep' subroutine which is jumped to from the main program when needed. Instead, when we quote the macro, the assembler assembles the required instructions at the current place in the program. This happens each time the macro is used. Thus, the instructions associated with the macro are repeated in the main program wherever the macro name was used.

The Archimedes BASIC assembler does not provide macros directly. However, we can implement a macro system quite easily using BASIC. This is

possible because of the way in which we are allowed to call BASIC functions from within assembler. We have seen this before with functions such as ASC() from assembler code:

```
MOV R0,#ASC("A")
```

As well as being allowed to use the pre-defined BASIC functions, we can also call user-defined functions (FNs). Thus, we can write statements like the following:

```
[
MUL R0,R1,R2
FNfreddy
ADD R0,R2,R3
]
```

Assembler will assemble the first instruction. It will then try to evaluate the function on the second line and, in doing so, will call FNfreddy, which is assumed to be defined elsewhere in the program.

When we are inside the function, we can use any BASIC statement including re-entering the assembler and assembling other instructions.

We are now in a position to try to implement a real macro. Let's try the 'beep' example. Type in listing 14.1. The macro is defined as a function at the end of the program, and is used several times throughout it. When the macro is called, assembler is re-entered and the instructions to make a 'beep' are assembled.

Listing 14.1. The 'Beep' macro.

```
 10 REM Macro calls in the assembler
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM macro 1024
 70
 80 FOR pass = 0 TO 3 STEP 3
 90 P% = macro
100 [
110 OPT pass
120
130 ADR R0,message1     ; Get address of first string into R0
140 SWI "OS_Write0"     ; Print string out
150 SWI "OS_NewLine"    ; Print a Newline
160
```

```
170 FNbeep              ; Call 'Beep' macro
180
190 ADR R0,message2     ; Get address of second string into R0
200 SWI "OS_Write0"     ; Print string out
210 SWI "OS_NewLine"    ; Print a Newline
220 SWI "OS_ReadC"      ; Wait for a key to be pressed
230
240 FNbeep              ; Call 'Beep' macro again
250
260 MOV PC,R14          ; Return to BASIC
270
280 ; Define the strings to be printed
290 .message1
300 EQUS "This is an example of a macro"
310 EQUB  0
320 .message2
330 EQUS "That beep was assembled from a MACRO !!"
340 EQUW  &0A0D
350 EQUS "Press a key to make another beep using macros"
360 EQUB  0
370 ]
380 NEXT
390 PRINT'''
391 CALL macro
392 END
400
410
420 REM Macro function which assembles the 'Beep' instructions
430 DEF FNbeep
440
450 LOCAL vdu,bell
460
470 vdu  = 256      : REM number of SWI block to perform VDU n
480 bell = 7        : REM Bell Character (VDU 7)
490
500 IF pass = 3  PRINT '"Expanding 'Beep' macro"
510
520 REM Re-enter the assembler to produce the Beep code
530
540 [ OPT pass
550 SWI vdu + bell
560 ]
570
580 PRINT
590 =0              : REM Return to main program
```

If you look at the assembler listing, you will see that the instruction in the 'beep' macro was assembled as if it was part of the main program.

This example may seem a little like using a sledgehammer to crack a nut! However, if the code contained in the macro is needed a lot, it saves us

from having to enter it each time. We can simply quote the macro name and leave the rest to the assembler. Also, variables can be passed to the function providing parameters for the macro. These could then vary the code which is assembled in the macro.

Conditional Assembly

Conditional assembly is a technique in which only parts of an assembler source program are assembled into machine code. Complete sections of code can be included, or omitted, from the final machine code program depending on the state of program variables. This may be useful for producing various versions of a program tailored to particular specific machine or user requirements.

The close relationship between BASIC and assembler on the Archimedes makes conditional assembly extremely easy to achieve. In the following example, the section of code is only assembled if the value of the 'printer' is true:

```
IF printer THEN
[
      SWI       "OS_WriteS"
      EQUS      "Output to printer?"
      EQUB      0
      SWI       "OS_ReadC"
      CMP       R0,#ASC"y"
      SWIEQ     256+2
]
ENDIF
```

The section of code asks if the output should be sent to a printer and, if so, enables it. If you don't have a printer, however, then a version of the machine code program could be produced with this entire section missed. This would simply require the 'printer' variable to be set to false.

Listing 14.2 gives a full example of conditional assembly. It is really a demonstration of the operation of the ARM's shift facilities previously described in Chapter Seven. The program asks the user which shift instruction is to be studied and then assembles appropriate instructions to do this using conditional assembly.

Listing 14.2. Conditional assembly – a demonstration.

```
 10 REM Combined demonstration - shifts and conditional assembly
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 REM Display menu of available shift operations
 70 MODE 15
 80 FOR shifts = 1 TO 6
 90 READ shift_name$
100 PRINT ;shifts;") ";shift_name$
110 NEXT
120
130 REM Ask user to select one type of shift
140 REPEAT
150 INPUT ' "Which shift is to be studied :",choice
160 UNTIL choice >0 AND choice <7
170
180 DIM demo_prog 512
190 FOR pass = 0 TO 3 STEP 3
200 P% = demo_prog
210
220 [
230 OPT pass
240 ; Routine is eneter with the number to be shifted
250 ; in R0. This is passed from A% when the routine
260 ; is called.
270
280 MOV R10,R14         ; Preserve R14 in R10
290 SWI "OS_WriteS"     ; Write string
300 EQUS "Before : %"   ; String to be output
310 EQUB 0              ; String terminator
320 BL print_binary     ; Call subroutine, print R0 in binary
330  SWI "OS_NewLine"   ; Output a New line
340
350 ]                   : REM Leave the assembly temporarily
360
370 REM Assemble appropriate shift instruction
380 IF choice = 1 THEN [ OPT pass : MOVS R1,R1,LSL R2 : ]
390 IF choice = 2 THEN [ OPT pass : MOVS R1,R1,ASL R2 : ]
400 IF choice = 3 THEN [ OPT pass : MOVS R1,R1,LSR R2 : ]
410 IF choice = 4 THEN [ OPT pass : MOVS R1,R1,ASR R2 : ]
420 IF choice = 5 THEN [ OPT pass : MOVS R1,R1,ROR R2 : ]
430 IF choice = 6 THEN [ OPT pass : MOVS R1,R1,RRX    : ]
440
450 REM Re-enter the assembler again
460 [
470 OPT pass
480
490 SWI "OS_WriteS"     ; Write string
```

```
500 EQUS "After  : %"  ; String to be output
510 EQUB 0             ; String terminator
520 BL print_binary    ; Print contents of R0 in binary
530 MOV PC,R10         ; Return to BASIC (addr preserved in R10)
540
550 .print_binary      ; Binary print subroutine
560 MOV R0,#0          ; explained in Chapter Eight
570 ADC R0,R0,#48
580 MOV R4,#1 << 31
590
600 .bits
610 TST R1,R4
620 SWIEQ 256+ASC"0"
630 SWINE 256+ASC"1"
640 MOVS R4,R4,LSR#1
650 BNE bits
660 SWI "OS_WriteS"
670 EQUS "  Carry = "
680 EQUB 0
690 SWI "OS_WriteC"
700 SWI "OS_NewLine"
710
720 MOV PC,14
730
740 ]
750 NEXT pass
760
770 PRINT '''
780 INPUT "Number to be shifted" , B$
790 B%=EVAL(B$)   : REM Number may be in hex, binary or decimal
800 REPEAT
810 PRINT
820 INPUT "Shift by how many places " , C%
830 CALL demo_prog
840 UNTIL FALSE
850
860 REM Names of the shift operations
870
880 DATA Logical shift left        (LSL)
890 DATA Arithmetic shift left     (ASL)
900 DATA Logical shift right       (LSR)
910 DATA Arithmetic shift right    (ASR)
920 DATA Rotate right              (ROR)
930 DATA Rotate right with extend (RRX)
```

Mixing Macros and Conditional Assembly

We can create programs which use both macro and conditional assembly techniques to achieve some very powerful results. We could, for example, write a macro called 'debug' which, when called, assembles instructions in

a program to output the contents of all the processor registers to the computer screen.

Within the macro function, conditional assembly could be used to test the value of a flag variable called 'debug'. The register outputting instructions would only be assembled if this flag was true. This allows us to include the debugging macro at key points in our program to help track down errors. When the program is working, we can set the 'debug' flag to false, then assemble the source, automatically omitting the debugging instructions.

Debugging Machine Code Programs

When we have written our machine code program, the chances are that it will not work! So what do we do next? The problem with machine code is that its low-level nature makes it very difficult to spot errors. We are usually faced with a large collection of machine code instructions and the only thing we know for certain is that they don't do what they should!

A useful technique, used in debugging, is to include instructions in the program which print messages on the screen. These could be simple diagnostic strings which let you know which part of a program is executing and what is going on. Also, the contents of key memory locations, or registers, could be output to aid in debugging. As stated, a program to help with this is included on the disc accompanying the book.

When debugging any machine code programs, the most important rule is:

'Before executing any machine code,
save the assembler source program'

Machine code programs do not observe any of the nicities of BASIC when they execute. In the event of an error, they will be more than happy to scribble all over the BASIC program containing the assembler source instructions. If this happens, then there is little that can be done other than starting again and re-writing the program.

The Debugger

To make the process of finding and correcting errors less painful, the designers of the Archimedes include a machine code debugging system within the operating system. This provides help in tracking down exactly what

your program is doing as it executes. We can then see why its behaviour is not what was expected and correct it.

The commands provided by the debugger are as follows:

*DEBUG	*QUIT
*BREAKSET	*BREAKCLR
*BREAKLIST	*CONTINUE
*MEMORY	*MEMORYI
*MEMORYA	*SHOWREGS
*INITSTORE	

Using the Debugger (*DEBUG and *QUIT)

All of the commands supported by the debugger can be used, like any other star command, from most applications. However, we can explicitly enter a 'debugging environment' from which the commands can be issued. This is done by typing '*DEBUG'. When in this environment, the prompt changes to:

```
DEBUG*
```

Commands can then be entered as required without the need to prefix them with the usual star.

To leave the debugging environment simply type 'QUIT'.

Breakpoints

The major debugging facility provided on the Archimedes is a system of breakpoints. A breakpoint is a trap which is set at a given instruction in a machine code program. When the ARM attempts to execute the instruction, the breakpoint is triggered, halting the program at the given point. The debugger is automatically re-entered and we can examine the register or memory contents and set further breakpoints. After this, we can resume the execution of the program from the instruction immediately following the breakpoint.

*BREAKSET <address>

This will set a breakpoint at a specified address. This will be triggered when the ARM attempts to execute the instruction at the given location. Since all ARM instructions are word-aligned, the address of the breakpoint should be word-aligned too.

When the breakpoint is triggered, the debugger will be re-entered and the status of the processor (including the contents of all registers) will be displayed on screen.

Note that the breakpoint should be set after the machine code program has been assembled. The breakpoint will be overwritten if the program is re-assembled and will therefore need to be set each time this occurs.

*BREAKLIST

BREAKLIST produces a list of addresses where the breakpoints are set.

*BREAKCLR {<address>}

This command will remove the breakpoint previously set at the specified address. The address is optional and, if omitted, will cause all breakpoints to be cleared. Confirmation is asked for before clearing the breakpoints.

*CONTINUE

CONTINUE can be used to resume execution of a program previously halted by a breakpoint trap.

Examining Memory and Registers

The next set of commands are used to examine, and in some cases modify, the contents of both memory and the processor registers. When memory addresses are given as arguments to the commands, they have the following form:

```
<addr/reg>
```

This means that either an immediate hexadecimal address can be quoted or, alternatively, the name of a processor register may be given. In the

latter case, the address used by the instruction will be that contained in the specified register.

When a range of memory is being specified, two such addresses will need to be given. The first address is the start of the range, the second is the end. An extra option is to prefix the second argument by a '+' character. This will be taken as the size of the range starting at the first address.

*MEMORY {B} <addr1/reg1> {<addr2/reg2>}

This command displays the specified range of memory. If the B option is used, then data will be displayed as bytes, otherwise it will displayed as words. All data is shown as hexadecimal quantities.

*MEMORYI <addr1/reg1> {<addr2/reg2>}

This is equivalent to the *MEMORY command, except that memory words are interpreted and displayed as ARM instructions. Thus, a disassembly of a machine code program can be produced.

Note, the debugger's disassembler knows about more than just the normal ARM instructions. In addition, it will disassemble co-processor instructions and floating point instructions. This means that when disassembling some chunks of memory, you may come across some very strange instructions which you will not recognise. These are not, however, anything to do with the ARM's instruction set.

*MEMORYA {B} <addr/reg1> {<data/reg2>}

This command is used to alter the contents of memory. If the optional B suffix is present, memory will be displayed and altered in bytes, otherwise words of memory will be used.

The address of the location to be altered is given by <addr/reg1>. If argument two is omitted, the debugger will then allow memory editing in interactive mode, starting from the specified address. In this mode the following may be entered:

Hexadecimal number
: Change the byte or word at the current address to the specified value. As confirmation the new value will be displayed.

RETURN Move to the next byte/word in the appropriate direction. By default the direction is to move forward in memory this can be changed, however.

+ Set 'direction' so that pressing RETURN moves onto the next byte/word in memory.

– Set 'direction' so that pressing RETURN moves back to the previous byte/word in memory.

<Anything else>
Quit the command.

As an alternative to interactive mode, a single byte/word of memory can be set to a given value by quoting the value as argument two in the command.

Note, a flexible memory editor is included on the accompanying disc for this book – details in Appendix J.

*INITSTORE {<data/reg1>}

This allows the contents of all the user memory to be initialised to contain the given data. The data is assumed to be a four-byte word which will be replicated throughout user memory. If the command is issued without the argument, then memory will be filled with &E1000090, which is the ARM's representation of an undefined instruction.

*SHOWREG

This command shows the processor status, including the contents of all the registers, when the last trap occurred. In most practical cases, this will be when the last breakpoint was triggered. The contents of the registers are given in hexadecimal.

15 : Interrupts and Events

The concept of an interrupt is a simple one, yet it is of fundamental importance to most computer systems. Interrupts are used to allow the various hardware devices and peripherals connected to the system, to gain the attention of the CPU when they require servicing.

For example, the keyboard will generate an interrupt whenever a key is pressed to it. This is a signal to the central processing unit that the keyboard matrix should be scanned, and the ASCII value of the key pressed entered into the keyboard buffer.

A useful analogy to draw when describing interrupts, is that of an office worker filling in forms when the telephone rings. Normally, the worker can get on with the main task of processing the forms. However, when the telephone rings, the person immediately breaks off the previous task and concentrates on answering the phone. When the call has been dealt with, the worker goes back to previous task at exactly the point at which they broke off. The work then continues as if the interruption never occurred.

This corresponds very closely with the interrupt system on a computer. Normally, the CPU gets on with the task of executing programs. However, if a peripheral device finds that it needs the services of the CPU for a particular reason, then it will cause an interrupt. The CPU, after completing its current instruction, will then break off its normal work and jump to a special routine in the operating system.

This routine, called the first level interrupt handler (FLIH), is responsible for finding the device which caused the interrupt and jumping to an appropriate routine to service it. The specific routine called by the FLIH, carries out the task requested by the interrupt and resets the device so that it does not continue signalling the same interrupt condition. The CPU then returns to the interrupted task and resumes it as if nothing had happened.

Without an effective interrupt system, the CPU would have to spend large amounts of its time checking around the various components of the system to see if any require servicing. For the majority of the time none would

require attention, and the time spent checking would therefore be wasted. Obviously, the effect of this would be a much slower and less powerful system for the user.

Interrupts on the Archimedes

The ARM supports two types of interrupts called:

Interrupt Requests	(IRQs)
Fast Interrupt Requests	(FIRQs)

Each of these has a separate interrupt control line entering the ARM chip. Connected to these lines are the devices which are potential sources of interrupts. Some of these devices are:

The disc interface
The keyboard interface
The video system
The Econet system (if fitted)
The serial interface
Various internal timers

The devices connected to the FIRQ line are defined as being high-priority systems like the disc interface. High-priority in this context means that when interrupts occur from these devices, it is vital they are serviced as quickly as possible. They therefore take precedence over all other activities.

The IRQ line, by comparison, is connected to devices for which a slight delay in servicing their interrupts is not so important. Thus, although IRQs are serviced very quickly by interrupting normal processing tasks, their service routines can themselves be interrupted by an FIRQ. The only time at which an FIRQ will not be serviced immediately, is when the ARM is already processing a previous FIRQ signal.

Disabling Interrupts

An exception to the above scheme comes when the user, or the operating system, disables one (or both) of the interrupt systems. This will normally be done when a particularly important piece of code is being run which must be allowed completed without interruption. For example, when

manipulating the control registers on a hardware device it is not always a good idea to let the device interrupt!

The ARM maintains two flags in its status register to control the action of the interrupts. These are a bits 26 and 27. If the IRQ bit is set in the register, then no IRQs will be allowed to interrupt the ARM. Similarly, if the FIRQ bit is set, then no FIRQs will be serviced.

When operating in user mode, programs are prevented from modifying these flags. However, in supervisor mode, setting or clearing the relevant bits of R15 will enable or disable the corresponding interrupt system.

The operating system also provides us with two SWI calls to control the interrupt system from user mode. These are:

```
SWI  "OS_IntOff"
SWI  "OS_IntOn"
```

If IntOff is used, then all interrupts, both IRQ and FIRQ, are disabled. IntOn must then be used to re-enable the interrupt system.

Interrupt Processing

When an interrupt occurs, the ARM always responds in the same uniform way. This is summarised as follows:

1) Finish executing the current ARM instruction

2) Switch to the appropriate ARM processor mode (either IRQ mode or FIRQ mode)

3) Move R15 into R14

4) Disable further interrupts. If IRQ, then only disable further IRQs, otherwise disable both IRQs and FIRQs

5) Jump to the appropriate interrupt vector (either IRQ_vec or FIRQ_vec)

The ARM automatically switches into the appropriate mode to process the interrupt. We have seen in Chapter Three that this causes some of the normal processor registers to be replaced by special private ones. Thus, when

the program counter and status flags are copied from R15 into R14, the normal user mode version of R14 is not used.

Instead, R15 will be stored in the appropriate private register for the mode, ie, either R14_IRQ or R14_FIRQ. The reason for taking a copy of R15 is to allow the interrupted program to be returned to after the service routine has completed.

The other private registers, which appear in interrupt modes, allow the interrupt routines to use some registers without having to explicitly save the contents of registers also used by other modes. In FIRQ mode there are seven private registers available. This is because FIRQ routines must execute very quickly, so we do not want the overhead of saving the previous contents of any registers used.

When the appropriate mode has been entered, the interrupts are disabled. This is to prevent subsequent interrupts from interrupting the service routine before the original interrupt has been dealt with. Note, however, that an IRQ service routine is allowed to be interrupted by an FIRQ.

The final action the ARM performs is to jump to the appropriate interrupt vector, either IRQ_vec or FIRQ_vec. The concept of vectors is covered in the next chapter. Basically, the action is to divert control to a constant known point (the vector). From here we then jump to the specific interrupt handling routine.

Returning From Interrupts

When the interrupt service routine has been performed, the operating system must return to the original program which was interrupted. This is done quite simply by using the following instruction:

```
SUBS R15,R14,#4
```

This restores the program counter so that the interrupted program can be resumed from exactly the point at which it was suspended. The 'subtract 4' calculation is required to correct for the effects of pipelining.

Providing that the interrupt handling routine has not corrupted any shared registers or workspace, the program will continue executing as if the interrupt had never happened. On the Archimedes, interrupts are occurring and being serviced continually without the user even realising it.

Writing Interrupt Routines

Normally, we don't need to concern ourselves with writing interrupt handlers. The operating system has routines which automatically take care of most internal interrupts.

The operating system also provides support for linking user routines to important system's events. This allows us to write code which will be called whenever the appropriate event occurs, without intercepting the interrupt system. (See Events and Vectors.)

Sometimes, however, we may need to write direct interrupt handling routines. Perhaps we need to intercept interrupts to gain priority over the operating system's handlers, or to handle interrupts from a new piece of hardware. In these cases, we must observe the following rules when writing an interrupt handling routine:

1) Do not re-enable interrupts in the handling routine. If this is done, a second IRQ/FIRQ could interrupt the processor before it has finished handling the first. In some cases this may be permissible, but it requires great care and should be avoided if at all possible.

2) The interrupt routine should terminate very quickly. If it keeps interrupts disabled for too long, then the normal Archimedes background activities will grind to a halt. The keyboard will lock, various software clocks will lose time, the sound system will cease to operate and the video system's flashing colours and mouse pointer will freeze.

3) All shared processor registers should be the same on exit from the interrupt routine as they were on entry. This is absolutely vital if the interrupted task is to be resumed correctly.

4) The interrupt handling routine should avoid calling operating system routines. It is possible that one of these routines was only half executed when it was interrupted by IRQ/FIRQ. If re-entered in the interrupt routine, workspace could be disturbed causing the routine to corrupt when resumed. Only operating system routines which do not suffer from this problem, (re-entrant routines) can be used in interrupt handling routines.

Events

As the computer operates, situations will frequently crop up which we would like to know about and act on, eg, when a character enters a buffer.

These situations are called events, and the operating system can be made to execute a user-supplied routine whenever an event occurs.

A full list of the events recognised by the system are given in figure 15.1. Initially, all events are disabled. However, this can be changed using a pair of *FX commands or their machine code OSBYTE equivalents:

*FX 13, <n>	Disable event<n>
*FX 14, <n>	Enable event <n>

Whenever an enabled event occurs, the operating system will call the event vector (number &10). A user-routine must be linked to this vector to handle the event. (See the next chapter for details about how to do this.) The handling routine is entered with the event number in register R0. This allows different events to be differentiated by the single event handler.

Event number	Cause of the event	Event entry information (R0 = event number)
0	An output buffer has become empty	R1 = Buffer number
1	Input buffer already full	R1 = Buffer number R2 = Character which couldn't be inserted
2	A character has been placed in an input buffer	R2 = ASCII value of new character
4	Vsync: scanning beam has reached bottom of screen	–
5	Interval timer crossed zero	–
6	Escape condition detected	–
7	RS423 receiving error	R1 = Serial device status R2 = Character received
8	Event generated by Econet	–
9	Event generated by the user	–
10	Mouse button has changed state	R1 = Mouse X co-ordinate R2 = Mouse Y co-ordinate

		R3 = Mouse button state R4 = Lower four bytes of real time centi-second value
11	Key pressed/Released event	R1 = zero if key pressed R1 = one if key released R2 = Key matrix number

Figure 15.1. Operating system events.

When writing event-triggered routines, the same rules should be observed as those used when writing interrupt routines.

16 : Vectors

Vectors are used to couple together a program which requires access to a routine and the routine itself. When using a vector system, a program does not call the required routine directly. All access is made through the vector. It is the vector which contains the address of the corresponding routine. For this reason, we therefore say that the vector provides indirect access to the routine.

Vectors are useful for two reasons. First, they allow programs to access standard routines without referencing their actual start address in memory. Thus, later on, if a routine needs to be moved to a different location, all we have to do is modify the address stored in the vector. All the existing software will still work and will not have to be individually modified.

The second advantage in using vectors, is that they can be intercepted. This means that the normal address which they contain is replaced by the address of a user-routine. Whenever any other program accesses services through the vector, the user routine will be executed instead of the normal one.

This is especially useful as most operating system tasks, eg, printing characters, disc access, error control, are all vectored. Thus, they can be intercepted to modify the system's behaviour in any way required.

ARM Hardware Vectors

The ARM processor itself makes use of certain vectors to deal with abnormal events which it cannot itself cope with. These are called the exception vectors and are located at the very beginning of memory at addresses &0000000 to &000001C. A list of the exception vectors is given in figure 16.1 which can be found on the next page.

&0000000	ARM reset
&0000004	Undefined instruction
&0000008	Software interrupt (SWI)
&000000C	Abort (pre-fetch)
&0000010	Abort (data)
&0000014	Address exception
&0000018	IRQ_Vec
&000001C	FIRQ_Vec

Figure 16.1. The ARM's exception vectors.

These vectors are different to the normal 'software vectors' because the ARM jumps to them directly if a specific event occurs in its internal hardware. For example, if an instruction causes the ARM to attempt to access non-existent memory then an address exception error will occur. In response to this, the ARM will stop executing the program and jump to location &0000014 – the address exception vector.

Each vector contains a branch instruction which causes the processor to jump again, this time to a suitable operating system routine to handle the exception.

Software Vectors

The operating system provides a whole series of vectors to provide access to its internal routines. These are listed in figure 16.2 and are *always* used whenever an operating system routine is called. Thus, when we use an SWI "OS_WriteC" instruction, the operating system accesses the write character routine via an internal vector.

If we intercepted this vector, then our routine would be entered each time a character is printed. When this happens, the processor registers will typically contain some relevant information. In this example, register R0 would contain the ASCII value of the character which is to be printed.

Sometimes, we might need the intercepting routine to completely replace the normal operating system one. More often, however, it will perform some function, then pass control back to the normal routine. The control system for vectors on the Archimedes provides support to allow either of these things to be done.

(&01)	ErrorV	(&0E)	ReadlineV
(&02)	IrqV	(&0F)	FSControl
(&03)	WrchV	(&10)	EventV
(&04)	RdchV	(&14)	INSV
(&05)	CliV	(&15)	REMV
(&06)	ByteV	(&16)	CNPV
(&07)	WordV	(&17)	UKVDU23V
(&08)	FileV	(&18)	UKSWIV
(&09)	ArgsV	(&19)	UKPLOTV
(&0A)	BGetV	(&1A)	MouseV
(&0B)	BPutV	(&1B)	VDUXV
(&0C)	GBPBV	(&1C)	TickerV
(&0D)	FindV	(&1D)	UpcallV

Figure 16.2. The operating system software vectors.

Intercepting Vectors

Before looking in detail at the various software vectors supported by the operating system, we shall examine how such vectors can be intercepted.

Each vector has a list of routines associated with it which wish to be called when the vector is used. Initially, each list only contains the default operating system routine. However, we can add our own routines to any list required. The operating system calls the routines in the vector's list in a reverse order. Thus, our added routine will be called before the operating system's default routine.

When our routine has been entered, we can perform any processing required. We can then either allow the next routine in the list to be called, or we can abort the list. Routines can therefore be added to the default operating system, or can replace them completely.

Claiming Vectors

There are two SWI calls which are specially designed to help in the interception of the software vectors. The first of these is:

```
SWI "OS_Claim"
```

This instruction is used to claim one of the vectors. Registers R0 to R2 must be set up in the following way before the call is made:

R0 The number of the vector which is to be intercepted
R1 Address of the routine to be added to the vector's list
R2 A value which will be passed in R12 when the routine is called

The operating system will then add the corresponding routine's address to the list of other routines to be called when the vector is used.

Releasing Vectors

Routines can be removed from a vector's call list by using:

```
SWI "OS_Release"
```

with the following registers set up:

R0 The number of the vector previously intercepted
R1 The address of the routine to be removed from the list
R2 The same value given in R2 when the vector was claimed

The operating system will then remove the specified routine from the vector's call list. The routine will no longer be entered when the vector is called. Other routines in the list will be unaffected.

Writing Vector Intercept Routines

There are some very important points to note when writing routines to intercept vectors:

1) The state of all registers must be same on exit as they were on entry to the routine. The exception to this is when a routine associated with a vector is expected to return some results in one of the registers.

2) Registers may be preserved on an internal stack while they are used. To push registers onto the stack, use:

```
STMFD R13!,{Register_list}
```

To pull the values from the stack, use:

```
LDMFD R13!,{Register_list}
```

3) An intercepting routine will be entered in supervisor IRQ or FIRQ mode depending on the type of vector.

4) When exiting from an intercept routine use:

```
MOV R15,R14
```

This will cause the operating system to enter the next routine in the vector's call list, or return if the end of the list is reached. This exit method should always be used if the default routine is to be entered after the intercepting one.

5) If an ABORT list exit is required then use:

```
LDMFD R13!,{R15}
```

This will ensure that the call is not passed on down the vector list. The normal use for this routine is when the user's intercepting routine completely replaces the default operating system one.

The Operating System Vectors

The following section contains information on the function and use of each of the operating system vectors.

For each vector, a series of entry conditions will be given. These define the contents of various registers on entry to the routine linked to the vector.

Sometimes, an exit condition is also given. This defines the state of the registers which would exist after the normal operating system routine had been called through the vector. For example, the insert character into buffer routine, which is called through INSV, returns with the carry flag set if buffer insertion failed. Some of the applications which call the routine will act upon these returned results. Any intercepting routine must, therefore, place appropriate values in the registers on exit.

Main Line System Vectors

Figure 16.3 shows the default operating system routines for this vector group. A vector is called whenever the corresponding routine is used. All access to the routines is directed through the appropriate vector.

The entry and exit conditions for these vectors are, in every case, exactly the same as those for the default routines. These vectors will not, therefore, be described any further. Full details of the entry/exit conditions can be found under the appropriate routines, which are described in the Advanced User Guide.

Vector number	Vector name	Default routine called
&05	CliV	OS_CLI
&06	ByteV	OS_BYTE
&07	WordV	OS_WORD
&08	FileV	OS_FILE
&09	ArgsV	OS_ARGS
&0A	BGetV	OS_BGET
&0B	BPutV	OS_BPUT
&0C	GBPBV	OS_GBPB
&0D	FindV	OS_FIND
&0F	FSCV	OS_FSC

Figure 16.3. The mainstream vectors.

(&01) ErrorV: Error Vector

On entry: R0=Pointer to error block

On exit: No information returned

This vector is called every time an error occurs on the Archimedes. Normally, it links to the error-handling routine reporting the error, or taking appropriate action. It may be intercepted to give user routines a warning of an impending error. However, it must pass the call on, so that the error handler is called to deal with the error.

(&02) IrqV: Interrupt Request Vector

On entry: No information passed

On exit: No information returned

This vector will be called in response to the ARM detecting an interrupt. It is the software entry point to the first-level interrupt handler. The default routine will discover the source of the interrupt and, if possible, call an appropriate handling routine.

If this vector is intercepted, then the user must be responsible for handling interrupts from every possible device. Alternatively, the user-routine will perform any processing necessary, then hand control back to the normal routine. In this way, the user can add to the interrupt system without having to replace it.

If non-standard interrupts are being used, it is essential that a handling routine is attached to this vector.

(&03) WrchV: Write Character Vector

On entry: R0 = ASCII code of character to be written

On exit: No information returned

Whenever a character is printed, this vector is used.

(&04) RdchV: Read Character Vector

On entry: No information passed

On exit: R0 = ASCII code of the character read

All calls to the operating systems read character routine are passed through this vector. It is assumed that the routine called by the vector will obtain the character, and return it in register R0.

(&0E) ReadLineV: Read a Line of Text Vector

On entry: R0 = Pointer to the text buffer
R1 = Maximum size of line
R2 = Lowest permissible ASCII character
R3 = Highest permissible ASCII character

On exit: Carry Set = Escape terminated entry
R1 = Length of buffer

All calls to the operating system's OS_ReadLine routine are directed through this vector.

(&10) EventV: Event Vector

On entry: R0 = Number of event
R1–R4 = Depend on event

On exit: No information returned

Events are covered in Chapter 15. Whenever an event occurs, this vector is called. Events are provided exclusively for the user. If any events are enabled, therefore, this vector must be linked to a suitable service routine.

(&14) INSV: Insert Character into Buffer Vector

On entry: R0 = Character to be inserted
R1 = Buffer number

On exit: R2 is undefined
Carry set = Insertion failed

Whenever a character is inserted into a system buffer, this vector is called.

(&15) REMV: Remove Character From Buffer Vector

On entry: R1 = Buffer number
Overflow set = Buffer to be examined only
Overflow clear = Character is to be removed

On exit: R0 = Next character to be removed (for examine buffer)
R2 = Character actually removed (for remove from buffer)
Carry set = Buffer was empty

Whenever a character is removed from a system buffer, this vector is the one called.

(&16) CNPV: Count/Purge Buffer Vector

On entry: R1 = Buffer number
Overflow set = Purge all characters from buffer
Overflow clear = Count characters in the buffer
Carry set = Return number of buffer entries, if counting
Carry clear = Return number of buffer spaces, if counting

On exit: R1 is undefined
R1 = Number of spaces/entries, if counting

(&17) UKVDU23V: Unknown VDU 23 Vector

On entry: R0 = VDU 23 number

On exit: R1 = Pointer to the VDU queue

This vector is called in response to an unrecognised 'VDU 23,n' command. The VDU command will be unrecognised if the value of 'n' is in the range 18 to 24 or 28 to 31.

This vector provides us with a very easy way to add new VDU23,n commands to the system.

(&18) UKSWIV: Unknown SWI Vector

On entry: R0 = SWI number

On exit: No information returned

This vector is called in response to an an SWI instruction being executed; the number of which is not known to the system. By trapping this vector, the user can easily add new SWI commands to those normally available.

(&19) UKVDU25V: Unknown PLOT Vector

On entry: R0 = PLOT number

On exit: No information returned

VDU25 is the graphics plot command. This is normally followed by a byte which defines the plot action to be taken; for example, plot a triangle. If the plot option used is unknown, however, then this vector is called.

Graphics applications could trap this vector to add new plot commands to the system.

(&1A) MouseV: Mouse Vector

On entry: No information passed

On exit: R0 = X position of mouse
R1 = Y position of mouse
R2 = Button status

The operating system directs all calls to OS_Mouse along this vector. The default routine investigates the state of the mouse, and returns information about it.

An alternative user-routine could intercept this vector and return similar information derived from another source. For example, if a joystick is added to the system, its position could be read by the intercepting routine and returned as mouse co-ordinates. Any application which uses the mouse would then work with a joystick.

(&1B) VDUXV: Special VDU Vector

On entry: VDU option requested

On exit: No information returned

Normally, VDU commands are sent directly to the VDU drivers. However, if bit five of the OSWRCH destination flag is set using *FX3, the screen VDU commands will be sent to this vector.

This provides a way of implementing a user-defined output stream. If bit five of the destination is set, then data usually sent to the screen, will pass to the new stream via the intercepting routine. This vector is used by the font manager.

(&1C) TickerV: 100 Hz Pacemaker Vector

On entry: No information passed

On exit: No information returned

The operating system calls this vector 100 times every second (once every centi-second). If intercepted, this vector can be used for a variety of time keeping functions.

(&1D) UpCallV: Warning Vector

On entry: No information passed

On exit: No information returned

This vector is called by the operating system when a filing system error has just occurred. When called, the error will not yet have been reported. It thus gives the application which issued the filing system command a chance to take corrective action.

For example, if a disc has been changed, then the application could prompt for the correct disc to be re-inserted, rather than reporting an error.

17 : OS SWI Routines

The ARM's SWI instruction was described in Chapter 11. In the following chapters, we will examine some of the most useful operating system routines which can be accessed using SWIs.

The SWI instruction is used to streamline and control access to operating system facilities. It removes the need to directly access routines, devices and workspace, thus making programs more independent of the operating system.

Many routines accessed using SWIs have a very similar interface to those provided on the earlier BBC micros and Master series machines. This gives the system a very familiar feel to people who have programmed in machine code on these machines.

A great many extra routines and functions exist on the Archimedes. Some control the extra Archimedes facilities, such as the mouse, stereo sound and enhanced graphics. Others provide services like character to number conversion and string input/output. These functions are often needed in programs, but were sadly missing from earlier machines.

The number and range of the SWI routines provided is vast, and we can't hope to cover them all. Instead only the most important ones were looked at. Several more SWI routines will be described, in other chapters. A complete list of the operating system SWIs is given in Appendix E. Full details of these can be found in the Advanced User Guide. The SWI routines covered here are collected into the following functionally related groups:

1) Input/output facilities
2) Conversion facilities
3) Systems functions
4) Controlling the WIMP environment
5) Managing the font system

For each SWI, its name, its number, and the entry/exit conditions are given together with a brief description of its purpose.

Input/Output Facilities

The following SWI calls are provided by the operating system to ease the problems of performing data input and output.

Character Input/Output

SWI "OS_WriteC" (&00)

On entry: R0 (low byte) = ASCII code of the character to be outputted

On exit: No information returned

This routine was called OSWRCH on the BBC micros and Masters. It performs the task of writing a single character, contained in the lower byte of register R0, into the output stream.

Any character can be written in this way. This allows control characters to be output to instruct the VDU drivers to perform special operations like graphics plotting.

OS_WriteC will be found in almost all programs which perform input/output. It has already been used in most of the example programs in this book.

SWI 256 + ASCII (&100-&1FF)

On entry: Nothing passed

On exit: No information returned

This is not a single SWI routine but a block of 256 routines. The routines do not have separate names, but are consecutively numbered starting at 256 continuing to 511.

We will often want to output a single fixed character, for example, CHR$(12) to clear the screen. It would be possible to load the appropriate ASCII code into register R0 and then call "OS_WriteC". However, to save us doing this each time, the operating system provides the block of 256 SWI calls.

Each of the SWIs in the block simply output one of the 256 characters in the ASCII set. For example, SWI number 256 outputs CHR$(0), number 257 outputs CHR$(1), and so on. To output character 'n', therefore, we simply use:

```
SWI 256+n
```

SWI 256+n is particularly useful as it doesn't corrupt any registers, and does not require any registers to be set up before calling it. It therefore provides an extremely easy way of outputting fixed individual characters.

SWI "OS_ReadC" (&04)

On entry: Nothing passed

On exit: R0 (low byte) = ASCII code of the character read.
Carry flag set if ESCAPE pressed

This routine was called OSRDCH on the earlier BBC and Master macros. It reads a single character from the input stream and places its ASCII code in the lower byte of register R0.

If the character entered is a 'special one', usually ESCAPE, then the carry flag is set to indicate this.

String Input/Output

A common requirement in many programs is to read or write a complete string of characters. The Archimedes operating system provides support for both of these operations.

SWI "OS_Write0" (&02)

On entry: R0 = Address of string to be output

On exit: R0 = Pointer to the byte after the end of the string

This routine uses register R0 to point to the address of a string in memory. This string may be any length, but must be terminated by a character of ASCII code '0'. The string may contain characters of any ASCII codes except, obviously, CHR$(0), as this is the terminator.

When called, "OS_Write0" will write each of the characters in the string to the output stream until the end of string marker is reached.

A common method of including strings in a program is to place them at the end of the code, using the EQUS directive, and 'EQUB 0' to add the terminator. The string can then be labelled, and the label address loaded into register R0 using the ADR directive. An example of this was given in Chapter 13 when the use of the EQU directives was described.

SWI "OS_WriteS" (&01)

On entry: Nothing passed

On exit: No information returned

This routine provides a quick way of writing fixed strings of characters to the output stream.

There are no entry or exit parameters. The string to be written is assumed to follow on directly from the SWI instruction in the next word of memory. Again, the string must be terminated by character of ASCII code 0.

When called, "OS_WriteS", will output the characters in the string until the terminator is reached. It will then modify the program counter so that the ARM resumes execution of the program from the word which immediately follows the end of the string. This makes programs look slightly strange, as the executable instructions seem to be split up by text messages. However, the operating system will take care of everything, and these type of programs really do work!

The routine is frequently used to embed fixed program messages or prompts in the code. For example, the following will display a prompt, then wait for a key to be pressed:

```
[
    SWI "OS_WriteS"
    EQUS ( "Press any key to continue:")
    EQUB 0
    SWI "OS_ReadC"
]
```

SWI "OS_ReadLine" (&0E)

On entry: R0 = Address of the buffer to hold string
R1 = Maximum length allowed for the string
R2 = Lowest permissible ASCII code entered into buffer
R3 = Highest permissible ASCII code entered into buffer

On exit: R1 = Length of the buffer
Carry flag set if ESCAPE was pressed during entry

Calling this routine will allow a program to read a complete line of text from the input stream into an area of memory.

On entry, register R0 must point to the memory area which is to act as a buffer for the characters. Characters will be accepted and stored sequentially in this buffer, providing their ASCII codes are in the range set by R2 and R3. The maximum number of characters which can be accepted is defined by R1. If an attempt is made to enter more than this number of characters, VDU 7 will be issued and the characters will not be accepted.

During the input process, pressing DELETE will remove the last character entered from both the screen and buffer. Also, pressing CTRL-U will cause all of the characters, previously entered on the line to be removed.

String entry is terminated when either a line feed or a cartridge return is entered. On exit, the specified memory area will contain the string. This will always be terminated by a character of ASCII code &0D, irrespective of how the input was terminated.

On BBC and Master micros, the ReadLine function is performed by OSWORD 0. This is available on the Archimedes for compatibility. However, the new routine should be used in preference to the old one.

An example of using ReadLine is given when an INPUT template is developed in Chapter 20.

SWI "OS_NewLine" (&03)

This routine simply writes a 'newline' to the output stream. A 'newline' is defined as being a line-feed character (&0A), followed by the return character (&0D).

Conversion Routines

A common requirement in assembly programs is the ability to convert between a numeric quantity and the string of characters representing the numeric value. If we want to print the number contained in a memory location, for example, we would need to convert the number into a string of decimal digits, then print the string.

On BBC micros and Master series machines, there is no support for performing these conversions in machine code programs. However, because it is such a common activity, the operating system on the Archimedes provides routines to carry out such conversions. Routines are provided to convert a string of numeric digits into an actual numeric quantity, and also to convert numbers into numeric strings using a variety of formats.

SWI "OS_ReadUnsigned" (&21)

On entry: R0 = Default base to be used in conversion
R1 = Pointer to string of digits to be converted

On exit: R2 = The value which the string was converted to

This routine will convert a string of numeric characters into an actual number. On entry to the routine, register R1 must have been set up to point to the string of digits to be converted. Register R0 should contain the number base to be used when converting the number. In addition to this, the string may contain a base number which over-rides the default one given in R0. This is done as follows:

<base>_<number> To select base for the conversion

or:

& To select hexadecimal for conversion

For example, the following strings are all suitable for conversion:

172	No base specified, therefore use default
2_10100101000	Specifies base two (binary)
8_777	Specifies base eight (octal)
&FFFF	Specifies hexadecimal

Any base from two to 36 may be used.

On exit from the routine, the value of the converted number is returned in register R2. Note that this routine will only convert unsigned numbers – negative quantities are not allowed. A routine is presented in Chapter 21 which shows an example of using the OS_ReadUnsigned routine to mimic the operation of BASIC's VAL statement. This program also contains additional code to allow positive and negative numbers to be converted.

SWI "OS_BinaryToDecimal" (&28)

On entry: R0 = Signed 32-bit number
R1 = Pointer to string buffer
R2 = Maximum length of buffer

On exit: Buffer contains converted numeric string
R2 = Length of numeric string in the buffer

This routine provides the reverse operation to the previous one. It is entered with R0 containing the signed 32-bit number to be converted. It will then convert this number into the equivalent string of numeric digits which represents it. These are stored in the string buffer pointed to by register R1. Register R2 is used to inform the routine how big the buffer is, and an error will be given if the converted number doesn't fit into the buffer.

On exit from the routine, the buffer will contain the appropriate string of numeric digits. If the number converted was negative, then the string will be preceded by a '–' character. Note that the string is *not* terminated, its length is returned in register R2.

'OS_BinaryToDecimal' is used in Chapter 21 to implement an equivalent to BASIC's STR statement in assembler.

Other Conversion Routines

The next group of SWIs provide similar functions to the previous routine, as they convert a number into an equivalent string. However, they allow the conversion to be performed in a variety of bases and formats. The SWI routines are divided into call blocks each of which performs the same function with a slightly different format. The names of each SWI in a block is almost

the same, differing only in the last character which selects the format used. The names used are as follows:

OS_ConvertHexN
OS_ConvertCardinalN
OS_ConvertIntegerN
OS_ConvertBinaryN
OS_ConvertSpacedCardinalN
OS_ConvertSpacedIntegerN

Where N is a numerical suffix to the name, used to specify the format.

The entry and exit conditions of all the routines are:

On entry: R0 = Number to be converted
R1 = Pointer to string buffer to contain result
R2 = Maximum length of buffer

On exit: Buffer contains the converted numeric string
R0 = Points to the string buffer
R1 = Points to the terminating zero byte in the buffer
R2 = Number of free bytes in the buffer

OS ConvertHexN (&D0 – &D4)

The SWI calls in the group are used to convert the given number into a hexadecimal string. The last character of the SWI name, N, may be 1, 2, 4, 6 or 8. This defines the number of hexadecimal digits produced in the resulting string. Leading zeros will be included to the left of the number, to pad it out to the specified number of digits.

Listing 17.1 contains a simple example of this routine. It uses "OS_ConvertHex8" to create an eight-character hexadecimal string representing the number &3E8.

Listing 17.1. Converting a number to a hexadecimal string.

```
10  REM Example of the 'number to hex string' routine
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  DIM buffer 32
70  DIM convert 256
```

```
 80  P%=convert
 90  [
100  MOV R0,#1000            ; Number to be converted (&3E8)
110  ADR R1,buffer           ; Address of string buffer
120  MOV R2,#32              ; Length of string buffer
130  SWI "OS_ConvertHex8"    ; Convert to hexadecimal string
140  SWI "OS_Write0"         ; Print the hexadecimal string
150  MOV PC,R14              ; Back to BASIC
160  ]
170  PRINT' "Numeric String is: ";
180  CALL convert
190  PRINT
```

OS_ConvertCardinalN (&D5 - &D8)

The SWI calls in the group are used to convert the given number into a decimal string. The number is assumed to be unsigned, ie, all the bits of the number are assumed to represent the number's magnitude.

The last character of the SWI name, N, may be 1, 2, 3 or 4. This specifies how many bytes of register R0 are occupied by the number to be converted. For example, using OS_ConvertCardinal4 will mean that a four-byte (32-bit) number is being converted. The converted string is not padded with leading zeros.

OS_ConvertIntegerN (&D9 - &DC)

This group of calls is exactly the same as the previous ones, except that the number given in R0 is assumed to be signed. It can thus be positive or negative using two's complement format.

OS_ConvertBinaryN (&DD - &E0)

The SWI calls in the group are used to convert the given number into a string of binary digits.

The last character of the SWI name, N, may be 1, 2, 3 or 4. This specifies how many bytes of register R0 are occupied by the number to be converted. For example, using OS_ConvertBinary3 will mean that a three-byte (24-bit) number is being converted. The converted string is always padded with zeros to obtain the required number of digits. Listing 17.2 contains an example of OS_ConvertBinary4.

Listing 17.2. Converting numbers to binary.

```
 10  REM Example of the 'number to binary string' routine
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM buffer 64
 70  DIM convert 256
 80  P%=convert
 90  [
100  MOV R0,#1000                ; Number to be converted
110  ADR R1,buffer               ; Address of string buffer
120  MOV R2,#64                  ; Length of string buffer
130  SWI "OS_ConvertBinary4"     ; Convert to binary string
140  SWI "OS_Write0"             ; Print the binary string
150  MOV PC,R14                  ; Back to BASIC
160  ]
170  PRINT' "String of binary digits is: ";
180  CALL convert
190  PRINT
```

OS_ConvertSpacedIntegerN (&E1 – &E4)

This group of calls is identical to OS_ConvertCardinalN except that a space is inserted at every three digits from the right. For example, the following number:

2100245673

would be converted to the string:

2 100 245 673

OS_ConvertSpacedCardinalN (&E5 – &E8)

This group of calls is identical to OS_ConvertIntegerN except that a space is inserted at every three digits from the right. For example, the number:

–30459210

would be converted to the string:

–30 459 210

System Calls

This next group of SWI calls are used to perform various system-related tasks. The first three (OSBYTE, OSWORD, OSCLI) are supported on the BBC and Master micros. These are entry points to routines which provide a whole range of functions. OSBYTE, for example, has an entry parameter which can select up to 256 different OSBYTE routines. Most of the routines available on the earlier machines are, where appropriate, included on the Archimedes. Several additional ones have also been added to control the extra features of the machine.

OSBYTE, OSWORD and OSCLI together now offer several hundred different functions. This is obviously too many to describe here! For this reason, only the method of accessing the three routines is described. Full details of the functions available can be found in the Advanced User Guide.

SWI "OS_BYTE" (&06)

On entry: R0 = Action code
R1 = Parameter one (if required)
R2 = Parameter two (if required)

On exit: R0 = Action code
R1 = May contain results (depends on routine called)
R2 = May contain results (depends on routine called)

OSBYTE is exactly equivalent to:

```
*FX  a,x,y
```

Where 'a' is a number in the range zero to 255 specifying the particular OSBYTE routine to be called. 'X' and 'y' are parameters which may be needed for certain routines.

Register R0 is used to pass the OSBYTE routine number (a). Registers R1 and R2 may be required to pass parameters. On exit from the routine, R0 is preserved. R1 and R2 may contain results from the particular routine called.

As an example, the command *FX12,1, which sets the keyboard auto-repeat rate to one-hundreths of a second, would be implemented in assembler as:

```
[
    MOV R0,#12
    MOV R1,#1
    SWI "OS_BYTE"
]
```

Note that the second OSBYTE parameter is not needed by this command, so the contents of register R2 are unimportant. Also the particular OSBYTE command used does not return any results, so the contents of the registers on exit are unimportant.

A complete list of the OSBYTE routines supported on the Archimedes is given in Appendix F.

SWI "OS_WORD" (&07)

On entry: R0 = Action code
R1 = Address of the OSWORD parameter block

On exit: The parameter block may be modified to return results

OSWORD calls a variety of routines which perform added functions to those provided by OSBYTE. The difference is that OSWORD routines typically need more than the two parameters used by OSBYTE, so a memory parameter block is used to pass them rather than the registers.

On entry, register R0 should contain the number of the OSWORD routine required. Register R1 must point to a parameter block in memory which contains the data for the routine.

As an example of the use of OSWORD, we shall use OSWORD 10 to read character definitions. The definition of all characters in the range 32 to 126 is read. For each character, the data is manipulated and used in a VDU23 statement to define the character to be upsidedown! A program which does all this is given in listing 17.3.

Listing 17.3. Manipulating character definitions using OSWORD 10.

```
10  REM Example of OSWORD to redefine characters
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  DIM redefine 256
70  DIM param_block 16 : REM Reserve for OSWORD parameter block
```

```
 80
 90 REM Define constants and register names
100 vdu    = 256
110
120 offset = 4
130 ascii  = 5
140
150 P%=redefine
160 [
170 MOV ascii,#32           ; Initial character to be redefined
180 .char_loop              ; Loop to redefine each character
190 MOV R0,#10              ; OSWORD 10
200 ADR R1,param_block      ; Pointer to parameter block
210
220 STRB ascii,[R1]         ; Store ASCII code in parameter block
230 SWI "OS_Word"           ; Call OSWORD
240
250 SWI vdu+23              ; Perform VDU23,ascii
260 MOV R0,ascii
270 SWI "OS_WriteC"
280
290 MOV offset,#8           ; Redefine the rows in the character
300 .redef_loop             ; in reverse order
310 LDRB R0,[R1,offset]     ; ie. row 1 as row 8
320 SWI "OS_WriteC"         ;     row 2 as row 7 ....
330 SUBS offset,offset,#1   ; and so on
340 BNE redef_loop
350
360 ADD ascii,ascii,#1      ; Increment character ASCII code
370 CMP ascii,#126          ; See if all characters processed
380 BLE char_loop           ; If not, redefine next character
390
400 MOV PC,R14 ; Back to BASIC
410
420 ]
430
440 CALL redefine
450
460 PRINT'''
470 PRINT "Try turning your monitor upsidedown !!!"
480 PRINT "Enter '*FX 20' to return to normal"
```

A complete list of the various OSWORD routines is given in Appendix G.

SWI "OS_CLI" (&05)

On entry: R0 = Address of command line string

On exit: Depends on the command executed

OSCLI is used to interpret and execute system commands. System commands are those which normally begin with a star character, for example, *CAT, *SHOW and so on. Every time one of these commands is used, OSCLI is called to process it.

On entry to the routine, register R0 points to a string in memory which contains the command to be executed. This is simply the series of characters in the command terminated by a carriage return (ASCII &0D).

When star commands are issued, the system normally allows special characters to be given, aliases to be used and so on. All of these features are also available when OSCLI is called from machine code.

Listing 17.4 contains an assembly language program which uses OSCLI to catalogue the disc.

Listing 17.4. Use OSCLI to catalogue a disc.

```
 10 REM Example of OSCLI to catalogue the disc
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 CLS
 70 DIM cli_com 256
 80
 90 FOR pass = 0 TO 3 STEP 3
100 P%=cli_com
110 [
120 OPT pass
130
140 ADR R0,command     ; Initialise pointer to command string
150 SWI "OS_CLI"       ; Call OSCLI
160 MOV PC,R14         ; Back to BASIC
170
180 .command
190 EQUS "CAT"         ; Command to be executed
200 EQUB &0D           ; Terminate with &0D
210 ]
220 NEXT
230
240 PRINT ''' "Executing *CAT from machine code now !!" '''
250 CALL cli_com
```

SWI "OS_ReadPoint" (&02)

On entry: R0 = X co-ordinate of point
R1 = Y co-ordinate of point

On exit: R2 = Colour of the specified point
R3 = Tint
R4 = Co-ordinate validity flag

This SWI call performs an equivalent function to OSWORD 9. It allows the logical colour of a point, at any graphics co-ordinate, to be determined.

On entry to the routine, registers R0 and R1 contain the x, y co-ordinates of the point.

On exit, register R2 contains the colour of the point. R3 contains the colour tint of the point in the top two bits. R4 contains zero if the point was on the screen and minus one if it was outside.

SWI "OS_EnterOS" (&16)

On entry: No parameters

On exit: No information returned

This call is used to switch the ARM processor from user mode to supervisor mode. This is needed if access is required to hardware devices, or if interrupts are to be manipulated. These activities can only occur if a program is running in supervisor mode.

When the call is issued, the processor mode switch comes into effect. Subsequent instructions are then executed in supervisor mode. Register R13 becomes a stack pointer for the operating system's stack, which can be used if required.

To return to user mode the following two instructions may be used:

```
TEQP PC,#0
MOVNV R0,R0
```

The second instruction is a null operation to allow the ARM to re-synchronise its register banks correctly.

SWI "OS_ValidateAddress" (&3A)

On entry: R0 = Start address of memory block to be checked
R1 = End address of memory block to be checked

On exit: Carry clear = Memory block valid
Carry set = Block contains an invalid address

Chapter Two described the memory management system on the Archimedes. It was noted that physical memory was not provided over the entire address space. This means that some memory addresses are illegal, as they do not correspond to physical memory. Also, some addresses cannot be accessed when the processor is in user mode.

This routine will check that every location in a block of memory is valid and can be accessed. The start and end addresses of the block are passed in registers R0 and R1. The carry flag indicates the block's validity on exit.

Interrupt Driven Routines

In Chapter 15 we saw how the Archimedes interrupt system worked. One common use of interrupts is to execute a machine code routine at specific time intervals, using interrupts from a hardware timer device.

This usually involves manipulating vectors, interrupts and the timer itself. To make life easier, the operating system provides three SWI calls which can set up timer-triggered routines. These are called:

OS_Call_After
OS_CallEvery
OS_RemoveTickerEvent

The first of these, OS_Call_After, will call a given routine after a certain time period has elapsed. The second, OS_CallEvery, will call a routine repeatedly at regular, definable, time intervals. The final routine, OS_RemoveTickerEvent, will cancel the previous command so that the routine is no longer called.

When writing routines to be called in this way, the usual rules about writing any interrupt driven routine should be observed.

The entry parameters for the three routines are given below.

SWI "OS_CallAfter" (&3B)

On entry: R0 = Number of centi-seconds after which call is to be made
R1 = Address of the routine to call
R2 = The value which register R12 will contain when the routine is called

SWI "OS_CallEvery" (&3C)

On entry: R0 = Time interval between calls to the routine
R1 = Address of the routine to call
R2 = The value which register R12 will contain when the routine is called

SWI "OS_RemoveTickerEvent" (&3D)

On entry: R0 = Address of routine to be stopped
R1 = The value of R12 used when the routine is called

18 : WIMPs

Controlling the WIMP Enviroment

A major feature of the Archimedes system is its support of a WIMP environment (Windows, Icons, Mice, Pull-down menus). The WIMP system provides an alternative to the traditional way of communicating with the computer using the keyboard. The user interacts with programs by moving a mouse pointer and pressing its buttons. Options are displayed graphically on the screen. These options are pointed to and selected by using the mouse.

Obviously, a great deal of work is involved in producing the graphics required in a WIMP environment. It would be very inconvenient, to say the least, if we had to write code in each application program to do all this! For this reason, the Arthur operating system includes a series of WIMP management routines which assist us when writing WIMP-based programs.

It is not feasible for the operating system to take over all responsibility for controlling the WIMP system. Different applications programs use different facilities in different ways. Trying to cater for every case would be impossible! Instead, a two-way dialogue is undertaken between the WIMP management system and our application program. The program instructs the WIMP environment to perform actions on its behalf, for example drawing a window, creating a menu and so on. The WIMP manager informs the program whenever a significant event occurs or whenever circumstances arise which it can't itself deal with.

The program would be informed, for example, when the mouse pointer enters a window or when a mouse button is pressed. As a result of some other action by the user, a part of the screen may need updating. For example, if the user moves a window then the WIMP may calculate that several other windows have become visible. It will, therefore, issue appropriate requests to the application program to ask it to redraw the affected areas. This redrawing may directly involve the application in some work, or may just require it to call other WIMP routines to the work for it.

The WIMP manager in the Arthur operating system on the Archimedes is very sophisticated. To describe every aspect of it would take a complete book in itself! In this chapter, therefore, we shall only look at some of the fundamental aspects of the system. We shall cover some of the most important routines which it provides, and see how these can be used to create some windows of our own.

Accessing the Mouse

Most of the user's interactions with the mouse are reported to us indirectly by the WIMP in the form of the events. However, there will be times when we want to access the mouse directly. For example, when the mouse is being used in a program without the full WIMP environment. The operating system provides an SWI routine, specifically for this purpose, which is separate from the WIMP system. The routine is called OS_Mouse and has the following entry and exit parameters:

SWI "OS_Mouse"

Syntax:

```
SWI"OS_Mouse"
```

On entry: No parameters

On exit: R0 = Current mouse x co-ordinate
R1 = Current mouse y co-ordinate
R2 = State of mouse buttons
R3 = Time of last button change

The value returned in register R2 is made up from three bits which reflect the state of the three mouse buttons. The bits are allocated as follows:

Bit	Button
0	Right button
1	Middle button
2	Left button

The mouse x and y co-ordinates are in the same range as the screen graphics co-ordinates, ie, 0 <= x <= 1279 and 0 <= y <= 1023. This makes it very easy to draw using the mouse as no scaling is required. Listing 18.1 uses OS_Mouse to implement a simple sketch pad. The mouse will draw on

the screen if the left button is pressed and the drawing colour can be changed by pressing the middle button. The graphics used in this program are explained in Chapter 24 where various graphics routines are covered.

Listing 18.1. A Simple sketch pad using the mouse.

```
 10 REM Sketch PAD using the mouse
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM sketch 256
 70
 80 REM Define constants and register names
 90 vdu  = 256     : REM Start of SWI block to perform VDU n
100 gcol = 18
110 plot = 25
120 dot  = 69
130
140 col  = 4
150 x    = 5
160 y    = 6
170
180 P% = sketch
190 [
200
210 SWI "OS_Mouse"          ; Get mouse data
220 MOV x,R0                ; Store x,y co-ords in other regs
230 MOV y,R1
240
250 TST R2,#%010            ; See if middle button pressed
260 ADDNE col,col,#1<<20    ; If so, then increment the colour
270
280 SWI vdu+gcol            ; Perform GCOL0,col (scaling 'col')
290 SWI vdu+0
300 MOV R0,col,LSR#25
310 SWI "OS_WriteC"
320
330 TST R2,#%100            ; See if left button pressed
340 BEQ sketch              ; If not loop back
350
360 ; This next section of code plots a point at
370 ; the co-ordinates in registers 'x' and 'y'
380 ; These were the current mouse co-ordinates
390
400 SWI vdu+plot
410 SWI vdu+dot
420 MOV R0,x
430 SWI "OS_WriteC"
440 MOV R0,x,LSR#8
450 SWI "OS_WriteC"
```

```
460  MOV R0,y
470  SWI "OS_WriteC"
480  MOV R0,y,LSR#8
490  SWI "OS_WriteC"
500
510  B sketch   ; Branch to keep sketching points
520
530  ]
540
550  MODE 15
560  *POINTER
570  E% =2<<25
580  CALL sketch
```

Initialising the WIMP

Before any WIMP routines can be used, the WIMP manager must be initialised. This sets up the screen and resets the manager. This is done using the SWI routine:

```
SWI "Wimp_Initialise"
```

The routine requires no parameters and performs all the initialisation required. It should be called once, just before an application starts using the WIMP environment.

WIMP Windows

A window under the WIMP system is a screen area in which an application may display graphics or text. Typically, the window will be surrounded on the screen by a 'systems area'. This allows the mouse to manipulate the window in a number of ways. For example, the window can be dragged to another area of the screen, its dimensions can be changed, and so on.

When a window is defined, we actually specify two areas. The first is the complete window size, called the window extent. This may be any size required and need not fit on the screen. The second area specified is the visible part of the window, called the work area. This is the area which will be seen on the screen and, if the window extent is larger, will only show a part of the total window contents. The system area around the window can contain items called scroll bars. These allow the user to scroll the work area over the entire window extent. In this way, the work area can be made to display any part of the total window extent area. A typical win-

dow is shown in figure 18.1. This also shows the functions of the various system areas surrounding the work area.

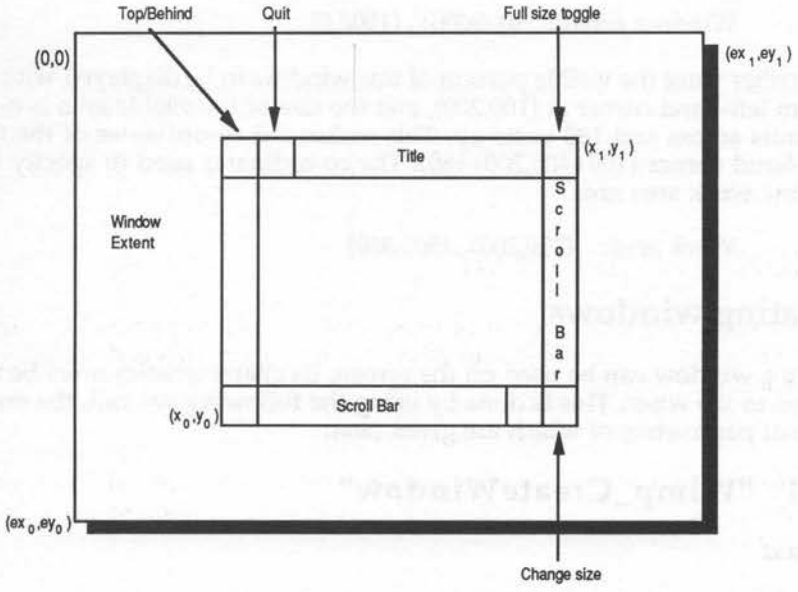


Figure 18.1. Layout of a typical WIMP window.

When we specify the work area, we do so by quoting the screen co-ordinates of the bottom-left and upper-right corners. This defines a rectangle on the screen in which the visible portion of the window will be displayed. The window extent is defined in a similar way. However, this time the co-ordinates are given relative to the top left-hand corner which is normally taken to be at 0,0. Thus, to create a window, which in total size is 'h' high and 'w' wide, we would specify:

(0,-h)

and also:

(W,0)

An example will help to clarify this. Suppose we want to create a window the total size of which is 2000 wide and 1500 high. The window extent would be specified as follows:

Window extent: (0,-2000) , (1500,0)

We further want the visible portion of this window to be displayed with its bottom left-hand corner at (100,200), and the size of the visible area is to be 400 units across and 180 units up. This makes the co-ordinates of the top right-hand corner (100+400,200+180). The co-ordinates used to specify the window work area are:

Work area: (100,200) , (500,380)

Creating windows

Before a window can be used on the screen, its characteristics must be described to the WIMP. This is done by using the following SWI call, the entry and exit parameters of which are given next:

SWI "Wimp_CreateWindow"

Syntax:

```
SWI "Wimp_CreateWindow"
```

On entry: R1 = Pointer to a window description block

On exit: R0 = Window handle.

The window handle is a number returned by the WIMP, which uniquely identifies the particular window. This is used to specify which window is to be operated on in other WIMP routines.

The window description block is simply an area of memory which holds all of the parameters necessary to define the window. The contents of the block are as follows:

Block + 0:	X co-ordinate of bottom-left corner of work area (x0)
Block + 4:	Y co-ordinate of bottom-left corner of work area (y0)
Block + 8:	X co-ordinate of top-right corner of work area (x1)
Block + 12:	Y co-ordinate of top-right corner of work area (y1)

Block + 16:	Scroll bar x position
Block + 20:	Scroll bar y position
Block + 24:	Handle to open window behind (–1 = top, –2 = bottom)
Block + 28:	Flags/status information
Block + 32:	Window title foreground colour
Block + 33:	Window title background colour
Block + 34:	Work area foreground colour
Block + 35:	Work area background colour
Block + 36:	Scroll bars outer colour
Block + 37:	Scroll bars inner colour
Block + 38:	Colour of window title background when highlighted
Block + 39:	Reserved
Block + 40:	X co-ord of bottom-left corner of window extent (Ex0)
Block + 44:	Y co-ord of bottom-left corner of window extent (Ey0)
Block + 48:	X co-ord of top-right corner of window extent (Ex1)
Block + 52:	Y co-ord of top-right corner of window extent (Ey1)
Block + 56:	Icon type flags for the title bar
Block + 60:	'Button type flags' for work area
Block + 64:	Sprite area control block
Block + 68:	Reserved – must be &00000000
Block + 72:	Window title string – maximum of 12 characters
Block + 84:	Number of icons initially defined for window
Block + 88:	Icon definitions (32 bytes per icon)

The work area and window extent co-ordinates are specified as described in the previous section. If a window extent is specified so that it doesn't completely contain the work area, then the WIMP will produce a 'bad work area extent' error message.

The scroll bar positions are the initial offsets of the work area within the window extent area. They specify exactly which part of the total window area is to be displayed initially in the work area. Note that these co-ordinates are given relative to the top-left corner of the window extent, which is usually at (0,0).

The window status flags define a further set of characteristics of the window. The options are listed in figure 18.2. Any combination of options can be used by including the corresponding bit into the final number used.

Control Flags

Bit	Meaning if set
0	Window has a title bar
1	Window can be moved about the screen
2	Window has a vertical scroll bar
3	Window has a horizontal scroll bar
4	Window can be redrawn entirely by WIMP (no user graphics)
5	Window is a 'pane' onto a tool window
6	Window can be moved so that parts of it are off the screen
7	Window has no 'back' or 'quit' boxes
8	'Scroll request' made if scroll bars clicked (auto-repeat)
9	'Scroll request' made if scroll bars clicked (debounced)

Status Flags

Bit	Meaning if set
16	Window is currently open
17	Window is 'on top', ie, not covered
18	Window has been toggled to full size

Figure 18.2. Window control flags.

The title bar 'icon' flags define exactly what is to be displayed as the title of the window. These flags are the same as those used to define the type of any icon within a window and will be described later. For most purposes, the value of this parameter will be 15, as this specifies that centered text, ie, the window title string, is to be displayed in the title bar.

The 'work area button type' and 'sprite control flags' will not be described, as they refer to more advanced facilities of the WIMP. These parameters may each be set to zero to de-select the corresponding features.

The parameter at block + 84 specifies how many icons the window is to contain initially. The bytes following this are used to store the definitions of any icons used. The creation of icons is dealt with in the next section.

Icons

An icon can best be described as a sensitive area within a window which is treated specially by the WIMP environment. Physically, an icon could be a

piece of text, a sprite or an anti-aliased font. Icons are an integral part of the WIMP window. A window can be defined to contain several icons at arbitrary co-ordinates. These automatically will be displayed whenever the part of the window containing them becomes visible.

In addition to displaying icons automatically, the WIMP can also be instructed to take special action when the mouse pointer and an icon interact. For example, we can be notified whenever the pointer passes over an icon or when it is selected by clicking the mouse button.

There are also a whole series of advanced facilities associated with icons. For example, the WIMP can create a menu structure comprised of icons and will then automatically handle the selection of items from the menu. Writable icons can be created which allow text to be entered into them from the keyboard under WIMP control.

For our purposes, we shall confine ourselves with simply looking at how a simple icon can be defined and included within a window definition.

Defining Icons

When a window is defined, any number of icons can be included within it. This is done by appending the relevant data onto the parameters in the window definition block. You will recall that the parameter stored at block + 84 defined how many icons the window was to contain. Following this are blocks of 32 bytes which contain the icon definition. The data in these 32-byte blocks is as follows:

Byte	
0	X co-ordinate of bottom-left corner of icon box
4	Y co-ordinate of bottom-left corner of icon box
8	X co-ordinate of top-right corner of icon box
12	Y co-ordinate of top-right corner of icon box
16	Icon control flags
20	Icon data

The icon box specifies the co-ordinates of the rectangle within the window which is to contain the icon. The icon control flags define the characteristics of the icon as follows:

Bit	Meaning when set
0	Icon contains text
1	Icon is a sprite
2	Icon has a border
3	Icon text is centred horizontally within box
4	Icon text is centred vertically within box
5	Icon has a filled background
6	Icon has anti-aliased font text
7	Icon requires application to redraw it
8	Icon data is 'indirected'
9	Icon text is right justified
10	If selected, don't cancel other selections
11	Reserved
12-15	Button type, controls icons response to being 'clicked'
16-20	Exclusive selection group of icon
21	Icon has been selected (inverted)
22	Icon cannot be selected by mouse (shaded)
23	Icon has been deleted

The final part of the icon block is the 12 bytes of actual icon data. This will depend on exactly what type of object the icon is. If the icon is text, then the data is a string of up to 12 bytes terminated by a character of ASCII code 13. If the icon is a sprite, then the data is the name of the sprite. Finally, if the icon is a writable object into which text can be entered, then the following rule applies:

Word	
0	Pointer to buffer to contain entered text
4	Pointer to validation string (minus one if none)
8	Length of buffer in bytes

When an icon is defined in a window in this way, it is allocated a handle number which identifies it in other operations. The handle is unique to the window containing the icon and is zero for the first icon defined, one for the second and so on.

Opening Windows

So far, we have seen how to describe the characteristics of a window to the WIMP manager. We have not, as yet, seen how we actually produce the window on the screen. This is done quite simply by asking the WIMP manager to 'open the window', using the following SWI routine:

SWI "Wimp_OpenWindow"

Syntax:

```
SWI "Wimp_OpenWindow"
```

On entry: R1 = Pointer to parameter block

On exit: Nothing returned

Once again, the routine makes use of a parameter block to pass information to the WIMP manager. The contents of this block are:

Block + 0	Handle of the window to be opened
Block + 4	X co-ordinate of bottom-left corner of work area (x0)
Block + 8	Y co-ordinate of bottom-left corner of work area (y0)
Block + 12	X co-ordinate of top-right corner of work area (x1)
Block + 16	Y co-ordinate of top-right corner of work area (y1)
Block + 20	Scroll bar x position
Block + 24	Scroll bar y position
Block + 28	Handle to open window behind (–1 = top, –2 = bottom)

The first parameter is the handle of the window to be opened and displayed on the screen. This is the number which was returned when the window was created.

The next six parameters are the familiar ones used when the window was created. They define where the window is to be placed on the screen, how big it is and which part of the total window area is to be displayed. These parameters may be the same as those used when the window was created. Alternatively, they can be changed to open the window anywhere on the screen and at any size.

The final parameter refers to where a window should be placed in respect to other windows which may already be on the screen. Specifying '–1' for example, will ensure the new window appears on top of existing ones.

When the Open Window request has been made, the WIMP will make the necessary calculations to display the window at the required position in the screen. However, it will not draw the window at this time. Instead, the graphics are said to be 'pending' and will be produced when the WIMP polling routine is called. This routine is described in the next section.

Polling the WIMP

Earlier we said that the application program and the WIMP manager took part in a two-way dialogue. So far, this dialogue has only been one-way, with our program telling the WIMP manager about the layout of windows and requesting them to be opened. The WIMP's 'poll' routine allows the WIMP to send information and requests back to the application program. The entry and exit conditions of the routine are:

SWI "Wimp_Poll"

Syntax:

```
SWI "Wimp_Poll"
```

On entry: R0 = Mask
R1 = Pointer to result block

On exit: R0 = Reason code
Result block contains data depending on reason code

When this routine returns, register R0 will contain a number which indicates which event or request the WIMP manager is informing us of. Each of the possible codes are listed in figure 18.3.

On entering Wimp_Poll, register R0 contained a mask. This allows some of the reason codes to be effectively masked out so that they are not returned to the user. Normally, however, the mask will be zero which allows all reason codes to be passed on.

When Wimp_Poll returns with a reason code, the result block which is pointed to by R1 on entry, will contain further information about the request or event.

Code	Reason
0	Null code – nothing has happened
1	Re-draw Window – request that application redraws a window
2	Open Window – request that application opens a window
3	Close Window – request that application closes a window
4	Mouse pointer has just entered a window
5	Mouse pointer has just left a window

6	The mouse buttons have just changed state
7	The user has completed a box drag operation
8	A key has been pressed on the keyboard
9	Option selected from a menu
10	Request to scroll user graphics in the work area

Figure 18.3. Reason codes returned by Wimp Poll.

Reason codes one, two, three and 10 are requests for the application to perform some operation which the WIMP could not directly handle. The remaining codes simply inform the application of events which have occurred which may be significant. These may be acted on, or ignored.

Let's look at some of the key reason codes returned by the WIMP in more detail. Full explanations of all the codes are explained in the Advanced User Guide.

Reason Code 1: Re-draw Window Request

This reason code indicates that, as a result of some user activity, a part of a window is not up-to-date. The application therefore requests a redraw of the appropriate section. This is done by asking the WIMP to calculate a full list of the screen rectangles, the contents of which must be redrawn by the application.

We shall restrict ourselves to creating windows which do not contain any user-controlled graphics. Such windows can still contain icons, but can be completely managed by the WIMP system. This reason code will not, therefore, occur in these circumstances and we will not consider it any further.

Reason Code 2: Open Window

This reason code means that the WIMP requires that a window should be opened at a specified position on the screen. This will be the case if an existing window has been moved across the screen, changed in size, scrolled and so on.

The results block returned with this reason code contains all the required data to open the window. This is in exactly the same format as the parameter block used by SWI Wimp_OpenWindow. All the application has

to do, therefore, is to execute SWI Wimp_OpenWindow, using the WIMP poll result block as the new parameter block.

Reason Code 3: Close Window

This reason code is issued when the user clicks the close window box. This means that the specified window should be removed from the screen, and the WIMP manager's list of active windows. The first word in the result block, which is returned with the code, contains the handle if the window to be closed.

The WIMP could immediately close the window itself. However, it issues this reason code so that the application can decide whether the window should be closed or not. The application could, for example, prompt for confirmation before closing the window.

If the application decides that the window should be closed, it can instruct the manager to do so using the routine with the following entry parameter:

SWI "Wimp_CloseWindow"

Syntax:

```
SWI "Wimp_CloseWindow"
```

On entry: R1 points to a parameter block

The first word in the parameter block is the handle of the window to be closed. This is compatible with the result block returned by SWI Wimp_Poll. This result block can, therefore, be used directly as the parameter block to SWI Wimp_CloseWindow.

The action of closing a window doesn't remove the window's definition from the WIMP manager's data tables. A closed window can still be reopened at a later data if required. To completely remove a window from the WIMP system we use the following:

SWI "Wimp_DeleteWindow"

Syntax:

```
SWI "Wimp_DeleteWindow"
```

This removes the definition of the window from the WIMP system, thus freeing the memory which it used to take up.

An Example of a Simple Window Program

The problem with the window system is that a great deal of material has to be understood and got right before you can get any results at all. It is for this reason that we have gone through all of the essential elements of the window manager before presenting any example programs. However, we can now put the theory into practice and produce a small program which demonstrates the use of windows.

The program is designed to be very simple so that the components of it can be easily understood. Several complete window programs are included on the Archimedes Welcome Disc and these show what can be done using the same window primitives, but on a larger scale.

Listing 18.2 creates a window each time the middle mouse button is pressed. These are displayed on the screen and the user can manipulate them, eg, move them, change their sizes, and so on, by using the mouse. Windows can be removed by clicking on 'Close Window'.

The title of each window is different and each includes an icon. This is simply a piece of text saying 'icon' surrounded by a box. The dimensions of the windows created are initially 300 x 120, however, the extent of the windows is much larger. This allows the window's size to be increased and the scroll bars to be used.

The data defining the windows and icons is created in the assembly program using EQU directives. Note that it is important to use the ALIGN directive to ensure that each block of data starts at a word-aligned address.

The ARM instructions in the program are minimal, the main task being to get all the data parameters correct. The program itself simply initialises the WIMP and then enters a polling loop. This executes SWI Wimp_Poll and tests the reason code returned. The following reason codes are recognised and acted on by the program:

Reason code	Action taken
1) Open Window	SWI "Wimp_OpenWindow" called
2) Close Window	Window closed and deleted using: SWI "Wimp_CloseWindow" SWI "Wimp_DeleteWindow"
3) Mouse button pressed	New window defined and opened using: SWI "Wimp_CreateWindow" SWI "Wimp_OpenWindow"

Listing 18.2. Example of creating windows.

```
 10 REM Producing windows using the WIMP manager
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM windows 1024
 70
 80 REM Define register names
 90 count   = 5
100 pointer = 4
110
120 FOR pass = 0 TO 3 STEP 3
130 P% = windows
140 [
150 OPT pass
160
170 MOV count,#0           ; Initialise window counter
180 SWI "Wimp_Initialise"  ; Initialise WIMP manager
190
200 .poll_loop             ; WIMP polling loop
210 MOV R0,#19             ; *FX 19 - wait for vertical sync
220 SWI "OS_Byte"
230 MOV R0,#0              ; Don't mask out any reason codes
240 ADR R1,result_block    ; Get address of result block into R1
250 SWI "Wimp_Poll"        ; Poll the WIMP
260 CMP R0,#0              ; See if NULL reason code
270 BEQ poll_loop          ; If so branch back and poll again
280
290 CMP R0,#2              ; See if it's reason code 2
300 SWIEQ "Wimp_OpenWindow"   ; If so, open specified window
310 BEQ poll_loop             ; Branch back to polling loop
320
330 CMP R0,#3                 ; See if it's reason code 3
```

```
340 SUBEQ count,count,#1        ; Decrement window count
350 SWIEQ "Wimp_CloseWindow"    ; Close window
360 SWIEQ "Wimp_DeleteWindow"   ; Delete window definition
370 BEQ poll_loop               ; Branch back to polling loop
380
390 CMP R0,#6                   ; See if it's reason code 6
400 BNE poll_loop  ; If not then branch back to polling loop
410 CMP count,#30  ; See if maximum number of windows are open
420 BGE poll_loop  ; If so then branch back to polling routine
430
440 ; Routine to create new window at mouse pointer co-ords
450
460 ADD count,count,#1      ; Increment the windows count
470 ADR R1,title_suffix     ; Get the address of the title suffix
480 MOV R0,count            ; Add window number string to title
490 MOV R2,#3
500 SWI "OS_BinaryToDecimal"
510
520 ADR R1,window_def       ; Get addr of window definition in R1
530 SWI "Wimp_CreateWindow"     ; Create new window
540
550 ; This section of code opens new window on the screen
560
570 ADR pointer,open_block ; Get block addr of open routine
580 STR R0,[pointer,#0]    ; Store window handle in block + 0
590
600 SWI "OS_Mouse"          ; Get mouse co-ordiantes
610 STR R0,[pointer,#4]     ; Store x co-ordinate at block +4
620 STR R1,[pointer,#8]     ; Store y co-ordinate at block +8
630 ADD R0,R0,#300          ; Calculate X+300
640 ADD R1,R1,#120          ; Calculate Y+120
650 STR R0,[pointer,#12]    ; Store X+300 in block+12
660 STR R1,[pointer,#16]    ; Store Y+120 in block+16
670 MOV R0,#0
680 STR R0,[pointer,#20]    ; Store '0' in block+20,24
690 STR R0,[pointer,#24]
700 MVN R0,#0      ; Store '-1' in block+28 (open window on top)
710 STR R0,[pointer,#28]
720 ADR R1,open_block       ; Put address of open_block in R1
730 SWI "Wimp_OpenWindow"   ; Open the new window on the screen
740 B poll_loop             ; Branch back to polling loop
750
760 ALIGN
770 ; Set up the definition parameters for the windows
780 .window_def
790
800 ;Work Area
810 EQUD 100                ; x0
820 EQUD 100                ; y0
830 EQUD 400                ; x1
840 EQUD 220                ; y1
850
860 ;Scroll Bar positions
```

```
 870 EQUD 0                  ; Horizontal
 880 EQUD 0                  ; Vertical
 890
 900 EQUD 0                  ; Handle to open new window behind
 910
 920 EQUD 31                 ; Window flags ; Has title bar
 930 ; Is moveable
 940 ; Has vertical scroll bar
 950 ; Has horizontal scroll bar
 960 ; Can be re-drawn without application
 970 ; Colours
 980 EQUB 1                  ; Title foreground
 990 EQUB 2                  ; Title background
1000 EQUB 3                  ; Work area foreground
1010 EQUB 4                  ; Work area background
1020 EQUB 5                  ; Scroll bars outer colour
1030 EQUB 6                  ; Scroll bars inner colour
1040 EQUB 7                  ; Hightlight colour
1050
1060 EQUB 0                  ; Reserved
1070
1080 ;Window Extent
1090 EQUD 0                  ;ex0
1100 EQUD -800               ;ey0
1110 EQUD 800                ;ex1
1120 EQUD 0                  ;ey1
1130
1140 EQUD 25                 ; Title bar flags
1150
1160 EQUD 0                  ; Work area button type
1170 EQUD 0                  ; Sprite area control
1180 EQUD 0                  ; Reserved
1190
1200 ;Window Title
1210 EQUS ("Window ")
1220 .title_suffix
1230 EQUS ("      ")
1240
1250
1260 EQUD 1              ; Number of icons in window
1270
1280 ; Define ICON bounding box
1290 EQUD 75             ; x0
1300 EQUD -75            ; y0
1310 EQUD 225            ; x1
1320 EQUD -25            ; y1
1330
1340 EQUD %1101          ; ICON flags
1350 EQUS "ICON"         ;ICON text
1360 EQUB 13
1370
1380
1390 ALIGN
```

```
1400  ; Block used when opening windows
1410  ; The data is filled in by the program
1420  .open_block
1430  EQUS STRING$(32,CHR$(0))
1440
1450  ALIGN
1460  ; Block used to return data from poll WIMP
1470  .result_block
1480  EQUS STRING$(32,CHR$(0))
1490
1500  ]
1510  NEXT
1520
1530  MODE 12
1540  GCOL128+15:CLG
1550  VDU19,15,0,0,0,0
1560  VDU 19,0,7,0,0,0
1570  *POINTER
1580  CALL windows
```

19 : Managing Fonts

The Archimedes includes an extension to the Arthur operating system called the font manager. This provides an alternative to the normal, limited, eight by eight characters usually displayed in the various screen modes. The font manager allows us to paint characters of variable size and proportion, in several high-quality typefaces anywhere on the screen. The characters are proportionally spaced, can be micro-justified and are displayed using special anti-aliasing techniques to reduce the effects of limited screen resolution.

There are a great many facilities provided by the font system and we can't hope to describe them all here. Instead, we will aim to cover the system in general and give an idea of its capabilities. Sufficient routines will be explained to allow us to use most features of the fonts in our own machine code programs. For full details of every routine provided by the font manager, refer to the Advanced User Guide.

The Character Fonts

The character definitions for the fonts are held in disc files. Two are supplied on the Archimedes Welcome Disc but it is possible for the user to define his/her own. The characters for the font are defined in several different point sizes within the files. This allows font characters to be printed in different sizes without losing definition, as would be the case if simple scaling is used.

When a font is requested, the font manager will load it from disc into a reserved area of memory known as the font cache. Future references to the data in the font can then be made without accessing the disc each time. The default size of the font cache is relatively small, and it is possible to run out of space when using the fonts. However, the cache size can be made larger using the operating system command:

```
*CONFIGURE FONTSIZE <n>
```

Where n is the new number of memory blocks which are allocated to the font manager.

The facilities of the font manager and font painter are accessed in two ways, by using SWI calls or VDU control codes. The VDU codes are more convenient in BASIC, whereas the SWI calls are more appropriate for machine code programs. For this reason we shall, on the whole, use SWI calls to manipulate the fonts.

Initialising a Font

Before we can use a font to output text to the screen, we must initialise it. The preferred way of doing this is by calling the SWI routine:

```
SWI "Font_FindFont"
```

This will locate the appropriate disc file which contains the required font definitions. The data is loaded into the manager's font cache and is then available for use. Any number of fonts can be initialised at one time, providing that there is sufficient space in the font cache. This allows several fonts of different sizes and typestyles to be used together.

The entry and exit conditions for "FindFont" are as follows:

SWI "Font_FindFont"

Syntax:

```
SWI "Font_FindFont"
```

On entry: R1 = Pointer to a string containing the font name
R2 = Required 'x' point size for the font
R3 = Required 'y' point size for the font
R4 = Screen x resolution (zero implies the default)
R5 = Screen y resolution (zero implies the default)

On exit: R0 = Font's handle

Remember that the font name is the path name on the disc which will locate the font's definition files.

If the exact point size specified is not available in the font definition file, the font manager will retrieve the nearest size to it. It will then perform

conversion algorithms to transform the font definitions to the character exact size required.

The screen x and y resolution control how the font point size is converted into screen co-ordinates. If these parameters are set to zero, then a default will be assumed which is suitable for the screen mode selected when the call is made. This is what normally happens, so it is important to select the screen mode before initialising the font.

When the "FindFonts" call has been made, the font handle will be returned in register R0. This is a number, unique to the font, which is used to identify the font in future operations.

Painting Text in Different Fonts

When initialised, the characters making up a font can be painted directly on the screen. The SWI call to do this is as follows:

SWI "Font_Paint"

Syntax:

```
SWI "Font_Paint"
```

On entry: R1 is a pointer to the string to be 'painted' on the screen
R2 = Plotting option
R3 = X co-ordinate
R4 = Y co-ordinate

The plotting option is a number in which individual bits select different functions. The function of each bit is as follows:

Bit 0:	Set	Justify the text
	Clear	Don't justify the text
Bit 1:	Set	Rub out previous screen contents before painting font
	Clear	No rub out used
Bit 2:	Set	Use absolute co-ordinates (no alternative to this)
Bit 4:	Set	(x,y) given as normal graphics co-ordinates
	Clear	(x,y) given as 1/72000th of an inch

Both the rub-out and justification options assume that a suitable box has been defined by previously moving the graphics cursor to the appropriate screen co-ordinates.

Listing 19.1 gives an simple example of painting a font. Remember that the disc containing the font definitions must be in the drive when the program is run, otherwise the font won't be found.

Listing 19.1 Painting text in the 'Trinity' font.

```
 10 REM Example of the painting text using the font manager
 20 REM Note : Welcome Disc must be in drive
 30 REM Shows problems of an undefined anti-aliasing palette
 40 REM (c) Michael Ginns 1988
 50 REM Dabs Press : Archimedes Assembly Language
 60 REM
 70
 80 *| SET THE DEFAULT FONT PATH NAME HERE
 90 *| NOTE : Check, using *FONTLIST, the complete font name of
100 *|        a font that has been loaded. It is to this that
110 *|        the font path name is prefixed.
120 *|        The resulting filename should locate
130 *|        the font file starting at the directory root
140
150 *SET Font$Prefix $.Fonts
160
170 DIM fonts 256
180
190 x_point_size = 480 : REM horizontal point size
200 y_point_size = 320 : REM vertical point size
210 handle = 10
220
230 FOR pass = 0 TO 3 STEP 3
240 P% = fonts
250 [
260 OPT pass
270
280 ADR R1,font_name        ; Path name of font file on disc
290 MOV R2,#x_point_size
300 MOV R3,#y_point_size
310 MOV R4,#0               ; Default x,y screen resolution
320 MOV R5,#0
330
340 SWI "Font_FindFont"     ; Get font into cache and initialise
350 MOV handle,R0           ; Font handle returned in R0
360
370
380 ADR R1,text             ; Text to be painted on screen
390 MOV R2,#2               ; Plot mode - absolute OS coords
400 MOV R3,#0               ; Paint text starting at (0,600)
```

```
410  MOV R4,#600
420  SWI "Font_Paint"        ; Paint the text
430
440  MOV PC,R14
450  MOV R0,handle           ; Put font handle into R0
460  SWI "Font_LoseFont"     ; Finished so inform manager
470
480  MOV PC,R14              ; Back to BASIC
490
500  .font_name              ; Name of font file on disc
510  EQUS "Trinity.Medium"
520  EQUB 0
530  .text                   ; Text message to be painted
540  EQUS "This is text produced by the FONT system"
550  EQUB 13
560  ]
570  NEXT
580
590  MODE 12
600
610  CALL fonts
```

The first section of the program initialises the font. This may take several seconds, and you should see the disc drive being accessed. After this, a text string is output to the screen at position 100,500. You may be surprised at the result of this! The characters are displayed in what seems to be a collection of random coloured dots. This effect is due to the anti-aliasing system which is covered in the next section.

Anti-aliasing

No matter how highly defined a font is, it will only be as good as the resolution of the screen mode in which it is displayed. The Archimedes has relatively high-definition screen modes, 640 x 256 pixels. However, the fonts are defined to a much higher resolution than this, so there is bound to be distortion when painting the characters. We can't, for example, illuminate half of a pixel to represent a very thin line, even though the font definition says that we could. Consequently, circles and angled lines tend to have very jagged edges. Figure 19.1 illustrates the problem.

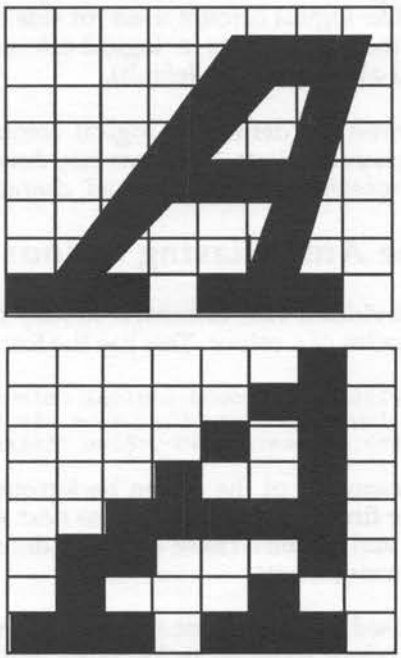


Figure 19.1. The problems of limited resolution.

To overcome these restrictions, we use a technique called anti-aliasing. Under this system, partially filled points are plotted as a pixel using a suitable shade of grey. For example, if a half pixel should be used, then a whole pixel whose colour is mid-way between the foreground and background colours will be plotted. In this way, the edges of the plotted characters are smoothed out.

The effectiveness of anti-aliasing depends on how many shades are available for representing incomplete pixels. Two colours, for example, could represent complete and half pixels but no smaller divisions. Adding another shade to this would allow quarter pixels to be represented, effectively doubling the apparent resolution of the displayed characters.

The Archimedes anti-aliasing system will support, at maximum, the use of 16 colours to represent part-filled pixels. The font painter will automatically plot different logical colours, when painting characters, to represent incomplete pixels. The logical colours used for this begin at the font foreground colour and include the next 'n' logical colours. N is the number of colours used for anti-aliasing (16 by default).

It is up to us, however, to define the logical colours to be appropriate shades of the foreground colour. This was not done in the previous example, and for this reason the 'multi-coloured' characters were produced.

Setting Up the Anti-aliasing Colour Palette

The font painter provides a VDU command to help in the re-definition of logical colours as shades of a colour. This has the form:

```
VDU 23,25,128+<background logical colour>,<foreground
logical colour>,<Red start>,<Green start>,<Blue start>,
<Red finish>,<Green finish>,<Blue finish>
```

The logical colour numbers of the screen background and the font foreground make up the first two parameters. The next six parameters specify two colours, 'start' and 'finish'. These are both defined in terms of their red, green and blue components.

The start colour is used as the darkest shade when painting the font. This will be the physical colour required as the background to the font. The 'finish' colour is the lightest shade to be used in painting the font. This will be the foreground colour. The font painter will then define each of the logical anti-aliasing colours to lighten, beginning at the start colour and ending at the finishing colour.

An example should make this clear. Suppose that we wanted to paint a font in white and on a black background. The screen background will normally be logical colour '0', so the first parameter to the VDU command is '128+0'. By default the logical colour for the foreground is '1' and so this is the second parameter to the command.

Next, we must define the physical 'start' and 'finish' colours. Since our background is black, the darkest shading colour is also black. This is our 'start' colour. This is represented in RGB terms as '0,0,0'. We also want the foreground colour to be white. This will, therefore, be our lightest shade

and is thus the 'finish' colour. The last three parameters will, therefore, be '255,255,255'. This is the RGB representation of white.

The complete command required is as follows:

```
VDU 23,25,128,1,0,0,0,255,255,255
```

Try typing this statement after listing 19.1 has been run, and see the 'random colours' in the font take on their correct shades of grey.

Listing 19.2 shows some effects of specifying different physical colours as the 'start' and 'finish' values of the shading range.

Listing 19.2. Demonstration of anti-aliasing shading.

```
  10 REM Example of the painting text using the font manager
  20 REM Note : Welcome Disc must be in drive
  30 REM Demo shows how anti-aliasing colours can be defined
  40 REM (c) Michael Ginns 1988
  50 REM Dabs Press : Archimedes Assembly Language
  60 REM
  70
  71 *SET Font$Prefix $.Fonts
  72
  80 DIM fonts 256
  90
 100 x_point_size = 480 : REM horizontal point size
 110 y_point_size = 320 : REM vertical point size
 120 handle = 10
 130
 140 FOR pass = 0 TO 3 STEP 3
 150 P% = fonts
 160 [
 170 OPT pass
 180
 190 ADR R1,font_name        ; Path name of font file on disc
 200 MOV R2,#x_point_size
 210 MOV R3,#y_point_size
 220 MOV R4,#0               ; Default x,y screen resolution
 230 MOV R5,#0
 240
 250 SWI "Font_FindFont"     ; Get font into cache and initialise
 260 MOV handle,R0           ; Font handle returned in R0
 270
 280 ADR R1,text             ; Text to be painted on screen
 290 MOV R2,#20              ; Plot mode - absolute OS co-ords
 300 MOV R3,#0               ; Paint text starting at (0,600)
 310 MOV R4,#600
 320 SWI "Font_Paint"        ; Paint the text
 330
```

```
340 MOV R0,handle          ; Put font handle into R0
350 SWI "Font_LoseFont"    ; Finished - inform manager
360
370 MOV PC,R14 ; Back to BASIC
380
390 .font_name ; Name of font file on disc
400 EQUS "Trinity.Medium"
410 EQUB 13
420 .text                  ; Text message to be painted
430 EQUS "This is text produced by the FONT system"
440 EQUB 13
450 ]
460 NEXT
470
480 MODE 12
490
500 CALL fonts
510
520 REM Re-define anti-aliasing colours
530
540 PRINT TAB(0,20) "Press a key to change colours:"
550
560 FOR colours = 0 TO 10
570 READ Rstart,Gstart,Bstart
580 READ Rfinish,Gfinish,Bfinish
590 VDU 23,25,128,1,Rstart,Gstart,Bstart,Rfinish,Gfinish,Bfinish
600 key = GET
610 NEXT
620
630 DATA 0,0,0,255,0,0          :REM Red
640 DATA 0,0,0,0,255,0          :REM Green
650 DATA 0,0,0,0,0,255          :REM Blue
660 DATA 0,0,0,255,255,0        :REM Yellow
670 DATA 0,0,0,255,0,255        :REM Magenta
680 DATA 0,0,0,0,255,255        :REM Cyan
690 DATA 0,0,0,255,255,255      :REM White
700 DATA 0,0,0,255,140,0        :REM Gold
710 DATA 48,0,48,255,0,255      :REM Magenta on purple
720 DATA 0,0,255,255,255,255    :REM White on blue
730 DATA 255,255,255,0,0,0      :REM Black on white
```

The Anti-aliasing Transfer Function

In the previous discussions, we assumed that the font painter uses 16 logical colours to produce anti-aliasing shading. This gives the best results for a given font. However, this may not always be desirable. The usual screen mode for producing fonts is mode 12. This offers the maximum number of colours at the highest resolution without involving the complexities of the 256 colour modes.

In mode 12 there are normally 16 logical colours available. However, if we paint characters using 16 colour levels for anti-aliasing, every colour in the mode is taken up. Each of the 16 colours will be redefined by the font painter to be a shade of the colour being painted. This means that we can only display fonts in one colour!

If single colour text is all that is required, the rest of this section can be ignored! However, if we want to paint different coloured text, on the same screen, then we must reduce the number of colours taken by the anti-aliasing function. Again, the font painter provides a VDU command to do this. The syntax of the command is as follows:

```
VDU 23,25,<bits>,threshold 1,threshold 2, ... threshold 7
```

In the command, bits specifies the number of bits to be used to represent the anti-aliasing colours and must be in the range one to four. The relationship between bits and colours used is given in figure 19.2.

Number of bits	Colours used for anti-aliasing
1	2
2	4
3	8
4	16

Figure 19.2 The relationship between bits and colours.

Using this command, we can specify that less than 16 colours should be used for anti-aliasing. If we do this, the threshold values are used to determine how the original 16 anti-aliasing colours should map to the reduced number of colours used. This is best illustrated by an example as follows:

```
VDU 23,25,2,4,8,12,0,0,0,0
```

This specifies two bits and therefore four colours to be used for anti-aliasing. We then define that, of the original 16 anti-aliasing colour numbers, any less than four will be translated into colour one, any between four and eight will be colour two, any between eight and 12 will be colour three and any greater than 12 will be colour four.

By restricting the number of colours used for anti-aliasing, we increase the number of colours in which we can display fonts on the same screen. For example, using four anti-aliasing colours means that we can now have

paint characters in four different physical colours. Each of these physical painting colours uses four logical colours for anti-aliasing shading. Thus, we again use the maximum number of 16 colours available in the mode.

In the next sections we see how to change the font painting colour. An example of setting the anti-aliasing transfer function is given there.

Changing the Painting Colour

To select the colour in which a font is painted we use:

```
VDU 17,<col>
```

Where col is the logical colour to be painted. The VDU sequence must be outputted together with the text being painted in the font. To do this, we include the two control characters 17 and col in the string pointed to by R1 when SWI Font_Paint is used.

Remember that the font will use logical colours col, col+1, col+2 ... col+n, where n is the number of colours used for anti-aliasing. The program in listing 19.3 paints characters in four different colours, each colour using four anti-aliasing shades.

Listing 19.3. Painting text in different colours.

```
 10 REM Example of the painting text using the font manager
 20 REM Note : Welcome Disc must be in drive
 30 REM Demo shows text Painting in several colours
 40 REM (c) Michael Ginns 1988
 50 REM Dabs Press : Archimedes Assembly Language
 60 REM
 70
 71 *SET Font$Prefix $.Fonts
 72
 80 DIM fonts 1024
 90
100 vdu = 256
110 x_point_size = 480 : REM horizontal point size
120 y_point_size = 320 : REM vertical point size
130 handle = 10
140
150 FOR pass = 0 TO 3 STEP 3
160 P% = fonts
170 [
180 OPT pass
190
200 ADR R1,font_name       ; Path name of font file on disc
```

```
210  MOV R2,#x_point_size
220  MOV R3,#y_point_size
230  MOV R4,#0              ; Default x,y screen resolution
240  MOV R5,#0
250
260  SWI "Font_FindFont"    ; Get font into cache and initialise
270  MOV handle,R0          ; Font handle returned in R0
280
290  ADR R1,text            ; Text to be painted on screen
300  MOV R2,#20             ; Plot mode - absolute OS co-ords
310  MOV R3,#0              ; Paint text starting at (0,600)
320  MOV R4,#600
330  SWI "Font_Paint"       ; Paint the text
340
350  ADR R1,text2           ; Text to be painted on screen
360  MOV R2,#20             ; Plot mode - absolute OS co-ords
370  MOV R3,#0              ; Paint text starting at (0,500)
380  MOV R4,#500
390  SWI "Font_Paint"       ; Paint the text
400
410  ADR R1,text3           ; Text to be painted on screen
420  MOV R2,#20             ; Plot mode - absolute OS co-ords
430  MOV R3,#0              ; Paint text starting at (0,400)
440  MOV R4,#400
450  SWI "Font_Paint"       ; Paint the text
460
470  ADR R1,text4           ; Text to be painted on screen
480  MOV R2,#20             ; Plot mode - absolute OS co-ords
490  MOV R3,#0              ; Paint text starting at (0,300)
500  MOV R4,#300
510  SWI "Font_Paint"       ; Paint the text
520
530  MOV R0,handle          ; Put font handle into R0
540  SWI "Font_LoseFont"    ; Finished - inform manager
550
560  MOV PC,R14             ; Back to BASIC
570
580  .font_name             ; Name of font file on disc
590  EQUS "Trinity.Medium"
600  EQUB 13
610
620  .text                  ; 1st message in colour 1
630  EQUB 17                ; Select text colour 1
640  EQUB 1
650  EQUS "This is Pink FONT Text"
660  EQUB 13
670
680  .text2                 ; 2nd message in colour 4
690  EQUB 17                ; Select text colour 4
700  EQUB 4
710  EQUS "This is Gold FONT Text"
720  EQUB 13
```

```
730
740 .text3                  ; 3rd message in colour 8
750 EQUB 17                 ; Select text colour 8
760 EQUB 8
770 EQUS "This is Blue FONT Text"
780 EQUB 13
790
800 .text4                  ; 4th message in colour 12
810 EQUB 17                 ; Select text colour 12
820 EQUB 12
830 EQUS "This is White FONT Text"
840 EQUB 13
850
860 ]
870 NEXT
880
890 MODE 12
900
910 REM Select 4 anti-aliasing colours ie, 2 bits and set
920 REM transfer function to reduce the 16 colours down to 4
930 REM This can be done in machine code using the VDU template
940
950 VDU 23,25,2,4,8,12,0,0,0,0
960
970 REM Define the physical Shading colours for each of the
980 REM 4 logical painting colours (1,4,8,12)
990
1000 VDU 23,25,128,1,0,0,0,255,0,255    : REM Colour 1 as pink
1010 VDU 23,25,128,4,0,0,0,255,140,0    : REM Colour 4 as gold
1020 VDU 23,25,128,8,0,0,0,96,96,255    : REM Colour 8 as blue
1030 VDU 23,25,128,12,0,0,0,255,255,255 : REM Colour 12 as white
1040
1050 CALL fonts
```

Losing Fonts

When a font is no longer required, the following SWI call should be made:

SWI "Font_LoseFont"

Syntax:

```
SWI "Font_LoseFont"
```

On entry: R0 = font's handle

This will inform the font manager that the font definition can be overwritten if extra cache memory is required for a new font.

20 : Templates and I/O

In the previous chapters we have seen how ARM's instructions may be used. We have also seen something of the facilities provided by the ARTHUR operating system. It should now be possible for us to sit down and write any machine code program which may be required. However, for the beginner this is not always an easy task. Even for the experienced programmer, it isn't always a good idea either!

Assembly programs are, by definition, very low-level. There is little structure imposed on the programmer and, unless you are very careful, programs start to grow haphazardly into a tangled mess of code. It's very easy to get bogged down with the details of instructions, registers, memory allocation and other implementation. This often results in obscure overall logic and program structure.

Programs written in this way are fine, as long as they work first time and never need modifying! Unfortunately, this is seldom the case. Trying to debug such a program is time-consuming and filled with difficulties. Often, a re-write is the only solution.

What we need is a more systematic way of turning high-level program designs into assembler statements. The use of 'templates' provides a partial solution to this problem. A template is a section of assembly code which implements, at least in outline, a single high-level statement or construct. In our case, we will consider templates to model statements in BBC BASIC.

When a template has been written, it can be included into our assembly code program each time the program design calls for the corresponding high-level construction to be used. For example, each time we need to use a FOR...NEXT loop in machine code, we can simply copy the relevant instructions from the FOR...NEXT template.

The use of templates has a number of advantages as follows:

1) We can design programs in terms of high-level constructions (usually BASIC). This allows us to get the overall logic and structure

of the program correct without having to worry about the details of the assembly code.

2) By using templates each time a construct is needed, we produce much more consistent code which is less likely to contain errors.

3) Any errors which do occur in the program are much easier to track down. If we know that a template is correct and we have used it consistently throughout, then we do not need to check each occurrence of it within the program.

4) Finally, the process of writing assembly programs is made easier and faster by using templates. There is no need to 're-invent' a section of code each time we use it.

Obviously, templates do not provide a complete solution to writing machine code programs. Programming in assembly language is different to BASIC and these differences must be understood. However, templates do help to give a little structure and order to our programs. Also, for beginners, they provide an excellent way of bridging the seemingly uncrossable gap between designing BASIC programs and machine code ones.

In the following chapters, templates are developed for many of the statements available in BASIC. The statements are logically grouped and the following are all covered:

Input/Output

INPUT
PRINT
SPC
GET
POS
TAB
INKEY
VPOS

String Manipulation

String representation
String assignment

String concatenation
String comparison

LEN
INSTR
LEFT$
STRING$
RIGHT$
VAL
MID$
STR$

Miscellaneous Statements

SGN
DIV
AND
EOR
ARRAYS
SOUND
ABS
MOD
OR
NOT

Control Constructs

IF...THEN...ELSE...ENDIF
LOGICAL AND/OR
REPEAT...UNTIL
WHILE...ENDWHILE
FOR...NEXT
CASE
PROCEDURES

Graphics

VDU
DRAW
RECTANGLE
CLS
POINT()

PLOT
BY
FILL
CLG
ON
MOVE
LINE
ORIGIN
COLOUR
OFF
POINT
CIRCLE
MODE
GCOL
WAIT

Before describing the templates, it is important to note a few general conventions relating to them.

Template Format

When presenting the templates, we shall often give only a fragment of an assembly code program. This shows the assembly statements which implement the template but do not necessarily include all the assembler formalities to make a complete assembled program. In other cases, where appropriate, a full program may be given which provides a real example of how the template may be used.

Register use

As well as specifying registers by number, we have seen that the assembler also allows us to specify registers by a name. This is done using a variable which has been set up to contain the number of the register with which it is associated. Thus, throughout our program we refer to a register called 'file_handle'. At the beginning of the program we could set 'file_handle' to one. This would cause the assembler to use register R1 whenever 'file_handle' is quoted.

This system makes programs more readable and will often be adopted in the template programs. When a template is presented as a program fragment, it is up to the programmer to allocate real register numbers to the

names used. This can be done in any way required as long as each unique register name has a unique register number associated with it.

There is an exception! This happens when we need to use specific registers. For example, when SWI calls are made, and specific registers are used, to pass data to and from the operating system. In these cases, fixed register numbers may be given in the code. Alternatively, names can still be used. In this case, the statements at which a register number is assigned to the name will be marked by a comment. This shows which register numbers are fixed and must not be changed.

Input/Output

The first set of assembly templates we will look at perform simple input/output operations. The operating system provides considerable support for this and SWI routines are frequently used.

INPUT

BASIC's input statement can be used to enter strings or numbers into programs. In our assembler template we restrict ourselves to entering strings. These can be processed using the VAL template to convert them to integer numbers if required.

To implement INPUT in assembly code, we use the operating system's OS_ReadLine routine. This allows a complete string to be entered and stored in memory. The parameters required for this are described in Chapter 17. The maximum line length for the input, and the maximum and minimum acceptable ASCII values of entered characters, can all be specified.

OS_ReadLine will accept characters from the input stream and store them consecutively in memory. Delete will remove the last character entered, and pressing CTRL-U will delete the whole input line. If more than the maximum permissible number of characters are entered, a 'beep' is issued and no further characters are accepted. The routine terminates when RETURN is pressed, or a new line (ASCII 10) is entered. The end of the string in memory is always marked by a return (ASCII 13) character.

It is vital that a suitable area of memory is reserved to act as a buffer for the entered characters. DIM or EQU are used for this in most cases. Listing 20.1 shows a very simple use of OS_ReadLine. It implements an endless loop which reads a string from the keyboard, then writes it out on screen.

Listing 20.1. INPUT template.

```
 10 REM Example of using the INPUT template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM input 256
 70
 80 REM Define names for registers used
 90 pointer    = 0 : REM Must use register R0
100 max_length = 1 : REM Must use register R1
110 min_ASCII  = 2 : REM Must use register R2
120 max_ASCII  = 3 : REM Must use register R3
130 base       = 4
140
150 REM Two pass assembly
160 FOR pass = 0 TO 3 STEP 3
170 P% = input
180 [
190 OPT pass
200
210 ADR pointer,buffer ; Put line buffer addr in pointer reg
220 MOV max_length,#20 ; Set max line length to 20 characters
230 MOV min_ASCII,#32  ; Minimum acceptable ASCII code is 32
240 MOV max_ASCII,#128 ; Maximum acceptable ASCII code is 127
250
260 SWI "OS_ReadLine"  ; Input a line of text
270
280 ; Print each character previously entered into the buffer
290
300 ADR base,buffer    ; Get line buffer start addr in base
310 .print_loop        ; Loop to output each char in buffer
320 LDRB R0,[base],#1  ; Get next char (uses post index addr)
330 SWI "OS_WriteC"    ; Output the character
340 CMP R0,#13         ; See if we are at the end of the line
350 BNE print_loop     ; If not branch to output next char
360 SWI "OS_NewLine"   ; Output a newline
370 B input            ; Repeat the entire program
380
390 .buffer            ; Reserve 32 spaces for the line buffer
400 EQUS STRING$(32,CHR$(0))
410
420 ]
430 NEXT
440
450 PRINT '' "Enter text lines now!" ''
460 CALL input
```

GET

The GET function makes the computer wait until a character is in the keyboard buffer, then returns the ASCII value of it. The operating system routine performing this task is called OS_ReadC. This returns, when a key has been pressed, with the ASCII code of the key in register R0. Full details are again given in Chapter 17:

```
[
SWI "OS_ReadC" ; Read Character - ASCII value in R0
]
```

INKEY

This statement is similar to GET except that it will wait for a key to be pressed OR until a pre-determined time interval has elapsed – which ever happens first. The command can also check whether or not a specific key is depressed on the keyboard. OSBYTE call number 129 is used to do this.

As usual with OSBYTE, the R0 register is used to pass the number of the routine to be used – in this case 129. To read a key within 't' centi-seconds, registers R1 and R2 are set up as follows:

```
R1 = t MOD 256
R2 = t DIV 256
```

When the routine returns, the contents of R1 and R2 contain a return result which shows what happened. This is interpreted as:

Contents of R2	Result
0	A key was pressed within the time limit. The ASCII value of the character is held in register R1
255	The specified limit expired before any key was pressed
27	The ESCAPE key was pressed

To check on whether or not a specific key is pressed, the call is used in a slightly different way. On entry to the routine, R1 and R2 are set up in the following way:

R1 = The negative INKEY number of the key
R2 = 255

A full list of the negative INKEY numbers for every key is included in the Archimedes User Guide, so we will not go into it here.

No time limit is specified when the routine is used in this way. It immediately terminates and returns whether or not the specified key was pressed at that moment. If registers R1 and R2 contain 255, then the specified key was pressed, otherwise the key was not pressed.

Listing 20.2 gives an example of using INKEY in machine code programs. It contains a loop which repeatedly waits for a key to be pressed within a time limit of one second. If a key is pressed then the character is echoed on the screen. The program also issues a beep after each call of INKEY, irrespective of whether a key was pressed or not. The effect is that the program will beep every second or after each key press.

Listing 20.2. Demonstration of INKEY from machine code.

```
 10 REM Example of the INKEY template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM inkey 256
 70
 80 REM Define constants
 90 vdu  = 256
100 beep = 7
110
120 P% = inkey
130 [
140
150 .loop
160 MOV   R0,#129
170 MOV   R1,#100
180 MOV   R2,#0
190 SWI   "OS_Byte"
200 CMP   R2,#0
210 MOVEQ R0,R1
220 SWIEQ "OS_WriteC"
230 SWI vdu + beep
240 B loop
250 ]
260
270 PRINT "Enter characters now !!"
280 CALL inkey
```

PRINT

The actions of the BASIC PRINT statement are too varied to be represented by a single assembly language template! There are a series of SWI calls which perform some of the PRINT facilities. Calls are included to perform the following:

Print single characters
Print strings of characters
Print signed integer numbers

Again, Chapter 17 contains details of the appropriate SWI routines which perform these operations.

POS and VPOS

These two functions return the horizontal and vertical positions of the text cursor on the screen. The operating system provides a routine to perform the same operation. It is an OSBYTE call, number 134.

OSBYTE 134 takes no entry parameters and returns with the cursor's x and y co-ordinates in registers R1 and R2 respectively. The code to obtain these positions is as follows:

```
[
MOV R0,#134     OSBYTE call number is 134
SWI "OS_Byte"   POS and VPOS returned in R1 and R2
]
```

This routine is used in the template for implementing the TAB() function.

SPC(n)

This statement takes a single integer argument and outputs that number of spaces on the screen. In assembly language, we represent this by a simple loop to output the correct number of spaces. This is shown in listing 20.3. The number of spaces required is assumed to be contained in the register called 'n'. In this example, 17 spaces are outputted followed by a '*' – so that you can see the effect of the spaces!

Listing 20.3. SPC(n) template.

```
 10 REM Example of the SPC(n) template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 n = 0          : REM Reg containing no. of spaces for output
 70
 80 vdu   = 256    : REM Start no. of SWI block to perfrom VDU n
 90 space = 32     : REM ASCII code for a space character
100 star  = 42     : REM ASCII code for a '*' character
110
120 DIM  spc 256
130
140 FOR pass = 0 TO 3 STEP 3
150 P% = spc
160 [
170 OPT pass
180
190 MOV n,#17      ; As an example, do SPC(17)
200
210 .space_loop    ; Loop to output required spaces
220 CMP n,#0       ; See if all space have been output
230 BEQ finished   ; If they have then branch to end of routine
240 SWI vdu+space  ; VDU 32
250 SUB n,n,#1     ; Dec 'n' (the no. of spaces to be output)
260 B space_loop   ; Branch back to beginning of routine
270 .finished      ; End of routine label
280
290 SWI vdu+star   ; VDU 42
300
310 MOV PC,R14     ; Back to BASIC
320
330 ]
340 NEXT
350
360 PRINT '' "Performing SPC(17)"
370 CALL spc
```

TAB

There are two forms of the TAB statement. The first takes a single argument specifying the horizontal position of the TAB. It then outputs enough spaces to reach this position on the screen. If the cursor is already beyond the specified position, then a newline is issued.

Obviously, in order to implement this statement, we must have some way of knowing where the text cursor is! BASIC uses the COUNT variable.

However, in machine code the nearest we can get is to read the text cursor's position using the routine in the POS template.

We can then subtract the current cursor position from the new TAB position and calculate the number of spaces. If the result is negative, we are already beyond the required TAB position and a new line must be outputted. Note that the spaces needed are outputted using the SPC statement template. You can see how the creation of standard templates is already becoming useful in creating more complex routines!

The assembly code routine to perform TAB is illustrated in listing 20.4. It assumes that the position to be TABed to is contained in the register called tab_pos. As an example, various strings are outputted from BASIC. After each the TAB(n) routine is called, column 32 is TABed to, and a star is outputted to show the new position.

Listing 20.4. TAB(n) template.

```
 10 REM Example of the TAB(n) template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 51 REM Define names for registers
 60 n = 1          : REM No. of spaces to be output
 61 tab_pos = 3    : REM Desired TAB position
 70
 80 vdu   = 256    : REM Start no. of SWI block to do VDU n
 90 space = 32     : REM ASCII code for a space character
100 star  = 42     : REM ASCII code for a '*' character
110
120 DIM  tab 256
130
140 FOR pass = 0 TO 3 STEP 3
150 P% = tab
160 [
170 OPT pass
180
190 MOV tab_pos,#32     ; As an example, do TAB(32)
200
210 MOV R0,#134         ; Use POS template to get cursor position
211 SWI "OS_Byte"       ; x position returned in register 'n'(R1)
212
213 SUBS   n,tab_pos,n ; Do currect postion - required position
214 MOVMI  n,tab_pos    ; If negative result restore position
215 SWIMI  "OS_NewLine"; and output a newline
216
217
220 ; Template to perfrom SPC(n)
```

```
221 ; 'n' is the number of space to reach required TAB position
222
230 .space_loop         ; Loop to output required spaces
240 CMP  n,#0           ; See if all space have been output
250 BEQ  finished       ; If so branch to end of routine
260 SWI  vdu+space      ; VDU 32
270 SUB  n,n,#1         ; Decrement 'n'
280 B    space_loop     ; Branch back to beginning of routine
290 .finished           ; End of routine label
300
301
310 SWI  vdu+star       ; VDU 42
320 SWI  "OS_NewLine"   ; Print a newline
330 MOV  PC,R14         ; Back to BASIC
340
350 ]
360 NEXT
370
380 PRINT "Hello !";
390 CALL tab
400 PRINT "That was a TAB(32)";
401 CALL tab
402 PRINT "Any number could be used - 32 is only an example";
403 CALL tab
404 PRINT "That line was already past position 32";
405 CALL tab
406 PRINT "And so was that one!";
407 CALL tab
```

The second form of the TAB statement is as follows:

```
TAB(x,y)
```

This causes the text cursor to move directly to the position x,y on the screen. Surprisingly, this is a much easier function to implement because the operating system provides a VDU command to do it for us! VDU 31,<x>,<y> will place the text cursor at the position x,y. In assembly language we simply output character 31, followed by the new position of the cursor. Listing 20.5 illustrates this. The registers called x and y are assumed to contain the new screen position. In this example the cursor is moved to position 10,15 on the screen.

Listing 20.5. TAB(x,y) template

```
10 REM Example of the TAB(x,y) template
20 REM (c) Michael Ginns 1988
30 REM Dabs Press : Archimedes Assembly Language
40 REM
50
60 REM Define register names
```

```
 70  x = 1                : REM x position
 80  y = 2                : REM y position
 90
100  vdu = 256            : REM Start no. of SWI block to do VDU n
110  MoveCursor = 31      : REM Control code - move cursor to (x,y)
120
130  DIM tab2 256
140  P% = tab2
150  [
160
170  MOV x,#10                  ; As an example, peform TAB(10,15)
180  MOV y,#15
190
200  SWI vdu + MoveCursor       ; Do 'move cursor' command (VDU 31)
210  MOV R0,x                   ; Put x pos into register R0
220  SWI "OS_WriteC"            ; Output x pos to VDU drivers
230  MOV R0,y                   ; Put y pos into register R0
240  SWI "OS_WriteC"            ; Output y pos to VDU drivers
250
260  MOV PC,R14 ; Back to BASIC
270
280  ]
290
300  CLS
310  PRINT "Performing TAB(10,15)";
320  CALL tab2
```

21 : Manipulating Strings

In this chapter we will look at the representation and processing of strings in machine code.

Representing Strings

In BASIC, we talk of string variables which contain sequences of characters. In machine code, we do a similar thing. There are 256 different characters available on the Archimedes, each of which has a unique number – its ASCII code. A single character can thus be represented by a single byte of memory containing its ASCII code. Strings can now be represented by storing the characters they contain in consecutive bytes of memory.

So far so good. However, we also need to know how many characters are contained in a string. BASIC solves this problem by storing the length of the string in memory alongside the string itself. The operating system, on the other hand, terminates all its strings with a special 'end of string marker', a character of ASCII code 13, 10 or 0.

We shall adopt the convention that all strings are terminated by a carriage return (ASCII 13). Strings in this form can be produced from BASIC using the following statement:

```
$<var> = <string>
```

Where var is a variable containing the address at which the string is to be stored, and string is the string itself. This will write the characters in the string into consecutive bytes of memory followed by a terminating carriage return (ASCII 13). For example:

```
DIM buffer 256
$buffer = "This string is stored in memory at buffer"
```

It is worth remembering that literal strings can also be stored in memory using the EQUS directive (see Chapter 13). Preceding this by a label defini-

tion will set the label to the string start address, which can then be loaded into a register using the ADR instruction.

String Manipulation Routines

Each of the various string manipulation templates is in fact presented within a complete program, illustrating how it can be used. The section of code which constitutes the template is marked within the program. The remaining instructions and statements are required only to illustrate the use of the template.

Some registers must always be set up to hold the parameters needed by the various string manipulation routines. The addresses where the strings are stored, for example, must be placed in the appropriate registers. Only after these registers have been set up, can the template be executed. This applies to the usage of all the templates. In the example programs, the template parameters are usually passed from BASIC to the appropriate registers using the corresponding integer variables. (For details of passing data to machine code routines, see Chapter Four.)

It is important to note that none of the routines validate the parameters passed to them. For example, if we ask for a string to be concatenated on to the end of another, then the routine will do it even if this creates a string which is too long for the space allocated, and overwrites other data. It is up to you to include checks on any parameters which could be invalid.

String Assignment

One of the simplest operations we can perform on a string is that of assignment. In our scheme, to assign the contents of one string to another, we simply copy the characters it contains to the memory area allocated to the new string.

Listing 21.1 does exactly this. Characters are accessed sequentially from the first string and stored in the memory area allocated to the second string. Note that the memory load and store instructions are used in byte mode to transfer individual characters. Also, post-index addressing with automatic write back is specified. This increments the addresses being used after each character is copied, so that they always point to the next character along.

The addresses of the source and destination string are assumed to be in the two registers called 'str1' and 'str2'. The destination string is the area of memory which is to contain the copy of the original string. Its previous contents are unimportant as they are overwritten.

Listing 21.1. String assignment.

```
10 REM Example of the string copy template
20 REM (c) Michael Ginns 1988
30 REM Dabs Press : Archimedes Assembly Language
40 REM
50
60 DIM copy 256
70
80 REM define names for the registers used
90 str1 = 0 : REM Source string addr passed in this register
100 str2 = 1 : REM Destination string addr in this register
110 char = 3
120
130 P% = copy
140 [
150 ; The addresses of the two strings are passed into
160 ; registers str1 and str2 from BASIC using A% and B%
170
180 ; ********* String Copy Template ********
190
200 .copy_loop          ; Loop to copy characters
210 LDRB char,[str1],#1  ; Get next character from string 1
220 STRB char,[str2],#1  ; Store in next space in string 2
230 CMP char,#13         ; Check for end of string marker
240 BNE copy_loop        ; If not got to end, then branch back
250
260 ; ********* Template ends *********
270
280 MOV PC,R14           ; Back to BASIC
290 ]
300
310 REM Reserve space for strings and put addresses in A%,B%
320 DIM string1 100
330 DIM string2 100
340 A%= string1
350 B%= string2
360
370 REPEAT
380 INPUT LINE ' "Enter the string to be copied : "  $string1
390 CALL copy
400 PRINT "Destination string contains   : "  $string2
410 UNTIL FALSE
```

String Concatenation

Listing 21.2 allows one string to be concatenated onto the end of another. This is equivalent to the BASIC statement:

```
A$ = A$ + B$
```

The routine works by copying characters from the source string, but this time it copies them on to the end of the destination string. Once again, the addresses of the two strings are assumed to be in registers 'str1' and 'str2'.

Listing 21.2. String concatenation.

```
10  REM Example of the string concatenation template
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  DIM concat 256
70
80  REM  Define names for registers used
90  str1 = 0   : REM Addr of string 1 passed in this register
100 str2 = 1   : REM Addr of string 2 passed in this register
110 char = 3
120
130 P% = concat
140 [
150 ; The addresses of the two strings are passed into
160 ; registers str1 and str2 from BASIC using A% and B%
170
180 ; ********* String Concatenation Template ********
190
200 .find_end               ; Loop to find end of first string
210 LDRB char,[str1],#1     ; Get next character from the string
220 CMP char,#13            ; Have we reached end of string marker
230 BNE find_end            ; If not then keep looking
240 SUB str1,str1,#1        ; Move pointer back to end of string
250
260 .copy_loop              ; Loop: string 2 on end of string 1
270 LDRB char,[str2],#1     ; Get next character from string 2
280 STRB char,[str1],#1     ; Store char in next space in string 1
290 CMP char,#13            ; Has end of string 1 been reached
300 BNE copy_loop           ; If not then keep copying
310
320 ; ********* Template ends *********
330
340 MOV PC,R14              ; Back to BASIC
350 ]
360
370 REM Reserve string space and place addresses in A%,B%
```

```
380 DIM string1 100
390 DIM string2 100
400 A%= string1
410 B%= string2
420
430 REPEAT
440 INPUT LINE '"Enter the first string   :"  $string1
450 INPUT LINE  "Enter string to be added :"  $string2
460
470 CALL concat
480
490 PRINT ' "Concatenating string 1 onto string 2"
500 PRINT   "Result is : " $string1
510 UNTIL FALSE
```

String Comparison

There are occasions when we want to perform comparison operations on strings. In BASIC we can write statements like:

```
IF name1$ > name2$ THEN PROCswap
```

This compares the two strings on the basis of the ASCII codes of the characters they contain.

The template to perform this is presented in listing 21.3. By way of an example, the program allows two strings to be entered, compares them using the routine, then outputs the result of the comparison.

The routine works by successively comparing the ASCII codes of each character pair from the two strings. Special care has to be taken when one string terminates before the other.

Listing 21.3. String comparison.

```
 10 REM Example of the string comparison template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM compare 256
 70
 80 REM Define names for registers used
 90 result = 0  : REM Result of comparison returned here
100 str1   = 1  : REM Addr of string 1 passed in this register
110 str2   = 2  : REM Addr of string 2 passed in this register
120 char1  = 3
```

```
130 char2  = 4
140
150 FOR pass =0 TO 3 STEP 3
160 P% = compare
170 [
180 OPT pass
190
200 ; The addresses of the two strings are passed into
210 ; registers str1 and str2 from BASIC using A% and B%
220 ; Result returned in R0 and passed back to BASIC via 'USR'
230
240 ; ********* String Comparison Template ********
250
260 MOV result,#0          ; Initially no comparison result
270 .comp_chars            ; Loop to compare characters
280 LDRB  char1,[str1],#1  ; Get next character from string 1
290 LDRB  char2,[str2],#1  ; Get next character from string 2
300 CMP char1,char2        ; Compare the two characters
310 MOVGT result,#1    ; If char1 > char2 THEN string1 > string2
320 MOVLT result,#2    ; If char1 < char2 THEN string2 > string1
330 BNE   done         ; If char1<>char2 comparison complete
340 CMP char1,#13      ; If end string 1 then string2 > string1
350 ADDEQ result,result,#2
360 CMP char2,#13      ; If end string 2 then string1 > string2
370 ADDEQ result,result,#1
380 CMP result,#0          ; See if comparison produced a result
390 BEQ comp_chars         ; If not, then keep comparing
400 .done
410
420 ; ********* Template ends *********
430
440 MOV PC,R14             ; Back to BASIC
450 ]
460 NEXT pass
470
480 REM Reserve space for strings and put addresses in A%,B%
490 DIM string1 100
500 DIM string2 100
510 B%= string1
520 C%= string2
530
540 REPEAT
550 INPUT LINE ' "Enter the first string :"  $string1
560 INPUT LINE   "Enter the second string :"  $string2
570
580 Result = USR(compare)
590
600 REM Display result of comparison
610 IF Result = 1 THEN PRINT $string1;" > ";$string2
620 IF Result = 2 THEN PRINT $string2;" > ";$string1
630 IF Result = 3 THEN PRINT $string1;" = ";$string2
640
650 UNTIL FALSE
```

LEN()

The LEN() function returns the current string length. The template for this operates by counting characters in the string until the end of string marker (character 13) is reached. The string length template is contained in listing 21.4. As an example of its use, the program prompts for a string to be entered, calls the LEN routine and then prints out the string length.

Listing 21.4. String length (LEN).

```
 10 REM Example of the LEN template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM len 256
 70
 80 REM Define names for the registers used
 90 length = 0 : REM String length returned in this register
100 str    = 1 : REM Address of string passed in this register
110 char   = 3
120
130 P% = len
140 [
150 ; Addr of strings passed to 'str' from BASIC via A%
160 ; String len returned in  R0 and passed to BASIC via 'USR'
170
180 ; ********* String Length Template ********
190
200 MOV length,#0                ; Initialise length
210 .find_end                    ; Loop to count characters
220 LDRB   char,[str],#1         ; Get next char from string
230 CMP    char,#13              ; Is it end of string marker
240 ADDNE  length,length,#1      ; If not increment length count
250 BNE find_end                 ; If not string end keep going
260
270 ; ********* Template ends *********
280
290 MOV PC,R14                   ; Back to BASIC
300 ]
310
320 REM Reserve space for the string and put address in A%
330 DIM string 100
340 B%= string
350
360 REPEAT
370 INPUT LINE ' "Enter the string :"  $string
380 PRINT "Length of the string is :"; USR(len)
390 UNTIL FALSE
```

LEFT$

Listing 21.5 emulates BASIC's LEFT$ function. It is passed a string and a number, n. It then returns a string consisting of the n left-most characters of the source string. n must be in the range 0 <= n <= LEN(string). If n = 0, an empty string is returned.

The routine works by copying n characters from the start of the source string. It terminates by adding character 13 to form a valid string.

Listing 21.5. LEFT$ template.

```
 10 REM Example of the LEFT$ template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM left 256
 70
 80 REM Define names for the registers used
 90 str1  = 0  : REM Source string addr passed in this reg
100 str2  = 1  : REM Destination string addr passed in this reg
110 n     = 2  : REM Contains 'number of characters to copy'
120 char  = 3
130 count = 4
140
150 P% = left
160 [
170 ; Addr of the two strings passed to registers str1,str2 from
180 ; BASIC via A%,B%. No. of chars to copy passed via C% to 'n'
190
200 ; ********* LEFT$ Template ********
210
220 MOV count,#0           ; Initialise count
230 .copy_loop             ; Loop to copy characters
240 CMP count,n            ; See if all characters copied
250                        ; IF NOT all copied THEN
260 LDRNEB char,[str1],#1  ; Get next character
270 STRNEB char,[str2],#1  ; Store char in destination string
280 ADDNE  count,count,#1  ; Inc 'characters copied' counter
290 BNE    copy_loop       ; Branch and process next character
300                        ; ENDIF
310 MOV    char,#13        ; Terminate the destination string
320 STRB   char,[str2]     ; with the end of string marker
330
340 ; ********* Template ends *********
350
360 MOV PC,R14             ; Back to BASIC
370
```

```
380 ]
390
400 REM Reserve space for strings and put addresses in A%,B%
410 DIM string1 100
420 DIM string2 100
430 A%= string1
440 B%= string2
450
460 REPEAT
470 INPUT ' "Enter the string :"  $string1
480 INPUT   "Enter number of characters :",num
490 C% = num : REM prepare to pass num, via C%, into routine
500
510 CALL left
520
530 PRINT   "Resulting string :"  $string2
540 UNTIL FALSE
```

RIGHT$

Listing 21.6 emulates BASIC's RIGHT$ function. It is passed a string and a number, n. It returns the n right-most characters of the source string. The value of n must be in the range 0 <= n <= LEN(string). If n = 0, an empty string is then returned.

This routine works in the same way as LEFT$, however, this time characters must be taken from the end of the source string. This is done by finding the end of the string marker first, then counting back the required number of characters.

Listing 21.6. RIGHT$ template.

```
 10 REM Example of the RIGHT$ template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM right 256
 70
 80 REM Define names for registers used
 90 str1  = 0 : REM Source string addr passed in this reg
100 str2  = 1 : REM Destination string addr passed in this reg
110 n     = 2 : REM Number of chars to be copied
120 char  = 3
130 count = 4
140
150 P% = right
160 [
```

```
170
180 ;Addrs of 2 strings passed to str1,str2 from BASIC via A%,B%
190 ; Number of chars to be copied passed to register 'n' via C%
200
210 ; ********* RIGHT$ Template ********
220
230 .find_end               ; Loop to find end of string
240 LDRB char,[str1],#1     ; Get next character
250 CMP char,#13            ; Is it the end of string marker
260 BNE find_end            ; If not then keep looking
270
280 .copy_characters        ; Loop to copy characters
290 LDRB char,[str1,#-1]!   ; Get next character from the right
300 STRB char,[str2,n]      ; store char in destination string
310 SUBS n,n,#1             ; Dec 'characters copied' counter
320 BPL copy_characters     ; If chars still to be copied, branch
330
340 ; ********* Template ends *********
350
360 MOV    PC,R14           ; Back to BASIC
370
380 ]
390
400 REM Reserve space for the strings and put addresses in A%,B%
410 DIM string1 100
420 DIM string2 100
430 A%= string1
440 B%= string2
450
460 REPEAT
470 INPUT ' "Enter the string :"  $string1
480 INPUT   "Enter number of characters :",num
490 C% = num  : REM prepare to pass num to routine via C%
500
510 CALL right
520
530 PRINT   "Resulting string :"  $string2
540 UNTIL FALSE
```

MID$

The MID$ function is used to extract characters from the middle of a string. It is passed a string and two numbers, p and n. It will then return n characters from the string, starting at position p. The template to do this is presented in listing 21.7. Note that p and n must be chosen, so that there are n characters in the source string starting at position p, ie, p+n <= LEN(string)

The routine is similar to the LEFT$ template, except that copying takes place starting at the position denoted by p.

Listing 21.7. MID$ template.

```
 10 REM Example of the MID$ template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM mid 256
 70
 80 REM Define names for registers used
 90 str1  = 0 : REM Source string addr passed in this reg
100 str2  = 1 : REM Destination string addr passed in this reg
110 n     = 2 : REM Contains number of characters to be copied
120 p     = 3 : REM Contains the position to start copying from
130 char  = 4
140 count = 5
150
160 P% = mid
170 [
180
190 ; Addresses of two strings passed into str1,str2 from BASIC
200 ; via A%,B%. Number of characters to be copied passed to
210 ; register n via C%. Start position passed to reg p via D%
220
230 ; ********* MID$ Template ********
240
250 MOV    count,#0          ; Initialise count
260 ADD    str1,str1,p       ; Add start position to string addr
270 CMP    p,#0              ; correct for '0' positions !!
280 SUBNE  str1,str1,#1
290
300 .copy_loop               ; Loop to copy characters
310 CMP    count,n           ; Have all chars have been copied
320                          ; IF not all copied THEN
330 LDRNEB char,[str1],#1    ; Get next character
340 STRNEB char,[str2],#1    ; Store char in destination string
350 ADDNE  count,count,#1    ; Inc 'characters copied' count
360 BNE    copy_loop         ; Branch to process next char
370                          ; ENDIF
380 MOV    char,#13          ; Terminate the destination string
390 STRB   char,[str2]       ; with end of string marker
400
410 ; ********* Template ends *********
420
430 MOV    PC,R14            ; Back to BASIC
440
450 ]
460
470 REM Reserve space for strings and put addresses in A%,B%
480 DIM string1 100
490 DIM string2 100
500 A%= string1
```

```
510  B%= string2
520
530  REPEAT
540  INPUT ' "Enter the string :"  $string1
550  INPUT   "Enter start position" , pos
560  INPUT   "Enter number of characters :",num
570  C% = num  : REM prepare to pass num to routine via C%
580  D% = pos  : REM prepare to pass pos to routine via D%
590
600  CALL mid
601
610  PRINT   "Resulting string :"  $string2
620  UNTIL FALSE
```

INSTR

INSTR takes two strings and attempts to find the position of the second string in the first. If it succeeds, the position of the second string is returned. If the string could not be found, 0 is returned. A number is also given, p, which signifies the position in the first string from which the search should begin. P must be in the range 0 < p <= LEN(string).

A template for INSTR is given in listing 21.8. The routine works by using two nested loops. The outer loop moves through each character in the source string. Starting from each of these position, the inner loop compares characters with the search string to see if they are the same. If all the characters in the search string are successfully matched, the string has been found and its start position is returned. However, as soon as two characters are found to be different, the comparison fails. The inner loop terminates and the outer loop moves to the next position.

Listing 21.8. INSTR template.

```
 10  REM Example of the INSTR template
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM instr   256
 70
 80  REM Define names for registers used
 90  count = 0 : REM Used to return result of INSTR
100  str1  = 1 : REM Source string addr passed in this reg
110  str2  = 2 : REM Destination string addr passed in this reg
120  n     = 3 : REM Contains the start position for INSTR
130  ptr1  = 4
140  ptr2  = 5
```

```
150 char1 = 6
160 char2 = 7
170
180 FOR pass = 0 TO 3 STEP 3
190 P% = instr
200 [
210 OPT    pass
220
230 ; Addrs of two strings passed to registers str1,str2 from
240 ; BASIC via B%,C%. Start position passed to reg n via D%.
250 ; Result returned in R0 and passed back to BASIC via 'USR'
260
270 ; ********* INSTR Template ********
280
290 CMP    n,#0             ; Correct for '0' start position !!
300 SUBNE  n,n,#1
310 MOV    count,n          ; Initialise count
320 ADD    str1,str1,n      ; Add start position to string addr
330
340 .compare_strings        ; Loop: compare strings at str1,str2
350 MOV    ptr1,str1        ; Make working copies of str1,str2
360 MOV    ptr2,str2
370
380 ADD    count,count,#1   ; Inc 'current position' count
390
400 .compare_chars          ; Loop: compare chars in strings
410 LDRB   char1,[ptr1],#1  ; Get character from string 1
420 LDRB   char2,[ptr2],#1  ; Get character from string 2
430 CMP    char2,#13        ; Has string 2 has been completed
440 BEQ    found_it         ; If so, it was found in string 1
450 CMP    char1,char2      ; Compare next two characters
460 BEQ    compare_chars    ; If same, compare next two chars
470
480 LDRB   char1,[str1],#1  ; Get char from string1, inc'ing str1
490 CMP    char1,#13        ; See if string1 has ended
500 BNE    compare_strings  ; If not, compare strings at new str1
510
520 MOV    count,#0         ; String 2 not found so return '0'
530 .found_it
540
550 ; ********* Template ends *********
560
570 MOV PC,R14
580 ]
590 NEXT
600
610 REM Reserve space for strings and put addresses in B%,C%
620 DIM string1 100
630 DIM string2 100
640 B%= string1
650 C%= string2
660
670 REPEAT
```

```
680  INPUT ' "Enter the string :"  $string1
690  INPUT    "Enter substring  :"  $string2
700  INPUT    "Enter start position :",pos
710  D% = pos : REM Prepare to pass pos to routine via D%
720
730  PRINT    "Result : "  USR(instr);
740
750  UNTIL FALSE
```

STRING$

String$ is used to create a new string by concatenating multiple copies of another string together. It takes the string to be copied and a number, n, which is the number of copies to be made. It returns a string consisting of 'n' copies of the original string.

The template to implement this is given in listing 21.9. It consists of two nested loops. The inner loop makes a copy of the source string on the end of the destination string. The outer loop repeats this to create the required number of copies.

Listing 21.9. STRING$ template.

```
 10  REM Example of the STRING$ template
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM replicate 256
 70
 80  REM Define names for registers used
 90  str1  = 0 : REM Source string addr passed in this reg
100  str2  = 1 : REM Destination string addr passed in this reg
110  count = 2 : REM No of copies to be made passed in this reg
120  ptr   = 3
130  char  = 4
140
150  FOR pass = 0 TO 3 STEP 3
160  P% = replicate
170  [
180  OPT pass
190
200  ; Addresses of two strings passed to str1,str2 via A% and B%
210  ; No. of copies of string is passed to reg 'count' via C%
220
230  ; ********* STRING$  Template ********
240
250  .rep_string            ; Loop to copy string 'n' times
```

```
260 CMP     count,#0          ; See if enough copies made
270 BEQ     finish            ; If so then branch to end
280 SUB     count,count,#1    ; Dec 'number of copies' counter
290 MOV     ptr,str1          ; Copy start of string pointer
300
310 .copy_string              ; Loop to characters in string
320 LDRB    char,[ptr],#1     ; Get next char from source string
330 CMP     char,#13          ; See if the end of string marker
340                           ; IF NOT end of string marker THEN
350 STRNEB char,[str2],#1     ; Store char in destination string
360 BNE     copy_string       ; Branch to process next character
370                           ; ENDIF
380 B       rep_string        ; Branch to copy string again
390
400 .finish
410 MOV     char,#13          ; Terminate the destination string
420 STRB    char,[str2]       ; with end of string marker
430
440 ; ********* Template ends *********
450
460 MOV PC,R14                ; Back to BASIC
470
480 ]
490 NEXT
500
510 REM Reserve space for strings and put addresses in A%,B%
520 DIM string1 100
530 DIM string2 100
540 A%= string1
550 B%= string2
560
570 REPEAT
580 INPUT LINE ' "Enter the first string :"  $string1
590 INPUT LINE  "Number of repeats :"  num
600
610 C% = num
620 CALL replicate
630
640 PRINT   "Result is : "$string2
650 UNTIL FALSE
```

VAL()

The VAL function interprets a string of characters as a sequence of numeric digits. It attempts to evaluate the number represented by these and, if it succeeds, returns the number.

The template to do this in machine code is given in listing 21.10. It uses an operating system SWI routine to perform the conversion. This routine,

however, can only deal with positive numbers. The template, therefore, contains some pre-processing code to check to see if a minus or plus sign precedes the numeric string. If this is the case, the sign of the number is noted and an unsigned string is passed to the SWI routine. After the conversion, the resulting number's sign is modified accordingly.

The string of digits can be prefixed by an optional code which specifies the base in which the number is given. This is done as follows:

```
<base>_<number>
```

The base is a number specifying the number base and can range from two to 36. For example, the following are all legal strings for processing by the VAL template:

2_11101010001	Base 2	(binary)
-2_10101010010	Base 2	(binary negative number)
16_FFEE	Base 16	(hexadecimal)
&FFEE	Base 16	(alternative for hexadecimal)
-8_777	Base 8	(octal negative number)
20_10G	Base 20	

Listing 21.10. VAL template.

```
 10 REM Example of the VAL template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM val 256
 70
 80 REM Define names for registers used
 90 str    = 1 : REM Address of string passed in this register
100 result = 2 : REM Numeric result returned in this register
110 neg    = 3
120 char   = 4
130
140 P% = val
150 [
160 ; The address of the string is passed into str via B%
170 ; The result of the conversion is produced in register R2
180
190 ; ********* VAL template *********
200
210 MOV   neg,#0            ; Initialise negative flag
220 .skip_spaces            ; Loop: skip leading spaces
230 LDRB  char,[str],#1     ; Get next character from string
240 CMP   char,#32          ; See if it is a space
```

```
250 BEQ   skip_spaces        ; If so branch to get next char
260
270 CMP   char,#ASC("-")     ; Is first non space char a '-'
280 MOVEQ neg,#1             ; If so set the negative flag
290 CMPNE char,#ASC("+")     ; If not '-' then see if it is '+'
300 SUBNE str,str,#1         ; If NOT '-' or '+' go back a char
310
320 MOV   R0,#10             ; Base is 10 - may be any from 2-36
330 SWI   "OS_ReadUnsigned" ; Call SWI to convert number
340
350 CMP   neg,#1             ; See if the negative flag is set
360 RSBEQ result,result,#0  ; If so, make result negative
370
380 ; ********* Template ends *********
390
400 MOV R0,result            ; Result in R0, return with 'USR'
410 MOV PC,R14               ; Back to BASIC
420 ]
430
440 REM Reserve space for the string and put address in B%
450 DIM string 100
460 B%= string
470
480 REPEAT
490 INPUT LINE ' "Enter the string :"  $string
500 PRINT "VAL of string is :" USR(val)
510 UNTIL FALSE
```

STR$

The STR routine performs the reverse operation to VAL. It takes an integer, n, and returns a string of numeric digits which represent n. If n is negative, the returned string will contain a minus sign as its first character. The template to perform STR$ is given in listing 21.11. It relies on a SWI call to perform the conversion.

Listing 21.11. STR$ template.

```
 10 REM Example of the STR$ template
 20 REM (c) Michael Ginns 1987
 30 REM DABS Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM ConvertStr 256
 70
 80 REM Define names for registers
 90 number = 0 : REM Used to pass no. for conversion (R0)
100 str    = 1 : REM Address of string passed in this register
110 char   = 4
```

```
120
130 P% = ConvertStr
140 [
150 ; Address of the string passed to R1 via B%
160 ; Number to be converted passed to R0 vi A%
170
180 ; ********* STR$ Template ********
190
200 MOV R2,#100                 ; Size of string buffer
210 SWI "OS_BinaryToDecimal"    ; Call convertion SWI
220 MOV char,#13                ; Terminate string by adding
230 STRB char,[R2,str]          ; end of string marker
240
250 ; ********* Template ends *********
260
270 MOV PC,R14                  ; Back to BASIC
280 ]
290
300 REM Reserve space for string and put address in B%
310 DIM string 100
320 B%= string
330
340 REPEAT
350 INPUT "Enter number to be converted :" A%
360
370 CALL ConvertStr
380
390 PRINT "String produced is : " $string
400 UNTIL FALSE
```

22 : Functions, Operators ...

This chapter contains templates for some miscellaneous BASIC statements. The first group are BASIC functions and operators. After these, the subject of implementing arrays in machine code is considered. Finally, we will take a brief look at making sound effects from machine code programs.

SGN

The BASIC SGN function takes one argument and returns a number indicating the sign of the argument in the following way:

−1	If argument is < 0
0	If argument is = 0
1	If argument is > 0

This is implemented very simply in assembly code as follows:

```
[
CMP number,#0        Compare the argument with zero
MOVEQ result,#0      If = 0 MOVE '0' into result
MOVGE result,#1      If > 0 MOVE '+1' into result
MVNLT result,#0      If < 0 MOVE '-1' into result
]
```

On entry to the routine, the argument should be placed into the register called number. On exit, the sign of the number (using the convention illustrated above) will be in the result register.

ABS

The ABS function complements SGN as it returns the magnitude of its argument while ignoring its sign. Put another way, ABS checks to see if the argument is negative and, if so, alters its sign to be positive. Again this is very simple to implement in assembly code as follows:

```
[
MOVS result,number        Move number to result register
RSBMI result,number,#0    If <0 then make positive
]
```

The argument is assumed to be contained in the register called 'number'. The routine initially performs result = number. It then checks to see if the result was negative and, if it is, executes result = 0 – result. This effectively reverses the sign, making the negative value positive again.

DIV and MOD

The DIV and MOD functions both perform integer division of two numbers. MOD returns the remainder of the division, and DIV returns the quotient.

We can produce a single assembly routine which will divide one 32-bit number by another and produce both quotient and remainder. Listing 22.1 does exactly this. It assumes that the two numbers to be used have been placed in registers 'number' and 'divisor'. It then performs 'number' divided by 'divisor'. The quotient and remainder of the result are placed in registers called 'quotient' and 'remainder'.

```
quotient = number DIV divisor
remainder = number MOD divisor
```

The routine consists of three main parts. The actual division is carried out by the program loop. This, however, can only deal with the division of positive integers. For this reason, the first program block stores the sign of each operand in turn, and then makes them positive. After the division occurs, the third program block corrects the results utilising the signs of the original numbers.

Listing 22.1. Template to perform DIV and MOD operations.

```
 10  REM Example of DIV and MOD templates
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM divide 256
 70
 80  REM Define names for registers used
 90  number    = 0
100  divisor   = 1
110  remain    = 2
120  quotient  = 3
```

```
130 place     = 4
140 dsign     = 5
150 msign     = 6
160
170 FOR pass = 0 TO 3 STEP 3
180 P%=divide
190 [
200 OPT pass
210
220 ; Division operands assumed to be present in registers:
230 ;  'number' and 'divisor'
240 ; In this example they are passed from BASIC via A% and B%
250
260 ANDS  msign,number,#1<<31  ; Produce sign of remainder
270 RSBMI number,number,#0     ; If negative, make positive
280 EOR   dsign,msign,divisor  ; Produce sign of quotient
290 CMP   divisor,#0           ; Check sign of divisor
300 RSBMI divisor,divisor,#0   ; If negative, make positive
310
320 MOV remain,#0              ; Initialise remainder
330 MOV quotient,#0            ; Initialise quotient
340 MOV place,#1<<31           ; Initialise place counter
350
360 .division_loop             ; Loop to process all 32 bits
370 MOVS number,number,ASL#1   ; Shift 1 place left and shift
380 ADC  remain,remain,remain  ; bit 31 into the remainder
390 CMP  remain,divisor        ; Is remainder > divisor
400 SUBGE remain,remain,divisor    ; If so do remainder-divisor
410 ORRGE quotient,quotient,place  ; and set appropriate bit
420 MOVS place,place,LSR#1     ; Move place counter 1 bit left
430 BNE division_loop      ; If all 32 bits not processed branch
440
450 CMP dsign,#0               ; Should quotient sign be neg
460 RSBMI quotient,quotient,#0 ; If so, make quotient negative
470 CMP msign,#0               ; Should remainder be negative
480 RSBMI remain,remain,#0     ; If so, make remainder negative
490
500 STR quotient,divres        ; Store results for BASIC to read
510 STR remain,modres
520
530 MOV PC,R14                 ; Back to BASIC
540
550 .divres                    ; Space for DIV and MOD results
560 EQUD 0
570 .modres
580 EQUD 0
590 ]
600 NEXT
610
620 REPEAT
630 PRINT '
640 INPUT "Number to be divided (dividend) :" A%
```

```
650  INPUT "Number to divide by  (divisor)  :" B%
660  CALL divide
670  PRINT ;A%; " DIV "; B%; "  =  "!divres
680  PRINT ;A%; " MOD "; B%; "  =  "!modres
690  UNTIL FALSE
```

Logical Operators: AND, OR, EOR

The use of AND and OR as a logical statement is dealt with in the template for the IF statement. Here, we consider the use of AND, OR and EOR as bit-wise operators. In BASIC we can write statements like:

```
result = operand1 AND operand2
result = operand1 OR operand2
result = operand1 EOR operand2
```

In assembly language we can perform the equivalent of these statements by using the following:

```
[
AND result,operand1,operand2
ORR result,operand1,operand2
EOR result,operand1,operand2
]
```

These statements perform the appropriate logical operation on operands one and two and place the result in the 'result' register. Full details of logical operators is given in Appendix C.

Logical Operator: NOT

The final logical operator provided by BASIC is the NOT function. This function takes a single argument and inverts all the bits in it to produce a result. For example:

```
result = NOT operand
```

This can be implemented in machine code easily using the processor's MVN instruction:

```
[
MVN result,operand
]
```

Arrays

The ARM processor provides excellent support for the handling of arrays. Here we shall only consider the implementation of one-dimensional integer arrays.

Dimensioning Arrays

Before an array can be used in BASIC, it must be dimensioned. This is done for two reasons. First, it allows BASIC to claim enough total memory to store the array's elements. Second, it informs BASIC of the individual dimension sizes of the array. This then allows it to check that subscripts used in future references to the array are legal. In our assembly language equivalent, we will not provide any automatic range checking of subscripts. Instead, it is left to the program using the array to make sure that it only accesses legal array elements.

In assembly code, therefore, the problem of dimensioning the array becomes the problem of reserving enough memory to hold the array. To store an 'n' element array we will require:

n * e bytes

Where 'n' is the number of elements and 'e' is the number of bytes required to store one array element. For example, suppose we want to reserve enough memory for an array defined as:

```
DIM freddy%(99)
```

This in fact creates a 100-element array (zero to 99) in which each element requires four bytes to store it. To store the complete array, therefore, we require the following:

100*4 bytes

The memory for an array can be reserved using the standard DIM<size> statement from BASIC. Alternatively, it can be reserved from within assembler by calling a user-defined function. This would take the number of bytes to be reserved as a parameter, and would increment P% by that amount. An example of this technique is given in listing 22.2.

Array Access

We now come to the problem of array access. The instructions to access memory are STR and LDR. These are fully described in Chapter 10. We shall use the pre-indexed form of addressing to access our arrays. In this mode, we can specify two addressing registers in an instruction, the contents of which are added together to give the address in memory of the accessed data. We shall use one register, called 'base', to contain the start address of our array. The other register, called 'index', will contain the number of the particular element we are accessing.

In an array of integers, each element will occupy one complete word (four bytes). To access the nth element, therefore, we must multiply the index by four before using it to access the data. This can be done within the instruction itself by specifying a two-place logical shift left of the index register. For example:

```
LDR destination,[base,index,LSL#2]
```

As an example of array access, listing 22.2 arbitrarily stores the numbers 200 to 300 consecutively in each of the 100 elements of an integer array. As it performs this action, it calls a 'print out' routine to display the operation being performed.

After waiting for a key to be pressed, the program then adds together all the numbers in the array – the equivalent of BASIC's SUM statement. Finally, the calculated total is displayed.

Listing 22.2. Array access in machine code.

```
 10  REM Example of the Array Access
 20  REM (c) Michael Ginns 1983
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM array_sum 1024
 70
 80  REM Define names for registers used
 90  base  = 5
100  index = 6
110  data  = 7
120  total = 8
130
140  FOR pass = 0 TO 3 STEP 3
150  P% = array_sum
```

```
160 [
170 OPT pass
180
190 MOV R10,R14         ; Preserve link register
200 ADR  base,freddy    ; Initialise base address of array
210 MOV  index,#0       ; Initialise array index
220 MOV  data,#200      ; Initial data value = 200
230
240 .loop1                          ; Loop to store data in array
250 STR  data,[base,index,LSL#2]    ; Store data in array(index)
260
270 SWI "OS_WriteS"     ; Output diagnostic data
280 EQUS "freddy( "
290 EQUB 0
300 MOV R0,index
310 BL print_it
320 SWI "OS_WriteS"
330 EQUS " )  =  "
340 EQUB 0
350 MOV R0,data
360 BL print_it
370 SWI "OS_NewLine"
380
390 ADD  data,data,#1       ; Increment data value
400 ADD  index,index,#1     ; Increment array index
410 CMP  index,#99          ; See if all elements accessed
420 BLE  loop1              ; If not then loop back
430
440                         ; Wait for a key to be pressed
450
460 SWI "OS_WriteS"
470 EQUS "Press any key to calculate SUM of the array:"
480 EQUB 0
490 SWI "OS_ReadC"
500 SWI "OS_NewLine"
510
520                         ; Calculate SUM the array
530 MOV total,#0            ; Initialise SUM total
540 MOV index,#0            ; Initialise array index
550
560 .loop2                  ; loop to sum elements of array
570 LDR  data,[base,index,LSL#2]    ; Access data in array(index)
580 ADD total,total,data            ; Add the data to the total
590 ADD index,index,#1              ; Increment the array index
600 CMP index,#99                   ; Are all elements done
610 BLE  loop2                      ; If not then loop back
620
630 MOV R0,total                    ; Print out the total
640 BL print_it
650
660
670 MOV PC,R10 ; Back to BASIC
680
```

```
690
700 ; Subroutine to print, in decimal, the number in R0
710 .print_it
720 ADR  R1,string_buffer
730 MOV  R2,#32
740 SWI  "OS_BinaryToDecimal"
750 MOV  R0,#0
760 STRB R0,[R1,R2]
770 ADR  R0,string_buffer
780 SWI  "OS_Write0"
790 MOV PC,R14
800
810 ; Reserve space for the array
820 .freddy
830 FN_work(100*4)
840
850 ; Reserve space for a string buffer
860 .string_buffer
870 EQUS STRING$(32,CHR$(0))
880
890 ]
900 NEXT
910
920 CLS
930 PRINT "Press any key to start"
940 pause = GET
950 CALL array_sum
960 PRINT
970 END
980
990 REM Function used to reserve space from the assembler
1000 DEF FN_work(number_of_bytes)
1010 P% = P% + number_of_bytes
1020 = pass
```

SOUND

The full sound system on the Archimedes is very different to that on the BBC micro. There is a full stereo wave synthesis system which can produce speech, sound effects, and play back sound samples.

The routines to control all this, therefore, are very complex and are beyond the scope of this book. However, on an extremely simple level, we can still make some use of the sound system.

The simplest equivalent of the BASIC SOUND command is an SWI call named "SoundControl". This is entered with the following parameters set up in registers R0 to R4:

R0: Channel number for sound
R1: Amplitude –15 (loudest) to zero (quietest)
R2: Pitch
R3: Duration

Listing 22.3 will produce a sound on channel one with a volume of –15, a pitch of 200 and a duration of 50.

Listing 22.3. Simple sound effects.

```
 10  REM Simple SOUND template
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM sound 256
 80  P% = sound
 90  [
100  MOV R0,#1              ; Channel 1
110  MOV R1,#15             ; +15 volume
120  RSB R1,R1,#0           ; Make R1 negative (-15)
130  MOV R2,#200            ; Pitch 200
140  MOV R3,#50             ; Duration 50
150  SWI "Sound_Control"    ; Use SWI routine to make sound
160
170  MOV PC,R14             ; Back to BASIC
180  ]
190  CALL sound
```

23 : Control Statements

Almost every computer program requires some use of control statements. These statements are used to make execution conditional on data, to implement loops and to select routes through multi-path code.

It is essential, therefore, that we can create equivalents to these high-level control statements in our assembly code programs. In this chapter we will consider how this may be done for each of BASIC's control statements.

For each control statement, a template is developed which will mimic its operation in assembly. The templates do not constitute complete programs. They do not even form complete assembly code routines to use within programs. Instead, they are skeletons which provide us with outlines for control statements.

For example, when developing an IF...THEN...ELSE template, it will depend on the application as to which condition is tested and what actions are taken by the THEN and ELSE clauses. Such a template is, therefore, given as a series of instructions which are general to all IF...THEN...ELSE statements with gaps left for the application specific instructions.

IF...THEN...ELSE...ENDIF

The complete block IF statement in BASIC is as follows:

```
IF <condition> THEN
        <statement 1>
ELSE
        <statement 2>
ENDIF
```

This can be implemented in assembly code using the CMP instruction and suitable branches. The general outline template for IF is shown in figure 23.1 on the next page.

The CMP instruction compares the two operands and sets various status register flags to indicate the results. If the comparison executes the first

branch, the processor will jump to the series of instructions which make up <statement 1>, (the THEN clause).

However, if the comparison results in the first branch not being taken, the instructions forming <statement> 2 will be executed, (the ELSE clause). After these instructions have been completed the unconditional branch will jump to the end of the construction, labelled by endif.

In this way we have two alternative execution paths depending on the result of a comparison, which is exactly what we want for a conditional statement like this.

```
[
    CMP reg1,reg2
    B<conditional suffix> then

            <statement 2>
    B endif

.then

            <statement 1>

.endif
]
```

Figure 23.1. Outline IF...THEN...ELSE template.

Note that the two operands for the IF statement test are assumed to be contained in registers reg1 and reg2. Obviously, this need not be the case, and the operands may need loading into the registers before executing the IF statement.

You will also have noticed that the first branch instruction is incomplete. We have not specified the conditional suffix to be used. For example, NE, EQ, GT, LT, and so on. This is deliberate as the suffix will depend on the comparison being made in the IF statement. For example, we could have:

```
IF A=B  THEN ....
IF A>B  THEN ....
IF A<=B THEN ....
```

We want the branch to be executed only when the appropriate relationship is true. The CMP instruction actually compares the two operands, but it is

the branch instruction's suffix which defines which relationship is true for the branch to be taken.

We are fortunate in that the ARM processor provides conditional suffixes to cover all the types of logical relations between two operands. To save having to work out how condition flags are set and which suffix should be used, look at the table in figure 23.2. This lists all the relationships between two operands which we may want to test for. With each it gives the corresponding conditional suffix to use with the branch instruction. It is assumed that the comparison CMP A,B has been made previously.

Condition of condition	Suffix used	Suffix for reverse
A = B	EQ	NE
A <> B	NE	EQ
A > B	GT	LE
A >= B	GE	LT
A < B	LT	GE
A <= B	LE	GT

Figure 23.2. Condition code for all possible logical comparisons.

An example should help to clarify things! Suppose we want to implement the following BASIC statements in assembler:

```
A = GET
IF A <65 THEN VDU 7 ELSE VDU A
```

This will accept a character from the keyboard. It will beep if the character is less than 65, ie, a numeric character. Otherwise, the character is printed out. The assembler equivalent using the IF template is given in listing 23.1.

Listing 23.1. Example of using the IF...THEN template.

```
 10  REM Example of the  'IF...THEN...ELSE'  template
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM conditional 256
 70
 80  REM Define register names and constants
 90  char = 0
100  vdu  = 256
110  beep = 7
```

```
120
130 FOR pass = 0 TO 3
140 P%= conditional
150 [
160 OPT pass
170
180 SWI   "OS_ReadC"   ; char = GET
190
200 CMP   char,#65     ; Compare char with 65
210 BLT   then         ; IF char < 65 branch to THEN clause
220
230 SWI "OS_WriteC"    ; ELSE output char
240 B     endif        ; branch to end
250
260 .then              ; THEN clause
270 SWI vdu+beep       ; VDU 7
280
290 .endif             ; End of statement
300
310 MOV PC,R14         ; Back to BASIC
320 ]
330 NEXT pass
340
350 PRINT '' "Type characters now!" '
360 REPEAT
370 CALL conditional
380 UNTIL FALSE
```

Multi-condition IF...THEN...ELSE Statements

A further modification of the simple IF statement is the inclusion of several conditions linked together by OR and AND. This too can be implemented in assembler language by modifying the general template.

OR

The OR condition can be implemented simply by adding extra compare and branch instructions after the first one. For example:

```
IF A<B OR C=D THEN <statement 1> ELSE <statement 2>
```

Assuming that the registers named A, B, C and D contain the appropriate values, this can be implemented as:

```
[
    CMP A,B
    BLT then
    CMP C,D
    BEQ then

            <statement 2>

    B endif

.then

            <statement 1>
.endif
]
```

If the first condition isn't met, the first branch to the THEN clause fails. However, instead of going to the ELSE clause as before, the second comparison is reached. If this comparison succeeds, we will still branch and execute THEN. Only if both comparisons fail, will the instructions in the ELSE clause be reached.

In this way statement 1 is executed if condition one is TRUE OR if condition two is TRUE. This idea can be extended to include any number of extra conditions required.

AND

To implement logical AND is slightly more complicated. We must force the processor to check several relationships and only execute the THEN clause if all of them are TRUE. If a single comparison succeeds, we must not branch immediately to THEN as the other conditions haven't been checked.

To solve the problem, we have to perform a certain amount of rearranging of the original IF statement. Consider the statement:

```
IF A=1 AND B=2 THEN <statement 1> ELSE <statement 2>
```

Statement one will be executed only if both of the relationships are TRUE, otherwise statement two will be executed. Thinking of this another way, statement two will be executed if either of the relationships are FALSE, otherwise statement one will be executed. This may seem a pointless exercise, but it allows us to re-write the statement as:

```
IF A<>1 OR B<>2 THEN <statement 2> ELSE <statement 1>
```

This is functionally identical to the first statement, but involves the OR operation, which we have already implemented.

In general, therefore, to implement AND, we swap over statements one and two in the template and reverse all of the conditions in the branch instructions. Referring back to the table in figure 23.2, the final column gives the suffix required to reverse the result of a condition. For example the opposite of BEQ (branch if equal) is BNE (branch if not equal).

Consider the following example:

```
IF A=B AND C<D THEN <statement 1> ELSE <statement 2>
```

This will be implemented in assembly code as:

```
[
    CMP A,B      Compare A and B
    BNE else     IF A<>B branch to else
    CMP C,D      Compare C and D
    BGE else     IF C>=D branch to else

                 THEN clause: reached only if A=B and C<D
        <statement 1>
    B endif      Jump to end

    .else        Reached if either condition fails

        <statement 2>

    .endif
]
```

Non-numeric Comparisons

So far in all of the descriptions, we have assumed that the condition statements test numerical quantities. However, strings can also be tested. In Chapter 21, a template routine was presented to compare to strings. This returned a number which indicates the result of the comparison. We can, therefore, call this routine from within the IF template and then use CMP to test the result returned from the comparison routine.

REPEAT...UNTIL

The REPEAT...UNTIL construction executes a series of statements UNTIL a condition is satisfied. To implement this in assembly code, we use similar ideas to those used with the IF statement. We set up a loop which includes a comparison to determine when it should terminate.

For example, supposing we wanted to implement the following:

```
REPEAT
    < statement >
UNTIL A=B
```

This would be done as follows:

```
[
.repeat
        < statement >

        CMP A,B
        BNE repeat
]
```

In general, the conditional suffix used with the branch instruction will depend on the comparison made. It can again be selected using figure 23.2.

It is important to note that in the previous example, although the terminating condition is UNTIL A=B, the branch instruction actually executes if A<>B (BNE). This is because for the loop to be repeated, the comparison must succeed and execute the branch. This is the opposite to the original REPEAT...UNTIL statement which terminates when the comparison succeeds.

We must, therefore, reverse the original comparison used in the REPEAT...UNTIL loop when writing our templates. This will automatically be allowed for if we use the last column of figure 23.2 when looking up the suffix for comparison.

WHILE...ENDWHILE

The WHILE loop is special in that the test is made at the beginning of each loop and, if it fails the first time, the statements in the loop are skipped.

An example of a WHILE loop is:

```
WHILE A=B
    <Statement>
ENDWHILE
```

This would be represented in assembly code as follows:

```
[
.while
        CMP A,B
        BNE endwhile

            <Statement>

        B while
.endwhile
]
```

Note that, once again, the suffix for the reverse of the WHILE condition is used in the branch instruction. This is because if the branch succeeds, the loop terminates.

FOR...NEXT

The FOR statement is a development of the simpler control loops which incorporate a counter to execute the code a specific number of times. An example of a FOR statement is as follows:

```
FOR num = start TO finish STEP s
    <Statement>
NEXT
```

The counter, in this case num, is called the control variable. It is initialised to the value of start, then incremented in the loop in steps of 's'. When it reaches, or exceeds, the value of 'finish', the loop terminates. An added complication is that 'start' may be greater than 'finish'. A negative step size will then be used to decrement the control variable.

The assembly code template of a FOR loop is given next. Only integer parameters may be used. Negative numbers must be represented in two's complement format. It is assumed that the loop parameters are contained in the appropriately named registers:

```
        MOV num,start ;Initialise control variable

.loop

        < statement >
        ADD num,num,s   Add step to control variable
        CMP s,#0        Is step size negative?
        BMI negative    Reached if positive step size
        CMP num,finish  Compare control variable with finish
        BLE loop        IF <= finish do loop again
        B end           Loop finished so jump to end

.negative               Reached if step size is negative

        CMP num,finish  Compare control variable with finish
        BGE loop        IF >= finish then loop again

.end                    end of loop.
```

The routine begins by moving the loop's start value into the control variable. It then performs one cycle of the loop. At the end of each cycle, the step size is added to the control variable. Note that, if the step size is a negative number, this will automatically decrease the control variable.

Finally, we check to see if the control variable has reached its terminating value. This is slightly more involved than might be expected. If the step size is positive, we check to see if the control variable is greater than or equal to the required finishing value. However, if a negative step size is being used, then we need to terminate the loop when the control variable becomes less than or equal to the finishing value. This explains why the template tests the step size to discover its sign, then branches to one of two different pieces of code to perform the comparison.

Listing 23.2 shows an example of a complete machine code FOR...NEXT loop. From BASIC, we enter the three loop parameters, start value, finish value and step size. These are passed to the routine using the resident integer variables. The machine code loop then prints out the value of the control variable on each iteration. Try entering different step sizes, both positive and negative, as well as various start and finish values. Confirm that, for each set of values, the routine behaves in exactly the same way as a BASIC FOR loop would.

Listing 23.2 A FOR...NEXT loop in assembly code.

```
10  REM Example of the FOR...NEXT loop template
20  REM (c) Michael Ginns 1987
30  REM DABS Press : Archimedes Assembly Language
```

```
 40 REM
 50
 60 DIM for 256
 70
 80 REM define names for registers used
 90 start  = 3
100 finish = 4
110 s      = 5
120 num    = 6
130
140 FOR pass = 0 TO 3 STEP 3
150 P%=for
160 [
170 OPT pass
180
190 MOV num,start                ;Initialise control variable
200
210 .loop
220
230 ; ********* Statements in the loop ********
240 ; ** These print the value of the control variable **
250
260 MOV R0,num                   ; Get value of control variable
270 ADR R1,string_buff           ; Pointer to string buffer
280 MOV R2,#32                   ; Length of string buffer
290 SWI "OS_BinaryToDecimal"     ; Convert number to a string
300 MOV R0,#0                    ; Terminate the string using
310 STRB R0,[R1,R2]              ; character CHR$(0)
320 MOV R0,R1                    ; Pointer to the number string
330 SWI "OS_Write0"              ; Output the string
340 SWI "OS_NewLine"             ; Output a NewLine
350
360 ; *********   Loop statements end  ********
370
380 ADD num,num,s                ;Add step to control variable
390 CMP s,#0                     ;Is step size negative?
400 BMI negative
410                              ;Reached if positive step size
420 CMP num,finish               ;Compare control var with finish
430 BLE loop                     ;IF <= finish do loop again
440 B   end                      ;Loop finished so jump to end.
450
460 .negative                    ;Reached if step size is negative
470 CMP num,finish               ;Compare control var with finish
480 BGE loop                     ;IF >= finish then loop again
490
500 .end                         ;End of loop
510
520 MOV PC,R14                   ;Back to BASIC
530
540 .string_buff
550 EQUS STRING$(32,CHR$(0))
560
```

```
570  ]
580  NEXT pass
590
600  REPEAT
610  INPUT " Enter the loop start value" , D%
620  INPUT " Enter the loop end value" , E%
630  INPUT " Enter the loop step size" , F%
640  CALL for
650  UNTIL FALSE
```

CASE Statement

The CASE statement is really a multi-clause IF statement. The assembler template of a slightly simplified CASE statement is given next. It assumes that the variable being tested is an integer held in the register called 'num'. It is also assumed that this is being tested against a series of constants C1...Cn. In practice, you can compare 'num' with other registers or derive values from memory:

```
        [
            CMP num,#C1
            BNE skip1
                    <Clause 1>
            B endcase

        .skip1
            CMP num,#C2
            BNE skip2
                    <Clause 2>
            B endcase

        .skip2
            CMP num,#C3
            BNE skip3
            . .
            . .
            . .

        .skip n
            CMP num,#Cn
            BNE otherwise
                    <Clause n>
            B endcase

    .otherwise
                    <Otherwise clause>

    .endcase
    ]
```

The value in 'num' is compared with each of the constants in turn. If the comparison fails, a branch instruction jumps to the next comparison in the structure. If a match is found, the instructions in the corresponding clause are reached and executed. After this, an unconditional branch instruction jumps to the end of the CASE structure.

If none of the comparisons in the CASE structure succeed, control falls through the routine to the instructions making up the default OTHERWISE clause.

An example of the use of the CASE template is given in listing 23.3. The program waits for a key to be pressed, then uses its ASCII value in a case statement. ASCII values 49 to 51 correspond to the numeric keys 1 to 3. For each of these values there is a corresponding clause in the CASE statement which simply prints out the number in words. Pressing any other key causes the OTHERWISE clause to be called which prints out a suitable message.

Listing 23.3. Example of the CASE template.

```
 10  REM Example of the CASE template
 20  REM (c) Michael Ginns 1987
 30  REM DABS Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM case 256
 70
 80  REM Define names for register used
 90  num = 0
100
110  FOR pass = 0 TO 3 STEP 3
120  P% = case
130  [
140  OPT pass
150
160  SWI "OS_ReadC"          ; Wait for key to be pressed
170
180                          ; First test
190  CMP num,#49             ; Is it '1'
200  BNE skip1               ; If not, skip to next test
210  SWI "OS_WriteS"         ; Clause 1
220  EQUS "One"              ; Output 'one'
230  EQUB 0
240  SWI "OS_NewLine"
250  B endcase               ; Branch to end of case statement
260
270  .skip1                  ; Second test
280  CMP num,#50             ; Is it '2'
290  BNE skip2               ; If not, skip to next test
```

```
300  SWI "OS_WriteS"          ; Clause 2
310  EQUS "Two"               ; Output 'two'
320  EQUB 0
330  SWI "OS_NewLine"
340  B endcase                ; Branch to end of case statement
350
360  .skip2                   ; Third test
370  CMP num,#51              ; Is it '3'
380  BNE  otherwise           ; If not, skip to 'otherwise' clause
390  SWI "OS_WriteS"          ; Clause 3
400  EQUS "Three"             ; Output 'Three'
410  EQUB 0
420  SWI "OS_NewLine"
430  B endcase                ; Branch to end of case statement
440
450  .otherwise               ; Otherwise clause
460  SWI "OS_WriteS"          ; Output message
470  EQUS "Only keys 1,2 or 3 please !"
480  EQUB 0
490  SWI "OS_NewLine"
500
510  .endcase
520
530  MOV PC,R14               ; Back to BASIC
540
550  ]
560  NEXT pass
570
580  PRINT '' "Enter characters now !!" '
590  REPEAT
600  CALL case
610  UNTIL0
```

Procedures

We have seen that the ARM processor gives at least partial support to the use of procedures in the form of the branch with link instruction, (BL). This allows us to call a sub-routine from an arbitrary point and return back to the point when the sub-routine terminates. The BL instruction was described in detail in Chapter 11.

The problem with this simple approach can be seen if we consider a real example. Supposing we call a procedure named 'freddy' and that this calls a second procedure named 'output'. Before calling 'output', the 'freddy' procedure must take a copy of its return address, usually held in register R14. If not done, it will be overwritten by the return address of the 'output' procedure when the second call is made.

At first, this may seem an acceptable scheme, and for small programs it is. However, in larger programs a procedure may call another which may call a third which in turn may call a fourth... and so on. In cases like this, as the chain of called procedures grows, it becomes more of a problem to store all the return addresses. Also, in dynamically recursive programs where a procedure calls itself, we will not know in advance the depth to which procedure calls will be nested. Keeping track of the various return addresses becomes impossible.

To solve these problems, we use a stack. In Chapter 12 we saw how the LIFO nature of the stack makes it ideal for storing data from nested structures. In this application the stack is used as follows.

Each time a procedure is called, it simply pushes its return address onto the stack. The data on the stack thus represents the return addresses of all active procedures, stored in the order in which they were called. The top element of the stack is always the return address of the most recently called procedure. When a procedure ends, therefore, it simply pulls the top element off the stack and returns to this address.

Local Variables

When implementing procedures, especially recursive ones, it is essential to use local variables. These are variables which can be used within the procedure without affecting the values of any outside variables. To implement a similar system in assembly language, we again call upon the stack.

On entry to a procedure, we push onto the stack, not only the procedure's return address, but also the contents of all the other working registers. Having done this, we can use the registers freely within the procedure as their contents outside it have been preserved on the stack. When the procedure ends, as well as pulling the return address off the stack, we also pull the stored values of the registers, thus restoring their original contents.

A procedure template is given in figure 23.3. A full ascending stack is used. However, any type of stack is acceptable. The stack pointer is assumed to have been set up in the register called 'sp'. Note, that the stack instructions have 'reg_list' specified within them. This is a list of registers which will be used as local variables within the procedure and need preserving:

```
.main_program
. .
BL subroutine        Call procedure
. .
.end_program

.subroutine

STMFA (sp)!,{R14,<reg_list} Stack return addr and regs
. .
<body of subroutine>
. .
LDMFA (sp)!,{R14,<reg_list} Unstack return address & regs

MOV R15,R14                 Return from subroutine
```

Figure 23.3. Template of a procedure using stacks.

Parameter Passing

The final consideration to be made as regards implementing procedures is parameter passing. In simple cases, it is possible to pass data to a procedure using the processor registers. This is done in the example program in the next section. However, in more complex procedures, this method rapidly becomes impractical as we soon run out of registers.

As an alternative, we can again use a stack. Before a procedure is called, the parameters for it are pushed on to the stack by the calling routine. When called, the procedure then pulls its parameters off the stack again. This provides a simple, but general, way of passing any number of parameters to a procedure. It also deals correctly with recursive procedure calls to any depth.

Example of Recursive Procedures

Listing 23.4 gives an example of a recursively-called procedure which uses local variables. Only three parameters are passed to the procedure, so registers are used.

The procedure draws a circle on the screen, then calls itself four times to create four more half-sized circles within the original. However, when these calls are made, as well as drawing the new circle, four further calls will be made. Each of these calls will produce four more calls and so on. This can be summarised by the following rule:

To draw a circle do the following:
Produce the circle on the screen
Draw four smaller circles (implies recursion)

Using this procedure once will cause a whole sequence of recursive procedure calls which will never end. However, we can add the rule that if a circle becomes too small the procedure may terminate without producing any further circles. The sequence of calls will then be limited, and the original procedure call will eventually end.

In order to see recursion in operation, the program will wait for a key to be pressed after drawing each circle. Try stepping through the program observing the sequence of procedure calls made. As a point of interest, try pressing ESCAPE to execute the program at full speed. The speed of the ARM processor is apparent when you consider that the program draws 1365 circles! Note, the graphics used to create the circles will be explained in the next chapter.

Listing 23.4. An example of a recursive procedure.

```
 10 REM Example of recursive procedure calls
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM pattern 256
 70 DIM stack_mem  1000
 80
 90 REM Define constants
100 vdu    = 256
110 plot   = 25
120 move   = 4
130 circle = 145
140 gcol   = 18
150
160 REM Define names for registers used
170 x = 1      : REM Used to pass x co-ord to circ procedure
180 y = 2      : REM Used to pass y co-ord to circ procedure
190 r = 3      : REM Used to pass radius to circ procedure
200 col =10    : REM circle colour - global so never stacked
210 sp = 12    : REM Stack pointer
220
230 FOR pass = 0 TO 3 STEP 3
240 P%=pattern
250 [
260 OPT pass
270
280 ADR sp,stack_mem   ; Set stack pointer bottom of stack
```

```
290  STMFA (sp)!,{R14}  ; Push BASIC's return addr onto stack
300
310  MOV col,#2         ; Set global variable 'col' to colour 2
320
330  MOV x,#640         ; Initial call of circ procedure
340  MOV y,#500         ; with parameters (640,500,492)
350  MOV r,#492
360
370  BL circ            ; Call the circ procedure
380
390  LDMFA (sp)!,{R14}  ; Pull BASIC's return addr back off stack
400
410  MOV PC,R14         ; Return to BASIC
420
430  ; Circ procedure starts here. It is defined as; circ(x,y,r)
440  ; It draws a circle of radius r at (x,y) & then recursively
450  ; calls itself 4 times to produce circles at:
460  ;   x+r,y
470  ;   x-r,y
480  ;   x,y+r
490  ;   x,y-r
500
510  .circ
520
530  STMFA (sp)!,{R0-R9,R14}    ; Stack return addr and R0-R9
540                             ; R0 to R9 can now be used freely
550
560  CMP r,#10          ; Compare radius with 10
570  BLT endproc        ; IF r < 10 then endproc
580
590  ; The next section of code simple draws a circle of
600  ; radius r at co-ordinates (x,y) in a new colour
610
620  SWI vdu+gcol       ; Perform GCOL 0,col MOD 127
630  SWI vdu+0
640  ADD col,col,#1     ; Increment col
650  AND R0,col,#127
660  SWI "OS_WriteC"
670
680  SWI vdu+plot       ; Perform MOVE x,y
690  SWI vdu+move
700  MOV R0,x
710  SWI "OS_WriteC"
720  MOV R0,x,LSR#8
730  SWI "OS_WriteC"
740  MOV R0,y
750  SWI "OS_WriteC"
760  MOV R0,y,LSR#8
770  SWI "OS_WriteC"
780
790  SWI vdu+plot       ; Perform PLOT 145,r,0
800  SWI vdu+circle
810  MOV R0,r
```

```
 820  SWI "OS_WriteC"
 830  MOV R0,r,LSR#8
 840  SWI "OS_WriteC"
 850  SWI vdu+0
 860  SWI vdu+0
 870
 880  MOV r,r,LSR#1       ; Half radius for next circles
 890
 900  SWI "OS_ReadC"      ; Wait for key press - may be removed
 910
 920  ADD x,x,r           ; Call circ(x+r,y,r)
 930  BL circ
 940
 950  SUB x,x,r,LSL#1     ; Call circ(x-r,y,r)
 960  BL circ
 970
 980  ADD x,x,r           ; Call circ(x,y+r,r)
 990  ADD y,y,r
1000  BL circ
1010
1020  SUB y,y,r,LSL#1     ; Call circ(x,y-r,r)
1030  BL circ
1040
1050  .endproc            ; End of circ procedure
1060
1070  LDMFA (sp)!,{R0-R9,R14}    ; Pull return addr/regs. R0-R9
1080
1090  MOV PC,R14          ; Return from procedure
1100
1110  ]
1120
1130  NEXT
1140  MODE 15
1150  PRINT "Press a key to step through program"
1160  PRINT "ESCAPE for full speed"
1170  CALL pattern
```

24 : Graphics Templates

At first sight, it may seem that all but the most simple graphics are beyond our reach in machine code. The calculations involved in just plotting the points on a straight line are bad enough, let alone creating filled triangles, circles and ellipses, or dealing with colour. However, all is not lost! The Archimedes designers have anticipated the problems!

When we issue graphics commands from BASIC, eg, 'CIRCLE 600,600,100', the BASIC interpreter does very little work. It simply interprets the command and sends the relevant data to the operating system. It is the operating system which actually performs the required graphics operation. As we have access to the operating system's routines from our machine code programs, we also have full access to the same graphics facilities.

The data for graphics operations, as well as a range of other functions, is passed to the operating system using special control characters. These are sent to the VDU drivers, in the same way as normal characters. However, the operating system intercepts these characters and interprets them as commands. When the operating system receives a command in this way, it may also intercept some of the following characters to get any data required for the command, for example, the co-ordinates to use in a graphics command. A complete list of all of the Archimedes control codes and their functions are given in Appendix H.

By sending the correct sequence of control characters to the VDU drivers, therefore, we can carry out many complex operations including graphics. An example may help to convince you! Consider the two BASIC statements:

```
MOVE 100,100
DRAW 1200,1000
```

BASIC requests the operating system to draw the line by issuing the following characters:

```
VDU 25,4,100,0,100,0
```

for the MOVE statement. And:

```
VDU 25,5,176,4,232,3
```

for the DRAW statement.

You can check this by entering the two VDU statements, and verifying that they draw the same line as the original BASIC statements.

All we have to do to produce graphics in machine code, therefore, is to be able to output characters. The BASIC statement VDU normally does this, so we shall begin by showing how this statement can be emulated from machine code.

VDU n

To print single ASCII-value characters, we use one of two possible SWI calls. We have already seen both calls used in previous examples:

```
SWI 256 + <ASCII>
SWI "OS_WriteC"
```

The first call is used when a fixed, known character needs to be outputted, like a graphics plot code. The number of the SWI call used is 256 plus the ASCII code of the character. So:

```
SWI 256 + 2
```

would perform the equivalent of VDU 2 and output character two – the control code to turn the printer on.

The second SWI call, OS_WriteC, is used where we do not know which character is to be outputted at assembly time. For example, graphics x,y co-ordinates may be calculated by a machine code program, then outputted using this call. When called, it outputs the character whose ASCII code is contained in the lower byte of the processor register R0. Both of these SWI routines are described fully in Chapter 17.

PLOT

PLOT is the workhorse of the graphics system. It has three parameters: an option code and a pair of x,y co-ordinates. The option code is in the range

zero to 255 and selects what you want to plot, eg, lines, triangles, circles, sprites and so on.

The VDU control code for PLOT is character 25. Thus, to perform PLOT from machine code we simply output character 25. After this we output the option code required and finally, the x,y co-ordinates to be used. Unfortunately, there is a problem! The graphics co-ordinates are in the range:

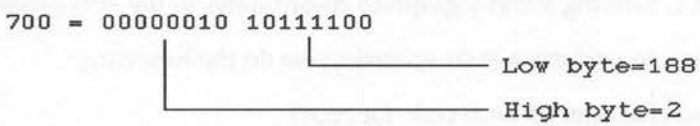
```
0 <= X <= 1279
```

and:

```
0 <= Y <= 1023
```

However, the maximum number that we can represent in one byte, and therefore one character, is 255. This means we can't send the co-ordinates as single characters.

Instead, we must send them as two pairs of two characters each. The first character in each pair is the low byte of the co-ordinate, and the following one is the high byte. For example, to send a co-ordinate of 700, we would do the following:



The characters sent, therefore, are:

```
VDU 188,2
```

The upper and lower bytes of a given number can be found using the DIV and MOD operators:

```
Lower byte = <co-ordinate> MOD 256
Upper byte = <co-ordinate> DIV 256
```

On the BBC micro this splitting of co-ordinates down into two separate bytes is insignificant. Being an eight-bit machine, it can only deal with byte-sized pieces of data anyway. However, on the Archimedes it is inconvenient to manipulate each piece of graphics data as two completely separate bytes. It is much more sensible to process and store graphics co-ordinates

as complete 32-bit words. When we need to plot them, then and only then, do we split the co-ordinates into two byte pieces. They are then sent to the VDU driver.

The routine shown in figure 24.1 takes a pair of co-ordinates, splits them into high and low bytes, then sends them to the VDU driver. The registers called 'x' and 'y' are assumed to contain the x and y co-ordinates which are to be output.

We will need this routine each time we perform a graphics operation. For this reason, it is presented as a subroutine, called "co_ord", which can be called from a main program whenever it is required.

```
.co_ord
MOV R0,x
SWI "OS_WriteC"
MOV R0,x,LSR#8
SWI "OS_WriteC"
MOV R0,y
SWI "OS_WriteC"
MOV R0,y,LSR#8
SWI "OS_WriteC"
MOV PC,R14
```

Figure 24.1. Sending x and y graphics co-ordinates to the VDU driver.

To summarise, to perform a PLOT operation we do the following:

1) Output character 25 (VDU code for PLOT)

2) Output the PLOT option code

3) Place the x and y co-ordinates in registers x and y

4) Call the co-ords sub-routine

Listing 24.1 below uses this technique to plot a line on the screen.

Listing 24.1. Example of a PLOT command from assembly code.

```
10  REM Example of the PLOT template
20  REM (c) Michael Ginns 1988
30  REM Dabs Press : Archimedes Assembly Language
40  REM
50
60  DIM plot example 256
```

```
 70
 80 REM Define constants and register names
 90 vdu  = 256 : REM Start of SWI block to perform VDU n
100 plot = 25
110 line = 5
120 x = 1      : REM passes x co_ord to co-ord routine
130 y = 2      : REM passes y co_ord to co-ord routine
140
150 FOR pass = 0 TO 3 STEP 3
160 P% = plot_example
170 [
180 OPT pass
190
200 MOV R10,R14             ; Preserves BASIC return addr in R10
210
220 SWI vdu+plot            ; VDU 25 (code for PLOT)
230 SWI vdu+line            ; VDU 5  (PLOT code for a line)
240 MOV x,#512              ; Use co_ordinates (512,640)
250 MOV y,#640
260 BL  co_ord              ; Call 'co_ord' subroutine
270
280 MOV PC,R10              ; Return to BASIC (addr in R10)
290
300 ; Subroutine to output pair of x,y co_ords to VDU drivers
310
320 .co_ord
330 MOV R0,x                ; Move x co-ord into register R0
340 SWI "OS_WriteC"         ; Output the x co-ord low byte
350 MOV R0,x,LSR#8          ; Put hi byte x co-ord in low byte R0
360 SWI "OS_WriteC"         ; Output the x co-ord high byte
370 MOV R0,y                ; Move y co-ord into reg R0
380 SWI "OS_WriteC"         ; Output the y co-ord low byte
390 MOV R0,y,LSR#8          ; Put hi byte y co-ord in low byte R0
400 SWI "OS_WriteC"         ; Output the y co-ord high byte
410 MOV PC,R14              ; Return from subroutine
420
430 ]
440 NEXT
450
460 MODE 0
470 CALL plot_example
```

It is important to stress that any graphics shape can be drawn using the correct option in a PLOT command. A full list of the various plot option codes can be found in Appendix I. BASIC commands like MOVE, CIRCLE, RECTANGLE, LINE and so on. All translate into one or more PLOT commands. These commands will be covered in the following sections.

SWI PLOT

In versions 1.20 and later of the Arthur operating system, a SWI call has been introduced which performs a PLOT operation directly. The theory behind this is exactly the same as that described above. The SWI routine simply save from having to send the PLOT data as separate control characters to the operating system via the VDU drivers. The routine is detailed below.

SWI "OS_Plot"

On Entry: R0= PLOT option code
R1= x co-ordinate
R2= y co-ordinate

For example, the line drawing program in listing 24.1 could be re-written using the PLOT SWI as follows:

```
[
MOV R0,#5          ; PLOT option code for a line
MOV R1,#512        ; Put x co-ord in R1
MOV R2,#640        ; Put y co-ord in R2
SWI "OS_Plot"
MOV PC,R14
]
```

'OS_Plot' take the plot option code and the x,y co-ordinates as complete 16-bit quantities. It then invokes an equivalent function to the previous co-ordinate plotting routine given in figure 24.1.

In subsequent graphics templates, however, we shall continue to use the original method of plotting data using the co-ordinate splitting and output subroutine for compatibility reasons.

However, should you wish to use "OS_Plot" then simply replace the statements to output the PLOT option code and call the co-ords subroutine with SWI "OS_Plot". Remember to set up registers R0 to R2 to contain the appropriate data for the PLOT operation before calling the SWI routine.

Some graphics operations, for example, defining graphics windows and moving the graphics origin have no equivalent using "OS_Plot". To implement these in assembly language, therefore, we must again return to

the method of outputting data using the appropriate VDU control codes. Once again, as the data may be greater than 255, the co-ordinates outputting routine will still be required.

MOVE x,y

MOVE x,y is directly equivalent to PLOT 4,x,y. Refer to the section on PLOT for details on how to perform this from machine code. As an example, the following MOVE statement could be implemented by the section of code given below as follows:

```
MOVE 600,500

[
    MOV x,#600   Put x co-ord in register named X
    MOV y,#500   Put y co-ord in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+4    PLOT code for a MOVE
    BL co_ord    Output co-ordinates
]
```

Note that it is assumed that the register names x and y have been associated with real register numbers. The constant vdu has been set to equal 256. Note also that the co-ord subroutine is being used. This must be included in any real program. These assumptions are common to many of the examples in the following sections, and will not be repeated each time.

The co-ordinates of the MOVE operation are supplied as immediate constants. However, they could be derived from anywhere, or calculated by the program. This applies to the example programs given with other graphics commands in the following sections too.

POINT x,y

POINT x,y is directly equivalent to PLOT 69,x,y. Again, refer to the section on PLOT for further details. As an example, the following POINT statement could be implemented by the section of code given below:

```
POINT 100,100

[
    MOV x,#100   Put x co-ord in register named X
    MOV y,#100   Put y co-ord in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+69   PLOT code for a POINT
```

```
        BL co_ord    Output co-ordinates
]
```

Notice again the reference to the co-ord subroutine.

DRAW x,y

DRAW x,y is directly equivalent to PLOT 5,x,y. As an example, the following DRAW statement could be implemented by the section of code given below:

```
DRAW 800,700

[
    MOV x,#800   Put x co-ord in register named X
    MOV y,#700   Put y co-ord in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+5    PLOT code for DRAW
    BL co_ord    Output co-ordinates
]
```

Once more it should be stressed that the co-ord subroutine is being used and must be included into any real program.

BY

MOVE, POINT and DRAW can all be followed with the BY command. This causes the operating system to treat the co-ordinates as being relative to the current graphics co-ordinates, rather than being absolute numbers.

To do the same in machine code, we use exactly the same routines for MOVE, DRAW and POINT, but we use a PLOT code which is four less than the values used before. For example:

```
MOVE BY x,y
PLOT 0,x,y
POINT BY x,y
PLOT 65,x,y
DRAW BY x,y
PLOT 1,x,y
```

LINE x1,y1,x2,y2

The BASIC LINE command takes two sets of co-ordinates: the start point of the line to be drawn, and the end point. The equivalent is MOVE followed by DRAW. For example:

```
LINE x1,y1,x2,y2
```

is equivalent to:

```
MOVE x1,y1
DRAW x2,y2
```

To implement LINE, therefore, we simple use the MOVE and DRAW templates given previously. Listing 24.2 gives an example of this to produce some interesting effects.

Listing 24.2. Example of the LINE template.

```
 10 REM Example of the LINE template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM lines 256
 70
 80 REM Define constants and register names
 90 vdu  = 256 : REM Start of SWI block to perform VDU n
100 plot = 25
110 move = 4
120 draw = 5
130 vsync= 19
140
150 x_cord = 3 : REM program's x co_ordinate
160 y_cord = 4 : REM program's y co_ordinate
170 x = 1      : REM passes x co_ordinate to co-ord routine
180 y = 2      : REM passes y co_ordinate to co-ord routine
190
200 FOR pass = 0 TO 3 STEP 3
210 P% = lines
220 [
230 OPT pass
240
250 .repeat
260 MOV x_cord,#0              ; Initialise 'x' co-ordinate
270 .draw_loop                 ; Loop to draw lines
280
290 RSB y_cord,x_cord,#1280    ; Obtain y co-ord (y = 1280 - x)
300
```

```
310 MOV R0,#vsync          ; *FX 19 to reduce screen flicker
320 SWI "OS_Byte"
330
340 ; The following code performs ; LINE 0,y,x,0
350
360                        ; MOVE 0,y
370 SWI vdu+plot           ; VDU 25 (code for PLOT)
380 SWI vdu+move           ; VDU 4  (PLOT code for a MOVE)
390 MOV x,#0               ; Use co_ordinates (0,y)
400 MOV y,y_cord
410 BL  co_ord             ; Call 'co_ord' subroutine
420
430                        ; DRAW x,0
440 SWI vdu+plot           ; VDU 25 (code for PLOT)
450 SWI vdu+draw           ; VDU 5  (PLOT code for a DRAW)
460 MOV x,x_cord           ; Use co_ordinates (x,0)
470 MOV y,#0
480 BL  co_ord             ; Call 'co_ord' subroutine
490
500 SWI "OS_NewLine"       ; Output a new line
510
520 ADD x_cord,x_cord,#16  ; Increment x co-ord
530 CMP x_cord,#1280       ; Are we at edge of screen
540 BLT draw_loop          ; If not, draw next line
550
560 B repeat               ; Keep repeating the whole program
570
580
590 ;Subroutine: output a pair of x,y co-ords to VDU drivers
600
610 .co_ord
620 MOV R0,x               ; Move x co-ord into register R0
630 SWI "OS_WriteC"        ; Output the x co-ord low byte
640 MOV R0,x,LSR#8         ; Put hi byte x co-ord in low byte R0
650 SWI "OS_WriteC"        ; Output the x co-ord high byte
660 MOV R0,y               ; Move y co-ord into register R0
670 SWI "OS_WriteC"        ; Output the y co-ord low byte
680 MOV R0,y,LSR#8         ; Put hi byte y co-ord in low byte R0
690 SWI "OS_WriteC"        ; Output the y co-ord high byte
700 MOV PC,R1              ; Return from subroutine
710
720 ]
730 NEXT
740
750 MODE 0
760 CALL lines
```

CIRCLE x,y,radius

BASIC's CIRCLE command takes three parameters; the x,y co-ordinates of the circle's centre and its radius.

The corresponding circle PLOT command (option code 145), however, works in a slightly different way. First we move to the centre of the circle and then use the PLOT 145 command with the co-ordinates of any point on the circle's circumference. Thus, to perform the equivalent of the BASIC CIRCLE command we do the following:

```
MOVE x,y
PLOT 145,radius,0
```

Note that the PLOT 145 code plots a circle using relative co-ordinates, ie, the co-ordinates given are added to those of the previous position visited. Thus, PLOT 145,radius,0 specifies a point at absolute co-ordinates (x + radius,y). This point is clearly on the circumference of the required circle.

Once again, to implement CIRCLE, we simply use the templates for MOVE and PLOT given earlier. For example, to implement the following CIRCLE command we would use the section of code given below:

```
CIRCLE 512,600,300

[
    MOV x,#512   Put x co-ord in register named X
    MOV y,#600   Put y co-ord in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+4    PLOT code for MOVE
    BL co_ord    Output co-ordinates
    MOV x,#300   Put radius in register named X
    MOV y,#0     Put 0 in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+145  PLOT code for a relative CIRCLE
    BL co_ord    Output co-ordinates
]
```

Listing 24.3 creates circles using this technique. It repeatedly plots 48 circles of gradually increasing radius, then 48 circles of gradually decreasing radius. After plotting each circle, the screen is scrolled giving some quite surprising results!

Listing 24.3. Example of the CIRCLE template.

```
 10 REM Example of the CIRCLE template
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 DIM circles 256
 70
 80 REM Define constants and register names
 90 vdu     = 256  : REM Start of SWI block to perform VDU n
100 plot    = 25
110 move    = 4
120 circle  = 145
130 gcol    = 18
140 vsync   = 19
150
160 x = 1      : REM passes x co_ordinate to co-ord routine
170 y = 2      : REM passes y co_ordinate to co-ord routine
180 radius = 3 : REM circle radius
190 count  = 4 : REM circle counter
200 inc    = 5 : REM radius increment value
210 col    = 6 : REM circle colour
220
230 FOR pass = 0 TO 3 STEP 3
240 P% = circles
250 [
260 OPT pass
270
280 MOV radius,#0          ; Initialise radius
290 MOV inc,#16            ; Initialise increment
300 MOV col,#114           ; Set colour
310
320 .repeat                ; loop to repeat entire program
330 MVN R0,#31             ; toggle increment between 16 and -16
340 EOR inc,inc,R0         ; by using EOR
350 MOV count,#0           ; Initialise circle counter
360
370 SWI vdu+gcol           ; Perform GCOL 0,col MOD 256
380 SWI vdu+0
390 AND R0,col,#127
400 SWI "OS_WriteC"
410 ADD col,col,#1         ; Increment col
420
430 .draw_loop             ; Loop to draw 48 circles
440
450 ADD radius,radius,inc  ; Change radius for each circle
460
470 MOV R0,#vsync          ; *FX 19 to reduce screen flicker
480 SWI "OS_Byte"
490
500 ; The following code performs ; CIRCLE 512,640,radius
```

```
510
520 MOV x,#640              ; MOVE 512,640
530 MOV y,#512
540 SWI vdu+plot            ; VDU 25 (code for PLOT)
550 SWI vdu+move            ; VDU 4  (PLOT code for a MOVE)
560 BL  co_ord              ; Call 'co_ord' subroutine
570
580 MOV x,radius            ; PLOT 145,radius,0
590 MOV y,#0
600 SWI vdu+plot            ; VDU 25 (code for PLOT)
610 SWI vdu+circle          ; VDU 145  (PLOT a relative CIRCLE)
620 BL  co_ord              ; Call 'co_ord' subroutine
630
640 SWI "OS_NewLine"        ; Output a new line
650
660 ADD count,count,#1      ; Increment the circle counter
670 CMP count,#48           ; Have 48 circles been drawn?
680 BLT draw_loop           ; If not, draw next circle
690
700 B repeat                ; Keep repeating the whole program
710
720
730 ; Subroutine: output a pair of x,y co-ords to VDU drivers
740
750 .co_ord
760 MOV R0,x                ; Move x co-ord into reg R0
770 SWI "OS_WriteC"         ; Output the x co-ord low byte
780 MOV R0,x,LSR#8          ; Put hi byte x co-ord in low byte R0
790 SWI "OS_WriteC"         ; Output the x co-ord high byte
800 MOV R0,y                ; Move y co-ord into register R0
810 SWI "OS_WriteC"         ; Output the y co-ord low byte
820 MOV R0,y,LSR#8          ; Put hi byte y co-ord in low byte R0
830 SWI "OS_WriteC"         ; Output the y co-ord high byte
840 MOV PC,R14              ; Return from subroutine
850
860 ]
870 NEXT
880
890 MODE 15
900 CALL circles
```

Filled Circles

A filled circle may be created in BASIC using the CIRCLE FILL command. To do this in assembly code, we use our normal circle plotting routine, but replace the 'plot relative circle' option (145) with that for a 'relative filled circle' (153). For example:

```
CIRCLE FILL 512,600,300
```

```
[
    MOV x,#512   Put x co-ord in register named x
    MOV y,#600   Put y co-ord in register named y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+4    PLOT code for MOVE
    BL co_ord    Output co-ordinates
    MOV x,#300   Put radius in register named x
    MOV y,#0     Put 0 in register named y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+153  PLOT code for a relative FILLED CIRCLE
    BL co_ord    Output co-ordinates
]
```

RECTANGLE x,y,w,h

This command takes four parameters. The first two parameters specify the co-ordinates of the bottom-left corner of the rectangle. The next two give the width and height of the rectangle respectively. Again, the word FILL can be used to produce a filled rectangle.

Strangely, there is a PLOT command available to draw a FILLED rectangle directly, but not an outline one! The filled rectangle is therefore easier to create, and we will deal with this one first.

Like the circle, the filled rectangle command can be converted into one MOVE and one PLOT operation, again relative co-ordinates are used in the PLOT command:

```
RECTANGLE FILL x,y,w,h
```

is equivalent to:

```
MOVE x,y
PLOT 97,w,h
```

The problem with drawing filled rectangles is again one of moving and plotting, both of which we can do from assembly language. An example shows the template needed to do this:

```
RECTANGLE FILL 200,100,800,700

[
    MOV x,#200   Put x co-ord in register named x
    MOV y,#100   Put y co-ord in register named y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+4    PLOT code for MOVE
```

```
    BL co_ord    Output co-ordinates
    MOV x,#800   Put rectangle width in register named X
    MOV y,#700   Put rectangle height in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+97   PLOT code for a relative filled rectangle
    BL co_ord    Output co-ordinates
]
```

Outline Rectangle

To draw an outline rectangle we must draw each of its four sides individually using the equivalent of the DRAW command. This can be done in the following way:

```
RECTANGLE x,y,w,h
```

which is equivalent to:

```
MOVE x,y
DRAW BY w,0
DRAW BY 0,h
DRAW BY -w,0
DRAW BY 0,-h
```

Note that the DRAW commands use relative co-ordinates. This translates to machine code very easily using the standard MOVE and DRAW BY templates described earlier.

Next is an example of the machine code routine required to mirror the operation of the statement:

```
RECTANGLE 100,200,500,300
```

This is somewhat long-winded. However, it is really no more complicated than the simple MOVE and DRAW which we have used before.

```
[
    MOV w,#500   Put rectangle width in register named W
    MOV h,#300   Put rectangle height in register named H

    MOVE X,Y
    MOV x,#100   Put X co-ord in register named X
    MOV y,#200   Put Y co-ord in register named Y
    SWI vdu+25   VDU 25 (code for PLOT)
    SWI vdu+4    PLOT code for MOVE
    BL co_ord    Output co-ordinates
```

```
        DRAW BY w,0
        MOV x,w Put rectangle width W in register named X
        MOV y,#0    Put 0 in register named Y
        SWI vdu+25  VDU 25 (code for PLOT)
        SWI vdu+1   PLOT code for DRAW relative
        BL co_ord   Output co-ordinates

        DRAW BY 0,h
        MOV x,#0    Put 0 in register named X
        MOV y,h     Put rectangle height H in register Y
        SWI vdu+25  VDU 25 (code for PLOT)
        SWI vdu+1   PLOT code for DRAW relative
        BL co_ord   Output co-ordinates

        DRAW BY -w,0
        RSB x,w,#0  Put -w in register named X
        MOV y,#0    Put 0 in register named Y
        SWI vdu+25  VDU 25 (code for PLOT)
        SWI vdu+1   PLOT for DRAW relative
        BL co_ord   Output co-ordinates

        DRAW BY 0,-h
        MOV x,#0    Put 0 in register named X
        RSB y,h,#0  Put -H in register named Y
        SWI vdu+25  VDU 25 (code for PLOT)
        SWI vdu+1   PLOT for DRAW relative
        BL co_ord   Output co-ordinates
]
```

FILL x,y

This command is followed by a pair of co-ordinates and flood fills the screen from this point. Its direct equivalent is PLOT 133,x,y. The routine to perform FILL, therefore, is simply the standard PLOT routine using an option code of 133. For example:

```
FILL 300,400

[
        MOV x,#300  Put X co-ord in register named X
        MOV y,#400  Put Y co-ord in register named Y
        SWI vdu+25  VDU 25 (code for PLOT)
        SWI vdu+133 PLOT code for FILL
        BL co_ord   Output co-ordinates
]
```

ORIGIN x,y

The VDU code to re-define the origin is 29. We do not need to use a plot command, instead we use VDU 29, then output the appropriate co-ordinates. For example:

```
ORIGIN 500,700

[
    MOV x,#500   Put X co-ord in register named X
    MOV y,#700   Put Y co-ord in register named Y
    SWI vdu+29   Perform VDU 29
    BL co_ord    Output new co-ordinates
]
```

MODE n

VDU 22 selects the new screen mode. It is followed by a single character, giving the number of the mode to be used. This maps very easily in machine code and involves simply outputting two characters. For example:

```
[
    MOV R0,#m    R0 should contain the new mode number M
    SWI vdu+22   Perform VDU 22
    SWI OS_WriteC"   Select mode M
]
```

CLS

The clear text screen function is carried out by VDU 12. In machine code this is simply:

```
[
    SWI vdu+12   Clear text screen
]
```

That's all there is to it!

CLG

This is similar to CLS, but this time the control character used is VDU 16. In assembly code we write:

```
[
    SWI vdu+16  Clear graphics screen
]
```

COLOUR

Listed below are the three variations of the COLOUR command:

1) COLOUR L
2) COLOUR L,P
3) COLOUR L,R,G,B

We shall develop assembler equivalents to each of these in turn.

COLOUR L

In this form, the statement simply selects colour L as the current text colour. It is equivalent to VDU 17,L – where L is the colour to be changed to. In assembly code, therefore, we would use the following instructions to select text colour two:

```
[
    MOV R0,#2   R0 contains the colour number to be used
    SWI vdu+17  Perform VDU 17,L
    SWI "OS_WriteC"
]
```

Listing 24.4 uses this template to output 63 star characters, each one in a different colour.

Listing 24.4. Printing coloured stars.

```
 10  REM Example of the COLOUR template
 20  REM (c) Michael Ginns 1988
 30  REM Dabs Press : Archimedes Assembly Language
 40  REM
 50
 60  DIM coloured 256
 70
 80  REM Define constants and names for the registers used
 90  vdu  = 256
100  col  = 17
110  star = 42
111
120  n = 1
130
```

```
140  P%= coloured
150  [
160
170  MOV n,#0
180  .star_loop          ; Loop to print 63 starts
190  MOV R0,n            ; Prepare to select colour in 'n'
200  SWI vdu+col         ; Perform COLOUR n
210  SWI "OS_WriteC"
220  SWI vdu+star        ; Output a '*'
230  ADD n,n,#1          ; Increment 'n'
240  CMP n,#63           ; See if 'n' has reached 63 yet
250  BLE star_loop       ; If not, output the next '*'
251  SWI "OS_NewLine"    ; Output a newline
260  MOV PC,R14          ; Back to BASIC
270  ]
280
290  MODE 15
300  CALL coloured
```

COLOUR L,P and COLOUR L,R,G,B

These forms of the COLOUR command are used to redefine the logical colours from the full palette of 4096 physical colours. The first statement defines logical colour L to be physical colour P. The second gives more control by allowing colour L to be redefined in terms of its RED, GREEN and BLUE components. Both statements are implemented using the VDU 19 control code. The VDU 19 sequence is shown below:

```
VDU 19,L,P,R,G,B
```

The possible effects of this are shown in figure 24.2. To perform any of these just issue VDU 19, followed by the appropriate parameter sequence.

Range for 'P'	Effect
0-15	Define logical colour L as physical colour P
16	Define logical colour L in terms of RED, GREEN and BLUE components using R,G and B
17	Define colour of first 'flash phase' for colour L
18	Define colour of second 'flash phase' for L
24	As before but define the border colour
25	As before but define colour L of the mouse pointer's colours

Figure 24.2. Possible effects using VDU 19.

For example, to redefine the colour black (0), as levels 200, 10, and 180 of RED, GREEN and BLUE respectively, we would write:

```
[
    SWI vdu+19      Perform VDU 19
    MOV R0,#0       R0=number of colour to be changed
    SWI "OS_WriteC"
    SWI vdu+16      16 specifies a RGB colour mapping
    MOV R0,#200     Amount of RED
    SWI "OS_WriteC"
    MOV R0,#10      Amount of GREEN
    SWI "OS_WriteC"
    MOV R0,#180     Amount of BLUE
    SWI "OS_WriteC"
]
```

GCOL a,c

This command changes the graphics colour to colour 'c' and specifies a plotting action of 'a'. The table in figure 24.3 shows the various plotting options available. The colour being plotted is 'c'.

0	Plot colour 'c' directly on the screen
1	OR colour on screen with 'c' and plot result
2	AND colour on screen with 'c' and plot result
3	EOR colour on screen with 'c' and plot result
4	Invert colour on screen
5	No colour plotted
6	AND colour on screen with NOT 'c' and plot result
7	OR colour on screen with NOT 'c' and plot result
8 - 15	As before but background colour treated as transparent
16 - 31	Use colour pattern 1
32 - 47	Use colour pattern 2
48 - 63	Use colour pattern 3
64 - 79	Use colour pattern 4
80 - 85	Use composite 'giant' pattern

Figure 24.3. Plotting options.

GCOL is performed by VDU 18,p,c. The assembler equivalent to this is simply to output character 18 followed by the two parameters. For example:

```
GCOL 3,5
```

```
[
    SWI vdu+18       VDU 18 (GCOL)
    MOV R0,#3        Action code 3
    SWI "OS_WriteC"  Output parameter 1
    MOV R0,#5        Colour 5
    SWI "OS_WriteC"  Output parameter 2
]
```

POINT

The POINT statement is used to find out the colour of a given point on the graphics screen. The operating system provides a direct equivalent to this, a SWI routine called OS_ReadPoint.

The routine is entered with the X co-ordinate of the point to be examined in register R0 and the Y co-ordinate in register R1. The call returns with the following information:

R2 = The colour of the point
R3 = The 'TINT' of the colour (256 colour mode)
R4 = Zero if the point was on the screen and
minus one if the point was not on the screen

Thus, to find the colour of the point at co-ordinates 700,560 we would use:

```
[
    MOV R0,#700          X co-ordinate
    MOV R1,#560          Y co-ordinate
    SWI "OS_ReadPoint"   Examine point
]
```

After executing these instructions, the information about the point would be contained in the appropriate registers, as just shown.

ON, OFF

These two commands are graphics-related in that they turn the screen cursor on and off. The operating system provides two routines specifically to perform these functions:

To turn cursor OFF:

```
[
    SWI "OS_RemoveCursors"
]
```

To restore the previous cursor state (ON):

```
[
    SWI "OS_RestoreCursors"
]
```

WAIT

This is the last of our graphics commands. It is used in animation to synchronise programs with the vertical synchronisation pulse of the monitor. It reduces the screen flicker produced if graphics are drawn during the screen re-fresh scan.

The equivalent to WAIT in machine code is OSBYTE 19. This is implemented as follows:

```
[
    MOV R0,#19      OSBYTE number 19
    SWI "OS_Byte"   Call OSBYTE
]
```

APPENDICES

A : Representing Numbers

When programming in assembly language we are dealing with the internal workings of the machine. We therefore need an understanding of how the computer represents and manipulates data at this level.

The central processing unit (CPU) at the heart of the computer does not even operate in the same number base as we do! We are used to counting in the decimal (base 10) number system. In decimal, numbers are represented by combining digits from the 10 possible numbers available in the base (zero to nine). The problem with this, as far as the computer is concerned, is that the various hardware elements of the system would have to manipulate signals which ranged over 10 distinct levels. This would be required to represent any one of the 10 possible decimal digits.

In practice this turns out to be extremely inconvenient. Instead, only two signal levels or states are used to represent data:

ON	(usually represented by plus five volts)
OFF	(usually represented by zero volts)

By convention, these two states are thought of as representing the digits one and zero respectively. As only two digits are used, we are operating in base two or the binary number system as it is called. A single digit in binary can only take the values zero or one – and is referred to as a bit, meaning Binary digIT.

Binary Numbers

Obviously, working with single bits would be incredibly limiting, not least because the computer would only be able to count up to two! To overcome this, we group several bits together in much the same way that we group decimal digits to represent more than the numbers zero to nine in decimal.

Consider the case of grouping two binary digits together. Each bit may be either a one or a zero, and so there are now four possible patterns or states which we can represent. These are shown in figure A.1.

00
01
10
11

Figure A.1. The four possible patterns available with two bits.

Each unique pattern can be used to represent one number, thus with two bits we could represent four different numbers. This same idea can be extended to use more bits to allow more and more numbers to be represented. For example, if eight bits are grouped together, then 256 different combinations of ones and zeros are possible and therefore 256 different numbers can be represented. See figure A.2.

Decimal	Binary
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
.	.
.	.
.	.
.	.
253	11111101
254	11111110
255	11111111

Figure A.2. Possible patterns using eight bits.

Blocks of eight bits have a special significance in computing and are referred to as bytes. Memory is frequently organised in bytes and thus one byte of memory can store one of 256 different patterns of data.

Listing A.1 allows you to enter a number, then it will enumerate all the data patterns possible with this number of bits. Any number of bits can be used

from one to 31. Try some values and see how rapidly the number of possible patterns increases as extra bits are added.

Listing A.1. Binary patterns.

```
 10 REM Binary pattern generator
 20 REM (c) Michael Ginns 1988
 30 REM Dabs Press : Archimedes Assembly Language
 40 REM
 50
 60 CLS
 70 REPEAT
 80 REPEAT
 90 INPUT "Enter the number of bits to be used (1-31) :" bits
100 UNTIL bits > 0 AND bits < 32
110
120 PRINT SPC(9); "Number     Pattern"
130
140 FOR number = 1 TO 2^bits
150 PROC_print_binary(number-1,bits-1)
160 NEXT
170
180 UNTIL FALSE
190
200 DEFPROC_print_binary(number,bits)
210 PRINT number; SPC(10);
220 FOR digit = bits TO 0 STEP -1
230 IF number AND 2^digit THEN PRINT "1"; ELSE PRINT "0";
240 NEXT
250 PRINT
260 ENDPROC
```

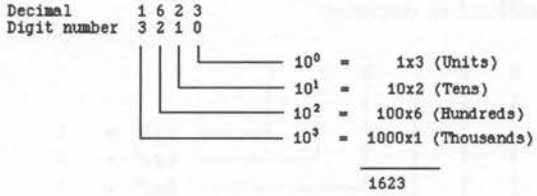
The Archimedes processor can deal directly with data consisting of 32 bits. We obviously need some scheme for defining which of the possible binary patterns corresponds to which numeric value. This is again done using ideas which are familiar from the decimal number system.

When we see the decimal number 1623 we intuitively know that this is the value one thousand six hundred and twenty three. However, we should stop to consider exactly how we calculated this value. The value is derived by multiplying the 'place value' of each digit in the number by the value of the digit itself. This gives us the actual value which each digit represents within the number. Finally, summing these values, we get the final value which the whole number represents.

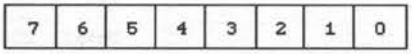
The place value, or weighting as it is called, is one for the right-most digit in the number. In decimal it then increases by a factor of 10 each time you

move another digit to the left. If we number the digits from left to right, starting at zero, then another way of thinking about this is that the weighting of a given digit is 10^n where n is the digit's number. Thus, the right-most digit has a weighting of $10^0 = 1$ and therefore represents units. The next digit's weight is $10^1 = 10$ and therefore it represents 10s... and so on.

Taking our original example of 1623, this can be broken down as follows:



In binary, we again number the digits. As with decimal, we start numbering from the right-most bit which is designated 'bit 0':



Because the binary system is really base two, the weight of each bit is calculated by raising two to the power of the bit's number. Thus, bit zero has a weight of $2^0 = 1$, bit one's weight is $2^1 = 2$, bit two's is $2^2 = 4$ and so on. Figure A.3 contains the weights for each of the bits in a complete byte:

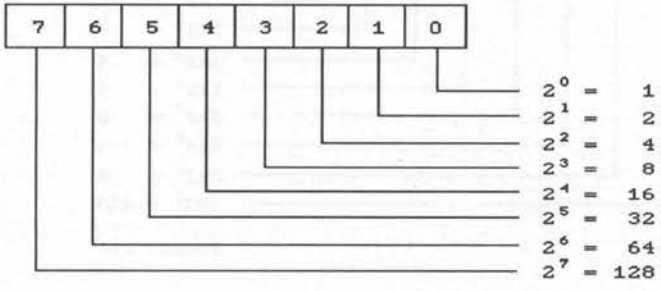


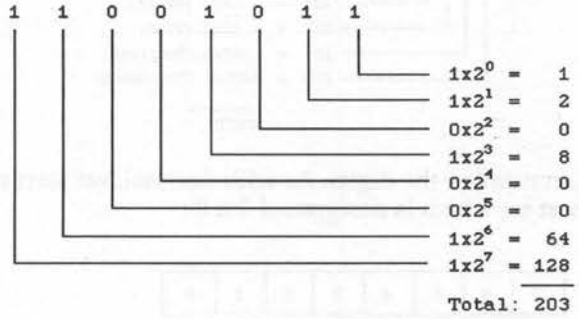
Figure A.3. The weightings of the eight bits in a byte.

Converting from Binary to Decimal

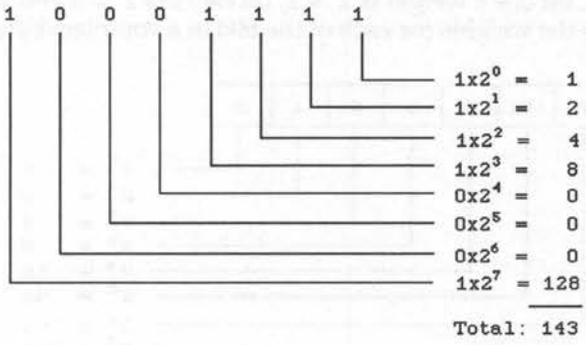
To convert from binary to decimal is very straightforward. We simply use the same method as we would in decimal, ie, multiply each digit in the number by its weight and then sum all of the results.

Working through an example should help to clarify matters:

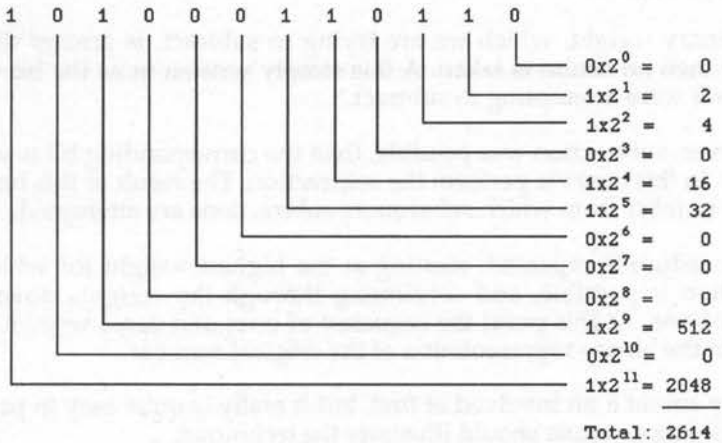
Convert binary 11001011 to decimal:



Similarly to convert binary 10001111 to decimal:



In the above examples the binary numbers were eight-bits long (one byte), however, the same process will work for any size number. For example, for 12 bits we have:



Using Binary on the Archimedes

Although it is very useful to be able to convert from binary to decimal by hand, BASIC on the Archimedes can accept binary numbers directly. For example, it is perfectly acceptable to say the following:

```
A = %10010101
```

This will set the variable A to the value 10010101 binary (149 decimal). Try printing the value or A afterwards to verify this.

Note the use of the % sign in the example. This specifies that the number following it is given in binary. This method of specifying binary numbers can be used wherever a numeric argument is required. For example:

```
PRINT %10010101
A = %10001 + %010101
[ OPT %101
```

Converting from Decimal to Binary

Converting from decimal to binary involves attempting to subtract successive binary weights.

If the binary weight, which we are trying to subtract, is greater that the number then no action is taken. A 0 is simply written in as the bit whose weight we were attempting to subtract.

If, however, subtraction was possible, then the corresponding bit is written as a one. In this case we perform the subtraction. The result of this becomes the new number from which subsequent subtractions are attempted.

This procedure is repeated, starting at the highest weight for which the subtraction is possible, and continuing through the weights down to a weight of one. At this point the sequence of ones and zeros written down will form the binary representation of the original number.

This may sound a bit involved at first, but it really is quite easy in practice. The following example should illustrate the technique.

Convert 231 decimal to binary:

Decimal number	Weight	Subtraction	Binary
231	128	231-128=103	1
103	64	103-64=39	1
39	32	39-32=7	1
7	16	-	0
7	8	-	0
7	4	7-4=3	1
3	2	3-2=1	1
1	1	1-1=0	1

Therefore, 231 is 11100111 in binary.

Hexadecimal

So far we have seen how numbers are represented by the computer in binary and how to convert between binary and decimal. There is, however, another system of representing numbers which is commonly used in computing. This is base 16, called hexadecimal or simply hex.

In decimal each digit making up a number could be chosen from a range of 10 possibilities (zero to nine). In binary, there were only two digits available (zero and one). Well, in hexadecimal there are 16 possible digits available! The digits zero to nine are used as normal, but we then have a problem as we need six extra symbols to represent 10 to 15. The letters A to F are used for this purpose, A=10, B=11 and so on.

As the base is 16, the weights of the digits in a hexadecimal number increase from right to left in powers of 16, ie, one, 16, 256, 4096 and so on. In the number hex 9F the F represents 15 units and the nine represents $9 \times 16^1 = 144$. The whole number therefore is 9+144 = 159 in decimal.

Figure A.4 shows the number zero to 15 expressed in decimal, binary and hexadecimal notations.

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Figure A.4. Numbers in decimal, binary and hexadecimal.

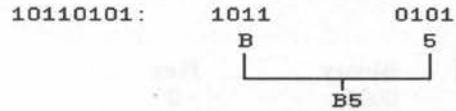
It is no coincidence that exactly four bits are required to represent a single hexadecimal digit, or that two hexadecimal digits can be stored exactly in one byte (eight bits). It is because of the close correspondence between hexadecimal and binary that it is so useful. It is much more concise than writing strings of ones and zeros, but it is more closely related to binary than the decimal number system.

Converting from Binary to Hexadecimal

The similarities between hexadecimal and binary are highlighted by the ease with which numbers can be converted between them.

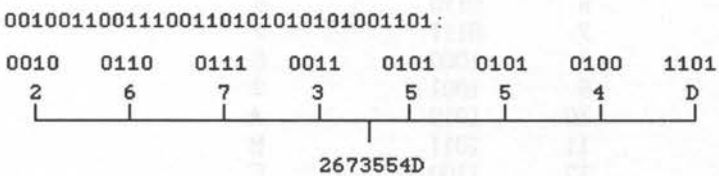
To convert a binary number into hexadecimal, we simply split it into groups of four bits starting at the right. Each of the four-bit binary numbers can then be directly translated into the corresponding hexadecimal digit shown previously in figure A.4.

For example:



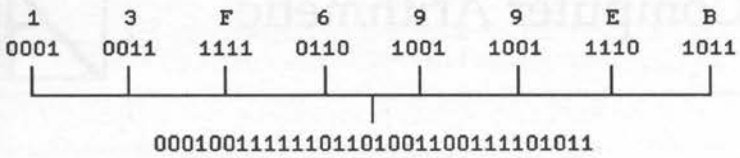
Therefore, 10110101 in binary equals B5 in hexadecimal.

Similarly, for a 32-bit number:



Converting from Hexadecimal to Binary

To represent hexadecimal numbers in binary, we simply apply the reverse process. Each hexadecimal digit in the number is converted into the corresponding four-bit binary number. These groups of four bits are then 'glued together' to form the complete binary number corresponding to the original hexadecimal one. For example consider the number &13F699EB:



Therefore, &13F699EB= 00010011111101101001100111101011.

The only point to be careful about is the treatment of leading zeros. When individual digits are converted into binary, it is important to always have four bits. Thus, in the above example, when the digit three was converted to binary, two leading zeros were added to retain four bits.

Hexadecimal on the Archimedes

In the same way that binary numbers are prefixed with a % sign, hexadecimal numbers are distinguished on the Archimedes by preceding them with an & sign. Thus, we can say:

```
A = &FF
```

alternatively:

```
PRINT &FE60
```

Also, because hexadecimal is so frequently used, the Archimedes has the facility to perform the reverse operation and print decimal numbers out in hexadecimal format. This is done by prefixing the decimal number by '~'. For example:

```
PRINT ~3584
```

This will cause the Archimedes to respond with E00 – the hexadecimal equivalent of 3584. Note that in teletext mode 7, the '~' character is displayed as a ' + ' character.

B : Computer Arithmetic

In Appendix A, we saw how the computer physically represents numbers as sequences of bits. The next step is to examine how these numbers can be manipulated by the computer to implement simple arithmetic.

After the complexities of binary representation, you could be forgiven for thinking that binary arithmetic is a problem which you could well do without! However, rest assured that arithmetic in binary is no more difficult that in decimal – just a bit different!

Addition

When adding together two bits there are four different cases possible. These are shown in figure B.1, together with the results of the addition on each case.

```
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 0   carry 1
```

Figure B.1. Rules of binary addition.

In the last case we added one to one which in decimal would give the result two. However, in binary this cannot be represented as a single digit, so we must carry one into the next column. This is similar to the case in decimal where the two digits to be added are, for example, eight and two. The result of this can't be given in a single decimal digit and so we write a zero and carry one into the next column. It is vital to remember to include the carried digit when performing the addition in the next column.

Some examples should illustrate the technique:

```
Example 1:      101     (5)
                001+    (1)
                ---     ---
                110     (6)
```

Note that in the right-most column, the addition 1 + 1 generated a carry into the next column. The next addition was therefore 0 + 0 + 1 (carry), which results in one and no carry.

Check that converting the binary result 110 back into decimal gives you the expected answer of six.

```
Example 2:      00110011    (51)
                01000011+   (67)
                --------    ----
                01110110    (118)
```

Again, the addition in the right-hand column is 1 + 1 which produces a carry. The addition for the next column then becomes 1 + 1 + 1 (carry). This is done in two parts; 1 + 1 gives a result of zero and produces another carry into the following column. Now adding the extra one carried from the previous column gives 0 + 1 = 1. Thus, the result is to write down one and also carry one.

```
Example 3:  10001001100111001100110011011010    (2308754650)
            10010101000110001100101001100110+   (2501429862)
            --------------------------------    ------------
            100011110101101011001011101000000   (4810184512)
```

In this example we are adding together two 32-bit numbers, but the result is a 33-bit value. This is because the result of the addition is too large to represent in 32 bits, and so a carry is generated from the left-hand column. This is perfectly acceptable on paper, however in practice, we may only have 32 bits available, so the result could not be stored. This condition is called an overflow and will usually cause an error or special corrective action to be taken.

Subtraction

Until now, our examination of the binary number system has dealt exclusively with positive numbers. However, when we try to tackle the

subtraction of binary numbers we, immediately raise the possibility that negative results may be generated. It is necessary, therefore, to look at the ways in which negative quantities could be represented in the binary system before attempting any subtraction.

This is also useful because, if we can represent negative numbers in binary, we can then perform the subtraction of two numbers simply by adding the negative of one of the numbers to the other. For example, the calculation:

23 – 10

Can be re-written as:

23 + (–10)

Providing we can form the binary equivalent to –10, we can use the ordinary rules of addition to add it to 23. The result of this will be equivalent to the original expression of 23 – 10.

Representing Negative Numbers in Binary

At first sight, it may seem very easy to modify the binary system to include negative numbers. After all, we only have to store whether the number is positive or negative, and this can be done in a single bit. While this is certainly true, things are unfortunately not that easy!

The following would be an example of simply using an extra bit to represent the sign of a number. The left-most bit (bit 31) is now designated the sign bit. If it is a one then the number is negative, if it is a zero then the number is positive. Remember that the sign bit does not play any part in representing the magnitude of the number, only bits zero to 30 are now used for this:

```
00000000000000000000000000000011   (+3)
10000000000000000000000000000011   (-3)
```

This system looks as if it should work. Unfortunately it breaks down when we attempt to apply arithmetic operations on negative numbers. For example, if we try to add +3 and –3 then we should get a result of zero.

```
00000000000000000000000000000011   (+3)
10000000000000000000000000000011   (-3) +
--------------------------------
10000000000000000000000000000110   (-6)
```

Converting the result back into decimal, we find that it is of magnitude six and that the sign is negative. Thus, using this system, we find that +3 added to –3 results in –6! Obviously, something more has to be done!

One's and Two's Compliment

The problems we were experiencing before are solved if we use the two's complement system of representing negative numbers. The idea of using a sign bit is retained but the other bits forming the number are modified so that the number performs correctly under arithmetic operations.

To form a negative number in two's complement format, simply perform the following steps:

1) Start with the number expressed as a normal positive binary value

2) Change, or invert, all the bits in the number. The number formed at this stage is called the one's complement

3) To form the two's complement, from the one's complement, simply add one to the number using the normal rules of binary addition

Again, taking the number–3 as an example the rules are applied as follows:

```
Convert +3 to binary:  00000000000000000000000000000011
Invert the bits:       11111111111111111111111111111100

Add one to number:     11111111111111111111111111111100
                                                      1 +
                       --------------------------------
                       11111111111111111111111111111101

Therefore -3 is:       11111111111111111111111111111101
```

Thus, we have formed the two's complement representation of –3. Note that the left-most bit is still used as a sign bit, and is a one to indicate a negative number.

The way in which the two's complement of a number is calculated may seem unbelievable at first - but it really does work. Our previous example, where we were adding +3 and –3, can now be tried again:

```
  00000000000000000000000000000011   (+3)
  10000000000000000000000000000011 + (-3)
-------------------------------------------
(1) 00000000000000000000000000000000   (0)
```

Ignoring the final carry, the result is zero - the correct answer!

In cases like this, it is perfectly acceptable to ignore the final carry as it is only an overflow of the sign bit from bit 31 – not a real overflow of the number itself.

A couple more examples of subtraction using two's complement may help to make things clear:

1) Perform 257–6:

```
Convert -6 to binary:  00000000000000000000000000000110  (+6)
Invert the bits:       11111111111111111111111111111001
Add one to number:     11111111111111111111111111111010  (-6)

Add 257 and -6:        00000000000000000000000100000001  (257)
                       11111111111111111111111111111010  ( -6)
                     (1)00000000000000000000000011111011  (251)
```

2) Perform 3 – 60

```
Convert 60 to binary:  00000000000000000000000000111100  (+60)
Invert the bits:       11111111111111111111111111000011
Add one to number:     11111111111111111111111111000100  (-60)

Add 3 and -60:         00000000000000000000000000000011   (+3)
                       11111111111111111111111111000100  (-60)
                       11111111111111111111111111000111  (-57)
```

C : Logic Operations

Logical operations deal only with quantities which are either true or false. They also yield results which are similarly restricted to being either true or false. The most useful logical operations are AND, OR and EOR.

Logical operators are ideally suited to binary, as true and false, can be directly represented in a single bit. By convention true is always one and false is always zero.

Having said that, logical operators only work with true or false values. It is perfectly acceptable to write expressions like the following:

```
7 AND 12 = 4
```

This may seem like a contradiction, but it isn't really! What the expression means is that the two numbers, seven and 12, should be expressed in binary form, and the logical operator AND should be applied between corresponding pairs of bits. This will yield a series of binary results which make up the bits in the numeric result. In this case the binary result, when converted back into decimal, is four. This will become clearer when we look at the individual logical operators in detail.

Logical AND

In English the operation A AND B can be described as:

'The result is true only if A is true AND B is true'

In binary, read one and zero for true and false respectively. The operation can be defined for all possibilities of A and B in the following truth table:

A	B	Result
0	0	0
0	1	0
1	0	0
1	1	1

With these four simple rules we can AND together any binary numbers simply by applying the rules to each pair of bits in turn. For example:

```
00000000010010101001000101110111          (4886903)
00000000100100010100101011101001 AND      (9521897)
--------------------------------------------------
00000000000000000000000001100001            (97)
```

The real usefulness of the AND operation is only apparent when you notice the following two properties:

Anything ANDed with zero is zero
Anything ANDed with one is left unchanged

This means that by carefully choosing the correct binary pattern, we can create a mask which will force certain bits in the number to zero when ANDed with *any* other number whileleaving the others unchanged.

A zero in the mask will ensure that the corresponding bit in the number will be set to zero. A one bit in the mask, therefore, will leave the corresponding bit unchanged.

We may, for example, want to zero the lower four bits in a number. This could be achieved using the following binary pattern as a mask:

11111111111111111111111111110000

ANDing this pattern with any other number would ensure that bits zero to three were all set to zero, but would leave the other bits in their original states. To show this is the case, let us try ANDing the mask with a number:

```
      11110101001000100110001011010110
mask  11111111111111111111111111110000 AND
-----------------------------------------
      11110101001000100110001011010000
```

Logical OR

In English the operation A OR B can be described as:

'The result is true if A is true OR if B is true'

Once again, the operation can be defined for all the possibilities of A and B as follows:

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	1

With these four basic rules we can OR together any binary numbers simply by applying the rules to each pair of bits in turn. For example:

```
00011000101010100000100101000000      (413796672)
00001000100011000010001000111010 OR   (143401530)
-------------------------------------  ----------
00011000101011100010101101111010      (414067478)
```

Like the AND operation, OR is useful in manipulating single bits in binary numbers while leaving the others unchanged. Specifically, the OR operation allows us to create a mask which will force any required bit to be a one.

This is possible because of the following properties of the OR operation:

Anything ORed with zero is left unchanged
Anything ORed with one is one

This means that a one in the mask will ensure that the corresponding bit in the number will be set to one. A zero bit in the mask will have no effect.

We may, for example, want to force bits one, three and five to be one. This could be achieved using the following binary pattern as a mask:

00000000000000000000000000101010

ORing this pattern with any number will set bits one, three and five, while leaving the others unchanged. Again, to prove the point, let's try an example as follows:

```
      10100001110101101100010100001000
mask  00000000000000000000000000101010 OR
      ----------------------------------
      10100001110101101100010100101010
```

Logical EOR

The EOR operation means Exclusive-OR. Its operation can be described in English as:

> 'The result is true if A is true OR if B is true but
> NOT if both A and B are true'

This is slightly more complex than the other two operations, but can still be defined as follows:

A	B	Result	
0	0	0	
0	1	1	
1	0	1	
1	1	0	(Note both A and B are true)

Another way of thinking about the EOR operation is that if the two bits are the same, then the result is zero. If they are different, then the result is one. Like AND and OR, the bits in two binary numbers can easily be EORed together using the four rules. For example:

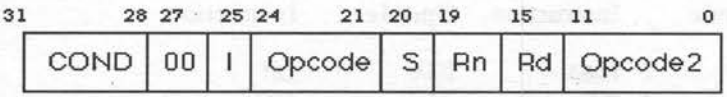
```
00011101010100100101001001011101      (491934301)
00111010101001010000000100011010 EOR (983892250)
--------------------------------------------------
00100111111101110101001101000111      (670520135)
```

A very useful property of the EOR operation is that if a number is EORed with a mask containing only ones, then the number will be inverted. For example, all the ones will be changed to zeros, and all the zeros will become ones. Thinking back to negative numbers in binary, the one's complement of a number was formed by inverting its bits. The EOR operation is therefore ideal for do this. For example, to form the one's complement of 10010110, we can EOR it with 11111111 to get 01101001.

D : Instruction Set Format

This section contains details of the internal formats used to represent each of the ARM instructions. Each instruction is 32 bits wide. This is subdivided into several fields to represent options and data within the instructions.

Data Processing Instructions



Field	Purpose
Cond	The conditional execution code (figure D.1)
Opcode	Defines which instruction it is (figure D.2)
Rd	The number of the destination register
Rn	The number of the register used as operand 1
Operand2	A register (possibly shifted) or an immediate constant

Bit	Function	
I	I=0	Operand 2 is a register
	I=1	Operand 2 is an immediate constant
S	S=0	Modify status flags on execution
	S=1	Leave status flags unchanged

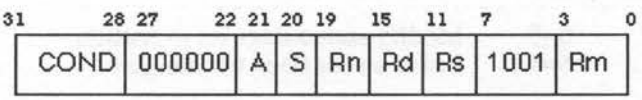
Cond Code	Condition Code	Cond	Condition
0000	EQ	1000	HI
0001	NE	1001	LS
0010	CS	1010	GE
0011	CC	1011	LT
0100	MI	1100	GT
0101	PL	1101	LE
0110	VS	1110	AL
0111	VC	1111	NV

Figure D.1. Condition codes.

Opcode	Instruction	Opcode	Instruction
0000	AND	1000	TST
0001	EOR	1001	TEQ
0010	SUB	1010	CMP
0011	RSB	1011	CMN
0100	ADD	1100	ORR
0101	ADC	1101	MOV
0110	SBC	1110	BIC
0111	RSC	1111	MVN

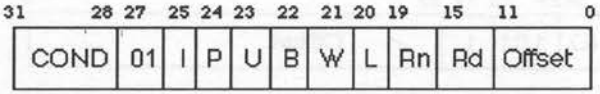
Figure D.2. Opcodes.

Multiply Instructions



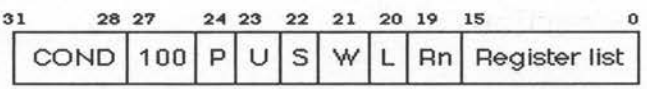
Bit	Function	
A	A=0	Multiply only
	A=1	Multiply and accumulate

Single Register Data Transfer



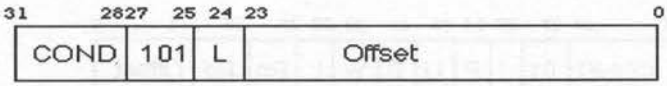
Bit	Function	
P	P=0	Post-indexed addressing
	P=1	Pre-indexed addressing
U	U=0	Offset subtracted from base address
	U=1	Offset added to base address
B	B=0	Word addressing used
	B=1	Byte addressing used
W	W=0	Do not perform writeback
	W=1	Perform writeback
L	L=0	Instruction is a store register (STR)
	L=1	Instruction is a load register (LDR)

Multiple Register Data Transfer



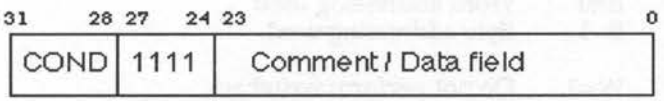
S	S=0	Do not load status register
	S=1	Load status register

Branch Instructions



L L=0 Ordinary branch
S=1 Branch with link

SWI Instructions



E : OS SWI Routines

SWI number in hex	SWI number in decimal	SWI name
000	0	OS_WriteC
001	1	OS_WriteS
002	2	OS_WriteO
003	3	OS_NewLine
004	4	OS_ReadC
005	5	OS_CLI
006	6	OS_Byte
007	7	OS_Word
008	8	OS_File
009	9	OS_Args
00A	10	OS_BGet
00B	11	OS_BPut
00C	12	OS_GBPB
00D	13	OS_Find
00E	14	OS_ReadLine
00F	15	OS_Control
010	16	OS_GetEnv
011	17	OS_Exit
012	18	OS_SetEnv
013	19	OS_IntOn
014	20	OS_IntOff
015	21	OS_CallBack
016	22	OS_EnterOS
017	23	OS_BreakPt
018	24	OS_BreakCtrl
019	25	OS_UnusedSWI
01A	26	OS_UpdateMemC
01B	27	OS_SetCallBack
01C	28	OS_Mouse

SWI number in hex	SWI number in decimal	SWI name
01D	29	OS_Heap
01E	30	OS_Module
01F	31	OS_Claim
020	32	OS_Release
021	33	OS_ReadUnsigned
022	34	OS_GenerateEvent
023	35	OS_ReadVarValue
024	36	OS_SetVarValue
025	37	OS_GSInit
026	38	OS_GSRead
027	39	OS_GSTrans
028	40	OS_BinaryToDecimal
029	41	OS_FSControl
02A	42	OS_ChangeDynamicArea
02B	43	OS_GenerateError
02C	44	OS_ReadEscapeState
02D	45	OS_EvaluateExpression
02E	46	OS_SpriteOp
02F	47	OS_ReadPalette
030	48	OS_ServiceCall
031	49	OS_ReadVduVariables
032	50	OS_ReadPoint
033	51	OS_UpCall
034	52	OS_CallAVector
035	53	OS_ReadModeVariable
036	54	OS_RemoveCursors
037	55	OS_RestoreCursors
038	56	OS_SWINumberToString
039	57	OS_SWINumberFromString
03A	58	OS_ValidateAddress
03B	59	OS_CallAfter
03C	60	OS_CallEvery
03D	61	OS_RemoveTickerEvent
0C0	192	OS_ConvertStandardDateAndTime
0C1	193	OS_ConvertDateAndTime
0D0	208	OS_ConvertHex1
0D1	209	OS_ConvertHex2
0D2	210	OS_ConvertHex4
0D3	211	OS_ConvertHex6

SWI number in hex	SWI number in decimal	SWI name
0D4	212	OS_ConvertHex8
0D5	213	OS_ConvertCardinal1
0D6	214	OS_ConvertCardinal2
0D7	215	OS_ConvertCardinal3
0D8	216	OS_ConvertCardinal4
0D9	217	OS_ConvertInteger1
0DA	218	OS_ConvertInteger2
0DB	219	OS_ConvertInteger3
0DC	220	OS_ConvertInteger4
0DD	221	OS_ConvertBinary1
0DE	222	OS_ConvertBinary2
0DF	223	OS_ConvertBinary3
0E0	224	OS_ConvertBinary4
0E1	225	OS_ConvertSpacedCardinal1
0E2	226	OS_ConvertSpacedCardinal2
0E3	227	OS_ConvertSpacedCardinal3
0E4	228	OS_ConvertSpacedCardinal4
0E5	229	OS_ConvertSpacedInteger1
0E6	230	OS_ConvertSpacedInteger2
0E7	231	OS_ConvertSpacedInteger3
0E8	232	OS_ConvertSpacedInteger4
0E9	233	OS_ConvertFixedNetStation
0EA	234	OS_ConvertNetStation
100+	256+	Write character with ASCII value SWI number – 256
200+	512+	Available to the user

F : OSBYTE Routines

Routine	Number	Function
&00	(0)	Display OS version information
&01	(1)	Write user flag
&02	(2)	Specify input stream
&03	(3)	Specify output stream
&04	(4)	Cursor key status
&05	(5)	Write printer driver type
&06	(6)	Write printer ignore character
&07	(7)	Write RS423 receive rate
&08	(8)	Write RS423 transmit rate
&09	(9)	Write duration of first colour
&0A	(10)	Write duration of second colour
&0B	(11)	Write keyboard auto-repeat delay
&0C	(12)	Write keyboard auto-repeat rate
&0D	(13)	Disable event
&0E	(14)	Enable event
&0F	(15)	Flush buffer
&12	(18)	Reset function keys
&13	(19)	Wait for vertical sync (vsync)
&14	(20)	Reset font definitions
&15	(21)	Flush selected buffer
&19	(25)	Reset group of font definitions
&1A to &69	(26) (105)	Reserved
&6A	(106)	Select pointer/activate mouse
&6B to &74	(107) (111)	Reserved
&70	(112)	Write VDU driver screen bank
&71	(113)	Write display hardware screen bank
&72	(114)	Write shadow/non-shadow state

Routine	Number	Function
&73	(115)	
to		Reserved
&74	(116)	
&75	(117)	Read VDU status
&76	(118)	Reflect keyboard status in LEDs
&78	(120)	Write keys pressed information
&79	(121)	Keyboard scan
&7A	(122)	Keyboard scan from 16 decimal
&7C	(124)	Clear escape condition
&7D	(125)	Set escape condition
&7E	(126)	Acknowledge escape condition
&7F	(127)	Check for end of file
&80	(128)	Get buffer/mouse status
&81	(129)	Read key with time limit
&86	(134)	Read text cursor position
&87	(135)	Read screen mode and character at text cursor position
&88	(136)	
to		Reserved
&89	(137)	
&8A	(138)	Insert character code into buffer
&8B	(139)	Write filing system options
&8C	(140)	
to		Reserved
&8E	(142)	
&8F	(143)	Issue module service call
&90	(144)	Set vertical screen shift and interlace
&91	(145)	Get character from buffer
&92	(146)	
to		Reserved
&97	(151)	
&98	(152)	Examine buffer status
&99	(153)	Insert character into buffer
&9A	(154)	
to		Reserved
&9B	(155)	
&9C	(156)	Read/write asynchronous communications state

Routine	Number	Function
&9D to &9F	(157) to (159)	Reserved
&A0	(160)	Read VDU variable value
&A1	(161)	Read battery backed RAM
&A2	(162)	Write battery backed RAM
&A3	(163)	Read/write general graphics information
&A5	(165)	Read output cursor position
&B1	(177)	Read/write input source
&B2	(178)	Read/write keyboard semaphore
&B5	(181)	Read/write RS423 input interpretation status
&B6	(182)	Read NOIGNORE state
&BF	(191)	Read/write RS423 busy flag
&C2	(194)	Read/write duration of second colour
&C3	(195)	Read/write duration of first colour
&C4	(196)	Read/write keyboard auto-repeat delay
&C5	(197)	Read/write keyboard auto-repeat rate
&C6	(198)	Read/write *EXEC file handle
&C7	(199)	Read/write *SPOOL file handle
&C8	(200)	Read/write BREAK and ESCAPE effect
&C9	(201)	Read/write keyboard status
&CA	(202)	Read/write keyboard status byte
&CB	(203)	Read/write RS423 input buffer minimum space
&CC	(204)	Read/write RS423 ignore flag
&D1	(209)	Reserved
&D2	(210)	Read/write sound suppression status
&D3	(211)	Read/write bell channel
&D4	(212)	Read/write bell sound information
&D5	(213)	Read/write bell frequency
&D6	(214)	Read/write bell duration
&D8	(216)	Read/write length of function key string
&D9	(217)	Read/write paged mode line count
&DA	(218)	Read/write bytes in VDU queue
&DB	(219)	Read/write TAB key code
&DC	(220)	Read/write escape character

Routine	Number	Function
&DD to &E0	(221) (224)	Read/write interpretation of input values 195 to 255
&E1 to &E4	(225) (228)	Read/write function key interpretation
&E5	(229)	Read/write escape key status
&E6	(230)	Read/write escape effects
&EB	(235)	Reserved
&EC	(236)	Read/write character destination status
&ED	(237)	Read/write cursor key status
&EE	(238)	Read/write numeric keypad interpretation
&F0	(240)	Read country flag
&F1	(241)	Read/write user flag
&F3	(243)	Read/write timer switch state
&F5	(245)	Read printer driver type
&F6	(246)	Read/write printer ignore character
&FD	(253)	Read last break type
&FE	(254)	Set effect of SHIFT on numeric keypad
&FF	(255)	Read/write startup options

G : OSWORD Routines

Routine	Number	Function
&00	(0)	Read line from input stream
&01	(1)	Read system clock
&02	(2)	Write system clock
&03	(3)	Read interval timer
&04	(4)	Write interval timer
&09	(9)	Read pixel logical colour
&0A	(10)	Read character definition
&0B	(11)	Read colour palette
&0C	(12)	Write colour palette
&0D	(13)	Read current/previous graphics co-ords
&0E	(14)	Read CMOS clock
&0F	(15)	Write CMOS clock
&15	(21)	Define hardware cursor and mouse parameters
&16	(22)	Write screen base address

H : VDU Control Codes

VDU Code	Ctrl	Extra bytes	Meaning
0	@	0	Does nothing
1	A	1	Sends next character to printer only
2	B	0	Enables printer
3	C	0	Disables printer
4	D	0	Writes text at text cursor
5	E	0	Writes text at graphics cursor
6	F	0	Enables VDU driver
7	G	0	Generates bell sound
8	H	0	Moves cursor back one character
9	I	0	Moves cursor on one space
10	J	0	Moves cursor down one line
11	K	0	Moves cursor up one line
12	L	0	Clears text area
13	M	0	Moves cursor to start of current line
14	N	0	Turns on page mode
15	O	0	Turns off page mode
16	P	0	Clears graphics area
17	Q	1	Defines text colour
18	R	2	Defines graphics colour
19	S	5	Defines logical colour
20	T	0	Restores default logical colours
21	U	0	Disables VDU drivers or deletes current line
22	V	1	Selects screen mode
23	W	9	Multi-purpose command
24	X	8	Defines graphics window
25	Y	5	PLOT
26	Z	0	Restores default windows
27	[	0	Does nothing
28	\	4	Defines text window
29	]	4	Defines graphics origin
30	^	0	Homes text cursor
31	_	2	Moves text cursor

I : Plot Codes

The groups of PLOT codes are as follows:

0 - 7	(&00 - &07)	Solid line including both end points
8 - 15	(&08 - &0F)	Solid line excluding final points
16 - 23	(&10 - &17)	Dotted line including both end points
24 - 31	(&18 - &1F)	Dotted line excluding final points
32 - 39	(&20 - &27)	Solid line excluding initial point
40 - 47	(&28 - &2F)	Solid line excluding both end points
48 - 55	(&30 - &37)	Dotted line excluding initial point
56 - 63	(&38 - &3F)	Dotted line excluding both end points
64 - 71	(&40 - &47)	Point plot
72 - 79	(&48 - &4F)	Horizontal line fill (left & right) to non-background
80 - 87	(&50 - &57)	Triangle fill
88 - 95	(&58 - &5F)	Horizontal line fill (right only) to background
96 - 103	(&60 - &67)	Rectangle fill
104 - 111	(&68 - &6F)	Horizontal line fill (left & right) to foreground
112 - 119	(&70 - &77)	Parallelogram fill
120 - 127	(&78 - &7F)	Horizontal line fill (right only) to non-foreground
128 - 135	(&80 - &87)	Flood to background
136 - 143	(&88 - &8F)	Flood to foreground
144 - 151	(&90 - &97)	Circle outline
152 - 159	(&98 - &9F)	Circle fill
160 - 167	(&A0 - &A7)	Circular arc
168 - 175	(&A8 - &AF)	Segment
176 - 183	(&B0 - &B7)	Sector
184 - 191	(&B8 - &BF)	Block copy/move

192 - 199	(&C0 - &C7)	Ellipse outline
200 - 207	(&C8 - &CF)	Ellipse fill
208 - 215	(&D0 - &D7)	Graphics characters
216 - 223	(&D8 - &DF)	Reserved for Acorn expansion
224 - 231	(&E0 - &E7)	Reserved for Acorn expansion
232 - 239	(&E8 - &EF)	Sprite plot
240 - 247	(&F0 - &F7)	Reserved for user programs
248 - 255	(&F8 - &FF)	Reserved for user programs

Within each block of eight the offset from the base number has the following meaning:

0	Move cursor relative (to last graphics point visited)
1	Draw relative using current foreground colour
2	Draw relative using logical inverse colour
3	Draw relative using current background colour
4	Move cursor absolute (ie, move to actual co-ordinate given)
5	Draw absolute using current foreground colour
6	Draw absolute using logical inverse colour
7	Draw absolute using current background colour

The above applies except for COPY and MOVE where the codes are as follows:

184 (&B8)	Move only, relative
185 (&B9)	Move rectangle relative
186 (&BA)	Copy rectangle relative
187 (&BB)	Copy rectangle relative
188 (&BC)	Move only, absolute
189 (&BD)	Move rectangle absolute
190 (&BE)	Copy rectangle absolute
191 (&BF)	Copy rectangle absolute

J : Programs Disc

A disc of software is available from Dabs Press to accompany this book. It contains all the example and tutorial programs listed in the previous chapters. In addition several other useful utility programs are available – a total of 74 programs!

In addition to the programs contained within this book, you will find the following programs invaluable aids:

A complete memory editor with ARM exception handler
A demonstration of interrupt driven colours on the Archimedes
A user friendly function key display/editor
A memory block movement utility
An ADFS disc sector editor
An RGB colour definer allowing creation of all 4096 colours
A memory block fill utility
A full scrolling ARM disassembler
Templates for implementing BASIC statements in ARM machine code
A string and byte memory search utility
Stack simulation program
All 65 tutorial programs listed in this book

All of the programs on the disc are available from a menu for ease of use. The extra programs on the disc show off the new features of the remarkable Archimedes machine. These are useful as stand alone utilities and also of interest in understanding how the different systems can be controlled.

The disc is available in 3.5in ADFS format and the programs are not copy protected in anyway, so you are free to integrate them into your own software as it develops. The disc is compatible with all versions of the Archimedes including the A305, A310, A410 and A440.

The cost of the disc is just £9.95 and it comes supplied with a small user guide which details how to use all the programs and provides additional documentation for the bonus programs.

To obtain your copy of the Archimedes Assembly Language programs disc send £9.95 to the address given below. Cheques and POs should be made payable to Dabs Press. Access and Visa card orders are acceptable by phone, or via Prestel or Telecom Gold by quoting your number and expiry date – and don't forget your address!

By post:

Dabs Press
76 Gardner Road
Prestwich
Manchester
M25 7HU

By phone:

061-773-2413

By electronic mail:

Messages to Telecom Gold 72:MAG11596 and Prestel 942876210

K : Dabhand Guides

The following Dabhand Guides and software packs are published or planned for 1988. Leaflets are available on all these products which go into considerably more detail than space here permits. Publication dates and contents are subject to change. All quoted prices are inclusive of VAT (on software, books are zero-rated), and postage and packing (abroad add £2 or £10 airmail). All are available from your local dealer or bookshop or in case of difficulty direct from Dabs Press – see page 360.

Books for the BBC Micros

Archimedes Operating System: A Dabhand Guide
By Alex and Nick van Someren
ISBN: 1-870336-48-8. Available Autumn 1988. 250 pages approx
Price: £14.95. 3.5in disc, £9.95. Book and disc together, £21.95

The book that is a must for every serious Archimedes owner. It describes how the Archimedes works and examines the Arthur operating system in microscopic detail, giving the programmer a real insight into getting the best from the Archimedes.

The book is intended for the serious machine code, or BASIC, programmer and includes sections on: the ARM instruction set, SWIs, graphics, Writing relocatable modules, vectors, compiled code, MEMC, VIDC, IOC and much, much more.

Master Operating System: A Dabhand Guide by David Atherton
ISBN 1-870336-01-1. Available now. 272 pages. Book : £12.95; 5.25in disc £7.95; 3.5in disc £9.95. Book and disc together, £17.95 (£19.95 with 3.5in)

The Master owners bible. Acclaimed reference guide for programmers and users of the BBC B+ and Master Series micros. Contains a wealth of information on the operating system, including all the * commands, OSBYTE and OSWORD calls, the Tube, filings systems and the differences between the various BBC micros. A&B Computing said it's *'invaluable'* – we agree!

C: A Dabhand Guide by Mark Burgess
ISBN: 1-870336-16-X. Available May 1988. Book, £14.95. 512 pages
Archimedes disc, £9.95. Book and disc £21.95

This massive 512 page book provides a comprehensive tutorial in C – fast becoming the *de facto* language for all micros. This book is ideal for the beginner and starts from first principles. It includes sections on all the major micros including the Master and the Archimedes.

VIEW: A Dabhand Guide by Bruce Smith
ISBN 1-870336-00-3. Publication : Available now. 248 pages
Book: £12.95. Disc: DFS 5.25in, £7.95 ADFS 3.5in, £9.95
Book and disc together, £17.95 (ADFS £19.95)

This top selling guide to VIEW, now in its second edition, has received rave reviews and is an absolute must for all VIEW users. This is what they said:

John Allen speaking on Radio London: *'It's very good...'*

Mike Williams, Beebug magazine June 1987: *'...much more to offer the competent VIEW user...practical and down-to-earth...for those who want a complete, thorough and readable guide to VIEW then Bruce Smith is your man.'*

Bill Penfold, Acorn User September 1987: *'This is the first computer book I've read in bed for pleasure rather than to cure insomnia.'*

ViewSheet and Viewstore: A Dabhand Guide by Graham Bell
ISBN 1-870336-04-6. Available now. 352 pages
Book: £12.95. Disc: DFS 5.25in, £7.95; ADFS 3.5in, £9.95
Book and disc together, £17.95 (ADFS £19.95)

A complete tutorial and reference guide for the Acornsoft ViewSheet spreadsheet and the ViewStore database manager, specifically written to appeal both to the beginner and to the more knowledgeable user. Also covers ViewPlot and OverView.

Master 512: A Dabhand Guide by Chris Snee
ISBN 1-870336-14-3 Publication: May 1988. 200 pages approx. Book: £14.95

At last, the book that all Master 512 owners have been waiting for. Covers:

What you get on the discs, DOS Plus versions, explanation of the filing system, DOS Plus CLI commands (syntax, abbreviations and errors), transient commands, file types, reserved extensions, reserved words, I/O, the 512 memory map, how a PC works, 8086 registers, MS-DOS, 512 Tube, the 80186 monitor, differences between DOS Plus and MS-DOS, making software work on the 512, colour limitations, hard disc set-up, PC disc formats, software compatibility, public domain software...

Bumper Assembler Bundle by Bruce Smith
Publication : Available Now. Two books, two discs and booklet, just £9.95

Five part package providing a complete tutorial in 6502 machine code at a third of their normal price. Full details on request.

Mini Office II: A Dabhand Guide by Bruce Smith and Robin Burton
ISBN: 1 870336 55 0. Publication Summer 1988. 300 pages approx.

A complete guide to this award winning software covering every aspect of using this powerful software package .

BBC and Master Software Packs

HyperDriver by Robin Burton

Software pack in ROM, £29.95. Sideways RAM version, only £24.95

HyperDriver is the ultimate printer ROM. And if you have a printer, then this will be the most significant purchase you can make. It's absurdly easy to use and provides you with many of the facilities missing from your current software including: on-screen preview, CRT graphics, NLQ font and user-definable macros to name but a few. No matter what you use your printer for, wordprocessing, spreadsheets, databases, programming you will have in excess of 80 * commands available for instant use from within applications such as VIEW, InterWord and so on.

The HyperDriver pack contains a 16k EPROM , and a Sideways RAM image on disc.A full and comprehensive 100-page manual and reference card complete this value for money package.

'The thought that's gone into the way HyperDriver is used with wordprocesors and a million over good design features make the value of this ROM stand out...an ingenious blessing.' Geoff Bains, March 1988, Beebug.

FingerPrint by David Spencer
Available Now! Disc & manual, DFS version, £9.95; ADFS version, £11.95

A unique single-step machine code tracing program allowing you to step through any machine code program. FingerPrint will even trace code situated in Sideways RAM/ROM – learn how BASIC works!

MOS Plus by David Spencer
Available Now! ROM, £12.95; disc for Sideways RAM, £7.95 (3.5in, £9.95)

For the Master 128. Provides ADFS *FORMAT, *VERIFY , *BACKUP, *CATALL and *EXALL in ROM and new * commands such as *FIND – which finds a file anywhere on an ADFS disc. A complete alarm system is present using the Master 128 alarm facility, as is an AMX Mouse driver.

'MOS Plus is an excellent product', Dave Somers, March 1988, Beebug

SideWriter by Mike Ginns
Available Now. 5.25in DFS disc, £7.95; 3.5in ADFS disc, £9.95;

For Sideways RAM owners this is a pop-up notepad which can be used from within any application. Notes taken in SideWriter can be saved to disc, transferred to a wordprocessor, or printed out.

Master Emulation ROM by David Spencer
Available Now. ROM version, £19.95 (disc for Sideways RAM, £14.95)

Provides model B and B+ owners with most of the features of the Master 128, such as the new * commands, the extended filing system operations including the temporary filing system, the *CONFIGURE system (using battery-backed Sideways RAM and/or a disc file), and if you have the hardware, Sideways or Shadow RAM. The only Master operating system software not covered in this ROM, is the extended graphics software. Works with all popular SRAM boards.

'...the whole system feels like a Master...most impressive is an almost complete emulation of the temporary filing system...' Bernard Hill, March 1988 Beebug.

Other Books from Dabs Press

AmigaDOS: A Dabhand Guide by Mark Burgess
ISBN 1-870336-47-X. Publication : July 1988. 300 pp approx. Price: £14.95

WordStar 1512: A Dabhand Guide – Including WordStar Express by Bruce Smith
ISBN 1-870336-17-8. Publication : Summer 1988. 300 pages approx
Book: £12.95. Disc: 5.25in, £7.95; book and disc together, £17.95

PCW 9512: A Dabhand Guide by John Atherton
ISBN 1-870336-50-X. Publication: Third quarter. 300 pages approx

Z88 Advanced User Guide by David Spencer
ISBN 1-870336-60-7. Publication: Third quarter. 300 pages approx

Z88 PipeDream: A Dabhand Guide by Rob Miller
ISBN 1-870336-61-5. Publication: Third quarter. 300 pages approx

WordPerfect: A Dabhand Guide by Mark Burgess
ISBN 1-870336-53-4. Publication: Fourth quarter. 350 pages approx

PostScript: A Dabhand Guide by Paul Martin
ISBN 1-870336-54-2. Publication: Fourth quarter. 300 pages approx

Ability Plus: A Dabhand Guide by Geoff Cox
ISBN 1-870336-51-8 . Publication: Third quarter. 300 pages approx

SuperCalc 3.1/3.2: A Dabhand Guide by A A Berk
Publication: Fourth quarter. 300 pages approx

Please note:

All future publications are in an advanced state of preparation. Content lists serve as a guide, but we reserve the right to alter and adapt them without notification. If you would like more information about Dabs Press, books and software, then drop us a line at 76 Gardner Road, Prestwich Manchester M25 7HU, and we'll send our latest catalogue.

Index

abort error, memory 23
ABS 270
absolute, addressing 115,124
ADC 69,87
ADD 69,85
addition 69,85,87
addition multi-word 87
addition, rules of 330
address bus 20,25,32
address exceptions 23,180
address space 21,23,205
addressing absolute 115,124
addressing byte mode 21,125,253
addressing indirect 116
addressing modes 115,117,125
addressing PC relative addressing 124,131
addressing post-indexed 117,122,253
addressing pre-indexed 117,121,124
addressing register 116,275
ADR 51,193,253
AL, conditional suffixes 62
ALIGN, assembler 154
ALU 27
AND 69,100,104,273,282,283,335
AND truth table 335
animation 318
anti-aliasing, fonts 230,235
anti-aliasing pallete, fonts 232,235
applications, stacks 146
ARM 13 14,17,173
array support 116,120,122,156,274,275
arrays, dimensioning 274
ARTHUR 35,138,158,190,208
ascending stacks 142,144
ASL, shifts 75,78
ASR, shifts 75,81
assembler 41,124,147
assembler ALIGN 154
assembler comments 49
assembler conditional assembly 161,164,166
assembler conditional suffixes 56,69,85,95, 132,280,281
assembler directives 52,147,152,154
assembler entering 43,162
assembler EQUB 152,153,154,221
assembler EQUD 152,221
assembler EQUS 152,153,154, 193, 221,243,252
assembler EQUW 152,154,221
assembler error reporting 148,149
assembler forward references 149
assembler, labels 50,52,131,149
assembler listings 44,47,147,148,163
assembler location counter P% 44,45, 50, 52,152,154,274
assembler macro assembly 161,166
assembler macro parameters 164
assembler object code 41,43,44
assembler OPT settings 147,148,151
assembler passes 150
assembler pseudo addressing 124
assembler reserving memory 45,152
assembler source code 41,43,46,147
assembly, offset 148,151
attributes, icon 216
auto-increment/decrement 126

B suffix 125,170
barrel shifter 28
base address/register 117,118,122,126,275
BASIC the assembler 41,52,150,161,164
BASIC functions 52,162,270

BIC 69 103
binary addition examples 331
binary arithmetic 330
binary numbers 322
binary on the Archimedes 325
binary pattern program 322
binary representation 320
binary signals 320
binary strings, conversion to 198
binary subtraction examples 334
binary subtraction 331
binary to decimal conversion 324
binary to hex conversion 328
binary weightings 323
bit 320
bits, grouping 321
bitwise logical operators 100,101,102,103, 273,335
BL 32,131,134,291
borrow in subtraction 90
branch offset 131
branches 33,39,131,149,180,280,285, 286, 290,291
branches, conditional 132,279
breakpoints, debugger 168
bus width 19
bus 19,25
byte mode, addressing 21,125,253
B 131
BY 304
byte 321

cache, font 226,238
CALL 48,155
CALL parameter block 155
CALL parameter types 156
carry digit 330
carry flag 32,34,61,77,79,81,83,84,87, 90, 92,106
case sensitivity 138
case statement example 290
case 289
CC, conditional suffixes 61,67
character strings 125,152,153,154,156,157,159,192,193,196,243
CIRCLE FILLED 309
CIRCLE template example 308
CIRCLE 120 293,301,307
CLG 313
CLS 313
CMN 63,99
CMP 58,63,69,95,106,270,279
co-ordinates, graphics 297,299,302, 317
co-processor instructions 170
colour re-definition 315
COLOUR 314
colour, fonts 231,235,236
comments, assembler 49
comparisons 283,285,289
comparsions 58,63,69,95,99,256,279
condition codes 55
conditional assembly an example 165
conditional assembly, assembler 161,164, 166
conditional branches 132,279
conditional execution of instructions 39,55,279
conditional suffixes AL 62
conditional suffixes CC 61,67
conditional suffixes CS 61
conditional suffixes EQ 58
conditional suffixes GE 64
conditional suffixes GT 65
conditional suffixes HI 63
conditional suffixes LE 65
conditional suffixes LS 63
conditional suffixes LT 64
conditional suffixes MI 60
conditional suffixes NE 58
conditional suffixes NV 62
conditional suffixes PL 60
conditional suffixes VC 59
conditional suffixes VS 59
conditional suffixes, assembler 56,69,85, 95,132,280,281
conditional suffixes 56,66,69,85,95, 117,132,281

control codes 297,351
conversion routines 195,267,268
conversion to binary strings 198
conversion to decimal strings 196,267
conversion to hex strings 197
conversion, binary to decimal 324
conversion, binary to hex 328
conversion, decimal to binary 326
conversion, hex to binary 328
copying characters 253,255,259,260, 261,265
COSINE 120
CS, conditional suffixes 61

data areas in machine code 152
data field, immediate operands 72
data processing operations 27,69
DCB 152 153
DCD 152
DCW 152
debugger 161,167
debugger breakpoints 168
debugger disassembly 170
debugger entering and exiting 168
debugger memory examination 169,170
debugger, register examination 169,171
debugger tracing programs 168 169,171
debugging machine code 166,167
decimal strings, conversion to 196,267
decimal to binary conversion 326
defining icons 215
descending stacks 142,144
destination operand 70,93,107,112
dimensioning arrays 274
direction of storage 127
directives, assembler 52 147,152,154
disabling events 177
disabling interrupts 173
disassembly, debugger 170
DIV 271
division 79 81,271
DRAW 297,298,304,305

empty, stacks 142,144
enabling events 177
enabling interrupts 173
end of string marker 243,252,256,258, 259,260
entering and exiting, debugger 168
entering, assembler 43,162
EOR 69 102,106,113,273,335,338
EOR truth table 338
EQ, conditional suffixes 58
EQUB, assembler 152,153,154,221
EQUD, assembler 152,221
EQUS, assembler 152,153,154,193,221, 243,252
EQUW, assembler 152,154,221
error handling 184
error reporting, assembler 148,149
event causes 177
events, disabling 177
events, enabling 177
events 176 177,186
exception vectors 179,180
executing machine code 48
execution of instructions, conditional 39, 55,279

FALSE 335
fetch execute cycle 24,27,33,39
fill, flood 312
FILLED, CIRCLE 309
FILLED, RECTANGLE 310
FIRQ 35,173,175
FLIH 172
floating point instructions 170
flood fill 312
FN, functions 162
font cache 226,238
font files 226,227
font handle 227
font painting example 229
font workspace 226
fonts anti-aliasing 230,235
fonts anti-aliasing pallete 232,235

fonts colour 231,235,236
fonts demonstration of anti-aliasing 233
fonts 226

fonts initialising 227
fonts losing fonts 238
fonts painting 228
fonts resolutions 230
fonts transfer functions 234
FOR...NEXT loop demonstration 287
FOR...NEXT 239,286
format of instructions 42,71,115,339
format of templates 242
forward references, assembler 149
full stacks 142,144
functions FN 162

GCOL options 316
GCOL316
GE, conditional suffixes 64
GET 245
graphics 120,135,209,293,297
graphics co-ordinates 297,299,302,317
grouping bits 321
GT, conditional suffixes 65

hard disc 23,24
hex digits 327
hex on the Archimedes 329
hex strings, conversion to 197
hex to binary conversion 328
hexadecimal 326
HI, conditional suffixes 63

icon attributes 216
icons, defining 215

IF...THEN example 281
IF...THEN...ELSE, multi-conditioned 282
IF...THEN...ELSE..ENDIF 279 289

illegal immediate operands 73
illegal instruction exceptions 180
immediate operands data field 72
immediate operands position field 72
immediate operands, illegal 73
immediate, operands71,74,107,117,119
indirect, addressing 116

initialising, fonts 227
INKEY 245
INPUT 194,243,270
input/output techniques 18,191,243
input/output, memory mapped 18
INSTR 263
instruction formats 42
instruction groups 68
integer variables 53
interrupt flags 33,35,174
interrupt intercept routines 176,205
interrupt program rules 176
interrupts on the Archimedes 35,172, 173,174
interrupts returning from 175,185
interrupts service routines 172,174, 176,205

interrupts, disabling 173
interrupts, enabling 173
interrupts 33 35,172,185,205
interrupts, software 138
interrupts, sources of 173
IRQ 35,173,175

labels, assembler 50,52,131,149
LDM 126 142,143
LDR 115 126,275
LE, conditional suffixes 65
LEFT$ 259,260
LEN 258,259,260,261
LIFO 140,145,292
LINE template example 305
LINE 301,305
link register 32,48,93,134,137,175,291
listings, assembler 44,47,147,148,163

loading, registers 115,126,253
local variables 292
location counter P%, assembler 44,45,50, 52,152,154,274
logical operators, bitwise 100,101,102, 103,273,335
loops 254,256,263,265,279,285,286
losing fonts 238
LS, conditional suffixes 63
LSL, shifts 75,77,275
LSR, shifts 75,79
LT, conditional suffixes 64

machine code, executing 48
macro assembly, assembler 161,166
macro parameters, assembler 164
masks 103,104,113,218,336,337,338
MEMEC 23
memory abort error 23
memory access 19,20,38,115
memory examination, debugger 169,170
memory management 23,205
memory mapped input/output 18
memory 18,29,115,205
MI, conditional suffixes 60
MID$ 261
MLA 69,109
mnemonics 41 42,69,74
MOD 271
mode flags 34,110,113
MODE 313
mouse 138,188,208,210,221
MOV 49,69,93,111,137,270
MOVE 297,301,303,304,305
multi-conditioned IF..THEN..ELSE 282
multi-word, addition87
MUL 69,78,107
multiple transfer options 127
multiple register transfer 125
multiplication 69,107,109,120,275
MVN 69,94,270,273

names in the assembler, register 42
NE, conditional suffixes 58
negative flag 32,33,34,60,113
negative number representation 60,78, 79,81,94,99,270,286,332,333,338
NOT 273
NV, conditional suffixes 62

O% 151
object code, assembler 41,43,44
offset assembly 148,151
offset field/register 117,118,122,123,124
offset, branch 131
OFF 318
ON 318
one's compliment 333,338
operands 27 29,42,69,70,74,91,107,110,117
operands immediate 71,74,107,117,119
operands register 29,70,74,107, 110,117,118
operands shifted 39,74,77,93,107, 117,120,275
operating system 158,180,190,297
OPT settings, assembler 147,148,151
options, multiple transfer 127
OR 273,282,336
OR truth table 337
ORIGIN 313
ORR 69,101
OSBYTE 200,245,247,318,346
OSCLI 202
OSRDCH 192
OSWORD 201,350
OSWRCH 191
OS_Args, swi 184
OS_BGet, swi 184
OS_BinaryToDecimal, swi196,269
OS_BPut, swi 184
OS_Byte, swi 160,177,184,200,245, 247,318,346
OS_CallAfter, swi 205
OS_CallEvery, swi 205
OS_Claim, swi 181

OS_CLI, swi 160,184,202
OS_ConvertBinaryN, swi 198
OS_ConvertCardinalN, swi 198
OS_ConvertHexN, swi 197
OS_ConvertIntegerN, swi 198
OS_ConvertSpacedCardinal, swi 199
OS_ConvertSpacedInteger, swi 199
OS_EnterOS, swi 204
OS_File, swi 184
OS_Find, swi 184
overflow flag 32,34,59

P suffix 113
P% 44,45,50,52,152,154,274
painting, fonts 228
parameter block, CALL 155
parameter types, CALL 156
passes, assembler 150
passing data to machine code 53,155, 159,253
PC relative addressing, addressing 124,131
pipelining 25,111,113,124,134,175
pixels 231
PL, conditional suffixes 60
PLOT example 301
PLOT 135,188,298,303,352
POINT 303,304
POINT() 317
position field, immediate operands 72
post-address modification 127
post-indexed, addressing 117,122,253
POS 247,249
pre-address modification 127,275
pre-indexed, addressing 117,121,124
preserving registers 130,137,175,176,292
PRINT 247
printer 164
private, registers 35,175
procedure parameters 293
procedures 291
processor modes 32,34,113,138,174,204
program counter 32,43,49,93,107,110, 111,112, 131,134,137,174
pseudo-addressing, assembler 124
pull 141,143
push 141,143

R14 32,48,93,134,137,175,291
R15 32,33,34,43,49,93,107,110,111,112, 131,134,137,174
range of immediate operands 71,119
re-entrant code 176
RECTANGLE 301,310,311
recursive programs 137,292,293
register examination, debugger 169,171
register list 126,143,159
register names in the assembler 42
register transfer, multiple 125
register usage in templates 242
register, operands 29,70,74,107,110, 117,118
registers, private 35,175
registers 29,35,38,42,53,110,115,126, 159,175,204
RECTANGLE FILLED 310
relative co-ordinates 304,307
relocatable programs 131,151
REPEAT...UNTIL 285
reserving memory, assembler 45,152
resolutions, fonts 230
restrictions, multiply instructions 107,109
returning data from machine code 53
returning from machine code 48
returning from, interrupts 175,185
RGB colour selection 316
RIGHT$ 260
RISC 13,34,37
ROR, shifts 75,83
rotate 83 84
RRX,,shifts 75,84
RSB 69 91
RSC 69,92
rules of addition 330

S, suffix 66,69,85,88,90,95,99,100, 104,106,113,137
SBC 69,90
screen memory 118

screen mode 234
service routines, interrupts 172,174, 176,205
SGN 270
shifted, operands 39,74,77,93, 107,117,120,275
shifts 28,39,61,74,77,93,120,164,275
shifts ASL 75,78
shifts ASR 75,81
shifts LSL 75,77,275
shifts LSR 75,79
shifts ROR 75,83
shifts RRX 75,84
sign bit 332
SINE 120
sketch pad 209
skipping instructions 111,113
software interrupts 138
SOUND 277
Sound_Control, swi 277
source code, assembler 41,43,46,147
sources of interrupts 173
SPC 247,249
special purpose registers 31,110
stack model 140
stack option codes 144
stack pointer 141,143
stack types 142,144
stacks applications 146
stacks, ascending 142,144
stacks, descending 142,144
stacks, empty 142,144
stacks, full 142,144
stacks 129,137,140,292,293
status flags 32,33,39,55,66,85,93,95,104,110, 113,159,174,279,281
status register 32,33,55,66,85,93,95,104, 110,113,159,174,279
STM 126,142,143
storing registers 115,126,253
STR$ 268
string assignment 253
string comparison 256,263,284
string concatenation 255,265
string information block (SIB) 156,157
string representation 252
string searching 263
string, termination 243,252,256, 258,259,260
STRING$ 265
strings, character 125,152,153,154,156, 157,159,192,193,196,243
STR 115 126,196,275
SUB 69,89,175
subroutines 32,93,134,135,146,161,291
subtraction 89,90
suffixes 42,56,66,69,85,113,117,125
SUM 275
supervisor mode 35,138,204
SVC 35
swi 256 + n 191,298
swi OS_Args 184
swi OS_BGet 184
swi OS_BinaryToDecimal 196,269
swi OS_BPut 184
swi OS_Byte 160,177,184,200,245, 247,318,346
swi OS_CallAfter 205
swi OS_CallEvery 205
swi OS_Claim 181
swi,OS_CLI 160,184,202
swi,OS_ConvertBinaryN 198
swi OS_ConvertCardinalN 198
swi OS_ConvertHexN 197
swi OS_ConvertIntegerN 198
swi OS_ConvertSpacedCardinal 199
swi OS_ConvertSpacedInteger 199
swi OS_EnterOS 204
swi OS_File 184
swi OS_Find 184
swi OS_plot 302
swi 138 159,180,187,190,343
swi Sound_Control 277

templates, format of 242
templates, register usage in 242
termination, string 243,252,256, 258,259,260

tracing programs, debugger 168,169,171
transfer functions, fonts 234
truth table, AND 335
truth table, EOR 338
truth table, OR 337

VC, conditional suffixes 59
VS, conditional suffixes 59

A Dabhand Guide

Learn how to get the most from the remarkable Archimedes micro by programming directly in the machine's own language – ARM machine code. This book covers all aspects of machine code/assembler programming specifically for the Archimedes range. The book is applicable to the A305, A310, A410 and A440 computers.

For those new to assembler programming, the book contains sections which take you step by step through topics such as the binary number system, 32-bit machine arithmetic and logic operation.

The inbuilt BASIC assembler and machine code debugger are fully explained. There is a large section on implementing BASIC equivalents in machine code, and much coverage of the 'Arthur' operating system. Details of the many facilities are provided, together with details of how to access them from machine code. Just some of the many areas covered are:

- The ARM (Acorn RISC machine) processor
- Descriptions of ALL processor instructions
- The BASIC assembler including macro assembly
- Implementing BASIC commands in machine code
- Accessing the 'Arthur' operating system
- Details of the new SWI calls
- Using the mouse, windows, and fonts
- Operating system vectors
- Interrupt handling
- Beginners tutorial section
- VDU codes, SWI numbers, instruction formats
- Example programs (also available on disc)

Mike Ginns read computer science at Reading University, and has been programming the BBC Micro and other computers in assembly language for over six years, contributing on many occasions to Acorn User magazine.

£14.95

ISBN 1-870336-20-8

9 781870 336208