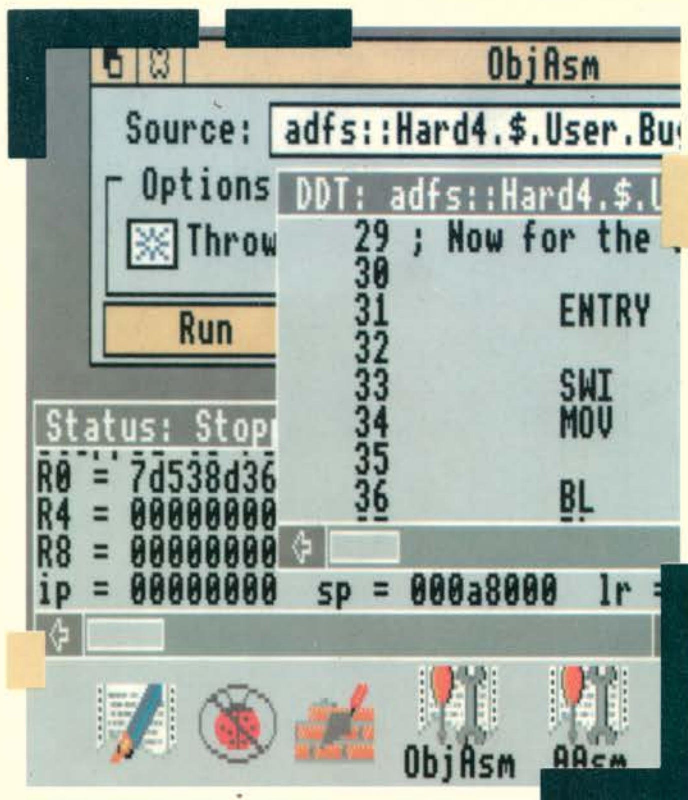
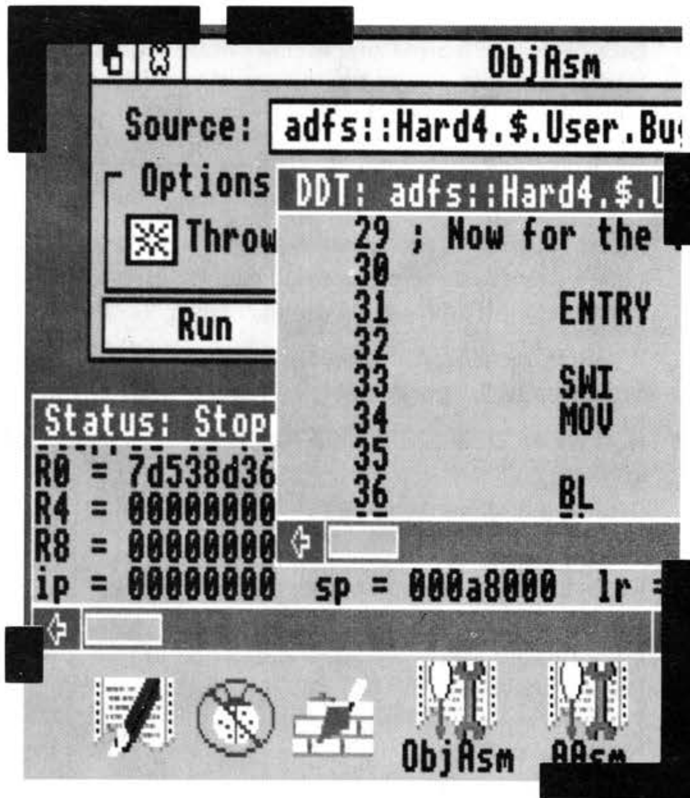
ACORN
ASSEMBLER RELEASE 2



Acorn

ACORN ASSEMBLER RELEASE 2



Acorn

Copyright © Acorn Computers Limited 1991

Published by Acorn Computers Technical Publications Department

Neither the whole nor any part of the information contained in, nor the product described in, this manual may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited.

The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers Limited in good faith. However, Acorn Computers Limited cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

If you have any comments on this manual, please complete the form at the back of the manual, and send it to the address given there.

Acorn supplies its products through an international dealer network. These outlets are trained in the use and support of Acorn products and are available to help resolve any queries you may have.

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

ACORN, ARCHIMEDES and ECONET are trademarks of Acorn Computers Limited.

UNIX is a trademark of AT&T.

Published by Acorn Computers Limited
Release 2
ISBN 1 85250 096 4
Part number 0470,589
Product number SKD36
Issue 1, May 1991

Contents

Contents iii

Introduction 1

Assembler tools 1

This user guide 2

Conventions used in this manual 3

Part 1 - Using the assemblers 5

Assemblers and the DDE 7

Using the assemblers through Make 7

Editor throwback 8

DDT debugging 11

Using FrontEnd on your programs 15

Making your own linkable libraries 16

AAsm 19

Starting AAsm 20

AAsm SetUp options 21

AAsm output 25

AAsm icon bar menu 26

Example AAsm session 27

AAsm managed by Make 28

AAsm command lines 28

ObjAsm 31

Starting ObjAsm 31

ObjAsm SetUp options 33

ObjAsm output 37

ObjAsm icon bar menu 38

Example ObjAsm session 39

ObjAsm command lines 40

Part 2 - Assembly language details 43

The ARM cpu 45

Registers 46
FIQ fast interrupt request 48
IRQ interrupt request 49
Address exception trap 49
Abort 50
Software interrupt 51
Undefined instruction trap 51
Reset 52
Vector summary 52
Modes of operation 52

Assembler language 55

Assembler language syntax 55
Symbols and labels 56
Expressions 57
Numeric constants 59
String constants 59
Boolean constants 59
Assembler operators 60
?Label 63
Operator summary 64

CPU instruction set 65

Conditional execution 65
Instruction timing 66
The barrel shifter 68
Shift types 70
Branch instructions 72
Data processing 73
Data processing instruction syntax 74
Data processing instruction summary 79
Single data transfer 79
Single data transfer instruction syntax 80
Block data transfer 81
Block data transfer instruction syntax 82
Stacking 84
Block data transfer: special points 88
Single data swap 89
Single data swap instruction syntax 90
Multiply and multiply-accumulate 90
Multiply instruction syntax 91
Supervisor calls 92
Supervisor calls instruction syntax 92
Coprocessor instructions 93
Coprocessor data operations 93
Coprocessor data operation instruction syntax 94
Coprocessor/memory transfers 94
Coprocessor/memory transfer instruction syntax 95
Coprocessor/register transfers 95
Coprocessor/register transfers instruction syntax 96
Summary of assembler mnemonic combinations 96
Further instructions 98

Floating point instructions 101

Programmer's model 101
Floating point status register 107
Floating Point Control Register 112
Assembler directives and syntax 114
The instruction set 115
Finding out more... 121

Directives 123
Number equating directives: */EQU 123
Register equating: RN 123
Coprocessor equating: CP 124
Coprocessor register equating: CN 124
Store-loading 124
ALIGN 125
LTORG 126
Laying out storage areas 126
Variables 128
Routines and local labels 130
Error handling 132
ORG 133
LEADR 133
END 133
GET 134
LNK 135
Objasm directives 135

Conditional and repetitive assembly 139
Conditional assembly 139
Repetitive assembly 141

Macros 143
Syntax 144
Local variables 145
MEXIT directive 146
Default values 146
Macro substitution method 147
Nesting macros 148
A division macro 148

Part 3 - Developing software for RISC OS 151

Writing relocatable modules in assembler 153
Assembler directives 154
Examples 155

Interworking assembler with C 157
Examples 157

Using memory efficiently 161
Guidelines 161
Recovery from lack of memory 161
Avoiding permanent loss of memory 162
Avoiding memory wastage 163
Using heap_alloc and heap_free 171

Part 4 - Appendices 173

Appendix A - Error messages 175

Appendix B - Directives syntax table 183

Appendix C - Example assembler fragments 185
Using the conditional instructions 185
Pseudo-random binary sequence generator 186
Multiplication by a constant 187
Loading a word from an unknown alignment 188
Sign/zero extension of a half word 188
Return setting condition codes 188
Full multiply 189

Appendix D - ARM datasheet 191

Description of signals 195

Programmers' Model 199

Introduction 199

Registers 199

Exceptions 201

Instruction Set 208

The condition field 208

Branch and branch with link (B, BL) 209

Data processing 211

Multiply and multiply-accumulate (ML) 218

Single data transfer (LDR, STR) 221

Block data transfer (LDM, STM) 226

Software interrupt 233

Co-Processor data operations 235

Co-Processor data transfers 237

Co-Processor register transfers 240

Undefined instructions 242

Instruction set summary 243

Instruction Speeds 243

Index 245

1 Introduction

Acorn Desktop Assembler is a development environment for producing RISC OS desktop applications and relocatable modules written in ARM assembly language. It consists of a number of programming tools which are RISC OS desktop applications. These tools interact in ways designed to help your productivity, forming an extendable environment integrated by the RISC OS desktop. Acorn Desktop Assembler may be used with its sister product, Acorn Desktop C, to provide an environment for mixed C and assembler development.

Acorn Desktop Assembler includes tools to:

- edit program source and other text files
- search and examine text files
- examine some binary files
- assemble small assembly language programs
- assemble and construct more complex programs under the control of makefiles, these being set up from a simple desktop interface
- squeeze finished program images to occupy less disk space
- construct linkable libraries
- debug RISC OS desktop applications interactively
- construct template files for RISC OS desktop applications.

Most of the tools in Acorn Desktop Assembler are also of general use for constructing applications in other programming languages, and are, for example, supplied with Acorn Desktop C. These non-language-specific tools are described in the accompanying *Acorn Desktop Development Environment* user guide.

Installation

Installation of Acorn Desktop Assembler is described in the accompanying *Acorn Desktop Development Environment* user guide.

Assembler tools

The assemblers provided include the following features:

- full support of the ARM instruction set
- global and local label capability

- powerful macro processing
- comprehensive expression handling
- conditional assembly
- repetitive assembly
- comprehensive symbol table printouts
- pseudo-opcodes to control printout

AAsm and Objasm

The Assembler AAsm produces binary image files which can be executed immediately, for example using a `*imagefile` command. A variant of AAsm, ObjAsm, creates object files which cannot be executed directly, but must first be linked using the Link tool. It is often most efficient to construct larger programs from several portions, assembling each portion with ObjAsm before linking them all together with Link. Object files linked with those produced by ObjAsm may be produced from some programming language other than assembler, for example C.

The Link tool is described in the accompanying *Acorn Desktop Development Environment* user guide, in the non-interactive tool section.

This user guide

This document is a reference guide to the Assemblers working as part of the Development Environment of Acorn Desktop Assembler. These assemblers are the only tools in the Acorn Desktop Assembler product which are not used for programming in other languages; the others are described in the accompanying *Acorn Desktop Development Environment* user guide. It is assumed that you are familiar with other relevant Archimedes documentation, such as the:

- Archimedes Welcome guide
- Archimedes User guide
- Programmer's reference manual

A good introduction to writing programs in Assembler on Archimedes is ARM *Assembly Language Programming* by P.J. Cockerell, (Computer Concepts/ MTC, 1987).

Note on program examples

Both general and specific examples of syntax and screen output are given but there are occasions where the full syntax of an instruction and its accompanying screen appearance would obscure the specific points being made. It follows, therefore, that not all the examples given in the text can be used directly since they are incomplete.

Conventions used in this manual

The Assembler has its own interpretations of the punctuation symbols and special symbols which are available from the keyboard. These are:

! " # $ % & ^ @ ( )

| | { } | : . , ; +

- / * = < > ? _

In order to distinguish between characters used in syntax and descriptive or explanatory characters, typewriter style typeface is used to indicate both text which appears on the screen and text which can be typed on the keyboard. This is so that the position of relevant spaces is clearly indicated.

The following typographical conventions are used throughout this manual:

Convention	**Meaning**
`filename`	Text that you must replace with the name of a file, register, variable or whatever is indicated.
&1C	Hexadecimal numbers are preceded with an ampersand.
`{instruction}`	Curly brackets *`{}`* enclose optional items in the syntax. For example, the Assembler AAsm accepts a three field source line which may be expressed in the form: *`{instruction} {label} {;comment}`*
`ALIGN`	Text that you type exactly as it appears in the manual. For example: `L321 ADD Ra,Ra,Ra,LSL #1 ;multiply by 3`

The abbreviation 'DDE' is used in later chapters to mean 'Desktop Development Environment'.

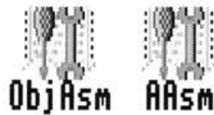
Part 1 - Using the assemblers

2 Assemblers and the DDE

The two versions of the ARM macro assembler, AAsm and ObjAsm, are the only tools included in Acorn Desktop Assembler which are specific to programming in assembly language, hence are described in this volume. All other tools, such as the editor and debugger, are described in detail in the accompanying *Acorn Desktop Development Environment* user guide.

AAsm and ObjAsm both fit into the non-interactive class of DDE tools, which means that, once you have started an assembly process and chosen a set of options, you cannot interact with it to modify its behaviour, except to view output and pause or stop it (interactive DDE tools, such as the DDT debugger, do allow interaction). All non-interactive DDE tools have several features in common. These are described in detail in the chapter entitled *General features* in the accompanying *Acorn Desktop Development Environment* user guide.

To load AAsm or ObjAsm onto the desktop, open a directory display on the DDE directory of your work disk and double click on !AAsm or !ObjAsm. The AAsm or ObjAsm icon then appears on the icon bar. These have the standard screwdriver and spanner appearance of all the non-interactive DDE tools:



From these icons you have access to the interface to set options and start assembly tasks unmanaged by Make. For more details of these interfaces and their use, see the chapters entitled *AAsm* and *Objasm* later in this volume.

Using the assemblers through Make

The DDE Make tool is designed to manage the efficient construction of programs and libraries, usually from several source files. It avoids needless re-processing of unaltered source files and ensures consistent construction by a method specified in a Makefile. For more Make details see the chapter entitled *Make* in the accompanying *Acorn Desktop Development Environment* user guide.

ObjAsm and AAsm, like the other non-interactive DDE tools, can be used by Make to process files. When managed by Make the assemblers are controlled by command lines issued by Make, and their icons need not be present on the icon bar. You don't need to double click on !ObjAsm or !AAsm before starting a Make job using these tools. The command lines issued by Make to the assemblers are calculated from the contents of the Makefile controlling the job in progress. The command lines understood by ObjAsm and AAsm are described in the chapters entitled *ObjAsm* and *AAsm* later in this volume.

If you have a machine with two megabytes or more you do not need to understand the details of the command lines contained in your Makefiles; you can adjust them using the same desktop interface as that available from each tool's icon. To do this you follow the Make **Tool options** menu item and click on the name of the tool concerned.

Editor throwback

During development of a program you may well find that you spend a high proportion of your time repeatedly editing, assembling, and testing programs. This development cycle can be speeded up by using Throwback to the SrcEdit editor to assist in removing assembly errors from your sources.

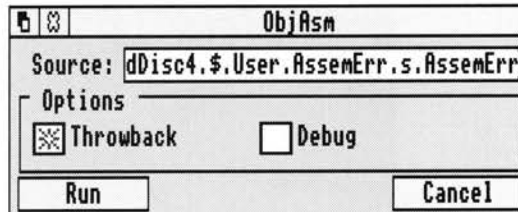
If SrcEdit and the DDEUtils module are loaded and you choose the assembler Throwback option, then perform an assembly of a file causing an assembly error, a browser window is presented by the editor. Double clicking Select on an error line in this browser window makes the editor open an edit window displaying the source file causing the error, with the offending line in view and highlighted, ready for correction. This facility can be used whether assembly is being performed managed by Make or by using the assembler icon bar interfaces.

Example throwback session

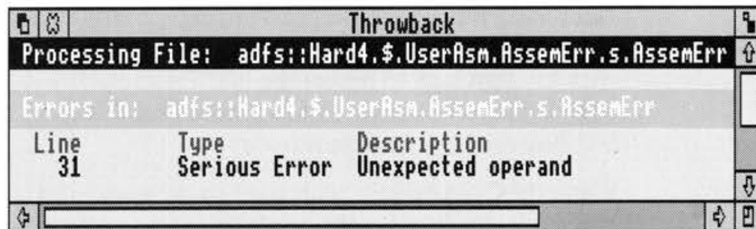
First double click on !SrcEdit, !ObjAsm and !Link in a directory display to load them as applications with icons on the icon bar. Next open a directory display on the subdirectory `User.AssemErr.s` to show the text file AssemErr containing the source of the program example of that name.

AssemErr is a simple assembly language program which when run prints `Hello World` on the screen. It is written to be assembled to an object file by ObjAsm then linked to form an executable image file with Link. Its source contains a simple error which will be detected by ObjAsm when you try to assemble it.

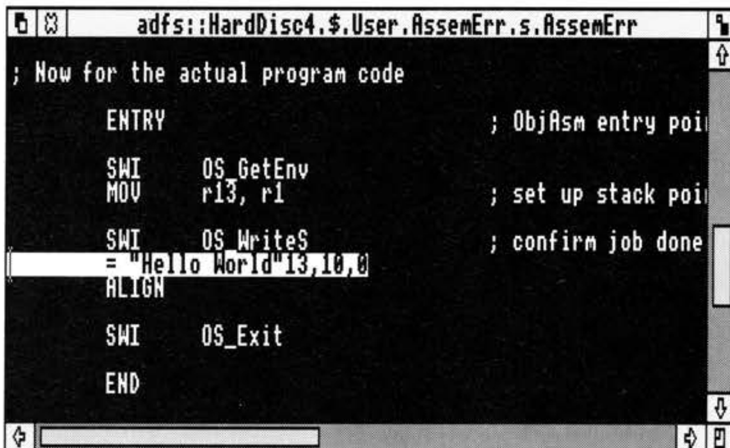
Drag the source file AssemErr to the ObjAsm icon. The ObjAsm SetUp dialogue box will appear with the Source filename initialised to the absolute file name. Ensure that the **Throwback** option is enabled: the correct dialogue box appearance is:



Click Menu on the dialogue box and ensure that the **Work directory** item on the menu displayed has the default setting of '^'. Next click on **Run** on the dialogue box to start assembly. This has the normal effect of removing the dialogue box and putting the ObjAsm output display on the screen, but almost immediately afterwards the assembler will produce an error and request SrcEdit to display a throwback error browser:

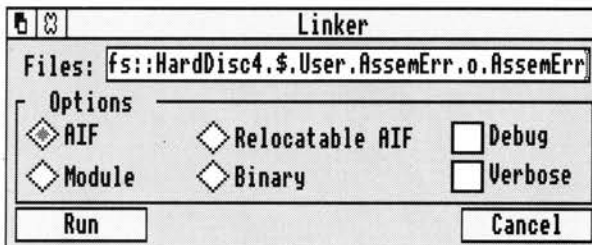


Double click Select on the assembler error message `Unexpected operand` in the browser. SrcEdit will display the source file with the line that caused the error clearly highlighted:



Examining this line closely shows that a comma is missing after the close quote. Insert this comma in SrcEdit and save the file. Click Select on the ObjAsm icon and click the **Run** icon to repeat the last assembly. If you have changed the AssemErr source correctly, the assembly should now complete with no errors and without bringing back the SrcEdit browser.

When the ObjAsm save dialogue box appears, click on the **OK** icon to save the object file produced in the o subdirectory next to the s subdirectory containing the source. Drag this object file to the Link icon on the icon bar. The Link SetUp dialogue box appears:



On this dialogue box and its associated menu ensure that the default output type of AIF is chosen, then click on the **Run** action icon. Save the resultant output file in a suitable directory such as the AssemErr subdirectory, then double click Select on its name. The image file should now run, printing the `Hello World` message in a RISC OS run window:

```
Run adfs::HardDisc4.$.User.AssemErr.!RunImage
Hello World

Press SPACE or click mouse to continue
```

DDT debugging

If you wish to debug your constructed program with the DDT debugger, you should use only the ObjAsm assembler, as AAsm does not provide sufficient symbol information in its output files to allow more than a few DDT features to work.

DDT can debug assembly language programs at machine level (ie displaying the current execution position on a disassembly of memory) or at source level (ie displaying the current execution position as a source file line). If you wish to debug at source level, the Debug option of ObjAsm must be enabled during assembly.

Insert the ObjAsm KEEP directive (without a following symbol name) in each source file to make this assembler output all symbol information. Your link operation to produce the executable image file to debug with DDT must have the Debug option selected. See the chapter entitled *Directives* for more details of KEEP.

Executing a binary produced in the above way, or dragging it from a directory display to the DDT icon, starts a DDT debugging session on it. See the chapter entitled *Desktop debugging tool* in the accompanying *Acorn Desktop Development Environment* user guide for more details of DDT.

Example DDT session

This session demonstrates machine level debugging of assembly language with DDT (see above for the meaning of machine level).

First double click on !DDT, !ObjAsm and !Link in a directory display to load them as applications with icons on the icon bar. Next open a directory display on the subdirectory `User.Buggy.s` to show the text file Buggy (containing the source of the program example of that name).

Buggy is a small assembly language program which when run is designed to print a list of four hexadecimal random numbers on the screen. It is first assembled with ObjAsm to an object file, then this object is linked with the PrintLib library in `User.PrintLib.o` to form an executable image file. A fault has been deliberately put in Buggy to illustrate the use of the DDT debugger. The directive KEEP is in the Buggy source file to retain symbol names so that they appear in DDT displays.

Drag the text source file of Buggy to the ObjAsm icon on the icon bar to bring up the ObjAsm SetUp dialogue box, then click **Run** to start assembly. Save the object file produced, then drag it to the Link icon. The Link SetUp box will appear. To link with the PrintLib library, drag this file from `User.PrintLib.o` to **Input files** . Ensure that the output format is AIF on the SetUp box, and click **Run**. Save the executable image file in a suitable place, then double click Select on it. The program should run, but incorrectly display only one random number:

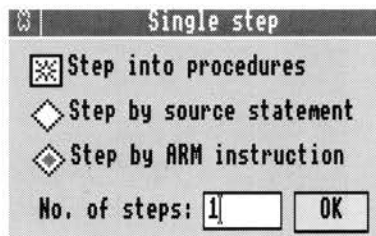
```
Run adfs::HardDisc4.$.User.Buggy.!RunImage

Four hexadecimal random numbers follow:
6F62A954

Press SPACE or click mouse to continue
```

Now we can determine what the problem is using DDT. A quick inspection of the source code in a SrcEdit window shows that the program is clearly intended to display four random numbers. You may be able to spot the fault in the program text, but here's a way of doing it with DDT.

Repeat the link step by clicking Select on the Link icon bar icon, this time selecting the Debug option on the SetUp box before clicking on **Run**. This produces a file with an associated icon which has a black bug displayed on it. Double clicking on this (or dragging it to the DDT icon) loads it into DDT. The two main DDT windows appear:

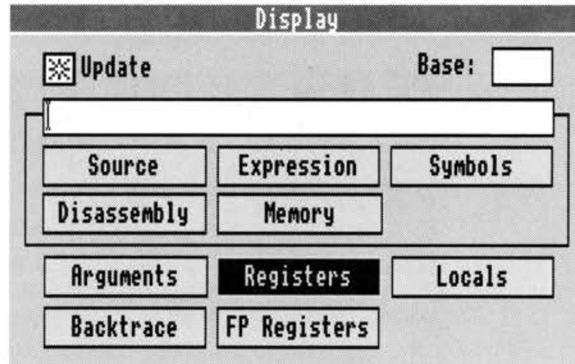
```
DDT: adfs::HardDisc4.$.User.Buggy.Buggy
00007ff4: 0000cc08 andeq   ip,r0,r8,lsl #&18
00007ff8: 00000002 andeq   r0,r0,r2
00007ffc: 0000003b dcb     ";", 0, 0, 0
00008000: fb000000 blnv    &00008008
00008004: fb000000 blnv    &0000800c
00008008: eb00000c bl      &00008040
0000800c: eb00001b bl      C$$code              ; &00008080
00008010: ef000011 swi     OS_Exit
00008014: 00000148 andeq   r0,r0,r8,asr #&02

Status: Initialisation

No source level debugging information
Can't set breakpoint on procedure main
RO area limit not on page boundary, last page not protected
```

Clicking Menu on either DDT window displays the DDT menu, from which you gain access to its many features. For a detailed description of these see the chapter entitled *Desktop debugging tool* in the accompanying *Acorn Desktop Development Environment* user guide. Click Select on the **Single step** item to bring up the single step dialogue box:



Set this as above to step into procedures by ARM instruction. Click once or twice on the **OK** icon and watch the program execution step forward. For the first few instructions you will not recognise the code executed, it is initialisation code

added by Link. Carry on until you reach `|C$$code|` – the Buggy code you assembled. Now return to the DDT menu, Select **Display**, and start an updated display of the ARM registers:



Continue single stepping, watching the registers change in the display window. An example pair of main windows is:

```
DDT: adfs::HardDisc4.$.User.Buggy.Buggy
 000080b4: 6c6c6f66 dcb     "foll"
 000080b8: 0d3a776f dcb     "ow:\r"
 000080bc: 0000000a dcb     "\n", 0, 0, 0
 000080c0: e3a01003 mov     r1,#&03
ex_loop
>000080c4: eb000009 bl      rand_word              ; &000080f0
 000080c8: eb000013 bl      print_hex              ; &0000811c
 000080cc: ef000001 swi     OS_WriteS
 000080d0: 00000a0d dcb     "\r\n", 0, 0

Status: Stopped after Single Stepping
Stepped to PC = 000080c4 (ex_loop + 0)
R0 = 04448241  R1 = 00000003  R2 = 04448205  R3 = 00008148
R4 = 00000000  R5 = 00000000  R6 = 00000000  R7 = 00000000
R8 = 00000000  R9 = 00000000  sl = 00000000  fp = 00008014
ip = 00000000  sp = 000a8000  lr = 00008090  pc = 000080c4
Flags: N = 0, Z = 0, C = 0, V = 0
```

When you reach the main top level printing loop, you should see what the problem is.

The procedures `rand_word` and `print_hex` obey the ARM Procedure Call Standard as applied to leaf procedures (those that call no others). This standard permits called procedures to alter r0 to r3 (any return integer being in r0). The main

top level printing loop incorrectly uses r1 as an index, which is then altered by `print_hex` to a negative number, terminating the loop. Change the register used for this index to r4, and all will be well:

```
Run adfs::HardDisc4.$.User.Buggy.!RunImage
Four hexadecimal random numbers follow:
51611466
B9F6E1AC
32E1B3F8
024F59D8

Press SPACE or click mouse to continue
```

Using FrontEnd on your programs

FrontEnd is a relocatable module supplied as part of the Acorn Desktop Assembler product which provides RISC OS desktop interfaces for non-interactive command line programs. The DDE non-interactive tools (such as AAsm and ObjAsm) are each command line programs supported in RISC OS by FrontEnd. This converts each tool into a fully multitasking windowed RISC OS application. For more details of non-interactive DDE tools see the chapters entitled *Working in the* DDE and *General features* in the accompanying *Acorn Desktop Development Environment* user guide.

You can use the power of FrontEnd to produce your own RISC OS applications. To do this you need to construct

- a suitable command line program;
- a Templates file (constructed with FormEd);
- a Sprites file (constructed with Paint);
- a !Run file;
- a !Help text file containing a short description of your program;
- a Messages text file;
- a Desc front end description file.

To be suitable, your command line program has to be non-interactive; ie started with a command line, then running to error or completion without any further user interaction, and outputting reports as screen text. An assembler such as ObjAsm fits this description, and an editor such as SrcEdit does not.

The Desc front end description file contains a specification of the appearance and function of the desktop interface to be provided for your program by FrontEnd. It is written in a special description language understood by FrontEnd. For more details of how to produce this file see the chapter entitled *Extending the* DDE in the accompanying *Acorn Desktop Development Environment* user guide. You may find it easier to make this file by altering a description belonging to one of the non-interactive tools rather than writing your own from scratch.

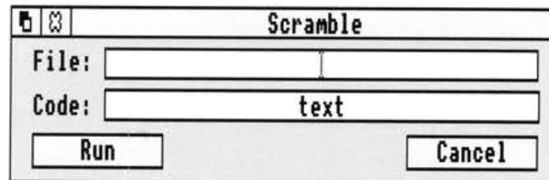
The Scramble example in your User directory is an example of a simple command line program written in assembler and provided with a desktop interface by FrontEnd.

`User.!Scramble.scramble` is the command line tool, with the corresponding assembly language source in `User.!Scramble.s`. It is a very basic command line program, knowing nothing of RISC OS windows or multitasking. It scrambles the contents of a text file to an unreadable jumble for security purposes. Repeating a scramble of a file with an identical code string unscrambles the file. The command line syntax of scramble is:

`*scramble` *`filename`* `-code` *`code_text`*

The Sprites file of Scramble has been adapted (by simply renaming the appropriate sprite to !Scramble) from that of one of the non-interactive DDE tools, so the Scramble icon bar icon has the familiar spanner and screwdriver appearance, but there is no need for your programs to have icons like this; just produce your own Sprites file with Paint.

After assembling `s.scramble`, double click on !Scramble in the `User` directory display to run it. Double click on !Scramble with the Shift key down to inspect the files which produce this effect. The Scramble SetUp dialogue box appears as:



Making your own linkable libraries

Linkable libraries, which are usually filed in o subdirectories like object files, are collections of many object files stored in one file. When presented to Link as an input file, the referenced object files within a library are linked into the output file, but those not needed are left out. A linkable library is therefore a recommended

way of storing a selection of useful procedures for re-use in a number of programs. You may find that this facility can save you a lot of time by avoiding continually 'reinventing the wheel'.

The tool used to construct and modify linkable libraries is LibFile. The tool DecCF can also be used to decode some information about an existing library.

The programming example PrintLib, which you can find in User.PrintLib, consists of three potentially useful procedures written in assembler which are intended to be assembled to object files using ObjAsm and then formed into a library with LibFile. They illustrate various programming points as well as how to construct a library.

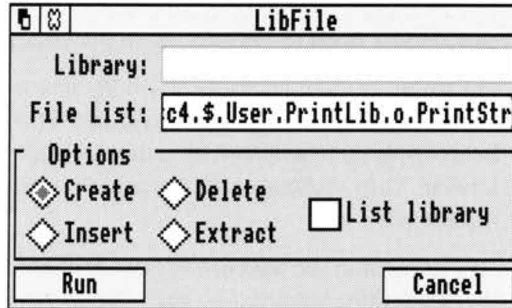
If you examine the assembler source files in `User.PrintLib.s` you will see that the procedure exported by each file obeys the ARM Procedure Call Standard. This ensures that they, and hence the PrintLib library, can be linked with other languages such as C. It is essential that procedures placed in a library have consistent register conventions, so that they can be re-used later without consulting their source text.

The PrintLib example is provided with both its assembly language source and the finished library. The facilities provided by this library are used in other programming examples. The procedures it exports are:

`print_string`	Print a null terminated string pointed to by r0.
`print_hex`	Print in hexadecimal an integer contained in r0.
`print_double`	Print in scientific format a double precision floating point number contained in r0,r1.

To reconstruct PrintLib from its sources, first double click on !ObjAsm and !LibFile in a directory display to load them as applications with icons on the icon bar. Then assemble `s.PrintStr`, `s.PrintHex` and `s.PrintDble` to corresponding object files by dragging each source file to the ObjAsm icon and saving the output

object files in the default places, ie `o.PrintStr`, `o.PrintHex` and `o.PrintDble`. Next drag `o.PrintStr` to the Libfile icon to make the LibFile SetUp dialogue box appear:



Ensure that the **Create** option is chosen as above. Drag the other two object files to **File List**, then click on **Run**. Finally save the library file produced: it is now ready to use.

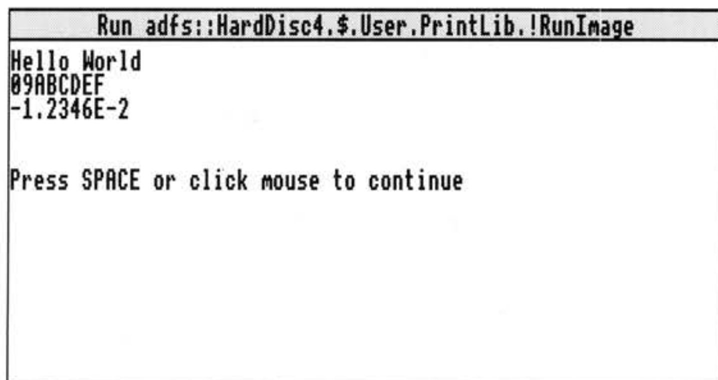
The assembly language source file `User.PrintLib.s.Test` is an example program making use of the procedures exported by PrintLib. To use it:

1 Double click on the !Link application to load it.

2 Assemble `s.Test` to `o.Test` with ObjAsm.

3 Link `o.Test` with the finished PrintLib library to produce an executable AIF image file.

Running the test program by double clicking on it should result in text output into a RISC OS output window:



3 AAsm

AAsm is one of the two ARM assemblers forming part of the Acorn Desktop Assembler product. It processes a text file containing program source written in ARM assembly language into an executable image file or relocatable module. AAsm multitasks under the RISC OS desktop, allowing other tasks to proceed while it operates.

An example use of AAsm is to construct a binary image file !RunImage in a RISC OS desktop application from a source file `s.myprog`. AAsm processes the source file directly to form !RunImage without the use of Link.

AAsm should not be used to assemble programs to be debugged using the DDT debugger - use ObjAsm instead. AAsm provides the most direct way of processing assembly language source into a runnable image file, and can be convenient for assembling small programs working unmanaged by the Make tool. The other assembler, ObjAsm, is more suitable for processing complex programs, as AAsm forces you to assemble a program in one chunk, using directives GET and LNK to join source files together at assembly time. This means that everything has to be reassembled if you make a change to one source file. ObjAsm has to be used if you wish to construct a program from a mixture of assembly language and a high level language such as C.

The controls of AAsm are similar to those of other non-interactive DDE tools (for a description of the common features of these tools see the chapter entitled *General features* in the accompanying *Acorn Desktop Development Environment* user guide). You adjust options for the next assembly operation on a setup dialogue box and menu which by default appear when you click Select on the main tool icon or drag a source file to it. Once you have set the required options you click on **Run** and the assembly starts. Output and text messages from the assembler can be displayed in one of two windows and menu options allow you to pause or stop the job at any time.

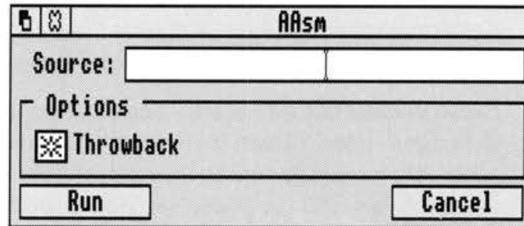
There is no file type to double click on to start AAsm - AAsm owns no filetype unlike, for example, Draw.

Starting AAsm

Like other non-interactive DDE tools, AAsm can be used under the management of Make, with its assembly options specified by the makefile passed to Make. For such managed use, AAsm is started automatically by Make, you don't have to load AAsm onto the icon bar.

To use AAsm directly, unmanaged by Make, first open a directory display on the DDE directory, then double click Select on !AAsm. The AAsm main icon appears on the icon bar.

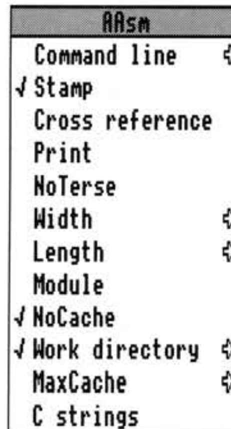
Clicking Select on this icon or dragging an assembly language source file from a directory display to this icon brings up the AAsm SetUp dialogue box:



Source will appear containing the name of the last filename entered there, or empty if there isn't one.

Dragging a file on to the icon will bring up the dialogue box and automatically insert the dragged filename as the **Source** file.

Clicking Menu on the SetUp dialogue box brings up the AAsm SetUp menu:



The SetUp dialogue box and menu specify the next assembly job to be done. You start the next job by clicking **Run** on the dialogue box (or Command line menu dialogue box). Clicking **Cancel** removes the SetUp dialogue box and clears any changes you have just made to the options settings back to the state before you brought up the SetUp box. The options last until you adjust them again or !AAsm is reloaded. You can also save them for future use with an option from the main icon menu.

AAsm SetUp options

When the SetUp dialogue box is displayed the **Source** writable icon contains the name of the sourcefile to be assembled. The sourcefile can be specified in two ways:

- If the SetUp box is obtained by clicking on the main AAsm icon, it comes up with the sourcefile from the previous setting. This helps you repeat a previous assembly, as clicking on the **Run** action button repeats the last job if there was one.
- If the SetUp box appears as a result of dragging a source file containing assembly language text to the main icon, the source file will be the same as the dragged source file.

When the SetUp box appears the Source icon has input focus, and can be edited in the normal RISC OS fashion. If a further source file is selected in a directory display and dragged to **Source**, its name replaces the one already there.

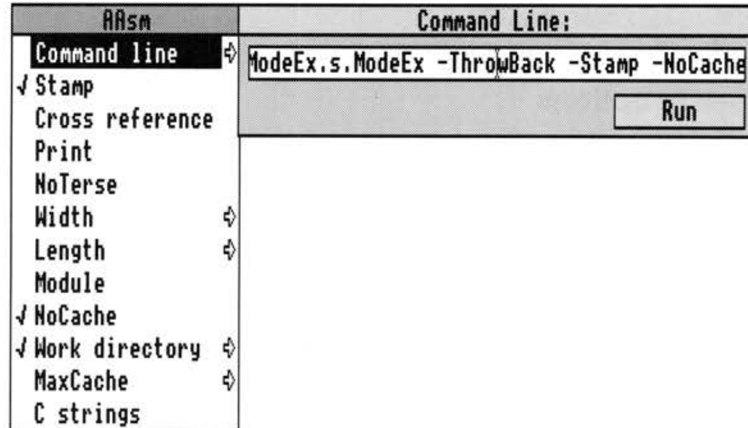
Setup dialogue box options

The **Throwback** option switches editor throwback on or off. When enabled, if the DDEUtils module and SrcEdit are loaded, any assembly errors cause the editor to display an error browser. Double clicking Select on an error line in this browser makes the editor display the source file containing the error, with the offending line highlighted. See the chapter entitled *SrcEdit* in the accompanying *Acorn Desktop Development Environment* user guide for more details.

Throwback is on by default.

Setup menu options

The AAsm RISC OS desktop interface works by driving an AAsm tool underneath with a command line constructed from your SetUp options. The **Command line** item at the top of the SetUp menu leads to a small dialogue box in which the command line equivalent of the current SetUp options is displayed:



Clicking on **Run** in this dialogue box starts assembly in the same way as clicking on **Run** in the main SetUp box. Pressing Return in the writable icon in this box has the same effect. Before starting assembly from the command line box, you can edit the command line textually, although this is not normally useful.

Stamp causes any output image files to have up to date timestamps.

Stamp is on by default.

When **Cross reference** is enabled (with a tick next to it) an alphabetically sorted cross reference of all symbols encountered is output after assembly. Note that the text output may be very large for a big program and so this option may not function on a machine with restricted memory.

Cross reference is off by default.

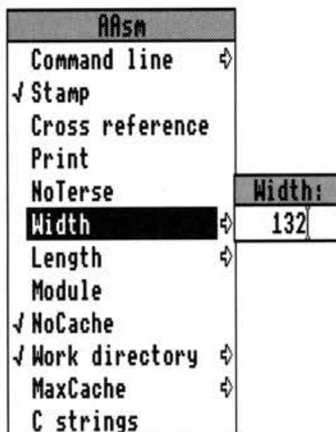
The **Print** option enables assembler source code to be viewed directly from the SrcEdit editor or from within the Assembler. This option turns on the Assembler screen listing, and during assembly the source code, object code, memory addresses and reference line numbers will be printed on the screen.

Print is off by default.

NoTerse modifies the effect of the **Print** option. If **NoTerse** is not enabled, **Print** only outputs the conditionally assembled parts of your program, but with **NoTerse** enabled (accompanied by a tick), conditionally non-assembled parts are listed as well.

NoTerse is off by default.

Width allows you to specify assembler output width:



This should be specified as an integer between 1 and 254. A width of 76 is suitable for a Mode 12 RISC OS window.

The default width is 131.

Length allows you to specify the number of lines per page for printer output. At the end of each page the assembler inserts a form feed character.

The default length is 60.

Enabling the **Module** option is the way to produce a relocatable module as an AAsm output file rather than an executable imagefile. This option is present with AAsm but not ObjAsm, since ObjAsm always produces linkable object files as its output, and producing a relocatable module from object files is enabled with the options of the Link tool.

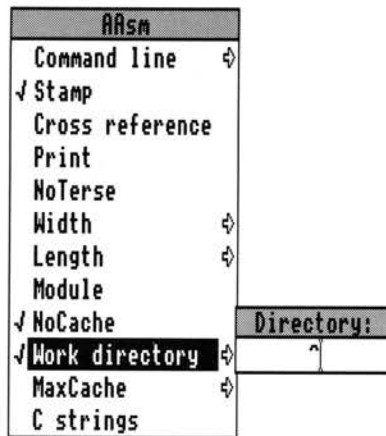
Module is off by default.

When **NoCache** is switched off cacheing is enabled. AAsm is a two pass assembler - it examines each source file twice. To avoid reading each source file twice from disk the assembler can cache the source in memory, reading it from disk for the first pass, then storing it in RAM for the second.

Cacheing is a very heavy user of memory, making it unsuitable for smaller machines.

NoCache is by default on - cacheing off.

Work directory allows you to specify the work directory:



The GET and LNK directives both result in the assembler loading source files specified with the directive. The work directory is the place where these source files are to be found. An example is a source file `adfs::4.$.user.s.foo` containing the line:

```
GET    s.macros
```

If the work directory is ^ then the file loaded is:

```
        adfs::4.$.user.s.^.s.macros
(ie     adfs::4.$.user.s.macros)
```

The work directory must be given relative to the position of the source file containing the GET or LNK, without a trailing dot.

The default work directory is ^.

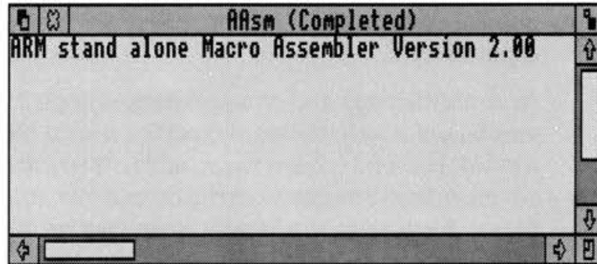
MaxCache allows you to specify the maximum amount of RAM to be used for cacheing source files (when **NoCache** is off). The maximum cache is specified in megabytes.

The default MaxCache is 8Mb - effectively unlimited.

C strings, when enabled, allows the assembler to accept C style string escapes such as '`\n`'. **C strings** is not enabled by default, as it results in '\' characters in string constants being interpreted in a different way compared to previous Acorn assemblers.

AAsm output

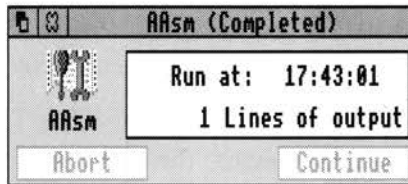
AAsm outputs text messages as it proceeds. These include source listings and symbol cross references (described in the previous section). By default any such text is directed into a scrollable output window:



This window is read-only: you can scroll up and down to view progress, but you cannot edit the text without first saving it. To indicate this, clicking Select on the scrollable part of this window has no effect.

The contents of the window illustrated above are typical of those you see from a successful assembly; the title line of the assembler with version number, followed by no error messages.

Clicking Adjust on the close icon of the output window switches to the output summary dialogue box. This presents a reminder of the tool running (AAsm), the status of the task (Running, Paused, Completed or Aborted), the time when the task was started and the number of lines of output that have been generated (ie those that are displayed by the output window).



Clicking Adjust on the close icon of the summary box returns to the output window.

Both the above AAsm output displays follow the standard pattern of those of all the non-interactive DDE tools. The common features of the non-interactive DDE tools are covered in more detail in the chapter entitled *General features* in the accompanying *Acorn Desktop Development Environment* user guide. Both AAsm output

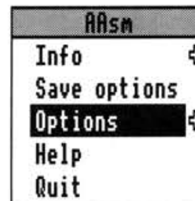
displays and the menus brought up by clicking Menu on them offer the standard features allowing you to abort, pause or continue execution (if the execution hasn't completed), and to save output text to a file or repeat execution.

AAsm error messages appear in the output viewer, with copies in the editor error browser when throwback is working. *Appendix A - Error messages* at the end of this manual contains a list of all the AAsm error messages together with brief explanations.

Assembly listings and cross references appearing in the output window are often very large for assemblies of complex source files. The scrolling of the output window is useful to view them, and to investigate them with the full facilities of the source editor. You can save the output text straight into the editor by dragging the output file icon to the SrcEdit main icon on the icon bar.

AAsm icon bar menu

The AAsm main icon bar menu follows the standard pattern for non-interactive DDE tools:



Save options saves all the current AAsm options, including both those set from the SetUp dialogue box and from the Options item on this menu. When AAsm is restarted it is initialised with these options rather than the defaults.

The **Options** submenu allows you to set the following options:

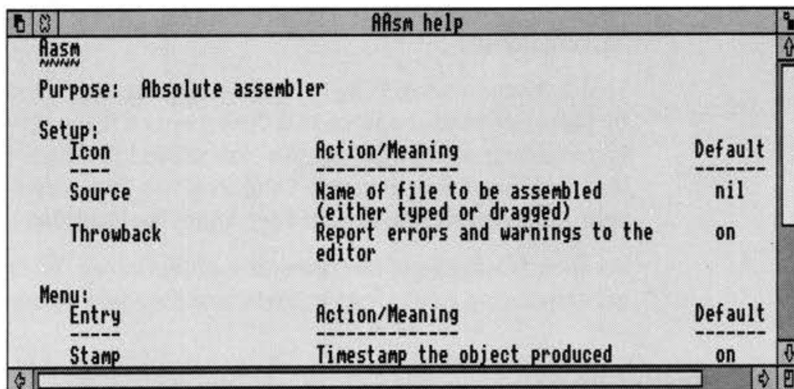
Display specifies the output display as either a text window (default) or as a summary box.

If **Auto run** is enabled, dragging a source file to the AAsm main icon immediately starts an assembly with the current options rather than displaying the SetUp box first.

If **Auto save** is enabled output image files are saved to suitable places automatically without producing a save dialogue box for you to drag the file from.

Both **Auto run** and **Auto save** are off by default.

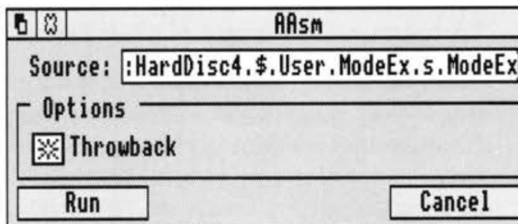
Clicking on **Help** on the main AAsm menu displays a short text summary of the various SetUp options, in a scrollable read-only window:



Example AAsm session

The programming example ModeEx is a relocatable module written in assembly language processed from text source to usable module with AAsm. The ModeEx relocatable module, when rmloaded, provides an extra screen mode, mode 29. Unless you alter the parameters in the source file, mode 29 is similar to mode 12 in having 16 colours, but has no borders, allowing 96 by 36 characters on the screen instead of 80 by 32. You may well find this mode a useful one in which to work with the DDE, as it offers a useful compromise between character size and information visible.

To construct this module, first double click on !AAsm to load it as an application with an icon on the icon bar. The source of ModeEx is in the subdirectory `User.ModeEx.s`. Drag the text source file to the AAsm icon to make the AAsm SetUp box appear:



Set the Module option on the SetUp menu to on, then click on **Run**. The module will then assemble, so you can save it and use it.

AAsm managed by Make

When Acorn Desktop Assembler is installed on your system, Make understands ObjAsm, not AAsm, as the tool to process assembly language source from `s` subdirectories.

To use AAsm managed by Make, ie driven by a recipe stored in a Makefile created by Make, store your source in a directory not having the recognised name `s`. When first creating your project in the New project dialogue box, specify AAsm as the tool creating the final target. Drag your top level source file to the **Insert** field of your Project dialogue box and set AAsm **tool options** in the normal way.

For general details of the operation of Make, see the chapter entitled *Make* in the accompanying *Acorn Desktop Development Environment* user guide.

AAsm command lines

AAsm, in common with the other non-interactive DDE tools, can be driven with a text command line without its RISC OS desktop interface appearing. This enables AAsm to be driven by Make as specified in textual makefiles.

You can use AAsm outside the RISC OS desktop from its command line, in the same way that it could be used in the previous Acornsoft Archimedes Assembler product. However, as all the useful AAsm features can now be more conveniently used from the RISC OS desktop there is little reason for you to do this. The desktop removes the need for you to understand the command line syntax.

The AAsm RISC OS desktop interface drives the AAsm tool underneath by issuing a command line constructed from your SetUp options. The **Command line** SetUp menu option allows you to view the command line constructed in this way.

If you have a machine with more than 1Mb of RAM, the Make tool allows you to construct makefiles with assembly operations specified using the AAsm desktop interface (by following the **Tool options** item of Make). You can therefore construct makefiles without understanding the command line syntax of AAsm.

The command line syntax of AAsm is documented here as a reference.

A command line just consisting of the tool name AAsm causes the assembler to drop into an obsolescent interactive mode, within which you issue commands with the assembler resident in memory. It is not recommended to use this mode, but its syntax is revealed by typing `help` once it is entered. Interactive mode is left by entering `Quit` (or `Q` for short). This mode of use is not covered further here.

The AAsm command line consists of the AAsm tool name followed by a series of keywords, some of which are followed by associated arguments. Each keyword starts with a minus sign (-) and is case independent, but is listed below with its minimum abbreviation in capital letters:

`-FRom filename`	Specifies the source file (the **Source** item of the SetUp box).
`-TO filename`	Specifies the output image or module file name.
`-Stamp`	Time stamps the output image or module file (the SetUp box **Stamp Output** option).
`-THrowback`	Enables source editor throwback when available (the SetUp box **Throwback** option).
`-Xref`	When combined with `-Quit` outputs a sorted cross reference (the SetUp menu **Cross reference** option).
`-Print`	Enables source file listing (the SetUp menu **Print** option).
`-NOTerse`	Enlarges source listing (the SetUp menu **NoTerse** option).
`-Width number`	Sets output width to an integer number of characters.
`-Length number`	Sets output page length to an integer number of lines.
`-Module`	Sets output file type to relocatable module (the SetUp menu **Module** option).
`-NoCache`	Do not cache source files (the SetUp menu **NoCache** option).
`-Desktop dirname`	Specifies the work directory in which to find GET or LNK files (the SetUp menu **Work directory** option).
`-Maxcache number`	Specifies the maximum cache size as an integer number of megabytes.
`-Quit`	Avoids entering interactive mode after assembly - recommended.
`-Closeexec`	Closes any open exec files if assembly fails.
`-Esc`	Enable C style string escapes (the SetUp menu **C strings** option).

`-FRom` and `-TO` have no effect unless both are specified. If both are specified, an assembly is performed immediately using the specified files. The parameters belonging to `-FRom` and `-TO` may be specified in this order without using the keywords.

4 ObjAsm

ObjAsm is one of the two ARM assemblers forming part of the Acorn Desktop Assembler product. It processes text files containing program source written in ARM assembly language into linkable object files. Object files can be linked by the Link tool with each other or with libraries of object files to form executable image files or relocatable modules. ObjAsm multitasks under the RISC OS desktop, allowing other tasks to proceed while it operates.

ObjAsm must be used to assemble programs to be debugged using the DDT debugger. It is more suitable than AAsm for the construction of large programs, as for ObjAsm the sources can be split into several files and only re-assembled to object files when you have altered them.

An example use of ObjAsm is to construct a binary image file !RunImage in a RISC OS desktop application from the two source files `s.interface` and `s.portable`. ObjAsm processes the source files to form `o.interface` and `o.portable`, which the Link tool processes to form !RunImage.

The controls of ObjAsm are similar to those of other non-interactive DDE tools, with the common features described in the chapter entitled *General features* in the accompanying *Acorn Desktop Development Environment* user guide. You adjust options for the next assembly operation on a setup dialogue box and menu which by default appear when you click Select on the main icon or drag a source file to it. Once you have set options you click on a Run action icon and the assembly starts. While the assembly is running output windows display any text messages from the assembler and allow you to stop the job if you wish.

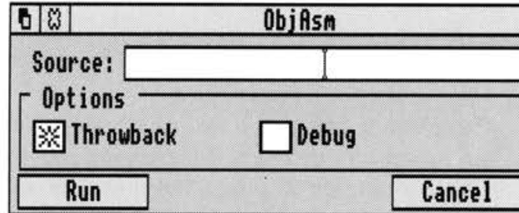
There is no file type to double click on to start ObjAsm - ObjAsm owns no file type unlike, for example, Draw.

Starting ObjAsm

Like other non-interactive DDE tools, ObjAsm can be used under the management of Make, with its assembly options specified by the makefile passed to Make. For such managed use, ObjAsm is started automatically by Make, you don't have to load ObjAsm onto the icon bar.

To use ObjAsm directly, unmanaged by Make, first open a directory display on the DDE directory, then double click Select on !ObjAsm. The ObjAsm main icon appears on the icon bar.

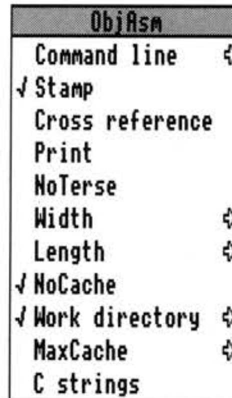
Clicking Select on this icon or dragging an assembly language source file from a directory display to this icon brings up the ObjAsm SetUp dialogue box:



Source will appear containing the name of the last filename entered there, or empty if there isn't one.

Dragging a file on to the icon will bring up the dialogue box and automatically insert the dragged filename as the **Source** file.

Clicking Menu on the SetUp dialogue box brings up the ObjAsm SetUp menu:



The SetUp dialogue box and menu specify the next assembly job to be done. You start the next job by clicking **Run** on the dialogue box (or Command line menu dialogue box). Clicking **Cancel** removes the SetUp dialogue box and clears any changes you have just made to the options settings back to the state before you brought up the SetUp box. The options last until you adjust them again or !ObjAsm is reloaded. You can also save them for future use with an option from the main icon menu.

ObjAsm SetUp options

When the SetUp dialogue box is displayed the **Source** writable icon contains the name of the source file to be assembled. The sourcefile can be specified in two ways:

- If the SetUp box is obtained by clicking on the main ObjAsm icon, it comes up with the sourcefile from the previous setting. This helps you repeat a previous assembly, as clicking on the **Run** action icon repeats the last job if there was one.
- If the SetUp box appears as a result of dragging a source file containing assembly language text to the main icon, the source file will be the same as the dragged source file.

When the SetUp box appears the Source icon has input focus, and can be edited in the normal RISC OS fashion. If a further source file is selected in a directory display and dragged to **Source**, its name replaces the one already there.

Setup dialogue box options

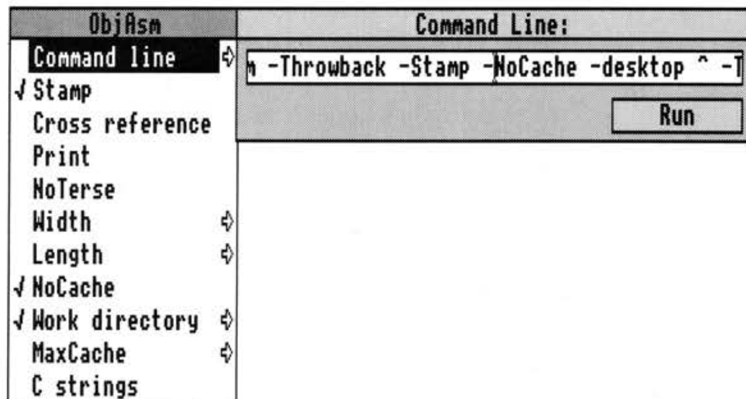
The **Throwback** option switches editor throwback on or off. When enabled, if the DDEUtils module and SrcEdit are loaded, any assembly errors cause the editor to display an error browser. Double clicking Select on an error line in this browser makes the editor display the source file containing the error, with the offending line highlighted. See the chapter entitled *SrcEdit* in the accompanying *Acorn Desktop Development Environment* user guide for more details.

Throwback is on by default.

The **Debug** option switches on or off the production of debugging tables. When enabled, extra information is included in the output object file which enables source level debugging of the linked image (as long as the Link Debug option is also enabled) by the DDT debugger. If this option is disabled, any image file finally produced can only be debugged at machine level. Source level debugging allows the current execution position to be indicated as a displayed line of your source, whereas machine level debugging only shows the position on a disassembly of memory.

Setup menu options

The ObjAsm RISC OS desktop interface works by driving an ObjAsm tool underneath with a command line constructed from your SetUp options. The **Command line** item at the top of the SetUp menu leads to a small dialogue box in which the command line equivalent of the current SetUp options is displayed:



The **Run** action icon in this dialogue box starts assembly in the same way as that in the main SetUp box. Pressing Return in the writable icon in this box has the same effect. Before starting assembly from the command line box, you can edit the command line textually, although this is not normally useful.

Stamp causes any output object files to have up to date timestamps.

Stamp output is on by default.

When **Cross reference** is enabled (with a tick next to it) an alphabetically sorted cross reference of all symbols encountered is output after assembly. Note that the text output may be very large for a big program and so this option may not function on a machine with restricted memory.

Cross reference is off by default.

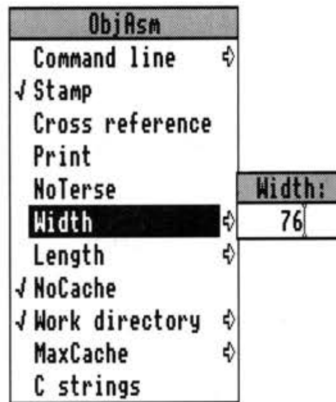
The **Print option** enables assembler source code to be viewed directly from the SrcEdit editor or from within the Assembler. This option turns on the Assembler screen listing, and during assembly the source code, object code, memory addresses and reference line numbers will be printed on the screen.

Print is off by default.

NoTerse modifies the effect of the **Print** option. Without **NoTerse** enabled, **Print** only outputs the conditionally assembled parts of your program, but with **NoTerse** enabled (accompanied by a tick), conditionally non-assembled parts are listed as well.

NoTerse is off by default.

Width allows you to specify assembler output width:



This should be specified as an integer between 1 and 254. A width of 76 is suitable for a Mode 12 RISC OS window.

The default width is 131.

Length allows you to specify the number of lines per page for printer output. At the end of each page the assembler inserts a form feed character.

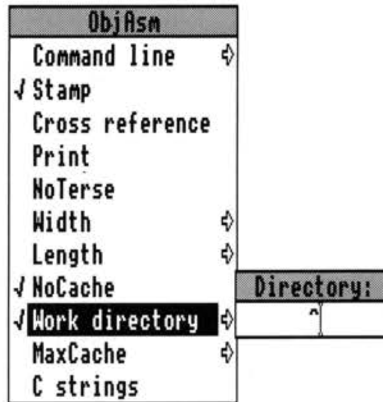
The default length is 60.

When **NoCache** is switched off cacheing is enabled. ObjAsm is a two pass assembler - it examines each source file twice. To avoid reading each source file twice from disk the assembler can cache the source in memory, reading it from disk for the first pass, then storing it in RAM for the second.

Cacheing is a very heavy user of memory, making it unsuitable for smaller machines.

NoCache is by default on - cacheing off.

Work directory allows you to specify the work directory:



The GET and LNK directives both result in the assembler loading source files specified with the directive. The work directory is the place where these source files are to be found. An example is a source file `adfs::4.$.user.s.foo` containing the line:

```
GET    s.macros
```

If the work directory is ^ then the file loaded is:

```
        adfs::4.$.user.s.^.s.macros
(ie     adfs::4.$.user.s.macros)
```

The work directory must be given relative to the position of the source file containing the GET or LNK, without a trailing dot.

The default work directory is ^.

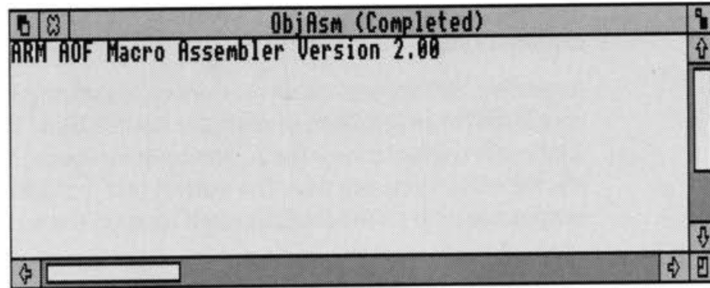
MaxCache allows you to specify the maximum amount of RAM to be used for cacheing source files (when **NoCache** is off). The maximum cache is specified in megabytes.

The default maximum cache is 8Mb - effectively unlimited.

C strings, when enabled, allows the assembler to accept C style string escapes such as '`\n`'. **C strings** is not enabled by default, as it results in '`\`' characters in string constants being interpreted in a different way compared to previous Acorn assemblers.

ObjAsm output

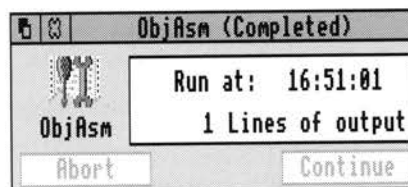
ObjAsm outputs text messages as it proceeds. These include source listings and symbol cross references (as described in the previous sections). By default any such text is directed into a scrollable output window:



This window is read-only: you can scroll up and down to view progress, but you cannot edit the text without first saving it. To indicate this clicking Select on the scrollable part of this window has no effect.

The contents of the window illustrated above are typical of those you see from a successful assembly; the title line of the assembler with version number, followed by no error messages.

Clicking Adjust on the close icon of the output window switches to the output summary dialogue box. This presents a reminder of the tool running (ObjAsm), the status of the task (Running, Paused, Completed or Aborted), the time when the task was started and the number of lines of output that have been generated (ie those that are displayed by the output window).



Clicking Adjust on the close icon of the summary box returns to the output window.

Both the above ObjAsm output displays follow the standard pattern of those of all the non-interactive DDE tools. The common features of the non-interactive DDE tools are covered in more detail in the chapter entitled *General features* in the accompanying *Acorn Desktop Development Environment* user guide. Both ObjAsm

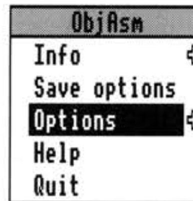
output displays and the menus brought up by clicking Menu on them offer the standard features allowing you to abort, pause or continue execution (if the execution hasn't completed) and to save output text to a file or repeat execution.

ObjAsm error messages appear in the output viewer, with copies in the editor error browser when throwback is working. *Appendix A - Error messages* at the end of this manual contains a list of all ObjAsm error messages together with brief explanations.

Assembly listings and cross references appearing in the output window are often very large for assemblies of complex source files. The scrolling of the output window is useful to view them, and to investigate them with the full facilities of the source editor, you can save the output text straight into the editor by dragging the output file icon to the SrcEdit main icon on the icon bar.

ObjAsm icon bar menu

The ObjAsm main icon bar menu follows the standard pattern for non-interactive DDE tools:



Save options saves all the current ObjAsm options, including both those set from the SetUp dialogue box and from the **Options** item on this menu. When ObjAsm is restarted it is initialised with these options rather than the defaults.

The **Options** submenu allows you to set the following options:

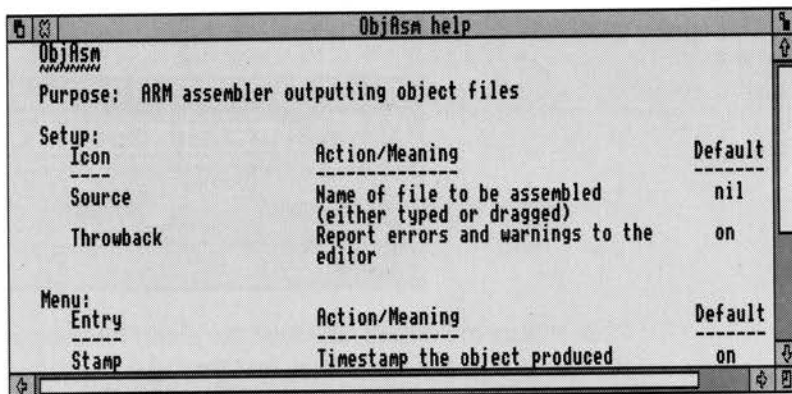
Display specifies the output display as either a text window (default) or as a summary box.

If **Auto run** is enabled, dragging a source file to the ObjAsm main icon immediately starts an assembly with the current options rather than displaying the SetUp box first.

If **Auto save** is enabled output image files are saved to suitable places automatically without producing a save dialogue box for you to drag the file from.

Both **Auto run** and **Auto save** are off by default.

Clicking on **Help** on the main ObjAsm menu displays a short text summary of the various SetUp options, in a scrollable read-only window:



Example ObjAsm session

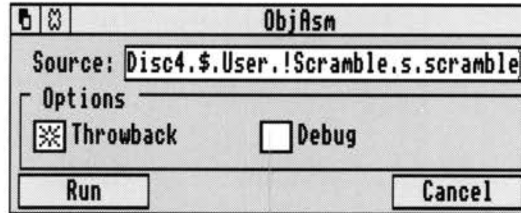
The programming example `User.!Scramble` is a non-desktop free standing command line program written in assembly language and given a RISC OS desktop interface (ie made into an application) by the FrontEnd module supplied as part of the DDE. Its purpose is to scramble the contents of text files for security. Repeating a scramble of a file with the same code text unscrambles it.

For more details of its support by the FrontEnd module, see the section entitled *Using FrontEnd on your programs* on page 15.

The assembly language source for scramble is in the subdirectory `User.!Scramble.s`. The code demonstrates the following points:

- ObjAsm directives needed for a free standing program;
- processing command lines from assembly language;
- random number generation;
- simple file handling;
- register usage by procedures.

To assemble scramble, first double click on !Objasm and !Link in a directory display to load them. Drag the scramble source text file to the ObjAsm icon. The SetUp dialogue box of ObjAsm appears. Check that the default SetUp options are enabled:



Click on **Run** to proceed, and save the object file produced in `!Scramble.o`. Drag the object file to the Link icon, and **Run** Link to produce an AIF executable image file, the link having the scramble object file as its only input file. The command line program is now ready for use.

Try copying the source file for scramble up into the subdirectory `User.!Scramble` as Test, and move out of the desktop to the command line by pressing F12. The command line syntax of scramble is:

`*scramble` *`filename`* `-code` *`code_text`*

As an experiment, try typing

```
*dir $.User.!Scramble
*scramble Test -code DDEIsFriendly
```

Now re-enter the desktop by leaving the command line (by pressing Return), and examine the text file Test with SrcEdit. It will be completely scrambled. Repeat the scramble from the command line with the same code text to unscramble the file.

ObjAsm command lines

ObjAsm, in common with the other non-interactive DDE tools, can be driven with a text command line without its RISC OS desktop interface appearing. This enables ObjAsm to be driven by Make as specified in textual makefiles.

You can use ObjAsm outside the RISC OS desktop from its command line, in the same way that it could be used in the previous Acornsoft Archimedes Assembler product. However, as all the useful ObjAsm features can now be more conveniently used from the RISC OS desktop there is little reason for you to do this. The desktop removes the need for you to understand the command line syntax.

The ObjAsm RISC OS desktop interface drives the ObjAsm tool underneath by issuing a command line constructed from your SetUp options. The Command line SetUp menu option allows you to view the command line constructed in this way.

If you have a machine with more than 1Mb of RAM, the Make tool allows you to construct makefiles with assembly operations specified using the ObjAsm desktop interface (by following the Tool options item of Make). You can therefore construct makefiles without understanding the command line syntax of Objasm.

The command line syntax of ObjAsm is documented here as a reference.

A command line just consisting of the tool name ObjAsm causes the assembler to drop into an obsolescent interactive mode, within which you issue commands with the assembler resident in memory. It is not recommended to use this mode, but its syntax is revealed by typing `help` once it is entered. Interactive mode is left by entering `Quit` (or `Q` for short). This mode of use is not covered further here.

The ObjAsm command line consists of the ObjAsm tool name followed by a series of keywords, some of which are followed by associated arguments. Each keyword starts with a minus sign (-) and is case independent, but is listed below with its minimum abbreviation in capital letters:

`-FRom filename`	Specifies the source file (the **Source** item of the SetUp box).
`-TO filename`	Specifies the output object file name.
`-Stamp`	Time stamps the output object file (the SetUp box **Stamp Output** option).
`-THrowback`	Enables source editor throwback when available (the SetUp box **Throwback** option).
`-Xref`	When combined with `-Quit` outputs a sorted cross reference (the SetUp menu **Cross reference** option).
`-Print`	Enables source file listing (the SetUp menu **Print** option).
`-NOTerse`	Enlarges source listing (the SetUp menu **NoTerse** option).
`-Width number`	Sets output width to an integer number of characters.
`-Length number`	Sets output page length to an integer number of lines.
`-NoCache`	Do not cache source files (the SetUp menu **NoCache** option).
`-Desktop dirname`	Specifies the work directory in which to find GET or LNK files (the SetUp menu **Work directory** option).

`-Maxcache number`	Specifies the maximum cache size as an integer number of megabytes.
`-Quit`	Avoids entering interactive mode after assembly - recommended.
`-Closeexec`	Closes any open exec files if assembly fails.

`-FRom` and `-TO` have no effect unless both are specified. If both are specified, an assembly is performed immediately using the specified files. The parameters belonging to `-FRom` and `-TO` may be specified in this order without using the keywords.

Part 2 - Assembly language details

5 The ARM cpu

The ARM (Advanced Risc Machine) is a 32-bit single chip microprocessor which has a reduced instruction set architecture. There are nine classes of instruction:

- branches
- data operations between registers
- single register data transfers
- multiple register data transfers
- supervisor calls
- multiplies
- coprocessor data operations
- coprocessor/memory transfers
- coprocessor/register transfers

The ARM has a 32-bit data bus and a 26-bit address bus. A 3-stage instruction pipeline allows an instruction to be executed while the next instruction is being decoded and the one after that is being fetched.

All instructions are designed to fit into one 32-bit word and all instructions can be made conditional. The processor can access two types of data:

- bytes (8 bits)
- words (32 bits)

The program counter (PC) is 24 bits wide and counts to &FFFFFF. However, two low-order bits (both zeros) are appended to the PC value and a 26-bit value is put on the address bus, thus quadrupling the total count to &3FFFFFC. The memory capacity of the ARM processor is 64 Mbytes, or 16 Mwords.

The PC is always a multiple of four because of the two appended zeros, and so it follows that instructions must be aligned to a multiple of four bytes. The instructions are given in one word and data operations are only performed on word quantities. Load and store operations can operate on either bytes or words and these instructions can put a full 26-bit address, with bits 0 and 1 set as required, on to the address bus.

Registers

The ARM normally operates in a mode of operation called User Mode, and in this environment the programmer sees a bank of sixteen 32-bit registers, R0 to R15. Eleven other registers exist and they are used when the ARM is in Interrupt Mode, Fast Interrupt Mode, or Supervisor Mode (see *The four modes of operation* below).

Of the sixteen registers R0-R15, only R14 and R15 are regarded as having specific purposes:

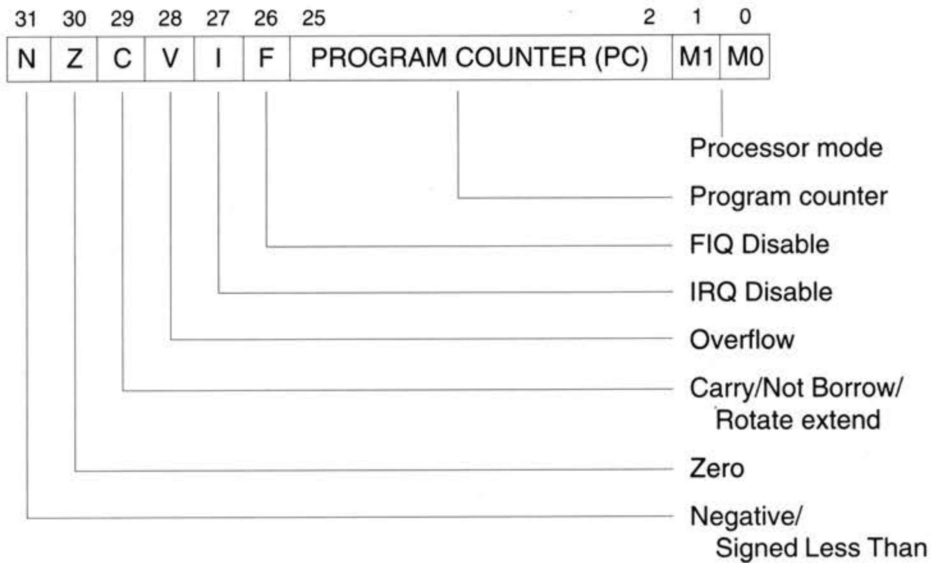
Register R15

R15 contains 24 bits of program counter (PC) and 8 bits of processor status register (PSR).

Special bits in some instructions allow the PC and PSR to be treated together, or separately, as required. *Allocation of bits within register* R15 on page 47 shows the allocation of the bits within the register R15.

User mode	SVC mode	IRQ mode	FIQ mode
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			
R8			R8_fiq
R9			R9_fiq
R10			R10_fiq
R11			R11_fiq
R12			R12_fiq
R13	R13_svc	R13_irq	R13_fiq
R14	R14_svc	R14_irq	R14_fiq
R15(PC/PSR)			

Figure 5.1 The four modes of operation



Processor mode		FIQ Disable		IRQ Disable	
00	User mode	0	Enable	0	Enable
01	FIQ mode	1	Disable	1	Disable
10	IRQ mode				
11	Supervisor mode				

Figure 5.2 *Allocation of bits within register* R15

Register R14

R14 is used as the subroutine Link register, and receives a copy of the return PC and PSR when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. Similarly, R14_svc, R14_irq and R14_fiq are used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within supervisor or interrupt routines.

In addition:

- FIQ processing state has seven private registers mapped to R8 to R14 (R8_fiq to R14_fiq).
- IRQ processing state has two private registers mapped to R13 and R14 (R13_irq and R14_irq).
- SVC processing state has two private registers mapped to R13 and R14 (R13_svc and R14_svc).

  The two private registers allow the IRQ and supervisor modes each to have a private stack pointer and link register. Supervisor and IRQ mode programs are expected to save the User state on their respective stacks and then use the User registers, remembering to restore the User state before returning.
- The PSR contains four condition flags:

N	Negative flag
Z	Zero flag
C	Carry flag
V	oVerflow flag

The condition flags may be altered in user mode. The I, F, and mode flags can only be changed directly in supervisor and interrupt modes; they are also modified when exceptions occur or SWI instructions are executed.

FIQ fast interrupt request

Note: The following sections on the ARM processor are mainly of interest to operating systems programmers, for example, when constructing relocatable modules. If you are writing applications, you can skip forward to the chapter entitled *Assembler language* on page 55.

The Fast Interrupt request (FIQ) exception is externally generated by pulling the FIQ pin LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect processor execution. It is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, so that the overhead of context switching is minimised.

The FIQ exception may be disabled by setting the F flag in the PSR (but note that this is not possible from user mode). If the F flag is clear ARM checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When ARM is FIQed it will:

1. save R15 in R14_fiq
2. force M0, M1 to FIQ mode and set the F and I bits in the PC word

3 force the PC to fetch the next instruction from address &1C.

To return normally from FIQ use:

SUBS PC,R14_fiq,#4

This will resume execution of the interrupted code sequence, and restore the original mode and interrupt enable state.

IRQ interrupt request

The Interrupt Request (IRQ) exception is externally generated by pulling the IRQ pin low. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect processor execution. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). If the I flag is clear ARM checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. When successfully IRQed ARM will:

1 save R15 in R14_irq

2 force M0,M1 to IRQ mode and set the I bit in the PC word

3 force the PC to fetch the next instruction from address &18.

To return normally from IRQ use:

SUBS PC,R14_irq,#4

Address exception trap

An address exception arises whenever a data transfer is attempted with a calculated address above &3FFFFFF. The ARM address bus is 26 bits wide, but an address calculation has a 32-bit result. If this result has a logic '1' in any of the top 6 bits it is assumed that the address overflow is an error, and the address exception trap is taken.

Note that a branch cannot cause an address exception, and a block data transfer instruction which starts in the legal area but increments into the illegal area will not trap (it wraps round to address 0 instead). The check is performed only on the address of the first word to be transferred.

When an address exception is seen ARM will:

1 if the data transfer was a store, force it to load. (This protects the memory from spurious writing.)

2 complete the instruction, but prevent internal state changes where possible. The state changes are the same as if the instruction had aborted on the data transfer.

3 save R15 in R14_svc

4 force M0,M1 to supervisor mode and set the I bit in the PC word

5 force the PC to fetch the next instruction from address &14.

Normally an address exception is caused by erroneous code, and it is inappropriate to resume execution. If a return is required from this trap, use `SUBS PC, R14_svc, #4`. This will return to the instruction after the one causing the trap.

Abort

The Abort signal comes from an external Memory Management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. ARM checks for an Abort at the end of the first phase of each bus cycle. When successfully Aborted ARM will respond in one of three ways.

Abort during an internal cycle

The ARM ignores aborts signalled during internal cycles.

Abort during instruction prefetch

If abort is signalled during an instruction prefetch (a Prefetch abort), the prefetched instruction is marked as invalid; when it comes to execution, it is reinterpreted as below. (If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, the abort will have no effect.)

Then ARM will:

1 save R15 in R14_svc

2 force M0, M1 to supervisor mode and set the I bit in the PC word

3 force the PC to fetch the next instruction from address &0C.

To continue after a Prefetch abort use `SUBS PC,R14_svc,#4`. The ARM will then re-execute the aborting instruction, so you should ensure that you have removed the cause of the original abort.

Abort during data access

If the abort command occurs during a data access (a Data Abort), the action depends on the instruction type.

- Single data transfer instructions (LDR, STR) are aborted as though the instruction had not executed.
- Block data transfer (LDM and STM) instructions complete, and if writeback is set, the base is updated. If the instruction would normally have overwritten the base with data (ie LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

Then ARM will:

1. save R15 in R14_svc
2. force M0, M1 to supervisor mode and set the I bit in the PC word
3. force the PC to fetch the next instruction from address &10.

To continue after a data abort, remove the cause of the abort, then reverse any auto-indexing that the original instruction may have done, then return to the original instruction with `SUBS PC,R14_svc,#8`.

Software interrupt

The software interrupt instruction is used for getting into supervisor mode, usually to request a particular supervisor function. ARM will:

1. save R15 in R14_svc
2. force M0, M1 to supervisor mode and set the I bit in the PC word
3. force the PC to fetch the next instruction from address &8.

To return from a SWI, use `MOVS PC,R14_svc`. This returns to the instruction following the SWI.

Undefined instruction trap

The undefined instruction trap may be used for software emulation of a coprocessor or for general purpose instruction set extension by software emulation (the floating point instruction set is implemented in software this way). If an undefined instruction or coprocessor instruction is encountered and is not claimed by any coprocessor, ARM will:

1. save R15 in R14_svc
2. force M0, M1 to supervisor mode and set the I bit in the PC word

3 force the PC to fetch the next instruction from address &4.

To return from this trap (after performing a suitable emulation of the required function), use `MOVS PC,R14_svc`. This will return to the instruction following the undefined instruction.

Reset

ARM can be reset by pulling its RESET pin HIGH. If this happens, ARM will:

1 stop the currently executing instruction and start executing no-ops. When RESET goes low again, it will:

2 save R15 in R14_svc

3 force M0,M1 to supervisor mode and set the F and I bits in the PC word

4 force the PC to fetch the next instruction from address &0.

Vector summary

The first eight words of store normally contain branch instructions pointing to the relevant routines. The FIQ routine may reside at &000001C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

Address	Definition
&0000000	Reset
&0000004	Undefined instruction
&0000008	Software interrupt
&000000C	Abort (prefetch)
&0000010	Abort (data)
&0000014	Address reception
&0000018	IRQ
&000001C	FIQ

Modes of operation

There are four modes of operation:

- user mode
- fast interrupt mode
- interrupt mode
- supervisor mode.

The mode in which the processor runs is determined by the state of bits 0 and 1 in the Processor Status Register. The processor has 27 physical registers, but the state of the mode bits determines which 16 registers, R0-R15, will be seen by the programmer. The four modes available are described below and shown in *The four modes of operation* on page 46.

Mode 0: User mode

User mode is the normal program execution state. Registers R0-R15 exist directly, and in this mode, only the N, Z, C and V bits of the PSR may be changed.

Mode 1: Fast interrupt mode

The FIQ processing state has seven private registers mapped to R8-R14 (R8_fiq-R14_fiq) and a fast interrupt will not destroy anything in R8-R14. Most FIQ programs, particularly those used for data transfer, will not need to use R0-R7, but if they do, then R0-R7 can be saved in memory using a multiple register data transfer instruction.

Mode 2: Interrupt mode

The IRQ processing state has two private registers mapped to R13 and R14 (R13_irq and R14_irq). If other registers are needed, their contents should be saved in memory using one of the multiple register data transfer instructions available for this purpose.

Mode 3: Supervisor mode

Supervisor mode (entered on SVC calls and other traps) also has two private registers mapped to R13 and R14 (R13_svc and R14_svc). If other registers are needed, they must be saved in memory.

Non-user modes

Non-user modes are privileged and allow trusted software to take control in a suitably protected system.

Changing operating modes

In the Assembler, the suffix P added to a CMN, CMP, TEQ or TST instruction causes the instruction to change the PSR directly. Such instructions can be used to change the ARM's mode, for example:

`TEQP R15,#2` changes to IRQ mode
`TEQP R15,#0` changes to user mode.

The action is to Exclusive OR the first operand with a supplied immediate field. R15 is the first operand. Whenever R15 is presented to the processor as the first operand, 24 bits are presented; the PSR bits are supplied as zero. The TEQ causes the immediate field value to be written into the register, and the P causes the PSR bits (now altered by the immediate field value) to be written back into R15. Since two of the PSR bits are the mode control bits, the processor assumes its new mode.

As the mode control bits cannot be set in user mode, this technique will not work in user mode. There are, however, two ways to pass from user mode to other modes:

- by receiving an external interrupt
- by making use of the SWI instruction.

Note: For more details of instructions executed immediately following a mode change see the section entitled *Forcing transfer of the user bank* on page 229 and the section entitled *Writing to* R15 on page 216.

Further information

An explanation of the ARM interrupt capability and of its operation modes is given in *Appendix* D - ARM *datasheet*.

6 Assembler language

This chapter describes the language syntax, symbols, labels, expressions, constants, and operators available in the Archimedes' Assemblers.

Assembler language syntax

Whenever the Assembler is required to generate opcodes representing program instructions, the general style of a three field line is used.

{label} {instruction} {;comment}

The label and instruction fields are separated by one or more spaces; however, if the line starts with a space, the label is absent.

{label} if present, this defines a symbol which is set equal to the address of the instruction assembled. If the instruction is absent, then the address used is the current value of the instruction location pointer. This may not be word-aligned, for example, when the last directive was one of the store-loading directives detailed in the section entitled *Store-loading* on page 124. However, using a label with an instruction ensures that the address generated is word-aligned.

(Symbols are described in the section entitled *Symbols and labels* below.)

{instruction} if present, this defines the instruction to be assembled.

(See the chapter entitled CPU *instruction set.*)

{;comment} if present, the comment is started by the first semi-colon on the line (ignoring semi-colons inside string constants). The semi-colon and the rest of the line are ignored by the Assembler.

A specific example of the three fields taken from an assembly listing is:

```
L321            ADD   Ra,Ra,Ra,LSL #1 ;multiply by 3
```

{label}	`L321`
{instruction}	`ADD   Ra,Ra,Ra,LSL #1`
{;comment}	`;multiply by 3`

It is worth noting two special cases of this syntax, both of which are ignored by the Assembler:

- a completely blank line is valid, and may be used to make the text more readable
- a line may start with a semi-colon in which case the entire line is treated as comment.

The Assemblers AAsm and ObjAsm treat a tab character (&09) in a source file as a space (&20), and will accept both linefeeds (&0A) and carriage returns (&0D) as line terminators. The combinations &0A &0D and &0D &0A are treated as a single line terminator.

Symbols and labels

A symbol is a group of alphanumeric characters which represents a number, logical value or string value. The values are assigned immediately by Assembler directives.

Symbols have the following characteristics:

- All symbols must start with a letter, A-Z or a-z. Lower-case letters may be used and will be treated as different from their upper-case counterparts.
- Numeric characters and the underscore character '_' may be embedded in the symbol.
- Symbols may also be the same as mnemonics, although this is not recommended, as it is likely to be confusing to the programmer. However, the Assembler will distinguish between a symbol and a mnemonic by their relative positions on the program line.
- Symbols can be any length (but the line length may not be more than 255 characters).
- All characters are significant.

A special syntax using enclosing | bars allows any character to be placed in a symbol. This allows the use of labels which are compatible with the output of compilers, which may use other characters within their symbols. The enclosing bars are not seen as part of the symbol. For example:

```
|C$$Code|
```

is one such symbol or label.

Labels

A label is a special type of symbol which the Assembler recognises by its position on the line, ie the first character of the label occupies the first column of the line. The number represented by the label is not always immediately known to either the programmer or the Assembler, but is generated as the assembly takes place.

Expressions

Expressions are combinations of simple values, unary and binary operators, and brackets.

Evaluating expressions

The order of evaluation of expressions is determined by:

1 bracketing
2 precedence rules
3 in the absence of brackets or precedence rules, evaluation is from left to right.

(For information on the precedence rules see the section entitled *Operator summary* on page 64.)

So:

- A bracketed sub-expression is always evaluated before being used as an operand to an operator.
- An operand with a binary operator on both sides is always used as an operand to the higher precedence operator, or if they have equal precedence, to the lefthand operator.

Types of operands

Every simple value has a type associated with it, as does every operand produced at any stage of the expression evaluation, including the expression result. The types of operand are:

- numeric values
- string values
- logical values
- program-relative values
- register-relative values.

For an expression to be syntactically correct, every operator must be provided with operands of the correct types.

Numeric values

Numeric values are unsigned integers in the range 0 to &FFFFFFFF. Overflow is ignored when doing calculations with numeric values (for example, -1 evaluates to &FFFFFFFF).

Comparisons are always unsigned comparisons, which may have counter-intuitive results in some cases (for example, -1>1 evaluates to 'true'). In a few places, this manual contains such statements as:

'The immediate value must lie in the range -4095 to 4095'.

The values are presented in this way for clarity, but the accurate interpretation of this example is:

'The immediate value must lie in the range 0 to &FFF, or &FFFFF001 to &FFFFFFFF'.

String values

String values are strings of 0 to 256 bytes, each of which may take any value in the range 0 to 255. The byte values are usually ASCII printable characters. The Assembler will convert a string of length 1 into a numeric value if necessary. See the section entitled *Numeric constants* on page 59 for further information on string conversion.

Program-relative values

Program-relative values are simply offsets from the program origin. All labels on instructions and stand-alone labels are therefore program-relative values. In the case where the program has a fixed origin, the distinction between numeric values and program-relative values disappears.

Register-relative values

Register-relative values are offsets from a base register, therefore the difference between two such values (having the same base register) is a numeric value.

- Simple register-relative values can be defined using the two operand form of the ^ and # directives (see the section entitled *Laying out storage areas* on page 126).
- Within the evaluation of an expression a register-relative value may acquire a base of a signed sum of registers, but by the time the evaluation of the expression is complete this must have collapsed to either a numeric value or an offset from a single register.

  Note: This is a technicality which probably need not worry the programmer.
- Register-relative values for which the base register is the program counter are always converted into program-relative values.

Numeric constants

The Assembler can accept numbers given to it in any of three forms:

Value	Type of constant
`123456`	decimal constants
`&A1F40`	hexadecimal constants
`n_xxxx`	number in the form base n

eg 2_101 is binary 101
n may be between 2 and 9

It will also evaluate a quoted ASCII character (for example, the character A) to a number if necessary.

Note that quoted ASCII characters are evaluated to their ASCII values, for example, '0' is evaluated to 48, not 0.

String constants

A string constant consists of an opening set of quotes, characters and closing quotes. The string can also contain embedded spaces, leading or trailing spaces, for example:

```
" a default string "
```

If the string needs to contain double quotes, then pairs of double quotes are used to represent these double quotes. For example:

```
"She said ""Hello""."
```

This represents the characters: She said "Hello".

$ character

The $ character may be used in a string, provided that it is represented by a pair of $ characters, for example:

```
=       "the price is $$$etc"
```

The $$ will be interpreted as a real $, and then $etc will be correctly interpreted as a parameter. See the section entitled *Variable substitution using* $ on page 129.

Boolean constants

The logical values 'true' and 'false' can be input to the Assembler as the logical constants {TRUE} and {FALSE}.

Assembler operators

The Assembler provides an extensive set of operators for use in expressions. Many of these operators resemble their counterparts in high level languages.

Binary operators

Binary operators act on two operands and are placed between the operands. For example:

```
VALUE-2
1:SHL:EXPONENT
```

Unary operators

Unary operators act on one operand and are placed before it. For example:

```
-VALUE
:LNOT:FLAG
:DEF:LABEL
```

Arithmetic operators

+	add/unary +	binary or unary
-	subtract/unary -	binary or unary
*	multiply	binary
/	divide	binary
:MOD:	remainder after division	binary

For the purposes of division, remainder and comparisons all values are treated as 32-bit unsigned integers in the range $0 - 2^{32} - 1$. The operators + and - act on numeric, program-relative and register-relative expressions, the others act only on numeric expressions.

Boolean logical operators

:LAND:	Logical AND	binary
:LOR:	Logical OR	binary
:LEOR:	Logical Exclusive OR	binary
:LNOT:	Logical NOT	unary

These perform the normal logical operations. Thus:

- `expr1 :LOR: expr2` gives TRUE if either expression is TRUE
- `expr1 :LEOR: expr2` gives TRUE if one of the expressions is TRUE but not both

- *expr1* `:LAND:` *expr2* gives TRUE if both expressions are TRUE and FALSE otherwise
- `:LNOT:` *expression* gives TRUE if the expression is FALSE, and vice versa.

Bitwise logical operators

:AND:	bitwise AND	binary
:OR:	bitwise OR	binary
:EOR:	bitwise Exclusive OR	binary
:NOT:	bitwise NOT	unary

These act on numeric expressions. The operation is done independently on each bit of the binary expansion(s) of the operand(s) to produce the binary expansion of the result.

The C language operator ~ can optionally be used in place of :NOT:.

Shift operators

:ROL:	ROtate Left	binary
:ROR:	ROtate Right	binary
:SHL:	SHift Left	binary
:SHR:	SHift Right	binary

These act on numeric expressions. The first operand is shifted or rotated by an amount given by the second operand. The shifts are logical rather than arithmetic, for example, `-1:SHR:1 = &7FFFFFFF`.

The C language operators << and >> can optionally be used in place of :SHL: and :SHR: respectively.

Relational operators

=	equal	binary
>	greater than	binary
>=	greater than or equal	binary
<	less than	binary
<=	less than or equal	binary
<>	not equal	binary
/=	not equal	binary

These act between two operands of the same type. The allowable types are:

- numeric
- program-relative
- register-relative

- string.

They produce a logical value.

When strings are used as operands in string comparisons, a lexical or dictionary ordering is used. The ordering of characters is the ASCII ordering. String a is less than or equal to string b if either string a is a leading substring of string b, or at the leftmost character position at which the two strings differ the character in string a is less than the corresponding character in string b. For example:

"A"	<	"B"	is TRUE
"A"	<=	"AB"	is TRUE
"B"	<=	"A"	is FALSE
"Label1"	<=	"Label2"	is TRUE

String operators

Concatenation (binary)

:CC: joins (concatenates) two strings.

`expression1 :CC:expression2`

where *expression1* and *expression2* are strings.

For example:

`"ABCD":CC:"EFGH"` gives `"ABCDEFGH"`

Slicing (binary)

`expression1 :LEFT: expression2`
`expression1 :RIGHT: expression1`

where *expression1* is a string and *expression2* is numeric.

`"sssss" :LEFT:n`

returns the n left-most characters from the string *"sssss"*.

`"sssss" :RIGHT:n`

returns the n right-most characters from the string *"sssss"*.

For example:

`"EGBDF" :LEFT:1` returns "E"
`"EGBDF" :RIGHT:1` returns "F"

Length (unary)

`:LEN: expression`

returns the length of a string expression.

Conversion (unary)

`:CHR: expression`

returns a string of length 1 having ASCII code expression. The expression must be numeric.

`:STR: expression`

returns an eight-digit hexadecimal string corresponding to an expression if the expression is numeric, or returns the string T or F if the expression is logical.

:BASE: (unary) — `:BASE: register-relative` or `PC-relative expression`

gives the number of the register.

:DEF:LABEL — `:DEF:label`

is used to determine if label is already defined as an assembly-time variable (by GBLL, GBLA or GBLS). It is a unary logical operator returning TRUE if label is so defined or FALSE if otherwise. An error is generated if label is used for another purpose.

:INDEX: (unary) — `:INDEX: register-relative` or `PC-relative expression`

gives the offset.

?Label

`?label` is used to find out how many bytes of code were produced on the label's defining line. For a label on a line containing an opcode mnemonic, the length is four; for a label on an otherwise blank line, the length is zero. For DCD, DCW, DCB, DCFS, DCFD and % directives, the length is the combined length of all the operands. For example:

```
STORE       &      1,2,3,4,5 ;5 words into STORE
STORELENGTH *      ?STORE    ;?STORE evaluates to 20
```

Operator summary

The precedence or relative binding of an operator is given as a number from 1 to 7, where 7 indicates the highest binding power. Note that unary operators are evaluated from right to left.

?	7	?A	Amount of code generated by line defining A
+	7	+A	Unary plus
-	7	-A	Unary negate
LNOT	7	:LNOT:A	Logical complement of A
NOT	7	:NOT:A	Bitwise complement of A
DEF	7	:DEF:A	Tests whether A is a defined assembly-time variable
LEN	7	:LEN:A	Length of string A
CHR	7	:CHR:A	ASCII string of A
STR	7	:STR:A	Hexadecimal string of A
*	6	A*B	Multiply
/	6	A/B	Divide
MOD	6	A:MOD:B	A modulo B
LEFT	5	A:LEFT:B	the left-most B characters of A
RIGHT	5	A:RIGHT:B	the right-most B characters of A
CC	5	A:CC:B	B concatenated on to the end of A
ROL	4	A:ROL:B	Rotate A left B bits
ROR	4	A:ROR:B	Rotate A right B bits
SHL	4	A:SHL:B	Shift A left B bits
SHR	4	A:SHR:B	Shift A right B bits
+	3	A+B	Add A and B
-	3	A-B	Subtract B from A
AND	3	A:AND:B	Bitwise AND on A and B
OR	3	A:OR:B	Bitwise OR on A and B
EOR	3	A:EOR:B	Bitwise exclusive OR on A and B
=	2	A=B	A equal to B
>	2	A>B	A greater than B
>=	2	A>=B	A greater than or equal to B
<	2	A<B	A less than B
<=	2	A<=B	A less than or equal to B
/=	2	A/=B	A not equal to B
<>	2	A<>B	A not equal to B
LAND	1	A:LAND:B	Logical AND on A and B
LOR	1	A:LOR:B	Logical OR on A and B
LEOR	1	A:LEOR:B	Logical EXCLUSIVE OR on A and B

7 CPU instruction set

This chapter describes the CPU instructions available in the Archimedes' Assemblers.

Conditional execution

Every ARM instruction is conditional so it will only be executed if the N, Z, C and V flags are in the correct state. The default condition is 'always execute' but other conditions can be requested by adding a two-character condition mnemonic to the standard form:

Mnemonic	Condition	Condition of flag(s)
EQ	EQual	Z set
NE	Not Equal	Z clear
CS	Carry Set / unsigned higher or same	C set
CC	Carry Clear / unsigned lower than	C clear
MI	negative (MInus)	N set
PL	positive (PLus)	N clear
VS	oVerflow Set	V set
VC	oVerflow Clear	V clear
HI	HIgher unsigned	C set and Z clear
LS	Lower or Same unsigned	C clear or Z set
GE	Greater or Equal	(N set and V set) or (N clear and V clear)
LT	Less Than	(N set and V clear) or (N clear and V set)
GT	Greater Than	((N set and V set) or (N clear and V clear)) and Z clear
LE	Less or Equal	(N set and V clear) or (N clear and V set) or Z set
AL	ALways	any
NV	NeVer	none

Note that the Assembler implements HS (Higher or Same) and LO (LOwer than) as synonymous with CS and CC respectively, giving a total of 18 mnemonics.

Conditional instruction sequence

Branches which are taken cause breaks in the pipeline. For this reason they often waste time, and can sometimes be replaced by a suitable conditional instruction sequence.

As an example, the coding of IF A=4 THEN B:=A ELSE C:=D+E might be conventionally achieved using five ARM instructions:

```
        CMP  R5, #4          ;test "A=4"
        BNE  LABEL           ;if not equal goto LABEL
        MOV  R6,R5           ;do "B:=A"
        B    LAB2            ;jump around the ELSE clause
LABEL   ADD  R0,R1,R2        ;do "C:=D+E"
LAB2                         ;finish
```

whereas, using the condition testing instructions, the same effect may be achieved using three instructions:

```
        CMP    5, #4         ;test "A=4"
        MOVEQ  R6,R5         ;if so do "B:=A"
        ADDNE  R0,R1,R2      ;else do "C:=D+E".
```

If the condition tested is true, the instruction is performed. If it is false, the instruction is skipped and the PC is advanced to the next memory word. This takes one S-cycle of processor time. The first of the examples above takes about twice as long as the second.

After the instruction is obeyed, the arithmetic logic unit (ALU) will output appropriate signals on the flag lines. On certain instructions, the flags set the condition code bits in the PSR; for other instructions, the flags in the PSR are only altered if the programmer permits them to be updated.

Instruction timing

All instruction timings are defined in terms of four types of processor cycle:

- sequential cycles
- non-sequential cycles
- coprocessor cycles
- internal cycles.

Sequential cycles (or S-cycles) are used when the processor needs to access a memory location that is the same as or one word after the memory location accessed in the previous cycle. On a typical 8MHz ARM2 Archimedes machine, they usually take 0.125 microseconds. They take longer (0.250 microseconds) if a four word boundary is crossed.

Non-sequential cycles (or N-cycles) are used when the processor needs to access a memory location that is unrelated to the memory address used in the previous cycle. On a typical 8MHz ARM2 Archimedes machine, they take 0.250 microseconds.

Coprocessor cycles (or C-cycles) are used when the processor needs to access a coprocessor. On a typical 8MHz ARM2 Archimedes machine, they take 0.125 microseconds.

Internal cycles (or I-cycles) are used when the processor does not need to access either memory or a coprocessor. On a typical 8MHz ARM2 Archimedes machine, they take 0.125 microseconds.

An instruction that is not executed because its condition has not been met always executes in one S-cycle. If the condition is met, the following table gives minimum instruction timings in terms of cycles and microseconds on a current Archimedes machine. However, programmers are advised to note the following facts:

- Instructions involving S-cycles will take longer than indicated if a four word boundary is crossed.
- Instructions involving coprocessors may take longer than indicated if the requested coprocessor is busy. For more details, refer to the documentation for the specific coprocessor.
- Programs are likely to take longer than indicated because of interrupts, VDU memory accesses and similar effects. This means that delay loops and similar devices should not in general be used, instead use the appropriate operating system routines. These timings should only be used for such purposes as deciding which is the fastest of a number of possible code fragments.
- Programs may run in less time than indicated on machines with greater processing power, for example a machine fitted with ARM3.

Instruction	Cycles used	Minimum timing (microseconds)
Data processing instructions	1S	0.125
	+ 1S for a register-controlled shift	+ 0.125
	+ 1S + 1N if R15 is written	+ 0.375
LDR	1S + 1N + 1I	0.500
	+ 1S + 1N if R15 is loaded	+ 0.375
STR	2N	0.500
LDM (of n registers)	nS + 1N + 1I	0.125 (n+3)
	+ 1S + 1N if R15 is loaded	+0.375
STM (of n registers)	(n-1)S + 2N	0.125(n+3)
B, BL or SWI	2S + 1N	0.500
MUL or MLA	1S + nI, where n depends on the value of the third operand as follows:	0.125(n+1)
	Operand	n
	0 or 1	1
	2 to 7	2
	8 to 31	3
	:	:
	2^(2x-3) to 2^(2x-1)-1	x
	:	:
	&8000000 to &1FFFFFFF	15
	&20000000 to &FFFFFFFF	16
CDP	1S	0.125
LDC or STC (n words)	(n-1)S + 2N	0.125(n+3)
MRC	1S + 1C	0.250
MCR	1S + 1I + 1C	0.375

The barrel shifter

The arithmetic logic unit has a 32-bit barrel shifter capable of various shift and rotate operations. Data involved in the data processing group of instructions (detailed in the section entitled *Data processing* on page 73) may pass through the barrel shifter, either as a direct consequence of the programmer's actions, or in other cases, as a result of the internal computations of the Assembler. The barrel shifter also affects the index for the single data transfer instructions (detailed in the section entitled *Single data transfer* on page 79).

The shift mechanism can produce the following types of operand:

Unshifted register

Syntax: `register`
For example: `R0`

Register shifted by a constant amount

A register shifted by a constant amount, in the range 0-31, 1-31 or 1-32 (depending on shift type).

Syntax: `register, shift-type #amount`
For example: `R0,LSR #1`

Value resulting from rotating register and carry bit one bit right

A value which is the result of rotating a register and the carry bit one bit right. Because the carry is included in the shift, 33 bits (rather than 32 bits) are affected. The shift type is always rotate right.

Syntax: `register,RRX`

For example: `R0,RRX`

Register shifted by n bits

A register shifted by n bits, where n is the least significant byte of a register. This form is not valid as an index in a single register transfer.

Syntax: `register,shift-type register`
For example: `R1,LSL R2`

8-bit constant rotated right by n * 2 bits

A constant constructed by rotating an 8-bit constant right by n * 2 bits, where n is a 4-bit constant. The shift type is always rotate right. This form is not valid as an index in a single register transfer.

Syntax: `#expression`
For example: `#&3FC`

Note that the rotation is invisible to the programmer, who should merely supply an immediate value for the data processing instruction to use.

The Assembler will evaluate the expression and reject any number which cannot be expressed as a rotation by an even amount of a number in the range 0-255. If possible, the Assembler always constructs it as an unrotated value, even if there are other possibilities.

Examples of valid immediate constants are:

```
#1
#&FF
#&3FC           ;This is &FF rotated right by 30
#&80000000      ;This is 2 rotated right by 2
#&FC000003      ;This is &FF rotated right by 6.
```

Examples of invalid constants are:

`#&101`	cannot be obtained by rotating an 8-bit value
`#&1FE`	an 8-bit value rotated by an odd amount but not an 8-bit value rotated by an even amount.

8-bit constant rotated right by n * 2 bits and specified explicitly

A constant constructed as in the point above, but specified explicitly. This form is not valid as an index in a single register transfer.

Syntax: *`#constant, rotate amount`*
For example: `#4,2`

The shift amount should be an even number in the range 0-30. This can be important for setting the carry flag on an operation which would otherwise not update it.

For example:

`MOVS   R0,   #4,2` produces the same result as

`MOVS   R0,   #1`

but because the first instruction does a rotate right of two bits the carry flag is cleared, whereas it is not altered by the second instruction.

Shift types

There are four shift types. These are:

LSL	Logical Shift Left
LSR	Logical Shift Right
ASR	Arithmetic Shift Right
ROR	Rotate Right

The mnemonic ASL (arithmetic shift left) may be freely interchanged with LSL (logical shift left).

Logical shift left

```
C <- - bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb <- - 0
```

Rm,LSL #*n*	Shift contents of *Rm* left by *n* which may be 0 to 31 bits.
Rm,LSL *Rs*	Shift contents of *Rm* left by the least significant byte of *Rs*.

If the shift amount is zero, no shift is performed and the carry flag is not altered. If the shift amount lies in the range 1 to 32, the carry flag is set to b(32-n). If the shift amount is greater than 32, the carry flag is set to zero.

Logical shift right

```
0 - -> bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb - -> C
```

Rm,LSR #*n*	Shift contents of *Rm* right by *n* which may be 1 to 32 bits.
Rm,LSR *Rs*	Shift contents of *Rm* right by the least significant byte of *Rs*.

If the shift amount is zero, no shift is performed and the carry flag is not altered. If the shift amount lies in the range 1 to 32, the carry flag is set to b(n-1). If the shift amount is greater than 32, the carry flag is set to zero.

Arithmetic shift right

```
- - ->xbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb - -> C
|     |      - - - - - - - - - - - - >
<- - -
```

Rm,ASR #*n*	Shift contents of *Rm* right by *n* which may be 1 to 32 bits.
Rm,ASR *Rs*	Shift contents of *Rm* right by the least significant byte of *Rs*.

The n most significant bits become equal to b31 (that is, on every single shift, bit 31 is duplicated).

If the shift amount is zero, no shift is performed and the carry flag is not altered. If the shift amount lies in the range 1 to 32, the carry flag is set to b(n-1). If the shift amount is greater than 32, the carry flag is set to b31.

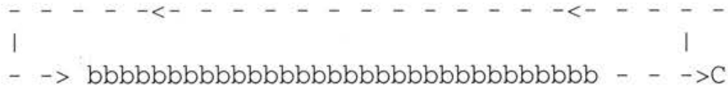
Rotate right

```
- - - - -<- - - - - - - - - - - -<- - - - -
|                                         |
- -> bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb - - ->C
```

Rm,ROR #*n*	Rotate contents of *Rm* right by *n* which may be 1 to 31 bits.
Rm,ROR *Rs*	Rotate contents of *Rm* right by the least significant byte of *Rs*.

If the LSB of *Rs* evaluates to zero, then no rotation is performed and the carry bit is not altered. Otherwise, the carry is set to the last bit rotated into b31.

Rotate right with extend



`Rm, RRX` Rotate right the contents of *Rm* and the carry flag by 1 bit only.

Branch instructions

The branch instruction takes a 24-bit word offset (equivalent to a 26 bit byte offset), allowing forward jumps of up to +&2000004 and backward jumps of up to -&1FFFFF8 to be made. This is sufficient to address the entire memory map as the calculation 'wraps round' between the top and bottom of memory. The programmer should provide a label from which the Assembler will calculate a 24-bit offset.

Branch

The instruction syntax is:

`B{condition} expression`

`{condition}`	One of the condition codes specified in the section entitled *Conditional execution* on page 65.
`expression`	A program-relative expression describing the branch destination.

For example:

```
B      LABEL   ;branch to LABEL
BNE    LABEL1  ;if not equal goto LABEL1
```

Note that in the absence of the condition mnemonic, the branch is always performed.

The ARM Assembler automatically handles the effects of pipelining and prefetching within the CPU. For example, the calculated jump offset in the following piece of code is 000000 even though the jump is to a label two PC locations ahead.

code generated	**Label**	**Mnemonic**	**Destination**
EA000000	L1	BEQ	L2
xxxxxxxx		xxx	
xxxxxxxx	L2	xxx	

Branch with link

The instruction syntax is:

`BL{condition} expression`

{condition}	One of the condition codes specified in the section entitled *Conditional execution* on page 65.
expression	A program-relative expression describing the branch destination.

Whenever branch with link is specified, 4 is subtracted from the contents of R15 (including the PSR) and the result is written to R14. Thus the value written into the link register is the address of the instruction following the branch and link instruction. Therefore, after branching to a subroutine, the program flow can return to the memory address immediately following the branch instruction by writing back the R14 value into R15. Subroutines can be called by a BL instruction. The subroutine should end with a

```
MOV    PC,R14
```

if the link register has not been saved on a stack or

```
LDMxx Rn,{PC}
```

if the link register has been saved on a stack addressed by `Rn`. (xx is the stack type, see the section entitled *Block data transfer* on page 81.)

These methods of returning do not restore the original PSR. If the PSR does need to be restored then

	`MOV PC,R14`	can be replaced by	`MOVS PC,R14`
or	`LDMxx Rn,{PC}`	by	`LDMxx Rn,{PC}^`

However, care should be taken when using these methods in modes other than user mode, as they will also restore the mode and the interrupt bits. In particular, restoring the interrupt bits may interfere unintentionally with the interrupt system.

Data processing

There are sixteen data processing instructions:

ADC	ADd with Carry
ADD	ADD
AND	bitwise AND
BIC	BIt Clear
CMN	CoMpare Negated
CMP	CoMPare
EOR	bitwise Exclusive OR

MOV	MOVe
MVN	MoVe Not
ORR	bitwise OR
RSB	Reverse SuBtract
RSC	Reverse Subtract with Carry
SBC	SuBtract with Carry
SUB	SUBtract
TEQ	Test EQuivalence
TST	TeST and mask.

Except in the cases of MOV and MVN, the operation is performed between a source register Rn and an operand. In the cases of MOV and MVN, only an operand is needed. The source register can be any one of the 16 registers, and the operand can be any operand that the barrel shifter can produce (see the section entitled *The barrel shifter* on page 68 for details). Note that any shifting is done before the operation is performed. Some instructions use the bit held in the ALU's carry flag and add it into the operation. The result of the operation is placed in the destination register, which may be any one of the 16 registers.

Each of these instructions contains a one bit field called the S bit, standing for 'set condition codes'. The result of the operation affects the N and Z flags, and may also affect the C and V flags. However, the ALU doesn't copy the contents of its flags to the relevant parts of the PSR unless the S bit is set. In the case of the four instructions CMN, CMP, TEQ and TST, the Assembler always sets the S bit since these instructions would be meaningless if their results were not copied to the PSR. In the case of the remaining 12 instructions, the programmer may request that the ALU flags are copied to the PSR by including the letter S in the source line. This forces the PSR update.

For example:

```
ADDS    R2,R0,R1  ;Add the contents of R1 to the
                  ;contents of R0, and put the result
                  ;in R2. Modify flags N, Z, C and V.
```

Data processing instruction syntax

The data processing instructions use three different types of syntax, depending on which opcode is being used:

MOV and MVN

`opcode{condition}{S} destination,operand`

`{condition}` A two-character condition mnemonic. In the absence of the condition mnemonic, AL is assumed.

{S}	optional: it sets the S bit in the instruction. If S is specified, N and Z are set according to the value placed in the destination register, and C is set to the last bit shifted out by the barrel shifter or is unchanged if no shifting took place. V is unchanged.
destination	must be a register.
operand	may be any of the operands that the barrel shifter can produce.
MOV	causes the operand to be placed unchanged in the destination register.
MVN	causes the operand to be evaluated and its bitwise inverse to be placed in the destination register.

For example:

```
MOV R0,R1,LSL#2
```

The contents of register 1 are shifted left by 2 bits and transferred to register 0.

```
MVN R2,R3
```

Register 2 is set to the bitwise inverse of the contents of register 3.

ADD, ADC, SUB, SBC, RSB, RSC, AND, BIC, ORR, EOR

opcode{condition}{S} destination,operand1,operand2

{condition}	A two-character condition mnemonic. In the absence of the condition mnemonic, AL is assumed.
{S}	optional: it sets the S bit in the instruction. If S is specified, then the N and Z flags are set according to the value placed in the destination register. For ADC, ADD, RSB, RSC, SBC and SUB, the C and V flags are set according to the result of the arithmetic operation. For AND, BIC, EOR and ORR, V is left unchanged, and C is set to the last bit shifted out by the barrel shifter or is unchanged if no shifting took place.
destination	must be a register.
operand1	must be a register.
operand2	may be any of the operands that the barrel shifter can produce.
ADD	addition is performed on operand1 and operand2. The result is stored in the destination register.

ADC — addition is performed on operand1 and operand2 and the carry flag. The result is stored in the destination register. This instruction can be used to implement multi-word additions. For example a 64 bit ADD:

```
ADDS R4,R2,R0 ;  Add least significant 32
              ;  bits updating carry
ADC  R5,R3,R1 ;  Add most significant 32
              ;  bits and carry from
              ;  previous
```

SUB — operand2 is subtracted from operand1. The result is stored in the destination register.

SBC — if the carry flag is set, operand2 is subtracted from operand1. If the carry flag is clear, operand1-operand2-1 is calculated. The result is stored in the destination register. This instruction can be used to implement multi-word subtractions. For example:

```
 SUBSR4,R2,R0 ; Do least significant
              ;  word of subtraction
 SBC R5,R3,R1 ;  Do most significant
              ;  word, taking account of
                 the borrow;
;This does the 64 bit subtraction;
;(R5,R4)=(R3,R2)-(R1,R0)
```

The result is stored in the destination register.

RSB — operand1 is subtracted from operand2. The result is stored in the destination register.

RSC — if the carry flag is set, operand1 is subtracted from operand2. If the carry flag is clear, operand2-operand1-1 is calculated. The result is stored in the destination register.

AND — a bitwise AND is performed on operand1 and operand2. The result is stored in the destination register.

BIC — bitwise inversion is performed on operand2, then a bitwise AND is performed on operand1 and the result of the inversion. The result is stored in the destination register.

ORR — a bitwise OR is performed on operand1 and operand2. The result is stored in the destination register.

EOR — a bitwise Exclusive OR is performed on operand1 and operand2. The result is stored in the destination register.

For example:

```
ADD R0,R1,R2        ;R0=R1+R2
ADDS R0,R1,#1       ;R0=R1+1 and set N,Z,C,V
```

For ADD and ADC carry is generated if 32 bit overflow occurred. For SUB, SBC, RSB and RSC carry is generated if and only if 32 bit underflow did not occur.

For ADD, ADC, SUB, SBC, RSB and RSC, the V flag is set if signed overflow occurred, ie if the carry into bit 31 was not equal to the carry out of bit 31.

CMN, CMP, TEQ, TST

opcode{condition}{P} operand1,operand2

{condition}	a two-character condition mnemonic. In the absence of the condition mnemonic, AL is assumed.
{P}	See below.
operand1	must be a register.
operand2	may be any of the operands that the barrel shifter can produce. There is no need to specify S as it is assumed by the Assembler. S may be specified in the syntax and it will be accepted provided that P has not also been specified. For example, `CMPSP` and `CMPPS` will not be accepted, but `CMPS` will be.
CMP	operand2 is subtracted from operand1. Flags N, Z, C and V are altered.
CMN	operand2 is added to operand1. This allows a negative data field to be created for a compare. Flags N, Z, C and V are altered.
TEQ	a bitwise exclusive OR is performed between operand1 and operand2.
TST	a bitwise AND operation is performed between operand1 and operand2. In the case of TEQ and TST, the N and Z flags are altered according to the result, V is unchanged, and C is set to the last bit shifted out by the barrel shifter or is unchanged if no shifting took place.

For example:

```
CMP R0,R1      ;Compare the contents of R0 with R1
CMP R0,#&80    ;Compare the contents of R0 with &80
```

{P} there are special forms for CMN, CMP, TEQ and TST in which the result of the operation is moved to the PSR even though the instruction has no destination register. In user mode, the N, Z, C and V flags are set from the top four bits of the result. In other modes, the N, Z, C, V, I and F flags are set from the top six bits of the result and the mode bits from its bottom two bits.

Invoking this special form is done by adding P to the instruction. One reason for wanting to modify R15 in this way would be to change modes.

For example:

```
TEQP R15, #0;change to user mode.
```

Note the treatment of R15 as the first operand, described in the second point below. It is unlikely that most applications will need to do this.

Using R15 as the destination or operand

Note that the CPU takes certain actions whenever the destination or any operand is R15. These are as follows:

- if R15 is the destination register, 24 bits are moved to R15 if the S bit is not set. These bits become the new PC. In user mode, 28 bits are moved to R15 if the S bit is set; these are the 24 PC bits and the N, Z, C and V flags. In other modes, all 32 bits are moved to R15 if the S bit is set.
- if R15 is the first operand in a two operand instruction, R15 is presented to the arithmetic logic unit (ALU) with the PSR bits set to zero.
- if the second or only operand is R15 (possibly shifted), R15 is presented to the barrel shifter or ALU with the PSR bits unchanged.
- if R15 is the rotation register, R15 is presented to the barrel shifter with the PSR bits set to zero.

Data processing instruction summary

Mnemonic	Meaning	Operation	Flags Affected
ADC	Add with Carry	Rd:= Rn + operand + C	N,Z,C,V
ADD	Add	Rd:= Rn + operand	N,Z,C,V
AND	And	Rd:= Rn AND operand	N,Z,C
BIC	Bit Clear	Rd:= Rn AND (NOT(operand))	N,Z,C
CMN	Compare Negated	Rn + operand	N,Z,C,V
CMP	Compare	Rn - operand	N,Z,C,V
EOR	Exclusive Or	Rd:= Rn EOR operand	N,Z,C
MOV	Move	Rd:= operand	N,Z,C
MVN	Move Not	Rd:= NOT operand	N,Z,C
ORR	Logical Or	Rd:= Rn OR operand	N,Z,C
RSB	Reverse Subtract	Rd:= operand-Rn	N,Z,C,V
RSC	Reverse Subtract with Carry	Rd:= operand-Rn-1+C	N,Z,C,V
SBC	Subtract with Carry	Rd:= Rn-operand-1+C	N,Z,C,V
SUB	Subtract	Rd:= Rn-operand	N,Z,C,V
TEQ	Test Equivalence	Rn EOR operand	N,Z,C
TST	TeST and mask	Rn AND operand	N,Z,C

Note that Rd is the destination register; Rn is a source register.

Single data transfer

This group of instructions is used for moving data between registers and memory. LDR (LoaD Register) loads a register from a memory location, while STR (STore Register) stores a register to a memory location. Both instructions may use pre-indexed or post-indexed addressing; in the case of pre-indexed addressing, write back may be used. (Write back means that the base register is updated.) The amount of data transferred may be either a word or a byte. Special versions of the post-indexed instructions also exist which cause the TRANS pin of the ARM to be active throughout the data transfer. These are useful for loading or storing user data areas from the supervisor state in a memory-managed system.

For register to register transfers, see the section entitled *Data processing* on page 73 and the MOV instruction in particular.

Single data transfer instruction syntax

There are two types of single data transfer instruction syntax.

Pre-indexed instruction (possibly with write back)

opcode{condition}{B} *register*,[*base*{,*index*}]{!}

Post-indexed instruction (always with write back)

opcode{condition}{B}{T} *register*,[*base*]{,*index*}

opcode may be LDR or STR, and must not be omitted.

{condition} may be any of the two-character condition mnemonics listed in the section entitled *Conditional execution* on page 65. If omitted, AL is assumed.

{B} if present the transfer will be of just one byte. If omitted, a full word is transferred. Note that transfers of words to or from non-word-aligned addresses have non-obvious and unspecified results. Note that a byte load will clear bits 8-31 of the destination register.

{T} if present, the TRANS pin will be active during the transfer. Note that T is invalid for pre-indexed addressing.

register destination of the load or the source of the store.

base must be a register. For pre-indexed addressing, base + index is the address to load from or store to. For post-indexed addressing, base is the address to load from or store to, and base + index is the value to write back to the base register.

{index} index to be added to or subtracted from the base register. If omitted, #0 is assumed. If used, it may have two forms:

#immediate value

The immediate value must lie in the range -4095 to 4095.

{-}index register{,shift}

The shift may be omitted, in which case no shifting is assumed. The allowed shift types are those listed in the section entitled *Shift types* on page 70, except that register controlled shifts are not allowed. The minus, if specified, means that the index value is to be subtracted.

An alternate form of the syntax where an expression provides the offset is:

opcode{condition}{B} register,expression{!}

The expression may be a program address (program-relative expression) or a register-relative expression. The Assembler will attempt to generate an instruction using the appropriate register as a base and an immediate offset to address the location given by evaluating the expression. The offset value must lie in the range -4095 to 4095, or the Assembler will signal an error.

{!} if present, write back will be done and the base register will assume the value of base+index, or base-index, as appropriate. Note that this is always done for post-indexed addressing.

If the contents of base are not destroyed by other instructions, the continued use of LDR (or STR) with write back will continually move the base register through memory in steps given by the index value. Note that `!` is invalid for post-indexed addressing, as write back is automatic in this case. For example:

```
STR   R1,PLACE                 ;store to address PLACE using
                               ;program-relative offset

STR   R1,[BASE,INDEX]!         ;store R1 at BASE+INDEX (both
                               ;register contents) and write
                               ;back address to BASE

STR   R1,[BASE],INDEX          ;store R1 at BASE and write
                               ;back BASE+INDEX to BASE

LDR   R1,[BASE, #16]           ;load R1 from contents of
                               ;BASE+16. Don't write back

LDR   R1,[BASE,INDEX,LSL #2]   ;load R1 from contents of
                               ;BASE+INDEX*4.
```

Note: Base may be the PC. In this case write back and post-indexing should not be used.

Block data transfer

This group of instructions is used for moving data between a number of registers and memory. LDM (LoaD Multiple registers) loads one or more registers from a block of memory, while STM (STore Multiple registers) stores one or more registers to a block of memory. The action of storing or loading may be preceded or followed by incrementing or decrementing the memory address. Write back to the base register may also be specified.

Block data transfer instruction syntax

opcode{condition}type base{!},{list}{^}

opcode	may be STM or LDM.
{condition}	any of the two-character conditional mnemonics listed in the section entitled *Conditional execution* on page 65. If omitted, AL is assumed.
type	two character mnemonic indicating one of eight instruction types. It may not be omitted. The types are FD, ED, FA, EA, IA, IB, DA and DB. The description of the eight instruction types differs depending on whether they are appended to STM or LDM:

STMDB	Decrement Before the store
STMDA	Decrement After the store
STMIB	Increment Before the store
STMIA	Increment After the store
LDMDB	Decrement Before the load
LDMDA	Decrement After the load
LDMIB	Increment Before the load
LDMIA	Increment After the load
STMFD	Push registers to a Full stack, Descending (Pre-Decrement)
STMED	Push registers to an Empty stack, Descending (Post-Decrement)
STMFA	Push registers to a Full stack, Ascending (Pre-Increment)
STMEA	Push registers to an Empty stack, Ascending (Post-Increment)
LDMFD	Pop registers from a Full stack, Descending (Post-Increment)
LDMED	Pop registers from an Empty stack, Descending (Pre-Increment)
LDMFA	Pop registers from a Full stack, Ascending (Post-Decrement)
LDMEA	Pop registers from an Empty stack, Ascending (Pre-Decrement)

- A full stack is one in which the stack pointer points to the last data item written to it.
- An empty stack is one in which the stack pointer points to the first free slot in it.

- A descending stack is one which grows from high memory addresses to low ones.
- An ascending stack is one which grows from low memory addresses to high ones.

Note that FD, ED, FA, EA are mnemonics that represent other instructions. In other words, the IA, IB, DA and DB forms of the multiple/load store instructions can be used to support all stack operations:

Stack instruction	**Equivalent instruction**
LDMED	LDMIB
LDMFD	LDMIA
LDMEA	LDMDB
LDMFA	LDMDA
STMFA	STMIB
STMEA	STMIA
STMFD	STMDB
STMED	STMDA

`base` any register. It contains the base address for the load or store. It must be specified.

`{!}` will force base to assume the value of base+4*(number of registers), or base-4*(number of registers), as appropriate.

`list` is a list of registers separated by commas, or a register range indicated by a hyphen, or a combination of both, for example:

```
{R1,R2,PC}
{R1-R10}
{R1-R9,R12,PC}
```

The braces ({ and }) around `list` are part of the assembler coding and do not indicate that the list is optional. Both the braces and the list itself must be specified.

`{^}` has different effects for STM and LDM.

`STM` causes the user mode registers to be transferred, whatever the current mode.

`LDM` if R15 is in the list of registers, only the 24 PC bits are normally loaded. Coding ^ causes the N, Z, C and V flags to be loaded as well as the PC in user mode, or all 32 bits

to be loaded in other modes. Thus, return from interrupt or return from SWI using LDM will normally have the ^ coded. For example:

```
LDMFD SP!, {R13,PC}^
```

Examples of LDM and STM are:

```
STMIA Rn,{R0,R1,R2,R3}
```

or:

STMIA Rn,{R0-R3}

This instruction saves register R0 at the address held in Rn and registers R1, R2 and R3 in the following three words of memory.

```
LDMIA Rn,{R0,R1,R2,R3}
```

or:

```
LDMIA Rn,{R0-R3}
```

Provided that the contents of Rn and the relevant memory locations have not been corrupted by another instruction, this LDMIA instruction reverses the effect of the above STMIA and recovers the contents of the four registers from memory.

{!} may be used to update the pointer Rn, so that it remains pointing to the memory location after the last update. For example:

```
STMIA Rn!,{R0,R1,R2,R3}
```

This instruction saves registers R0 to R3 as above, then increments Rn by 16 so that it points to the next word above that used to store R3.

To recover the register contents would now require:

```
LDMDB Rn!,{R0,R1,R2,R3}
```

Stacking

ARM registers can be saved to, and popped from, a stack.

Push to stack

Various forms of STM (store multiple registers) and LDM (load multiple registers) may be used to save the ARM registers on a stack. The opcodes generated for the various styles of stacking and unstacking are no different from those of the STMDB,

DA, IB, IA and LDMDB, DA, IB, IA instructions, but the syntax is different. (For information on Block data transfer instructions types, see the section entitled *Block data transfer instruction syntax* on page 82.)

There are four types of instruction which push register values on to a stack. They are:

`STMFD`	Full stack, Descending
`STMED`	Empty stack, Descending
`STMFA`	Full stack, Ascending
`STMEA`	Empty stack, Ascending

Write back is almost always required in stacking applications, but it must be coded explicitly.

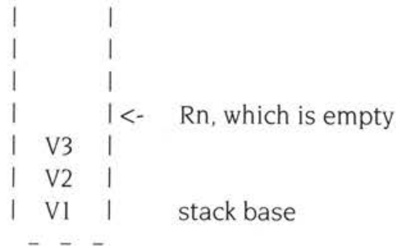
An example using STMEA is given below:

```
STMEA Rn!,{R6,R3,R7,R8}
```

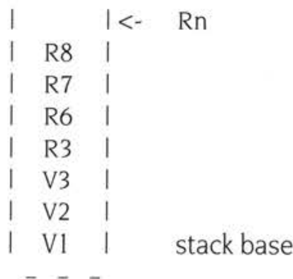
which may also be written:

```
STMEA Rn!,{R6-R8,R3}
```

Prior to the instruction, it is assumed that a stack holding three values already exists, and that Rn is ready to push more values on to it:



The stack is ascending, and the location currently pointed to is deemed to be empty. Then, after `STMEA Rn!,{R6,R3,R7,R8}` the stack grows.



Notice that register values are stacked in register order. This is always the case and cannot be altered. The lowest-numbered register always occupies the lowest memory location and registers are placed on or removed from the stack starting with the lowest-numbered register. This can be seen in the next example which shows the order of stacking following two full stack descending instructions.

An example using STMFD is given next:

```
STMFD Rn!,{R6,R3,R7,R8}
STMFD Rn!,{R0-R4}
```

```
|      |<-   Rn before 1st instruction
  - - -
|  R8  |
|  R7  |
|  R6  |
|  R3  |<-   Rn after 1st instruction
|  R4  |
|  R3  |
|  R2  |
|  R1  |
|  R0  |<-   Rn after 2nd instruction
|      |
```

Pop from stack

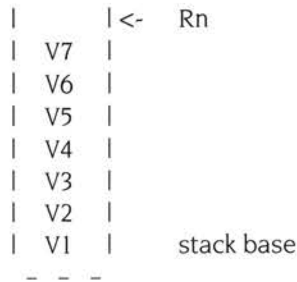
There are four types of instruction which pop register values from a stack. They are:

`LDMEA`	Empty stack, Ascending
`LDMFA`	Full stack, Ascending
`LDMED`	Empty stack, Descending
`LDMFD`	Full stack, Descending

A worked example of LDMEA is given below:

To pop all values from the following stack (set up by the earlier example `STMEA Rn!,{R6-R8,R3}`), use:

```
LDMEA Rn!,{R1-R7}
```



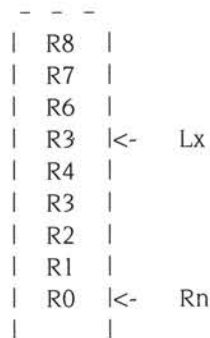
The following transfer would take place:

V1 -> R1
V2 -> R2
V3 -> R3
V4 -> R4
V5 -> R5
V6 -> R6
V7 -> R7
Rn = stack base

The following is an example of LDMFD:

To recover one set of the saved registers from the following stack (set up by the earlier example: `STMFD Rn!,{R0-R4}`), use:

```
LDMFD Rn!,{R0-R4}
```



After the pop operation, Rn will point to location Lx.

Block data transfer: special points

There are special cases to consider when using block data transfers.

When the base register is in the list of registers

- The base register may be stored and if write back is not in operation, no problem will occur.
- If write back is in operation, the STM is performed in the following order:
  1. write lowest-numbered register to memory
  2. perform the write back
  3. write other registers to memory in ascending order.

  Thus, if the base register is the lowest-numbered register in the list, its original value is stored. Otherwise, its written back value is stored.
- If the base register is loaded the pop operation will continue successfully. The entire block transfer runs on an internal copy of the base, and will not be aware that the base register has been loaded with a new value.

When R15, the PC register, is in the list of registers

- When R15 is stored, the PSR is saved as well.
- When R15 is loaded, the PSR is only included if the symbol ^ is coded following the register list. The part of the PSR included will in any case only be that which may be modified in the currently selected ARM mode. For example:

```
LDMFD SP!,{FP,PC}^
```

When the base register is R15

- When the PC is used as the base register, the PSR bits form part of the 32-bit address. Unless all flags are zero and the interrupts enabled, an address greater than &3FFFFFF will be formed. This will cause an address exception which will cause control to be transferred to the address exception trap address, as described in the section entitled *Address exception trap* on page 49.
- Write back is switched off when the PC is the base register.

Other points

- The register list is always effectively sorted into ascending order. This means that instruction sequences such as:

```
STMIA R0,{R1,R2}
LDMIA R0,{R2,R1}
```

  do not swap the contents of R1 and R2.

- In order to force the saving of the user mode registers when executing in a different mode, ^ should be coded following the register list.

  For example:

  ```
  STMFD R0,{R0-R15}^
  ```

- Registers are transferred to or from the stack starting with the lowest-numbered register (PC last) independent of stack type, so that if a data abort occurs during the instruction the PC is preserved.

Single data swap

The SWP instruction is supported by the assembler, and is introduced for the ARM3 microprocessor. It is not supported by ARM2. When executed on a machine containing an ARM2 it causes an undefined instruction trap, so you should only use SWP in code specifically intended for ARM3 machines, not in code written to run on all Archimedes computers.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. The action of the SWP instruction is a memory read and a memory write to the same address, with both transfers locked together (ie the processor cannot be interrupted until both operations have completed). This instruction is particularly useful for implementing software semaphores.

The only addressing mode supported is the swap address contained in a base register. Two other registers are specified in the instruction - the destination and source registers. The destination register is set to the value read from memory, the source register being written to memory. If the same register is specified as both source and destination, its contents are correctly swapped with memory. ARM3 has a memory cache which is updated by SWP, but data is always swapped directly with external memory.

A byte swap (SWPB) places the selected byte from memory in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros.

Using R15 is not recommended. If used as the base register, an address exception will result unless all flags are clear and interrupts are enabled. If used as source, both PC and PSR are saved, the pc being 12 bytes on from the address of the SWP instruction. If used as the destination, the PSR bits are not altered.

Single data swap instruction syntax

`SWP {condition} {B} destination, source, [base]`

`{condition}`	may be any of the two-character condition mnemonics listed in the section entitled *Conditional execution* on page 65. If omitted, AL is assumed.
`{B}`	if present then the transfers will each be of one byte. If omitted, a full word is swapped. Note that swaps of words to or from non-word-aligned addresses have non-obvious and unspecified results.
`destination`	must be a register.
`source`	must be a register.
`base`	must be a register.

Examples of SWP are:

```
SWP     R0,R1,[R10]  ;load R0 with the word pointed to by
                     ;R10, and store R1 at the same address

SWPB    R2,R3,[R10]  ;load R2 with the byte pointed to by
                     ;R10, and store bits 0 to 7 of R3 at
                     ;the same address

SWPEQ   R0,R0,[R10]  ;conditionally swap the word pointed
                     ;to by R10 with the contents of R0
```

Multiply and multiply-accumulate

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd:= Rm*Rs while the multiply accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2's complement) or unsigned integers.

Operand restrictions

Owing to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The Assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result.

The destination register (Rd) should also not be R15. R15 is protected from modification by these instructions, so the instruction will have no effect, except that it will put meaningless values in the PSR flags if the S bit is set.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

PSR flags

Setting the PSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to bit 31 of the result, Z is set if and only if the result is zero), the V flag is unaffected by the instruction (as for logical data processing instructions), and the C flag is set to a meaningless value.

Using R15 as an operand

R15 may be used as one or more of the operands, though the result will rarely be useful. When used as Rs the PC bits will be used without the PSR flags, and the PC value will be 8 bytes on from the address of the multiply instruction. When used as Rn, the PC bytes will be used along with the PSR flags and the PC will again be 8 bytes on from the address of the instruction. When used as Rm, the PC bits will be used together with the PSR flags, but the PC will be the address of the instruction plus 12 bytes in this case.

Multiply instruction syntax

```
MUL{condition} {S} Rd,Rm,Rs
MLA{condition} {S} Rd,Rm,Rs,Rn
```

`{condition}`	two-character condition mnemonic.
`{S}`	set condition codes if S present.
`Rd,Rm,Rs,Rn`	must be registers. (Rd must not be R15 and must not the same as Rm.)

For example,

```
MUL R1,R2,R3           ; R1:=R2*R3

MLAEQS R1,R2,R3,R4     ; conditionally R1:=R2*R3+R4,
                       ; setting condition codes
```

The multiply instruction may be used to synthesize higher precision multiplications, for instance to multiply two 32 bit integers and generate a 64 bit result:

```
mul64
  MOV    a1,A,LSR #16;    a1:= top half of A
  MOV    D,B,LSR #16;     D := top half of B
  BIC    A,A,a1,LSL #16;  A := bottom half of A
  BIC    B,B,D,LSL #16;   B := bottom half of B
  MUL    C,A,B            ;  low section of result
  MUL    B,a1,B           ;)     middle sections
  MUL    A,D,A            ;)        of result
  MUL    D,a1,D           ;) high section of result
  ADDS   A,B,A            ;  add middle sections
                          ;( couldn't use MLA as we need C
                          : correct)
  ADDCS  D,D,#&10000;     carry from above add
  ADDS   C,C,A,LSL #16;   C is now bottom 32 bits of product
  ADC    D,D,A,LSR #16;   D is top 32 bits
```

(A, B are registers containing the 32 bit integers; C, D are registers for the 64 bit result; al is a temporary register. A and B are overwritten during the multiply.)

Supervisor calls

These instructions are used extensively in ARM-based systems to communicate with the operating system and device drivers.

Supervisor calls instruction syntax

The syntax is: `SWI` *`expression`*

The CPU will save the contents of R15 less 4 in R14 of the SVC register set, then set the PSR register mode bits to SVC mode and set flag I. The PC will then be loaded with the value 8 causing a jump to that address to be made.

The CPU will ignore the expression, but it may be decoded by other system software and used to determine what action is to be taken. The expression may have up to 24 bits (that is, take values 0-&FFFFFF). For example:

```
SWI        &1
=          "Hello world",10,13,0
ALIGN
.......code........
.......continues...
 ..................
```

This will cause RISC OS to send the message "Hello world" to the output terminal.

The significance of the Assembler directive ALIGN is explained in the chapter entitled *Directives*.

Coprocessor instructions

The ARM can work with up to 16 external coprocessors, which (if present) will execute the instructions listed below. If the requested coprocessor is absent, these instructions will be regarded as undefined. The undefined instruction trap can then take appropriate action (for example emulating the requested instruction in software or telling the user that the program won't run in a machine without the coprocessor.)

Coprocessor number 1 is the floating point coprocessor. The floating point emulator works by trapping and emulating undefined instructions destined for coprocessor 1. The coprocessor 15 instructions are used by ARM3 as instructions to control cache operation.

The Assembler provides support for coprocessors at two levels. Firstly, it provides a set of generic coprocessor instructions, detailed below. Secondly, it provides specific floating point instructions; see the chapter entitled *Floating point instructions* for details.

All the generic coprocessor operations include a coprocessor number symbol and one or more coprocessor register symbols. These should be defined using the CP and CN directives respectively. (See the chapter entitled *Directives*.)

All coprocessor instructions are conditional. Whether they are executed depends on the ARM's condition flags, not on any coprocessor status register.

Coprocessor data operations

These instructions tell the coprocessor to perform some internal operation. ARM does not wait for the operation to complete, and no result is communicated back to ARM.

The instruction syntax assumes that the coprocessor contains up to 16 registers, and that the operation can be specified by:

- a four bit coprocessor opcode
- three coprocessor registers
- three bits of additional information.

While the interpretation of these 19 bits is purely up to the coprocessor, it is recommended that coprocessors adhere to this standard as closely as possibly.

Coprocessor data operation instruction syntax

```
CDP{condition}coproc,operation,destination,operand1,operand2{,info}
```

{condition}	two character condition mnemonic.
coproc	is the number of the coprocessor which is to handle the instruction. It must be a symbol defined via the CP directive.
operation	is the operation requested. It should be a numeric expression in the range 0-15.
destination	is the number of the coprocessor's destination register. It must be a symbol defined via the CN directive.
operand1	The number of the two coprocessor operand registers.
operand2	They must be symbols defined via the CN directive.
{,info}	is additional information to be passed to the coprocessor. It should be a numeric expression in the range 0 - 7.

Coprocessor/memory transfers

These instructions transfer one or more words of data between memory and a coprocessor.

The instruction syntax assumes that the coprocessor contains up to 16 registers and that the register(s) to be transferred can be specified by a register number and one bit of length information. Again, the interpretation of these five bits is up to the coprocessor, but it should adhere to this interpretation as closely as possible.

These instructions have pre-indexed and post-indexed forms, with the former having the option of writing back the new base register value (as with LDR and STR, this always happens for the post-indexed form).

Coprocessor/memory transfer instruction syntax

There are two types of these transfer instructions.

Pre-indexed instruction (possibly with write back)

```
opcode{condition}{L}coproc,register,[base{,#offset}]{!}
```

Post-indexed instruction (always with write back)

```
opcode{condition}{L}coproc,register,[base]{,#offset}
```

opcode	is LDC to load coprocessor register(s) from memory, or STC to store coprocessor register(s) to memory.
{condition}	is a two character condition mnemonic.
{L}	if coded, this causes a bit to be set in the instruction telling the coprocessor to do a 'long' load or store. How this is interpreted is up to the individual coprocessor.
coproc	is the coprocessor number, a symbol defined via the CP directive.
register	is the (first) coprocessor register to be transferred. It must be a coprocessor register symbol defined via the CN directive.
base	is an ARM register.
offset	is a value in the range -1020 to 1020. It must be divisible by 4.
{!}	if coded, indicates that write back is to occur to the base register.

The first word is transferred to or from the address base + offset. The second word (if it exists) then uses an address four higher, and so on. The number of words transferred is part of the coprocessor specification.

If R15 is specified as the base register, the value used is the PC without the PSR flags. The PC holds the address of the instruction plus 8 bytes.

Coprocessor/register transfers

These instructions transfer a word from an ARM register to a coprocessor, or vice versa.

The instruction syntax assumes that the coprocessor contains up to 16 registers and that the operation to be done can be specified by:

- a three bit coprocessor opcode
- two coprocessor registers
- three bits of additional information.

As usual, coprocessors should adhere as closely as possible to this convention.

Coprocessor/register transfers instruction syntax

opcode{condition}coproc,operation,armreg,operand1,operand2{,info}

opcode	should be MRC to perform the requested operation on the operands and transfer the result to the ARM register, or MCR to transfer the ARM register to the operands in the way specified by the operation and additional information.
{condition}	is a two character condition mnemonic.
coproc	is the coprocessor number, a symbol defined via the CP directive.
operation	is the operation requested. It should be a numeric expression in the range 0 to 7.
armreg	is an ARM register.
operand1 *operand2*	are coprocessor registers. They must be symbols defined via the CN directive.
{,info}	is additional information to be passed to the coprocessor. It should be a numeric expression in the range 0 to 7.

If *armreg* is R15 in a MRC instruction, bits 31, 30, 29 and 28 of the result are transferred to the N, Z, C and V flags respectively. Bits 27 to 0 of the result are ignored.

If *armreg* is R15 in a MCR instruction, both the PC and PSR are transferred.

Summary of assembler mnemonic combinations

The main AAsm and ObjAsm assembler mnemonic combinations are shown in the table below. All the root instructions may be followed by one of the condition codes listed in the section entitled *Conditional execution* on page 65; the condition code is always placed after the root instruction and before any other suffixes.

Branch group

B	Branch
BL	Branch with Link

Data processing group

ADC	Add with Carry
ADD	Add
AND	Bitwise And
BIC	Bit Clear
CMN	Compare Negated
CMP	Compare
EOR	Bitwise Exclusive Or
MOV	Move
MVN	Move Not
ORR	Bitwise Or
RSB	Reverse Subtract
RSC	Reverse Subtract with Carry
SBC	Subtract with Carry
SUB	Subtract
TEQ	Test Equivalence
TST	Test and Mask

S may follow these mnemonics
S: Set condition codes.

P may follow CMP, CMN, TST or TEQ.
P: causes CMP, CMN, TEQ and TST to set the PSR directly.

S is included by the Assembler for CMP, CMN, TEQ and TST.

Single register transfer group

LDR	Load register from memory location
STR	Store register from memory location

B or T may follow these mnemonics.
B: perform a byte transfer, not a word transfer.
T: Set the Translate bit.

Multiple register transfer group

LDM	Load multiple registers
STM	Store multiple registers

Followed by one of the suffixes shown below:

DA	Decrement after
DB	Decrement before
IA	Increment after
IB	Increment before
EA	Empty stack, ascending

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

Single data swap (from ARM3)

SWP	Swap register and memory contents.
B	may follow this mnemonic, implying byte transfer.

Multiplies

MUL	Multiply
MLA	Multiply and accumulate

S may follow these mnemonics
S: Set condition codes

Supervisor call

SWI	Software interrupt

Coprocessor data operations

CDP	Perform internal coprocessor operation

Coprocessor/memory transfer

LDC	Load coprocessor register from memory location
STC	Store coprocessor register to memory location

L may follow these mnemonics
L: Perform long transfer

Coprocessor/register transfers

MCR	Move ARM register to coprocessor register
MRC	Move coprocessor register to ARM register

Further instructions

The above completes the description of all the basic ARM instructions. However, the Assembler understands a number of other instructions, which it translates into appropriate basic ARM instructions.

Extended range immediate constants

In the case of an instruction such as

```
MOV    R0,    #VALUE
```

the Assembler will evaluate the expression and produce a CPU instruction to load the value into the destination register. This may not in fact be the machine level instruction known as MOV, but the programmer need not be aware that an alternative instruction has been substituted. A common example is

```
MOV    Rn,    #-1
```

which the CPU cannot handle directly (as -1 is not a valid immediate constant). The Assembler will accept this syntax, but will convert it and generate object code for

```
MVN    Rn,    #0
```

which results in Rn containing -1. Such conversions also takes place between the following pairs of instructions.

- BIC/AND
- ADD/SUB
- ADC/SBC
- CMP/CMN

The ADR instruction

Syntax: `ADR`*`{condition} register, expression`*

This produces an address in a register. ARM does not have an explicit 'calculate effective address' instruction, as this can generally be done using ADD, SUB, MOV or MVN. To ease the construction of such instructions, the Assembler provides an ADR instruction.

The expression may be register-relative, program-relative or numeric.

register-relative	`ADD register, register2, #constant`
	or
	`SUB register, register2, #constant`
	will be produced, where `register2` is the register that the expression is relative to.

program-relative	`ADD register, PC, #constant`
	or
	`SUB register, PC, #constant`

will be produced.

numeric — `MOV register, #constant`

or

`MVN register, #constant`

will be produced.

In all three cases, an error will be generated if the immediate constant required is out of range.

If the program has a fixed origin (that is, if the ORG directive has been used), the distinction between program-relative and numeric values disappears. In this case, the Assembler will first try to treat such a value as program-relative. If this fails, it will try to treat it as numeric. An error will only be generated if both attempts fail.

ADR {condition} L

This form of ADR is provided by ADRL and allows a wider collection of effective addresses to be produced. ADRL can be used in the same way as ADR, except that the allowed range of constants is any constant specified as an even rotation of a value less than &10000. Again program-relative, register relative and numeric forms exist. The result produced will always be two instructions, even if it could have been done in one. An error will be generated if the necessary immediate constants cannot be produced.

Literals

Literals are intended to enable the programmer to load immediate values into a register which might be out of range as MOV/MVN arguments.

Syntax: `LDR` *`register,=expression`*

The Assembler will then take certain actions. It will:

- if possible, replace the instruction with a MOV or MVN,
- otherwise, generate a program-relative LDR and if no such literal already exists within the addressable range, place the literal in the next literal pool.

Program-relative expressions and imported symbols are also valid literals in ObjAsm. See the section entitled LTORG on page 126 for further information.

Floating point instructions

The Assembler recognises a standard set of floating point instructions and translates them into the appropriate coprocessor instructions. See the next chapter entitled *Floating point instructions* for details.

8 Floating point instructions

The Acorn RISC machine has a general coprocessor interface. The first coprocessor available is one which performs floating point calculations to the IEEE standard. To ensure that programs using floating point arithmetic remain compatible with all Archimedes machines, a standard ARM floating point instruction set has been defined. This can be implemented invisibly to the customer program by one of several systems offering various speed performances at various costs. The current 'bundled' floating point system is the software only floating point emulator module. Floating point instructions may be incorporated into any assembler text, provided they are called from user mode. These instructions are recognised by the Assembler and converted into the correct coprocessor instructions.

Programmer's model

The ARM IEEE floating point system has eight 'high precision' *floating point registers*, F0 to F7. The format in which numbers are stored in these registers is not specified. Floating point formats only become visible when a number is transferred to memory, using one of the formats described below.

There is also a *floating point status register* (FPSR) which, like the ARM's combined PC and PSR, holds all the necessary status and control information that an application is intended to be able to access. It holds *flags* which indicate various error conditions, such as overflow and division by zero. Each flag has a corresponding *trap enable bit*, which can be used to enable or disable a 'trap' associated with the error condition. Bits in the FPSR allow a client to distinguish between different implementations of the floating point system.

There may also be a *floating point control register* (FPCR); this is used to hold status and control information that an application is not intended to access. For example, there are privileged instructions to turn the floating point system on and off, to permit efficient context changes. Typically, hardware based systems have an FPCR, whereas software based ones do not.

Available systems

Floating point systems may be built from software only, hardware only, or some combination of software and hardware. The following terminology will be used to differentiate between the various ARM floating point systems already in use or planned:

System name	System components
Old FPE	Versions of the floating point emulator up to (but not including) 4.00
FPPC	Floating Point Protocol Convertor (interface chip between ARM and WE32206), WE32206 (AT&T Math Acceleration Unit chip), and support code
New FPE	Versions of the floating point emulator from 4.00 onwards
FPA	ARM Floating Point Accelerator chip, and support code

The results look the same to the programmer. However, if clients are aware of which system is in use, they may be able to extract better performance. For example, compilers can be tuned to generate bunched FP instructions for the FPE and dispersed FP instructions for the FPA, which will improve overall performance.

Precision

All basic floating point instructions operate as though the result were computed to infinite precision and then rounded to the length, and in the way, specified by the instruction. The rounding is selectable from:

- Round to nearest
- Round to +infinity (P)
- Round to –infinity (M)
- Round to zero (Z).

The default is 'round to nearest'; in the event of a tie, this rounds to 'nearest even'. If any of the others are required they must be given in the instruction.

The working precision of the system is 80 bits, comprising a 64 bit mantissa, a 15 bit exponent and a sign bit. Specific instructions that work only with single precision operands may provide higher performance in some implementations, particularly the fully software based ones.

Floating point number formats

Like the ARM instructions, the floating point data processing operations refer to registers rather than memory locations. Values may be stored into ARM memory in one of five formats (only four of which are visible at any one time, since P and EP are mutually exclusive):

IEEE Single Precision (S)

Figure 8.1 Single precision format

- If the exponent is 0 and the fraction is 0, the number represented is ±0.
- If the exponent is 0 and the fraction is non-zero, the number represented is $\pm 0.\mathit{fraction} \times 2^{-126}$
- If the exponent is in the range 1 to 254, the number represented is $\pm 1.\mathit{fraction} \times 2^{\mathit{exponent} - 127}$
- If the exponent is 255 and the fraction is 0, the number represented is ±∞.
- If the exponent is 255 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

IEEE Double Precision (D)

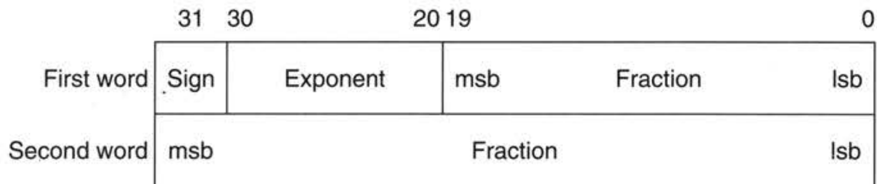


Figure 8.2 Double precision format

- If the exponent is 0 and the fraction is 0, the number represented is ±0.
- If the exponent is 0 and the fraction is non-zero, the number represented is $\pm 0.\mathit{fraction} \times 2^{-1022}$
- If the exponent is in the range 1 to 2046, the number represented is $\pm 1.\mathit{fraction} \times 2^{\mathit{exponent} - 1023}$
- If the exponent is 2047 and the fraction is 0, the number represented is ±∞.
- If the exponent is 2047 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

Double Extended Precision (E)

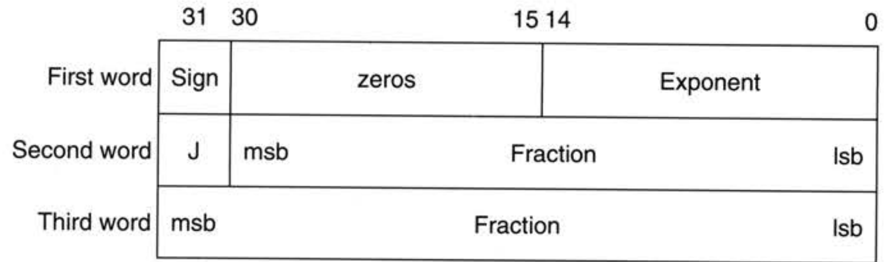


Figure 8.3 Double extended precision format

- If the exponent is 0, J is 0, and the fraction is 0, the number represented is ±0.
- If the exponent is 0, J is 0, and the fraction is non-zero, the number represented is $\pm 0.fraction \times 2^{-16382}$
- If the exponent is in the range 0 to 32766, J is 1, and the fraction is non-zero, the number represented is $\pm 1.fraction \times 2^{exponent - 16383}$
- If the exponent is 32767, J is 0, and the fraction is 0, the number represented is ±∞.
- If the exponent is 32767 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

Other values are illegal and shall not be used (ie the exponent is in the range 1 to 32766 and J is 0; or the exponent is 32767, J is 1, and the fraction is 0).

The FPPC system stores the sign bit in bit 15 of the first word, rather than in bit 31.

Storing a floating point register in 'E' format is guaranteed to maintain precision when loaded back by the same floating point system in this format. Note that in the past the layout of E format has varied between floating point systems, so software should not have been written to depend on it being readable by other floating point systems. For example, no software should have been written which saves E format data to disc, potentially loaded into another system. In particular, E format in the FPPC system varies from all other systems in its positioning of the sign bit. However, for the FPA and the new FPE, the E format is now defined to be a particular form of IEEE Double Extended Precision and will not vary in future.

Packed Decimal (P)

	31							0
First word	Sign	e3	e2	e1	e0	d18	d17	d16
Second word	d15	d14	d13	d12	d11	d10	d9	d8
Third word	d7	d6	d5	d4	d3	d2	d1	d0

Figure 8.4 *Packed decimal format*

The sign nibble contains both the significand's sign (top bit) and the exponent's sign (next bit); the other two bits are zero.

d18 is the most significant digit of the significand, and e3 of the exponent. The significand has an assumed decimal point between d18 and d17, and is normalised so that for a normal number $1 \leq d18 \leq 9$. The guaranteed ranges for d and e are 17 and 3 digits respectively; d0, d1 and e3 may always be zero in a particular system. A single precision number has 9 digits of significand and a maximum exponent of 53; a double precision number has 17 digits in the significand and a maximum exponent of 340.

The result is undefined if any of the packed digits is hexadecimal A - F, save for a representation of ±∞ or a NaN (see below).

- If the exponent's sign is 0, the exponent is 0, and the significand is 0, the number represented is ±0.

  Zero will always be output as +0, but either +0 or –0 may be input.
- If the exponent is in the range 0 to 9999 and the significand is in the range 1 to 9.999999999999999999, the number represented is $\pm d \times 10^{\pm e}$.
- If the exponent is &FFFF (ie all the bits in e3 - e0 are set) and the significand is 0, the number represented is ±∞.
- If the exponent is &FFFF and d0 - d17 are non-zero, a NaN (not-a-number) is represented. If the most significant bit of d18 is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

All other combinations are undefined.

Expanded Packed Decimal (EP)

	31							0
First word	Sign	e6	e5	e4	e3	e2	e1	e0
Second word	d23	d22	d21	d20	d19	d18	d17	d16
Third word	d15	d14	d13	d12	d11	d10	d9	d8
Fourth word	d7	d6	d5	d4	d3	d2	d1	d0

Figure 8.5 *Expanded packed decimal format*

The sign nibble contains both the significand's sign (top bit) and the exponent's sign (next bit); the other two bits are zero.

d23 is the most significant digit of the significand, and e6 of the exponent. The significand has an assumed decimal point between d23 and d22, and is normalised so that for a normal number $1 \leq d23 \leq 9$. The guaranteed ranges for d and e are 21 and 4 digits respectively; d0, d1, d2, e4, e5 and e6 may always be zero in a particular system. A single precision number has 9 digits of significand and a maximum exponent of 53; a double precision number has 17 digits in the significand and a maximum exponent of 340.

The result is undefined if any of the packed digits is hexadecimal A - F, save for a representation of ±∞ or a NaN (see below).

- If the exponent's sign is 0, the exponent is 0, and the significand is 0, the number represented is ±0.

  Zero will always be output as +0, but either +0 or –0 may be input.
- If the exponent is in the range 0 to 9999999 and the significand is in the range 1 to 9.99999999999999999999999, the number represented is $\pm d \times 10^{\pm e}$
- If the exponent is &FFFFFFF (ie all the bits in e6 - e0 are set) and the significand is 0, the number represented is ±∞.
- If the exponent is &FFFFFFF and d0 - d22 are non-zero, a NaN (not-a-number) is represented. If the most significant bit of d23 is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

All other combinations are undefined.

This format is not available in the old FPE or the FPPC. You should only use it if you can guarantee that the floating point system you are using supports it.

Floating point status register

There is a floating point status register (FPSR) which, like ARM's combined PC and PSR, has all the necessary status for the floating point system. The FPSR contains the IEEE flags but not the result flags – these are only available after floating point compare operations.

The FPSR consists of a system ID byte, an exception trap enable byte, a system control byte and a cumulative exception flags byte.

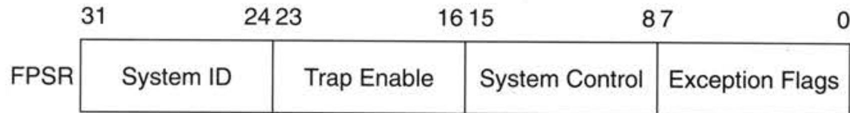


Figure 8.6 Floating point status register byte usage

System ID byte

The System ID byte allows a user or operating system to distinguish which floating point system is in use. The top bit (bit 31 of the FPSR) is set for **hardware** (ie fast) systems, and clear for **software** (ie slow) systems. Note that the System ID is read-only.

The following SysId's are currently defined:

System	System ID
Old FPE	&00
FPPC	&80
New FPE	&01
FPA	&81

Exception Trap Enable Byte

Each bit of the exception trap enable byte corresponds to one type of floating point exception, which are described in the section entitled *Cumulative Exception Flags Byte* on page 109.

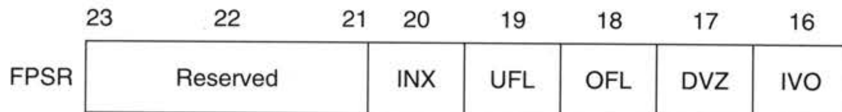


Figure 8.7 Exception trap enable byte

If a bit in the cumulative exception flags byte is set as a result of executing a floating point instruction, and the corresponding bit is also set in the exception trap enable byte, then that exception trap will be taken.

Currently, the reserved bits shall be written as zeros and will return 0 when read.

System Control Byte

These control bits determine which features of the floating point system are in use.

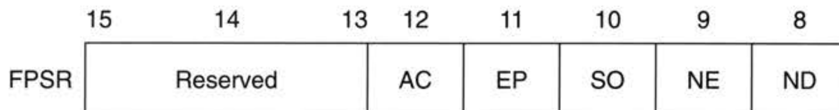


Figure 8.8 *System control byte*

By placing these control bits in the FPSR, their state will be preserved across context switches, allowing different processes to use different features if necessary. The following five control bits are defined for the FPA system and the new FPE:

ND	No Denormalised Numbers
NE	NaN Exception
SO	Select Synchronous Operation of FPA
EP	Use Expanded Packed Decimal Format
AC	Use Alternative definition for C flag on compare operations

The old FPE and the FPPC system behave as if all these bits are clear.

Currently, the reserved bits shall be written as zeros and will return 0 when read. Note that all bits (including bits 8 - 12) are reserved on FPPC and early FPE systems.

ND – No Denormalised Numbers Bit

If this bit is set, then the software will force all denormalised numbers to zero to prevent lengthy execution times when dealing with denormalised numbers. (Also known as abrupt underflow or flush to zero.) This mode is not IEEE compatible but may be required by some programs for performance reasons.

If this bit is clear, then denormalised numbers will be handled in the normal IEEE-conformant way.

NE – NaN Exception Bit

If this bit is set, then an attempt to store a signalling NaN that involves a change of format will cause an exception (for full IEEE compatibility).

If this bit is clear, then an attempt to store a signalling NaN that involves a change of format will not cause an exception (for compatibility with programs designed to work with the old FPE).

SO – Select Synchronous Operation of FPA

If this bit is set, then all floating point instructions will execute synchronously and ARM will be made to busy-wait until the instruction has completed. This will allow the precise address of an instruction causing an exception to be reported, but at the expense of increased execution time.

If this bit is clear, then that class of floating point instructions that can execute asynchronously to ARM will do so. Exceptions that occur as a result of these instructions may be raised some time after the instruction has started, by which time the ARM may have executed a number of instructions following the one that has failed. In such cases the address of the instruction that caused the exception will be imprecise.

The state of this bit is ignored by software-only implementations, which always operate synchronously.

EP – Use Expanded Packed Decimal Format

If this bit is set, then the expanded (four word) format will be used for Packed Decimal numbers. Use of this expanded format allows conversion from extended precision to packed decimal and back again to be carried out without loss of accuracy.

If this bit is clear, then the standard (three word) format is used for Packed Decimal numbers.

AC – Use Alternative definition for C flag on compare operations

If this bit is set, the ARM C flag, after a compare, is interpreted as 'Greater Than or Equal or Unordered'. This interpretation allows more of the IEEE predicates to be tested by means of single ARM conditional instructions than is possible using the original interpretation of the C flag (as shown below).

If this bit is clear, the ARM C flag, after a compare, is interpreted as 'Greater Than or Equal'.

Cumulative Exception Flags Byte

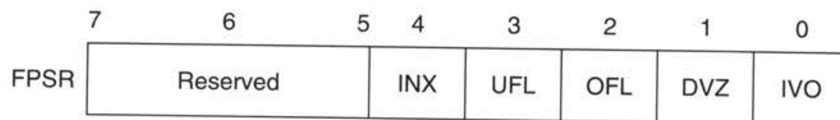


Figure 8.9 Cumulative exception flags byte

Whenever an exception condition arises, the appropriate cumulative exception flag in bits 0 to 4 will be set to 1. If the relevant trap enable bit is set, then an exception is also delivered to the user's program in a manner specific to the operating

system. (Note that in the case of underflow, the state of the trap enable bit determines under which conditions the underflow flag will be set.) These flags can only be cleared by a WFS instruction.

Currently, the reserved bits shall be written as zeros and will return 0 when read.

IVO – invalid operation

The IVO flag is set when an operand is invalid for the operation to be performed. Invalid operations are:

- Any operation on a trapping NaN (not-a-number)
- Magnitude subtraction of infinities, eg $+\infty + -\infty$
- Multiplication of 0 by $\pm\infty$
- Division of 0/0 or ∞/∞
- x REM y where $x = \infty$ or $y = 0$
  (REM is the 'remainder after floating point division' operator.)
- Square root of any number < 0 (but $\sqrt{(-0)} = -0$)
- Conversion to integer or decimal when overflow, ∞ or a NaN operand make it impossible

  If overflow makes a conversion to integer impossible, then the largest positive or negative integer is produced (depending on the sign of the operand) and IVO is signalled
- Comparison with exceptions of Unordered operands
- ACS, ASN when argument's absolute value is > 1
- SIN, COS, TAN when argument is $\pm\infty$
- LOG, LGN when argument is ≤ 0
- POW when first operand is < 0 and second operand is not an integer, or first operand is 0 and second operand is ≤ 0
- RPW when first operand is not an integer and second operand is < 0, or first operand is ≤ 0 and second operand is 0.

DVZ – division by zero

The DVZ flag is set if the divisor is zero and the dividend a finite, non-zero number. A correctly signed infinity is returned if the trap is disabled.

The flag is also set for LOG(0) and for LGN(0). Negative infinity is returned if the trap is disabled.

OFL – overflow

The OFL flag is set whenever the destination format's largest number is exceeded in magnitude by what the rounded result would have been were the exponent range unbounded. As overflow is detected after rounding a result, whether overflow occurs or not after some operations depends on the rounding mode.

If the trap is disabled either a correctly signed infinity is returned, or the format's largest finite number. This depends on the rounding mode and floating point system used.

UFL – underflow

Two correlated events contribute to underflow:

- *Tininess* – the creation of a tiny non-zero result smaller in magnitude than the format's smallest normalised number.
- *Loss of accuracy* – a loss of accuracy due to denormalisation that **may** be greater than would be caused by rounding alone.

The UFL flag is set in different ways depending on the value of the UFL trap enable bit. If the trap is enabled, then the UFL flag is set when tininess is detected regardless of loss of accuracy. If the trap is disabled, then the UFL flag is set when both tininess and loss of accuracy are detected (in which case the INX flag is also set); otherwise a correctly signed zero is returned.

As underflow is detected after rounding a result, whether underflow occurs or not after some operations depends on the rounding mode.

INX – inexact

The INX flag is set if the rounded result of an operation is not exact (different from the value computable with infinite precision), or overflow has occurred while the OFL trap was disabled, or underflow has occurred while the UFL trap was disabled. OFL or UFL traps take precedence over INX.

The INX flag is also set when computing SIN or COS, with the exceptions of SIN(0) and COS(1).

The old FPE and the FPPC system may differ in their handling of the INX flag. Because of this inconsistency we recommend that you do not enable the INX trap.

Floating Point Control Register

The Floating Point Control register (FPCR) may only be present in some implementations: it is there to control the hardware in an implementation specific manner, for example to disable the floating point system. The user mode of the ARM is not permitted to use this register (since the right is reserved to alter it between implementations) and the WFC and RFC instructions will trap if tried in user mode.

You are unlikely to need to access the FPCR; this information is principally given for completeness.

The FPPC system

The FPCR bit allocation in the FPPC system is as shown below:

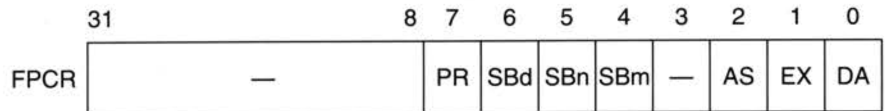


Figure 8.10 FPCR *bit allocation in the* FPPC *system*

Bit		**Meaning**
31-8		Reserved – always read as zero
7	PR	Last RMF instruction produced a partial remainder
6	SBd	Use Supervisor Register Bank 'd'
5	SBn	Use Supervisor Register Bank 'n'
4	SBm	Use Supervisor Register Bank 'm'
3		Reserved – always read as zero
2	AS	Last WE32206 exception was asynchronous
1	EX	Floating point exception has occurred
0	DA	Disable

Reserved bits are ignored during write operations (but should be zero for future compatibility.) The reserved bits will return zero when read.

The FPA system

In the FPA, the FPCR will also be used to return status information required by the support code when an instruction is bounced. You should not alter the register unless you really know what you're doing. Note that the register will be read sensitive; **even reading the register may change its value, with disastrous consequences.**

The FPCR bit allocation in the FPA system is **provisionally** as follows:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FPCR	RU	—		IE	MO	EO	—		OP				—	S1		

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(cont'd)	OP	DS			SB	AB	RE	EN	PR	RM		OP	S2			

Figure 8.11 FPCR *bit allocation in the* FPA *system*

Bit		**Meaning**
31	RU	Rounded Up Bit
30		Reserved
29		Reserved
28	IE	Inexact bit
27	MO	Mantissa overflow
26	EO	Exponent overflow
25, 24		Reserved
23-20	OP	AU operation code
19	PR	AU precision
18-16	S1	AU source register 1
15	OP	AU operation code
14-12	DS	AU destination register
11	SB	Synchronous bounce: decode (R14) to get opcode
10	AB	Asynchronous bounce: opcode supplied in rest of word
9	RE	Rounding Exception: Asynchronous bounce occurred during rounding stage and destination register was written
8	EN	Enable FPA (default is **off**)
7	PR	AU precision
6, 5	RM	AU rounding mode
4	OP	AU operation code
3-0	S2	AU source register 2 (bit 3 set denotes a constant)

Note that the SB and AB bits are cleared on a read of the FPCR. Only the EN bit is writable. All other bits shall be set to zero on a write.

Assembler directives and syntax

The precision letter determines the format used to store the number in memory, as follows:

Letter	Precision	Memory usage
S	Single	1 word
D	Double	2 words
E	Extended	3 words
P	Packed BCD	3 words
EP	Extended Packed BCD	4 words

For details of these formats see the section entitled *Floating point number formats* on page 102.

Floating point number input

A floating point number recognised by the assemblers consists of an optional sign, followed by an optional mantissa part followed by an optional exponent part. One or other of the mantissa part and the exponent part must be present. The mantissa part consists of a sequence of zero or more decimal digits, followed by an optional decimal point followed by a sequence of zero or more decimal digits. If present, the mantissa must contain a non-zero number of digits overall. The exponent part begins with 'e' or 'E', followed by an optional sign, followed by a sequence of one or more decimal digits.

Examples are:

```
1
0.2
5E9
E-2
-.7
+31.415926539E-1
```

The value generated represents the mantissa multiplied by ten to the power of the exponent, where the mantissa is taken to be one if missing, and the exponent is taken to be zero if missing. All reading is done to double precision, and is then rounded to single precision as required. The required precision is determined by the context as shown in the sections *Floating point store loading directives* and *Floating point literals* below.

NOFP directive

If you know that your code should not use floating point instructions and want to ensure that you don't accidentally include them, you can use the NOFP directive. It must occur before any floating point instructions or directives.

Syntax: `NOFP`

Floating point register equating: FN

The directive FN is used to assign a floating point register number 0-7 to a symbol.

Syntax: *`label`* `FN` *`numeric expression`*

Floating point register numbers are taken to be constants when included in arbitrary expression, but only floating point register names are valid when a floating point register is required.

Floating point store loading directives

Directives DCFS and DCFD are provided to load store with respectively single and double precision floating point numbers. Single precision floating point numbers occupy one word of store, double precision floating point numbers occupy two words, but are not constrained to be double word aligned.

Syntax: *`label`* `DCFx` *`floating point number{,floating point number}`*

where the syntax of floating point numbers is defined in the section *Floating point number input* above.

`?label` will have the value of the number of bytes of code generated by its defining line in a way analogous to DCD.

The instruction set

Floating point coprocessor data transfer

`op{condition}prec         Fd,addr`

`op`	is LDF for load, STF for store
`condition`	is one of the usual ARM conditions
`prec`	is one of the usual floating point precisions
`addr`	is `[Rn]{,#offset}` or `[Rn,#offset]{!}` (`{!}` if present indicates that write-back is to take place.)
`Fd`	is a floating point register symbol (defined via the FN directive).

Load (LDF) or store (STF) the high precision value from or to memory, using one of the five memory formats. On store, the value is rounded using the 'round to nearest' rounding method to the destination precision, or is precise if the destination has sufficient precision. Thus other rounding methods may be used by having previously applied some suitable floating point data operation; this does not compromise the requirement of 'rounding once only', since the store operation introduces no additional rounding error.

The offset is in words from the address given by the ARM base register, and is in the range −1020 to +1020. In pre-indexed mode you must explicitly specify write-back to add the offset to the base register; but in post-indexed mode the assembler forces write-back for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if write-back will take place.

Floating point literals

LDFS and LDFD can be given literal values instead of a register relative address, and the Assembler will automatically place the required value in the next available literal pool. In the case of LDFS a single precision value is placed, in the case of LDFD a double precision value is placed. Because the allowed offset range within a LDFS or LDFD instruction is less than that for a LDR instruction (−1020 to +1020 instead of −4095 to +4095), it may be necessary to code LTORG directives more frequently if floating point literals are being used than would otherwise be necessary.

Syntax: `LDFx Fn, = ` *`floating point number`*

where the syntax of floating point numbers is defined in the section entitled *Floating point number input* on page 114.

Floating point coprocessor multiple data transfer

The LFM and SFM multiple data transfer instructions are supported by the assemblers, but are not provided by the old FPE or the FPPC system. Executing these instructions on such systems will cause undefined instruction traps, so you should only use these instructions in software intended for machines you are confident are using the new FPE or the FPA system.

The LFM and SFM instructions allow between 1 and 4 floating point registers to be transferred from or to memory in a single operation; such a transfer otherwise requires several LDF or STF operations. The multiple transfers are therefore useful for efficient stacking on procedure entry/exit and context switching. These new instructions are the preferred way to preserve exactly register contents within a program.

The values transferred to memory by SFM occupy three words for each register, but the data format used is not defined, and may vary between floating point systems. The only legal operation that can be performed on this data is to load it back into floating point registers using the LFM instruction. The data stored in memory by an SFM instruction should not be used or modified by any user process.

The registers transferred by a LFM or SFM instruction are specified by a base floating point register and the number of registers to be transferred. This means that a register set transferred has to have adjacent register numbers, unlike the unconstrained set of ARM registers that can be loaded or saved using LDM and STM. Floating point registers are transferred in ascending order, register numbers wrapping round from 7 to 0: eg transferring 3registers with F6 as the base register results in registers F6, F7 then F0 being transferred.

The assembler supports two alternative forms of syntax, intended for general use or just stack manipulation:

op{condition} Fd,count,addr

op{condition}stacktype Fd,count,[Rn]*{!}*

op	is `LFM` for load, `SFM` for store.
condition	is one of the usual ARM conditions.
Fd	is the base floating point register, specified as a floating point register symbol (defined via the FN directive).
count	is an integer from 1 to 4 specifying the number of registers to be transferred.
addr	is `[Rn]{,#`*offset*`}` or `[Rn,#`*offset*`]{!}` (`{!}` if present indicates that write-back is to take place).
stacktype	is FD or EA, standing for Full Descending or Empty Ascending, the meanings as for LDM and STM.

The offset (only relevant for the first, general, syntax above) is in words from the address given by the ARM base register, and is in the range –1020 to +1020. In pre-indexed mode you must explicitly specify write-back to add the offset to the base register; but in post-indexed mode the assembler forces write-back for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if write-back will take place.

Examples:

```
SFMNE   F6,4,[R0]           ;if NE is true, transfer F6, F7,
                            ;F0 and F1 to the address
                            ;contained in R0

LFMFD   F4,2,[R13]!         ;load F4 and F5 from FD stack -
LFM     F4,2,[R13],#24      ;equivalent to same instruction
                            ;in general syntax
```

Floating point coprocessor register transfer

```
FLT{condition}prec{round}     Fn,Rd
FLT{condition}prec{round}     Fn,#value
FIX{condition}{round}         Rd,Fn
WFS{condition}                Rd
RFS{condition}                Rd
WFC{condition}                Rd
RFC{condition}                Rd
```

{round}	is the optional rounding mode: P, M or Z; see below.
Rd	is an ARM register symbol.
Fn	is a floating point register symbol.

The value may be of the following: 0, 1, 2, 3, 4, 5, 10, 0.5. Note that these values must be written precisely as shown above, for instance '0.5' is correct but '.5' is not.

FLT	Integer to Floating Point	Fn := Rd	
FIX	Floating point to integer	Rd := Fm	
WFS	Write Floating Point Status	FPSR := Rd	
RFS	Read Floating Point Status	Rd := FPSR	
WFC	Write Floating Point Control	FPC := R	Supervisor Only
RFC	Read Floating Point Control	Rd := FPC	Supervisor Only

The rounding modes are

Mode	**Letter**
Nearest	(no letter required)
Plus infinity	P
Minus infinity	M
Zero	Z

Floating point coprocessor data operations

The formats of these instructions are:

```
binop{condition}prec{round}   Fd, Fn, Fm
binop{condition}prec{round}   Fd, Fn, #value
unop{condition}prec{round}    Fd, Fm
unop{condition}prec{round}    Fd, #value
```

binop	is one of the binary operations listed below
unop	is one of the unary operations listed below
Fd	is the FPU destination register
Fn	is the FPU source register (binops only)
Fm	is the FPU source register
#value	is a constant, as an alternative to Fm. It must be 0, 1, 2, 3, 4, 5, 10 or 0.5, as above.

The binops are:

ADF	Add	Fd := Fn + Fm
MUF	Multiply	Fd := Fn × Fm
SUF	Sub	Fd := Fn – Fm
RSF	Reverse Subtract	Fd := Fm – Fn
DVF	Divide.	Fd := Fn/Fm
RDF	Reverse Divide	Fd := Fm/Fn
POW	Power	Fd := Fn to the power of Fm
RPW	Reverse Power	Fd := Fm to the power of Fn
RMF	Remainder	Fd := remainder of Fn / Fm (Fd := Fn – integer value of (Fn/Fm)*Fm)
FML	Fast Multiply	Fd := Fn × Fm
FDV	Fast Divide	Fd := Fn / Fm
FRD	Fast Reverse Divide	Fd := Fm / Fn
POL	Polar angle	Fd := polar angle of Fn, Fm

The unops are:

MVF	Move	Fd := Fm
MNF	Move Negated	Fd := –Fm
ABS	Absolute value	Fd := ABS (Fm)
RND	Round to integral value	Fd := integer value of Fm
SQT	Square root	Fd := square root of Fm
LOG	Logarithm to base 10	Fd := log Fm
LGN	Logarithm to base e	Fd := ln Fm
EXP	Exponent	Fd := e to the power of Fm
SIN	Sine	Fd := sine of Fm

COS	Cosine	Fd := cosine of Fm
TAN	Tangent	Fd := tangent of Fm
ASN	Arc Sine	Fd := arcsine of Fm
ACS	Arc Cosine	Fd := arccosine of Fm
ATN	Arc Tangent	Fd := arctangent of Fm
URD	Unnormalised Round	Fd := integer value of Fm (may be abnormal)
NRM	Normalise	Fd := normalised form of Fm

Note that wherever Fm is mentioned, one of the floating point constants 0, 1, 2, 3, 4, 5, 10, or 0.5 can be used instead.

FML, FRD and FDV are only defined to work with single precision operands. These 'fast' instructions are likely to be faster than the equivalent MUF, DVF and RDF instructions, but this is not necessarily so for any particular implementation.

Rounding is done only at the last stage of a SIN, COS etc – the calculations to compute the value are done with 'round to nearest' using the full working precision.

The URD and NRM operations are only supported by the FPA and the new FPE.

Floating point coprocessor status transfer

op{condition}prec{round} Fm, Fn

op is one of the following:

CMF	Compare floating	compare Fn with Fm
CNF	Compare negated floating	compare Fn with –Fm
CMFE	Compare floating with exception	compare Fn with Fm
CNFE	Compare negated floating with exception	compare Fn with –Fm

{condition} an ARM condition.

prec a precision letter

{round} an optional rounding mode: P, M or Z

Fm A floating point register symbol.

Fn A floating point register symbol.

Compares are provided with and without the exception that could arise if the numbers are unordered (ie one or both of them is not-a-number). To comply with IEEE 754, the CMF instruction should be used to test for equality (ie when a BEQ or BNE is used afterwards) or to test for unorderedness (in the V flag). The CMFE instruction should be used for all other tests (BGT, BGE, BLT, BLE afterwards).

When the AC bit in the FPSR is clear, the ARM flags N, Z, C, V refer to the following after compares:

N	Less than	ie Fn less than Fm (or –Fm)
Z	Equal	
C	Greater than or equal	ie Fn greater than or equal to Fm (or –Fm)
V	Unordered	

Note that when two numbers are not equal, N and C are not necessarily opposites. If the result is unordered they will both be clear.

When the AC bit in the FPSR is set, the ARM flags N, Z, C, V refer to the following after compares:

N	Less than
Z	Equal
C	Greater than or equal or unordered
V	Unordered

In this case, N and C are necessarily opposites.

Finding out more...

Further details of the floating point instructions (such as the format of the bitfields within the instruction) can be found in the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.

9 Directives

This chapter describes the various directives available in the Archimedes' Assemblers.

Number equating directives: */EQU

Numeric values are assigned to symbols by the * or EQU directive. Program-relative values can also be assigned in this way.

Syntax: *label * numeric or program-relative-expression*

For example:

```
LINEFEED     *      &0A                  ;equate LINEFEED as &0A
MASK         EQU    &FF00FFF             ;create a mask
FRAMESIZE    *      4*((framebase+3)/4)  ;calculate FRAMESIZE
                                         ;from framebase
LABEL        SWI    16
LABEL2       *      LABEL-4
```

Register equating: RN

The directive RN is used to assign a register number 0-15 to a symbol.

Syntax: *label* RN *numeric expression*

For example:

```
Reg2         RN     2
TempStore    RN     3
SL           RN     4
sl           RN     SL
```

A register name is taken to be a constant when included in an arbitrary expression, but only register names are valid where a register is required.

All register names must be defined. Many examples in this manual assume that PC, R0, R1, R2, and so on are valid register names. To make this the case it is first necessary to use the RN directive at the beginning of the source code, thus:

```
R0      RN  0
R1      RN  1
R2      RN  2
:       :   :
:       :   :
:       :   :
R15     RN  15
PC      RN  15
```

It is suggested that a separate source file of standard register definitions be produced and included in any assembly using the GET directive.

Coprocessor equating: CP

The directive CP is used to assign a coprocessor number 0-15 to a symbol.

Syntax: `label` CP `numeric expression`

Like register names, coprocessor names are taken to be constants when included in arbitrary expression, but only coprocessor names are valid where a coprocessor is required.

Coprocessor register equating: CN

The directive CN is used to assign a coprocessor register number 0-15 to a symbol.

Syntax: `label` CN `numeric expression`

Like register and coprocessor names, coprocessor register names are taken to be constants when included in arbitrary expressions, but only coprocessor register names are valid where a coprocessor register is required.

There is also a directive FN, for defining floating point registers. See the chapter entitled *Floating point instructions* for details.

Store-loading

This places data in store at the current instruction location and advances the instruction location pointer.

The line takes the general form `{label} directive expression list`

directive Directive is either:

DCD or &	which defines one or more words (aligned)
DCW	which defines one or more half-words (16-bit numbers)
DCB or =	which defines one or more bytes.

Note that in AAsm, DCD can only take numeric expressions but that in ObjAsm, DCD can take a program-relative or external expression, even when the code does not have an absolute origin.

expression list list of one or more numeric expressions separated by commas. In the case of DCB or =, the list may also include string expressions, which cause the characters of the string to be loaded into consecutive bytes in store. For example:

```
TABLE1   DCD   VALUE1,VALUE2;load 2 words into Table1
TABLE2   =     1,2,3,4,5,6 ;load 6 bytes into Table2
MESSAGE  =     "Turn off motor"
ERRORM   =     99,"Error number 99",0
TABLE4   =     """a sentence within quotes"""
TABLE5   =     1,2,3,"a","b",4,5,6
PROMPT   =     ">" ;loads 62 into one byte of memory
PROMPT2  DCW   ">" ;loads 62, and then 0 into 2 bytes
PROMPT3  &     ">" ;loads 62,then 0,0,0 into 4 bytes
```

Loading memory with zeroes has its own directive:

Syntax: *{label} % numeric expression*

For example:

```
BLANKS      %  &400   ;store 1K of zeroes
```

ALIGN

After using store-loading directives such as:

```
=      "a long string"   ;messages
=      1,2,3,4,5         ;a long list
%      VALUE4/SIZE       ;nulls
```

the program counter doesn't necessarily point to a word boundary, which it must do if the file is to continue with program instructions. The alignment of the program counter to a word boundary is automatic if an instruction mnemonic is

encountered after the tables. The Assembler will insert up to three zero bytes to achieve automatic alignment. However, there are occasions when alignment needs to be forced.

The directive ALIGN on its own will set the instruction location to the next word boundary. However, ALIGN can take two optional parameters:

Syntax: `ALIGN` *{power-of-two}{,offset-expression}*

{power-of-two}	defines the boundary.
{,offset-expression}	defines the offset from the boundary.

4 is the power-of-two default and 0 is the offset-expression default, so ALIGN on its own will increment the program counter to the next word boundary. Other values will force the program counter to align to any particular boundary needed by the programmer. These extra arguments will only rarely be needed.

LTORG

The directive LTORG (literal origin) is used to start a literal pool, an area in which to place literals. (See the section entitled *Further instructions* on page 98.) Literals are addressed using PC relative addressing, so large programs may need several LTORG directives.

The Assembler generates a default LTORG at every LNK or END directive in files which are not part of a nested piece of assembly. See the sections END, GET and LNK below.

Laying out storage areas

The ^ and # directives

The Assembler can lay out areas of memory, storage areas, or data structures. The start address of such an area is given by the ^ directive.

Syntax: ^ *expression*

The origin of the storage area is set to *expression*, and a storage-area location counter @ is also set to *expression*. The expression must be fully evaluable on the first pass of the assembly, but may be program-relative. In the absence of a ^ directive, the @ counter is set to zero.

Space in the storage area is reserved by the # directive.

Syntax: *{label}* # *expression*

For example:

```
LABEL1     #       n     ;reserve n bytes
 .......
 .......code
 .......code
 .......
LABEL2     #       4     ;reserve 4 bytes, attached to
                         ;the end of LABEL1's store
```

Every time # is encountered, the label is given the value of @ and then @ is incremented by the number of bytes reserved. The @ counter may be set to another value any number of times by the repeated use of ^ and so storage areas can be easily established anywhere in memory.

Extension to the ^ directive

A special extension of ^ allows a register to be attached to the base address of a storage area:

Syntax: ^ *expression,register*

The register introduced by this extra parameter is taken to be implicit in all symbols defined by any # directives which follow until cancelled by another ^ directive. In this case, the expression must be an absolute value. For example:

```
SB                  RN    10      ;SB is register 10
                    ^     0,SB    ;@=0
Start               #     0       ;ie [SB,#0]
Frame               #     4       ;ie [SB,#0]
StaticBase          #     4       ;ie [SB,#4]
StaticBase_Offset   *     StaticBase-Start
```

The subsequent # directives are, therefore, generating register-relative symbols. This means that later in the source program, it becomes possible to quote any symbol containing an implicit register name in a load or store instruction and the pre-indexed form of opcode will be generated.

For example, the valid line

```
    LDR RO [SB,#StaticBase_Offset]
```

can be replaced by the shorter line

```
    LDR R0,StaticBase
```

and the same code will be generated by the Assembler.

Counter values

The current value of the Assembler's program location counter is referred to by the dot symbol '.', while the current value of the storage-area location counter is, as has already been noted, the '@' symbol. Since these symbols are not particularly obvious (especially when appearing in expressions), they may be replaced by {PC} and {VAR} respectively.

Variables

Symbols have a fixed value attached to them, derived from the first or second pass of the assembly process. It is also possible to define symbols which have values which change as the assembly proceeds. Such symbols are called variables, and the Assembler has two types:

- local variables
- global variables

Global variables can operate over the entire source file, whereas local variables are only accessible within the confines of a macro expansion. Local variables are described in chapter entitled *Macros*.

Declaring variables

Variables must be declared before they are used. The three types of global variable are arithmetic, logical and string. These are declared by the following directives:

Directive	Meaning
GBLA	Define an arithmetic variable
GBLL	Define a logical variable
GBLS	Define a string variable

These symbols may be used in expressions like normal symbols.

Syntax: `GBLx` *`variable_name`*

Altering the value of global and local variables

The directives SETA, SETL and SETS are provided to alter the values of both global and local variables.

Directive	Meaning
SETA	Set the value of an arithmetic variable (global or local)
SETL	Set the value of a logical variable (global or local)
SETS	Set the value of a string variable (global or local)

Syntax: *variable name* SETx *expression*

For example:

```
count          SETA      count+1
message        SETS      "media error"
```

count and message can be used as required in the source file:

```
space          #         count
string         =         message
```

Variable substitution using $

Any attempt to use count and message as labels will, quite rightly, cause the syntax checker to issue error messages. This is because they have been declared as global variables and cannot, therefore, be accepted as labels. However, if the $ symbol is prefixed to them, variable substitution will take place before the line is passed to the syntax checker. Logical and arithmetic variables will be replaced by the result of applying :STR: to them. String variables will be replaced by their value. For example:

```
               GBLS      A
               GBLA      B
               GBLL      C
;three variable types declared
A              SETS      "Labname"
B              SETA      1
C              SETL      {TRUE}
;and duly set
;without $ they are rejected as labels
A              ADD       R0,R0,R1 ;syntax error!
;with $ they are accepted
$A             AND       R0,R1,#8
L$B            AND       R2,R3,#16
$C             AND       R4,R5,#32
```

After the Assembler has performed variable substitution, its own internal conception of the last three lines of source can be considered as:

```
Labname        AND       R0,R1,#8
L00000001      AND       R2,R3,#16
T              AND       R4,R5,#32
```

Other useful variables

There are five special variables. These are:

{PC}	current value of Assembler's program location counter
{VAR}	current value of the storage-area location counter
{TRUE}	logical constant true
{FALSE}	logical constant false
{OPT}	value of the currently set printer output option.

The variables {PC} and {VAR} have already been explained (see the section entitled *Counter values* on page 128). The other three variables take the bracketed form of {*name*}.

{OPT}

A simple but extremely useful way of using {OPT} is to use it to store the currently set printer options, force a temporary change in printing mode, and then, later in the source code, to restore the original value of {OPT}. For example:

```
              GBLA    AS_WAS
AS_WAS        SETA    {OPT}
;start of long section of code
;eg a macro
              OPT     2           ;turn off listing!
              .....lots of code......
              OPT     AS_WAS      ;restore print option
;end of long section of code
```

Routines and local labels

Although labels may not begin with a digit, there is a special form of local label which bears a number in the range 0-99. However, the scope of this type of local label is limited by the ROUT directive.

Beginning a local label area

The syntax to begin a new local label area is:

{label} ROUT

in the label and instruction fields respectively. The start of the source is the start of the first local label area. The extent of a local label area is from its ROUT directive up to the next ROUT directive or end of assembly.

Defining a local label

The local label definition syntax is:

number{routinename}

in the label field. The number must lie in the range 0-99. The parameter routinename need not be present, but if it is, it will be checked against the label on the last ROUT directive. If no label is present on the last ROUT directive, and yet a routinename has been provided, an assembly error will be generated.

Referencing a local label

The syntax for the local label reference is:

%{x}{y}n{routinename}

%	introduces a local label reference. % may be used anywhere where an ordinary label reference is valid.
{x}{y}	The optional letters x and y tell the Assembler the direction and/or level for the search of the location of the local label.
{x}	any one of the following options:

absent	look backwards and forwards through the sourcefile for the label
B	look backwards for the label
F	look forwards for the label

Searches for a local label will never go outside the current local label area; that is, they will never go either forwards or backwards past a ROUT directive. The same local label may be defined many times. The Assembler always uses the first matching local label that it finds in its search.

{y} any one of the following options:

absent	look at this macro and outer nested levels
A	look at all macro levels
T	look only at this macro level.

n	The number *n* is the number given to the local label.
{routinename}	its use makes the source listing more readable. If present, the Assembler will check it against the routine's label.

For example:

```
NORMLABEL     ROUT     ;The routine is between the ROUTs.
              .........;Its name is NORMLABEL, but the
              . .......;naming of the routine is
              .........;optional
              .........
00            .........;Local label 00
              .........
              BEQ %00NORMLABEL ;Branch if equal to 00
              .........
01            .........;Local label 01
              .........
NEXTROUTINE   ROUT
```

Local labels can be used anywhere in the source file and are particularly useful for solving the problem of unique macro labels. See the section entitled A *division macro* on page 148.

Error handling

As an aid to error trapping, the ASSERT directive is provided for use inside and outside macros

Syntax: `ASSERT` *`logical expression`*

For example:

```
ASSERT TEMP1 < TEMP
```

If the logical expression returns a true result then nothing happens but a false result will generate an error during the second pass of the assembly. The error message is "`Assert failed at line xxxxxx`".

A similar directive ! is inspected on both passes of the Assembler. This time an arithmetic expression is evaluated

! *`arithmetic expression, string expression`*

If the arithmetic expression is:

= 0 — no action is taken on pass 1 and the string is printed out as a warning on pass 2. No error is generated.

<> 0 — an error is produced and assembly halts after pass 1. The arithmetic expression is evaluated on pass one, so forward referencing is not permitted. The string expression is printed as an error.

ORG

ORG is only intended for use with AAsm.

The program's starting point is determined by the ORG directive.

Syntax: ORG *numeric_expression*

For example:

```
        ORG   &100
```

Or:

```
START *       &100
        ORG   START
```

Only one ORG directive is allowed in the entire source and no ARM instructions or Assembler store directives can precede the ORG directive. In the absence of an ORG directive, the program is considered to be relocatable and the program location counter is initially set to 0.

Otherwise:

- ORG sets the program location counter, the symbol for which is '.' or {PC}.
- and also sets the load and execute address for the code file if you are using AAsm.

LEADR

AAsm (but not ObjAsm) has a directive called LEADR which sets the load and execute address. Its purpose is to enable a default run address to be set for relocatable binary output.

LEADR can be used with or without the ORG directive to indicate the address at which the program should load and run. If ORG is present, then LEADR will override its effect on load and execute addresses.

Syntax: LEADR *numeric_expression*

For example:

```
        LEADR &8000
```

END

The Assembler stops processing an input file on encountering the END directive and any source code after END or LNK will be ignored by the Assembler. Failing to end a file with an END or LNK directive is an error.

The END directive and assembly

If the input file was part of a nested piece of assembly invoked by a GET directive, then assembly will continue within the file containing the GET, at the line following the GET directive. Otherwise, the current pass will stop.

If this was the first pass, and no errors have been generated, then assembly will proceed to the second pass starting again in the original source file.

GET

The GET directive in the source file is used to include a secondary source file within the current assembly.

Syntax: `GET` *`filename`*

Once assembly of the secondary source file is complete, assembly continues in the original source file. The secondary source file must be terminated by an END or LNK directive, and may include further GET directives.

In the following example, the primary file is called file_a:

```
SYM1          *       SYM2+100
                ......................
                ......file_a code....
                ......................
                GET   file_b
                ......................
                ......more file_a...
                ........code.........
                ......................
                END
```

This is the secondary file, file_b :

```
SYM2          *       200
                .............
                .............
                END
```

Symbol SYM1 takes the value 300. There are two points to notice in this example:

- file_b has no ORG statement and so the program counter merely continues to increment as file_b is assembled. Had the secondary file been given an ORG of its own, an error would have been flagged.
- file_b must have an END directive, whereupon control passes back to file_a.

LNK

Syntax: `LNK` *`filename`*

During assembly, a secondary file can be called via the GET or LNK directives. In order to prevent control passing back to the primary file once the secondary file has been assembled, the LNK directive is used in place of GET.

LNK is generally used to split large source files into sequences of smaller more manageable ones.

Objasm directives

ObjAsm is the Assembler which creates Acorn Object Format code (AOF). It uses a number of directives not used by AAsm. These are:

- AREA
- IMPORT
- EXPORT
- STRONG
- ENTRY
- KEEP
- AOF
- AOUT

ObjAsm also accepts more extensive operands to the DCD directive.

External expressions

An external expression is an imported symbol plus an optional numeric expression, for example:

```
IMPORT      LinkSymbol
B           LinkSymbol + 4
DCD         LinkSymbol
```

Note that when using ObjAsm:

- external expressions and program-relative symbols not defined in the current area are valid operands to the branch and branch and link instructions.
- external expressions and program-relative symbols not defined in the current area are not valid in general expressions.

Using literals

Program-relative expressions and external expressions are also valid literals in ObjAsm.

AREA

This directive gives a name plus optional attributes and alignment to the area in which the code or data following the directive is to be put.

The basic form of the directive is `AREA` *`symbol`*. The symbol is the name of an area and, as such, it is an external symbol which can be used in the link phase of processing. Other programs may import the symbol and make use of it. The value of the symbol may be taken to be offset zero from the start of the area.

A list of attributes may follow the symbol. These are:

```
AREA  symbol{,attr}{,attr}....{,ALIGN=expression}
```

The attributes, many of which are self-explanatory, are as follows:

`REL`	Relocatable: this area may be relocated by the Linker.
`CODE`	This area contains code (and is therefore read only).
`DATA`	This area contains read-write data.
`READONLY`	This area may not be written.
`COMDEF`	Common area definition (only used by Fortran 77).
`COMDEF`	A common area (only used by Fortran 77)
`NOINIT`	This is an initialised data area.

IMPORT

Syntax: `IMPORT` *`symbol{,WEAK}`*

IMPORT is followed by a symbol which is treated as a program address. It provides the Assembler with a name which may be referred to but which is not defined within this assembly. It must, therefore, be imported at link time from another piece of the Acorn object format code, when its value will be ascertained and used. If the option WEAK is coded, then the Linker will not fault an unresolved reference to this symbol at link time.

EXPORT

Syntax: `EXPORT` *`symbol`*

EXPORT is also followed by a symbol. This time the symbol is being declared for use by other Acorn object format files at link time.

STRONG

Syntax: `STRONG` *`symbol`*

STRONG is a variant of EXPORT. Set the STRONG attribute on the symbol for special interpretation by the Linker. See the *Link* chapter in the accompanying *Desktop Development Environment* user guide for more details.

ENTRY

Syntax: `ENTRY`

The directive ENTRY causes the program's execution to start from this address. It signals to the whole program (which is contained in the various Acorn object format files) that the address computed for ENTRY (ie the value of the program location counter when ENTRY is assembled) is the execute address for the entire program.

KEEP

Syntax: `KEEP` *`{symbol}`*

The Linker will not normally keep track of symbols it does not need. To force the Linker to retain symbols it would otherwise consider unnecessary, the Link option Debug should be selected. ObjAsm's own directive KEEP has the function of declaring a symbol which is not needed by the Acorn object format, but which can be maintained in the Acorn object format symbol table. If the symbol is not specified, then all program relative symbols will be kept.

In this way symbols of use to the DDT debugger can be stored and will not be lost.

DCD

In ObjAsm, DCD or & will accept program-relative expressions and external expressions for its operands, as well as the numeric expressions used by AAsm. For example:

```
        IMPORT   Fred
Label   DCD      Fred+2
```

AOF and AOUT

If Objasm detects unix style assembler input it will output unix a.out format linkable object files. AOUT forces a.out output. These features are only of use when porting assembly language between Acorn RISC OS and RISC IX. Unix style assembler is documented in the RISC IX *Programmers' Reference Manual*.

10 Conditional and repetitive assembly

This chapter describes the features available within the Assembler for constructing conditional assembly statements and conditional looping statements.

Conditional assembly

The | and | directives mark the start and finish of sections of the source file which are to be assembled only if certain conditions are true. The basic construction is IF. . .THEN. . .ENDIF, however, ELSE is also supported, giving the full IF. . .THEN. . .ELSE. . .ENDIF conditional assembly.

The start of the section is

| *logical expression*

and is known as the IF directive.

|

is the ELSE directive and

|

is the ENDIF directive.

The two main ways of using these directives are:

```
[ logical expression
......................
.......code..........
......................
]
```

The code will only be assembled if the logical expression is true, it will be skipped if the logical expression is false.

```
[ logical expression
........................
..first piece of code..
........................
|
........................
..second piece of code.
........................
]
```

If the logical expression is true, the first piece of code will be assembled and the second skipped. If the expression is false, the first piece of code will be skipped and the second assembled.

Conditional assembly and the TERSE command

Lines conditionally skipped by these directives are not listed if:

- TERSE ON is given to the action prompt
- TERSE ON is given by default.

If -NOTERSE is given to the command line, or TERSE OFF is given to an action prompt, then conditionally skipped code will be listed.

Using ELSE, IF and ENDIF directives

A block, which is being conditionally assembled, can contain several [|] directives; that is, conditional assembly can be nested. It is also valid to place more than one ELSE directive within an IF block.

An example of a notional data storage routine is given below. This routine can either use a disc or a tape data storage system. To assemble the code for tape operation, the programmer prepares the system by altering just one line of code, the label SWITCH.

```
DISC      *     0
TAPE      *     1
SWITCH    *     DISC
          ..........
          ...code...
          .........
          [ SWITCH=TAPE
          .........................
          ...tape interface code...
          .........................
          ]
```

```
[ SWITCH=DISC
.........................
...disc interface code...
.........................
]
...code continues...
....................
```

or alternatively,

```
[ SWITCH=TAPE
.........................
...tape interface code...
.........................
|
.........................
...disc interface code...
.........................
]
...code continues...
.....................
```

The IF construction can be used inside macro expansions as easily as it is used in the main program.

Repetitive assembly

It is often useful for program segments and macros to produce tables. To do this, they must be able to have a conditional looping statement. The Assembler has the WHILE . . .WEND construction. This produces an assembly time (not runtime) loop.

Syntax: `WHILE` *`logical expression`*

to start the repetitive block and

```
WEND
```

to end it.

For example:

```
            GBLA   counter
counter     SETA   100

            WHILE counter >0
            DCD    &$counter
counter     SETA   counter-1
            WEND
```

produces the same result as the following (but is shorter and less prone to typing errors):

```
            DCD    100
            DCD    99
            DCD    98
            DCD    97
            :
            :
            :
            DCD    2
            DCD    1
```

Since the test for the WHILE condition is made at the top of the loop, it is possible that the source within the loop will not generate any code at all.

Listing of conditionally skipped lines is as for conditional assembly.

11 Macros

Macros give the programmer a means of placing a single instruction in his/her source which will be expanded at assembly time to several assembler instructions and directives, just as if these instructions and directives had been written by the programmer within the source at that point.

For example, one might wish to define a TestAndBranch instruction. This would normally take two ARM instructions. So we tell the Assembler, by means of a macro definition, that whenever it meets the TestAndBranch instruction, it is to insert the code we have given it in the macro definition. This is of course a convenience: we could just as easily write the relevant instructions out each time, but instead we let the Assembler do it for us.

The Assembler determines the destination of the branch with a macro parameter. This is a piece of information specified each time the macro is coded: the macro definition specifies how it is used. In the TestAndBranch example, we might also make the register to be tested a parameter, and even the condition to be tested for. Thus our macro definition might be:

```
        MACRO
$label  TestAndBranch $dest,$reg,$c;this is called the macro prototype
                                   ; statement
$label  CMP $reg,#0                ;these two lines are the ones that
        B$cc $dest                 ;will be substituted in the source.
        MEND                       ;this says the macro definition is
                                   ; finished
```

A use of the macro might be:

```
Test    TestAndBranch       NonZero,R0,NE
        :
        :
        :
NonZero
```

The result, as far as the Assembler is concerned, is:

```
Test    CMP                 R0,#0
        BNE                 NonZero
        :
        :
        :
NonZero
```

Syntax

Syntax: `MACRO`

The fact that a macro is about to be defined is given by the directive MACRO in the instruction field.

This is immediately followed by a macro prototype statement which takes the form:

```
{$label} macroname{$parameter}{,$parameter}{,$parameter}..
```

`{$label}`	if present, it is treated as an additional parameter.
`{$parameter}`	Parameters are passed to the macro as strings and substituted before syntax analysis. Any number of them may be given.

The purpose of the macro prototype statement is to tell the Assembler the name of the macro being defined. The name of the macro is found in the opcode field of the macro prototype statement.

The macro prototype statement also tells the Assembler the names of the parameters, if any, of the macro. Parameters may occur in two places in the macro prototype statement. A single optional parameter may occur in the label field, shown as $label above. This is normally used if the macro expansion is to contain a program label, and is merely an aid to clarity, as can be seen in the TestAndBranch example. Any number of parameters, separated by commas, may occur in the operand field. All parameter names begin with the character $, to distinguish them from ordinary program labels.

The macro prototype statement can also tell the Assembler the default values of any of the parameters. This is done by following the parameter name by an equals sign, and then giving the default value. If the default value is to begin or end with a space then it should be placed within quotes. For example:

```
$reg   = R0
$string= " a string "
```

It is not possible to give a default value for the parameter in the label field.

For example:

```
                MACRO
$label          MACRONAME $num,$string,$etc
                ...............
                ...............
$label          ....lots of....
                .....code......
                =      $num
                =      $string
                =      "the price is $etc"
                =      0
                MEND
```

- MACRONAME is the name of this particular macro and $num, $string and $etc are its parameters. Other macros may have many more parameters, or even none at all.
- The body of the macro follows after MACRONAME, with $label being optional even if it was given in the macro prototype statement.
- $etc will be substituted into the string "the price is " when the macro is used.
- The macro ends with MEND.

The macro is called by using its name and any missing parameters are indicated by commas, or may be omitted entirely if no more parameters are to follow. Thus, MACRONAME may be called in various ways:

```
MACRONAME9,"disc",7
```

or:

```
MACRONAME9
```

or:

```
MACRONAME,"disc",
```

Local variables

Local variables are similar to global variables, but may only be referenced within the macro expansion in which they were defined. They must be declared before they are used. The three types of local variable are arithmetic, logical and string. These are declared by:

Directive	Local variable type	Initial state
LCLA	Arithmetic	zero
LCLL	Logical	FALSE
LCLS	String	null string.

New values for local variables are assigned in precisely the same way as new variables for global variables: that is, using the directives SETA, SETL and SETS.

Syntax: *variable name* SETx *expression*

Directive	Local variable type
SETA	Arithmetic
SETL	Logical
SETS	String

MEXIT directive

Normally, macro expansion terminates on encountering the MEND directive, at which point there must be no unclosed WHILE/WEND loops or pieces of conditional assembly. Early termination of a macro expansion can be forced by means of the MEXIT directive, and this may occur within WHILE/WEND loops and conditional assembly.

Default values

Macro parameters can be given default values at macro definition time. In the example of the macro 'MACRONAME' already used:

```
        MACRO
$label  MACRONAME $num,$string,$etc
        ...............
        ...............
$label  ....lots of....
        .....code......
        =      $num
        =      $string
        =      "the price is $etc"
        =      0
        MEND
```

it is possible to write `$num=10` in the macro prototype statement. Then, when calling the macro, a vertical bar character '|' will cause the default value 10 to be used rather than the value $num.

Syntax: *$parameter=default value*

For example:

```
MACRONAME |,"disc",7
```

will be equivalent to:

```
MACRONAME 10,"disc",7
```

Note that this default is not used when the macro argument is omitted - the value is then empty.

Macro substitution method

Each line of a macro is scanned so it can be built up in stages before being passed to the syntax analyser. The first stage is to substitute macro parameters throughout the macro and then to consider the variables. If string variables, logical variables and arithmetic variables are prefixed by the $ symbol, they are replaced by a string equivalent. Normal syntax checking is performed upon the line after these substitutions have been performed.

An important exception to these values is that vertical bar characters ('|') prevent substitution from taking place in some circumstances. To be specific, if a line contains vertical bars, substitution will be turned off after this first vertical bar, on again after the second one, off again after the third, and so on. This allows the use of dollar characters in labels (see the section entitled *Symbols and labels* on page 56 for details).

In certain circumstances, it may be necessary to prefix a macro parameter or variable to a label. In order to ensure that the Assembler can recognise the macro parameter or variable, it can be terminated by a dot '.' The dot will be removed during substitution.

For example:

```
            MACRO
$T33        MACRONAME
            ...............
            ...............
$T33.L25    ....lots of....
            .....code......
            MEND
```

If the dot had been omitted, the Assembler would not have related the $T33 part of the label to the macro statement and would have accepted $T33L25 as a label in its own right, which was not the intention.

Nesting macros

The body of a macro can contain a call to another macro; in other words, the expansion of one macro can contain references to macros. Macro invocation may be nested up to a depth of 255.

A division macro

As a final example, the following macro does an unsigned integer division:

```
; A macro to do unsigned integer division. It takes four
; parameters, each of which should be a register name:
;
; $Div:  The macro places the quotient of the division in
;        this register - ie $Div := $Top DIV $Bot.
;        $Div may be omitted if only the remainder is
;        wanted.
; $Top:  The macro expects the dividend in this register
;        on entry and places the remainder in it on exit -
;        ie $Top := $Top MOD $Bot.
; $Bot:  The macro expects the divisor in this register on
;        entry. It does not alter this register.
; $Temp: The macro uses this register to hold intermediate
;        results. Its initial value is ignored and its
;        final value is not useful.
;
; $Top, $Bot, $Temp and (if present) $Div must all be
; distinct registers. The macro does not check for division
; by zero; if there is a risk of this happening, it should
; be checked for outside the macro.
;

        MACRO
$Label  DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot        ;Produce an error if the
        ASSERT  $Top <> $Temp       ; registers supplied are
        ASSERT  $Bot <> $Temp       ; not all different.
        [       "$Div" /= ""
        ASSERT  $Div <> $Top
        ASSERT  $Div <> $Bot
        ASSERT  $Div <> $Temp
        ]
$Label  MOV     $Temp,$Bot          ;Put the divisor in $Temp.
        CMP     $Temp,$Top,LSR #1   ;Then double it until
90      MOVLS   $Temp,$Temp,LSL #1  ; 2 * $Temp > $Top.
        CMP     $Temp,$Top,LSR #1
```

```
        BLS     %b90
        [       "$Div" /=""
        MOV     $Div,#0             ;Initialise the quotient.
        ]
91      CMP     $Top,$Temp          ;Can we subtract $Temp?
        SUBCS   $Top,$Top,$Temp     ;If we can, do so.
        [       "$Div /= ""
        ADC     $Div,$Div,$Div      ;Double $Div & add new bit
        ]
        MOV     $Temp,$Temp,LSR #1  ;Halve $Temp.
        CMP     $Temp,$Bot          ;And loop until we've gone
BHS     %b91                        ; past the original divisor.
        MEND
```

The statement:

```
Divide      DivMod      R0,R5,R4,R2
```

would be expanded to:

```
        ASSERT  R5 <> R4            ;Produce an error if the
        ASSERT  R5 <> R2            ; registers supplied are
        ASSERT  R4 <> R2            ; not all different
        ASSERT  R0 <> R5
        ASSERT  R0 <> R4
        ASSERT  R0 <> R2
Divide  MOV     R2,R4               ;Put the divisor in R2.
        CMP     R2,R5,LSR #1        ;Then double it until
90      MOVLS   R2,R2,LSL #1        ; 2 * R2 > R5.
        CMP     R2,R5,LSR #1
        BLS     %b90
        MOV     R0,#0               ;Initialise the quotient.
91      CMP     R5,R2               ;Can we subtract R2?
        SUBCS   R5,R5,R2            ;If we can, do so.
        ADC     R0,R0,R0            ;Double R0 & add new bit.
        MOV     R2,R2,LSR #1        ;Halve R2.
        CMP     R2,R4               ;And loop until we've gone
        BHS     %b91                ; past the original divisor.
```

Similarly, the statement:

```
        DivMod      ,R6,R7,R8
```

would be expanded to:

```
        ASSERT  R6 <> R7            ;Produce an error if the
        ASSERT  R6 <> R8            ; registers supplied are
        ASSERT  R7 <> R8            ; not all different.
        MOV     R8,R7               ;Put the divisor in R8.
```

```
        CMP     R8,R6,LSR #1        ;Then double it until
90      MOVLS   R8,R8,LSL #1        ; 2 * R8 > R6.
        CMP     R8,R6,LSR #1
        BLS     %b90
91      CMP     R6,R8               ;Can we subtract R8?
        SUBCS   R6,R6,R8            ;If we can, do so.
        MOV     R8,R8,LSR #1        ;Halve R8.
        CMP     R8,R7               ;And loop until we've gone
        BHS     %b91                ; past the original divisor.
```

Note:

- Conditional assembly is used to reduce the size of the assembled code (and increase its speed) in the case where only the remainder is wanted.
- Local labels are used to avoid multiply defined labels if DivMod is used more than once in the assembler source.
- The letter 'b' is used in the local label references (indicating that the Assembler should search backwards for the corresponding local labels) to ensure that the correct local labels are found.

Part 3 - Developing software for RISC OS

12 Writing relocatable modules in assembler

Relocatable modules are the basic building blocks of RISC OS and the means by which RISC OS can be extended by a user.

The relocatable module system provides mechanisms suitable for

- providing device drivers
- extending the set of RISC OS *commands
- providing shared services to applications (eg the shared C library)
- implementing 'terminate and stay resident' (TSR) applications.

All these projects require code either to be more persistent than standard RISC OS applications or to be used by more than one application, hence resident in the address space of more than one application. If your program does not have these requirements it is not recommended to put it in modules, as relocatable modules are more persistent consumers of system resources than applications, and are also more difficult to debug. The DDT debugger is not currently able to debug relocatable modules.

This chapter is not intended to provide a complete set of the technical details you need to know to construct any relocatable module. For more information on such details, see the RISC OS *Programmer's Reference Manual*. The points covered here are intended to provide help for constructing relocatable modules specifically in assembly language.

For more details of memory management in relocatable modules, see the chapter entitled *Using memory efficiently*.

Unlike the construction of relocatable modules in high level languages, no substantial standard portions of code are supplied for you by the DDE tools. This means that you have to construct the module header table, workspace routines, etc. yourself.

Note that some of the relocatable module entry points are called in SVC mode. Such routines may use SWIs implemented by other parts of RISC OS, but unlike being in user mode, SWIs corrupt r14, so this must be stored away. Floating point instructions should not be used from SVC mode.

Assembler directives

The two ARM assemblers, AAsm and ObjAsm, can both be used to construct modules but have different uses depending on the type of module required.

AAsm modules

AAsm can be used to directly construct relocatable modules from source by assembling with the **Module** setup menu option enabled. As no linking step occurs, all the source files of your module must join themselves into one 'lump' using the GET and LNK directives.

The code that assembles to the lowest address must contain your module header table. This starts with a couple of lines containing the LEADR directive:

```
Module_LoadAddr * &fffffa00
     LEADR Module_LoadAddr
```

The table of entry points relative to the module base then follows. For example:

```
Module_BaseAddr
     DCD   RM_Start     -Module_BaseAddr
     DCD   RM_Init      -Module_BaseAddr
     DCD   RM_Die       -Module_BaseAddr
     DCD   RM_Service   -Module_BaseAddr
     DCD   RM_Title     -Module_BaseAddr
     DCD   RM_HelpStr   -Module_BaseAddr
     DCD   RM_HC_Table -Module_BaseAddr
```

ObjAsm modules

ObjAsm can be used to assemble a module from a set of source files, a link step being required to join the output object files to form the usable module.

The separation of routines into separately assembled files has several advantages. Since DDT cannot be used to debug modules, it can be useful to link routines into test applications, debug them there with DDT, then link them into the module.

It can be a good idea to construct a module with the module header and the small routines/data associated with it in one source file, to be linked with the code forming the body of the module.

Such a module header file must be linked so that it is placed first in the module binary. To do this it should contain an AREA directive at its head such as:

```
AREA |!!!Module$$Header|, CODE, READONLY
```

Areas are sorted by type and name; a name beginning with '!' is placed before an alphabetic name, so the above can be used to ensure first placing.

The module header source needs to contain IMPORT directives making available any symbols referenced in the module body. In addition, the initialisation routine should call __RelocCode, a routine added by the linker which relocates any absolute references to symbols when the module is initialised. If the module header source contains the initialisation routine, it must use the IMPORT directive to make __RelocCode available.

The module header must be preceded by the ENTRY directive:

```
    ENTRY

Module_BaseAddr
    DCD   RM_Start    -Module_BaseAddr
    DCD   RM_Init     -Module_BaseAddr
    DCD   RM_Die      -Module_BaseAddr
    DCD   RM_Service  -Module_BaseAddr
    DCD   RM_Title    -Module_BaseAddr
    DCD   RM_HelpStr  -Module_BaseAddr
    DCD   RM_HC_Table -Module_BaseAddr
```

Examples

The Acorn Desktop Assembler product is supplied with two versions of the source for an example relocatable module; one to be assembled with AAsm from one source file; the other to be assembled with ObjAsm from two source files then linked. The source of the AAsm example is in `User.ModeEx.s`, that of the ObjAsm example in `User.SkelRM.s`.

Both versions of the source produce a relocatable module with exactly the same function - providing an extra soft screen mode. This has to be done via service call handling, and to be useful must be persistent, so providing a typical use of relocatable modules. For more details of the function of ModeEx, see the section entitled *Example AAsm session* on page 27.

The SkelRM version (assembled with ObjAsm) has its module header separated from the main module body. `User.SkelRM.s.SkelRM` is the source file producing the module header, and may be useful for you to copy and edit to form headers for your own modules.

13 Interworking assembler with C

Interworking assembly language and C – writing programs with both assembly language and C parts – requires using both the Acorn Desktop Assembler and Acorn Desktop C products if you want to do more than just try the examples supplied with Acorn Desktop Assembler.

Interworking assembly language and C allows you to construct top quality RISC OS applications. Using this technique you can take advantage of many of the strong points of both languages. Writing most of the bulk of your application in C allows you to take advantage of the portability of C, the maintainability of a high level language, and the power of the C libraries and language. Writing critical portions of code in assembler allows you to take advantage of all the speed of the Archimedes and all the features of the machine (eg the complete floating point instruction set).

The key to interworking C and assembler is writing assembly language procedures that obey the ARM Procedure Call Standard (APCS). This is a contract between two procedures, one calling the other. The called procedure needs to know which ARM and floating point registers it can freely change without restoring them before returning, and the caller needs to know which registers it can rely on not being corrupted over a procedure call. Additionally both procedures need to know which registers contain input arguments and return arguments, and the arrangement of the stack has to follow a pattern that debuggers, etc. can understand. For the specification of the APCS, see *Appendix* F - ARM *procedure call standard* in the accompanying *Acorn Desktop Development Environment* user guide.

Examples

The following programs have been provided to demonstrate how to write programs combining assembly language and C.

PrintLib

The linkable object library `User.PrintLib.o.Library` contains three object files, each containing a screen printing routine. The three procedures, each written in assembly language with sources in `User.PrintLib.s`, are `print_string`, `print_hex` and `print_double`. They print null terminated strings, integers in hexadecimal, and double precision floating point numbers in scientific format respectively. For more details on constructing this library, see the section entitled *Making your own linkable libraries* on page 16.

Each routine is written to obey the APCS, so it can be called from assembler, C, or any other high level language obeying the APCS. The sources for PrintLib illustrate several aspects of the APCS, such as the distinction between leaf and non-leaf procedures and how floating point arguments are passed into a procedure.

A small example C program using the routines in PrintLib is supplied with the Acorn Desktop Assembler product. Its C source is in the file `User.PrintLib.c.hello`. Since a C compiler is not supplied with Acorn Desktop Assembler, the object file `User.PrintLib.o.hello` produced by compiling the C program is also supplied. To try this example, merely link `o.hello` with `o.PrintLib` to produce an executable AIF file, then run this by double clicking on its name in a directory display. A standard RISC OS command line output window appears containing text printed by the assembly language library routines as a result of arguments passed from C:

```
Run adfs::HardDisc4.$.User.PrintLib.!RunImage
hello world
09ABCDEF
-1.2346E-1
1.0000E1
-1.0000E-1
1.0000E0
0
1.0000E100

Press SPACE or click mouse to continue
```

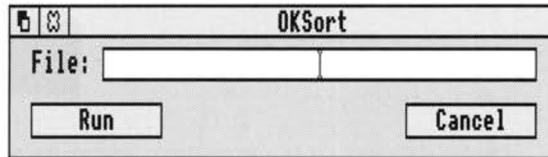
OKSort

`OKSort` is a simple command line program which sorts words into alphabetic order in a text file specified in its command line. It contains no knowledge of the RISC OS desktop, being provided with a desktop interface by the FrontEnd module. For more details of using FrontEnd, see the section entitled *Using FrontEnd on your programs* on page 15.

The command line syntax of OKSort is:

`OKSort` *`filename`*

The FrontEnd setup dialogue box of OKSort is:



Note that only an input filename is specified, the input file being overwritten by the sorted output version.

The command line tool executable image file is constructed from a C file containing the bulk of the program and an assembly language file containing code for the routine cistrcmp, a routine particularly critical to the execution speed of `OKSort`. Though this speed optimisation may not be very useful in this particular example, it serves to illustrate how interworking allows access to the most important advantages of both C and assembler.

The source files are found in User.!OKSort.c and User.!OKSort.s respectively. Since a C compiler is not supplied with Acorn Desktop Assembler, the object file `User.!OKSort.o.OKSort` is also supplied. This is produced by compiling the C source of `OKSort` and partially linking it with a C library. One point illustrated in the assembler file is the use of the AREA directive rather than the EXPORT directive to provide a symbol for C to reference.

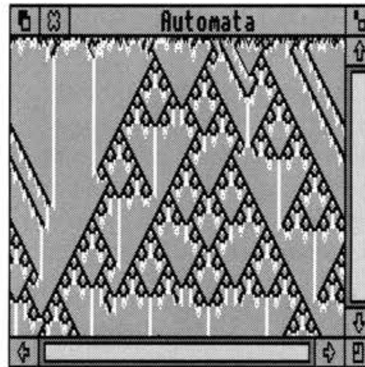
Automata

Automata is a complete RISC OS desktop application coded in both C and assembler. The RISC OS desktop user interface of Automata is coded in C, making use of the RISC_OSLib library facilities. The speed critical sprite construction routines are written in assembly language.

The source files for Automata are found in `User.!Automata.c` and `User.!Automata.s` respectively. Since a C compiler and associated libraries are not supplied with Acorn Desktop Assembler, the object file `User.!Automata.o.Automata` is also supplied. This is produced by compiling the C source of Automata and partially linking it with the ANSI C library and RISC_OSLib library. To construct the executable image !RunImage of Automata, assemble the assembler source of Automata with ObjAsm and link the resultant object file with the supplied one.

Double clicking on !Automata in the User directory display starts the application, putting its icon on the icon bar:

Clicking Select on the main icon brings up one or more scrollable windows containing one dimensional cellular automata patterns:



14 Using memory efficiently

This chapter provides basic information on memory management by RISC OS applications. It is intended to provide some specialist knowledge to help you write efficient programs for RISC OS, and to provide some practical hints and tips.

All the information in this chapter relating to programs written in C refers to the Acorn Desktop C product.

Guidelines

Follow the guidelines in this section to make the best use of available memory. The guidelines are explained in more detail on the following pages.

- **Use recovery procedures** – Your program should keep the machine operational. Don't allow your program to lock up when memory runs out; your program should indicate that it has run out of memory (with an error or warning message) and only stop subsequent actions that use more memory. Ideally, ensure that actions which free up memory have enough reserved memory to run in.
- **Return unwanted memory** – You should return any memory you have no further use for. Claiming memory then not returning it can tie up memory unnecessarily until the machine is re-booted. RISC OS has no garbage collection, so once you have asked for memory RISC OS assumes that you want it until you explicitly return it, even if your program terminates execution. Language libraries often provide you with protection from this, as long as memory is claimed from them.
- **Don't waste memory** – You should avoid wasting memory. It is a finite resource, often wasted in two ways:
  - by permanently claiming memory for infrequent operations
  - by fragmenting it, so that although there is enough unused memory, it is either in the wrong place, or it is not in large enough blocks to use.

Recovery from lack of memory

An important consideration when designing programs for RISC OS is the recovery process, not just from user errors, but also from lack of system resources.

An example of a technique that can be designed into an application is to make an algorithm more disc-based and less RAM-based on detection of lack of memory. This could allow you to continue using an application on a small machine (especially one with a hard disc) at the expense of some speed.

When implementing your code, expect the unexpected and program defensively. Be sure that when the system resources you need (memory, windows, files etc) are not available, your program can cope. Make sure that, when a document managed by your program expands and memory runs out, the document is still valid and can be saved. Don't just check that your main document expansion routines work; check that **all** routines which require memory (or in fact any system resource) fail gracefully when there is no more.

Centralising access to system resources can help: write your program as if every operating system interface is likely to return an error.

Avoiding permanent loss of memory

Permanent loss of memory is mainly a problem for applications or modules written entirely in assembly language. When interworking assembler routines with C or another high level language you should use memory handed to you by the high level language library (eg use `malloc` to get a memory area from C and pass a pointer to it as an argument to your assembler routine). The language library automatically returns such areas to RISC OS on program exit. Additional types of program requiring care to avoid memory loss are those expected to run for a long time (eg a printer spooler) and those making use of RMA directly through SWI calls.

When using the RMA for storage directly through SWI calls, especially for items in linked lists, consider using the first word as a check word containing four characters of text to identify it as belonging to your program. When a block of RMA is deallocated, the heap manager puts it back into a list of free blocks, and in so doing overwrites the first word of the block.

This technique therefore serves two purposes:

1. after your program has been run and exited, your check word can be searched for, showing up any blocks you have failed to deallocate
2. it avoids problems when accidentally referencing deallocated memory.

A typical problem of referencing deallocated blocks results from using the first word as a pointer to your program's next block, then accidentally referencing a wild pointer when it is overwritten.

You can use the following BASIC routine to search for any lost blocks:

```
100 REM > LostMemory checks for un-released blocks
110 RMA%=&01800000: RMAEnd% = RMA% + (RMA%!12)
120 FOR PossibleBlock% = RMA%+20 TO RMAEnd%-12 STEP 16
130    REM Now loop looking for "Prog"
140    IF PossibleBlock%!0 = &676F7250 THEN
150       PRINT "Block found at &";~PossibleBlock%
160    ENDIF
170 NEXT PossibleBlock%
180 END
```

When writing relocatable module initialisation code you should check that memory and other system resources are returned if initialisation is unable to complete and is going to return with V set. It is often useful to construct module finalisation code as a mirror image of initialisation code so that it can be jumped to when initialisation is going to return an error and cleaned up. A typical algorithm is:

Initialisation

Claim main workspace: If error then keep this error and goto *Exit3*
Claim secondary workspace: If error then keep this error and goto *Exit2*
Claim tertiary workspace: If error then keep this error and goto *Exit1*
Return

Finalisation

	Set kept error to null
	Release tertiary workspace
Exit1	Release secondary workspace
Exit2	Release main workspace
Exit3	Get kept error (if there was one)
	Return

Avoiding memory wastage

The key factor in writing programs that use memory efficiently and don't waste it is understanding the following:

- how SWI XOS_Module and SWI XOS_Heap work if you are constructing a relocatable module or are using the RMA from an application
- how C flex and `malloc` work when writing a C program (parts of which may be written in assembler).

This understanding will lead you to writing programs that will work in harmony with the storage allocator. See the following section for a description of C memory allocation.

The C storage manager

Understanding the C storage manager may be useful to writers of assembly language for two reasons: to assist in constructing part C and part assembler programs; to assist in constructing their own memory allocation routines, both as an example algorithm and as an allocator that may be running for other applications at the same time as their own.

Normal C applications (ie those not running as modules) claim memory blocks in two main ways:

- from `malloc`
- from `flex`.

The `malloc` heap storage manager is the standard interface from which to claim small areas of memory. It is tuned to give good performance to the widest variety of programs.

In the following sections, the word *heap* refers to the section of memory currently under the control of the storage manager (usually referred to as `malloc`, or the `malloc` heap).

The flex facility is available as part of RISC_OSLib, and can be useful for claiming large areas of data space. It manages a shifting set of areas, so its operation can be slow, and address-dependent data cannot be stored in it. However, it has the following advantages:

- it doesn't waste memory by fragmenting free space
- it returns deallocated memory to RISC OS for use by other applications.

Allocation of malloc blocks

All block sizes allocated are in bytes and are rounded up to a multiple of four bytes. All blocks returned to the user are word-aligned. All blocks have an overhead of eight bytes (two words). One word is used to hold the block's length and status, the other contains a guard constant which is used to detect heap corruptions. The guard word may not be present in future releases of the ANSI C library. When the stack needs to be extended, blocks are allocated from the `malloc` heap.

When an allocation request is received by the storage manager, it is categorised into one of three sizes of blocks

- small $0 \rightarrow 64$
- medium $65 \rightarrow 512$
- large $513 \rightarrow 16777216$.

The storage manager keeps track of the free sections of the heap in two ways. The medium and large sized blocks are chained together into a linked list (overflow list) and small blocks of the same size are chained together into linked lists (bins). The overflow list is ordered by ascending block address, while the bins have the most recently freed block at the start of the list.

When a small block is requested, the bin which contains the blocks of the required size is checked, and, if the bin is not empty, the first block in the list is returned to the user. If there was not a block of the exact size available, the bin containing blocks of the next size up is checked, and so on until a block is found. If a block is not found in the bins, the last block (highest address) on the overflow list is taken. If the block is large enough to be split into two blocks, and the remainder is a usable size (> 12 including the overhead) then the block is split, the top section returned to the user and the remainder, depending on its size, is either put in the relevant bin at the front of the list or left in the overflow list.

When a medium block is requested, the search ignores the bins and starts with the overflow list. This is searched in reverse order for a block of usable size, in the same way as for small blocks.

When a large block is requested, the overflow list is searched in increasing address order, and the first block in the list which is large enough is taken. If the block is large enough to be split into two blocks, and the size of the remainder is larger than a small block (> 64) then the block is split, the top section is returned to the overflow list, and bottom section given to the user.

Should there not be a block of the right size available, the C storage manager has two options:

1. Take all the free blocks on the heap and join adjacent free blocks together (coalescing) in the hope that a block of the right size will be created which can then be used
2. Ask the operating system for more heap, put the block returned in the overflow list, and try again.

The heap will only be coalesced if there is at least enough free memory in it to make it worthwhile (ie four times the size of the requested block, and at least one sixth of the total heap size) or if the request for more heap was denied. Coalescing causes the following:

- the bins and overflow list are emptied;
- the heap is scanned;
- adjacent free blocks are merged;
- the free blocks are scattered into the bins and overflow list in increasing address order.

Deallocation of malloc blocks

When a block is freed, if it will fit in a bin then it is put at the start of the relevant bin list, otherwise it is just marked as being free and effectively taken out of the heap until the next coalesce phase, when it will be put in the overflow list. This is done because the overflow list is in ascending block address order, and it would have to be scanned to be able to insert the freed block at the correct position. Fragmentation is also reduced if the block is not reusable until after the next coalesce phase. It is worth noting that deallocating a block and then reallocating a block of the same size can not be relied upon to deliver the original block.

Reallocation of malloc blocks

You should be cautious when using `realloc`. Reallocating a block to a larger size will usually require another block of memory to be used and the data to be copied into it. This means that you cannot use the whole of the heap as both blocks need to be present at the same time.

If consecutive calls keep increasing the block size until all memory is used up, then only about a third of the heap is likely to be available in one block. A typical course of events is:

1. The first block is present (block A).
2. It is extended to a larger sized block (block B). Block A must still be present (see above).
3. It is again extended to a larger sized block (block C). Block B must still be present (see above). However, block A also still exists because it is too small to use, and cannot be coalesced with another block because block B is in the way.

Wimp slots and the C flex system

A typical C application running under the Wimp has a single contiguous application area (wimp slot) into which are placed the following:

- program image
- static data
- stack
- `malloc` data.

The initial wimp slot size is set by the size of the Next slot (in the Task display window) when the application is started, or by *WimpSlot commands in the !Run file associated with the C application. If the `malloc` heap is full, and the flex system has not been initialised and the operating system has free memory, the wimp slot grows, raising its highest address. Once enlarged by `malloc`, the wimp slot never reduces again until program termination.

The stack is allocated on the heap, in 4K (or as big as needed) chunks: the ARM procedure call standard means that disjoint extension of the stack is possible. The only other use that the ANSI library makes of the `malloc` heap is in allocating file buffers, but even this usage can be prevented by making the appropriate calls to the ANSI library buffer handling facilities (`setvbuf`). The operation of the `malloc` heap is described above and is designed to provide good performance under heavy use. Its design is such that small blocks can be allocated and freed rapidly.

Any `malloc` heap tends to fragment over time. This is particularly serious in the following circumstances:

- no virtual memory
- multitasking – if memory is not in use, it should be handed to other applications
- if a program runs out of memory it must not crash, but must recover and continue.

These are just the conditions under which a desktop application operates!

Because of this, the flex facilities are available as part of RISC_OSLib (the RISC OS-specific C library provided with Acorn Desktop C). These provide a shifting heap, intended for the allocation of large blocks of memory which might otherwise destroy the structure of a `malloc`-style heap.

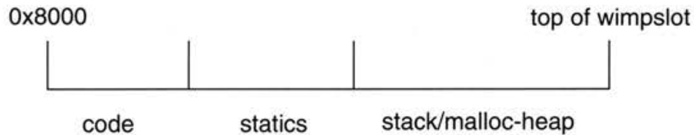
Flex works by increasing the size of the application area, using space above that reserved for use by `malloc`. Once the flex system is initialised the `malloc` heap cannot grow, unless you enable this (see later). The benefits of using flex can be seen in Draw, Paint and Edit, which are all written in C using early versions of RISC_OSLib. Their application areas expand when new files are added, contract when files are discarded, and do not suffer from needless incremental application area growth over time.

The implementation of flex is quite simple. There is no free list as memory is shifted whenever a block is destroyed or changed in size. New blocks are always allocated at the top. When blocks are deallocated or resized, those above are moved. This means that deallocating or changing the size of a block can take quite a long time (proportional to the sum of the sizes of the blocks above it in memory). Flex is also not recommended for allocation of small blocks. Its other limitation is that as flex blocks can be shifted, you should not use them for address-dependent data (eg pointers or indirected icon data).

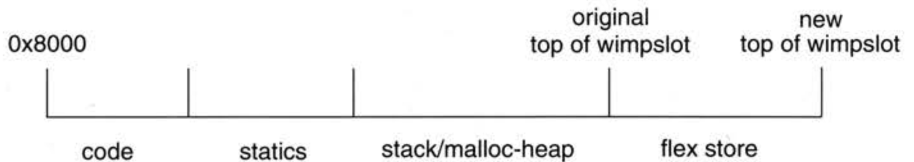
In addition to the facilities described above, RISC_OSLib also provides an obsolete `malloc`-like allocator of non-shifting blocks called heap.

Two facilities are provided, because no one storage manager can solve all problems in the absence of Virtual Memory. A program which works adequately with `malloc` should feel no compulsion to use anything else. The use of flex, however, particularly in desktop applications such as editors (which are likely to be resident on the desktop for a long period of time) can go a long way towards improving their memory usage.

The model of a C application's memory layout is as follows:



If the application uses flex store as supported by RISC_OSLib, the model is:



To expand the `malloc` heap when a flex store area is being used the flex area has to be moved. To achieve this, `malloc` calls a flex function to move the flex blocks. The flex function called is registered with the C library, and may be a dummy function which does not move flex. If a dummy function is registered or flex cannot be successfully moved, then `malloc` itself returns a `0` to indicate failure.

The Acorn Desktop C version of RISC_OSLib registers a dummy flex-moving function during `flex_init()`, inhibiting `malloc` heap expansion after `flex_init()` has been called. This is registered with a call to the function `_kernel_register_slotextend()`.

A functional flex-moving function performs the relocation, sets a pointer to the newly available space, and returns the size of the memory thus obtained (which may be less than that requested by `malloc`).

Allowing `malloc` heap expansion to move flex makes the use of pointers into flex blocks potentially hazardous when the pointers are set before, but used after, the following:

- calls to `flex_alloc`, `flex_free`, `flex_extend`
- calls to `malloc` and `_kernel_alloc`

- calls to any functions which may cause stack extension (since stack extension uses the malloc-heap for this purpose)

Consider the following code fragment:

```
#define FLEX_SIZE 1024 /* for example */
#define OFFSET    42   /* for example */

static void nonleaf_function(char *p)
{
    /* declaration of local vars, and calls to other functions here */
    /* use of p happens here ... */
}

static void access_flex_store(void)
{
    char *message;

    flex_alloc((flex_ptr)&message, FLEX_SIZE);
    nonleaf_function(message+OFFSET);
}
```

Notice that when the value of the char pointer `message+OFFSET` is passed by value to the function `nonleaf_function()`, use of `p` in this function may no longer be valid, since stack extension may have happened during the function call, which may have caused the allocated flex store to move.

Working in this Environment

1. If you have an existing binary, linked with a version of stubs pre-dating the 3.1b intermediate release, such as that included with ANSI C Release 3, then you do not get an extending wimpslot, and hence no new problems arise (the shared C library 'knows' which stubs the application was linked with). You must make your initial wimpslot large enough to accommodate your stack/heap needs. This is important for old applications which rely on `malloc` returning 0 when the application's initial wimpslot is exhausted.
2. If you link with the Acorn Desktop C version of stubs, but do not use the flex functions in RISC_OSLib, you get a wimpslot extendable by `malloc`, and have no new problems. When more heap is required your wimpslot may be increased by the C library (but will not shrink when `free()` is called).
3. If you link with the Acorn Desktop C version of stubs, and use the flex functions in RISC_OSLib, then your malloc-heap will (by default) not be allowed to grow. You must make your initial wimpslot large enough to accommodate your stack/heap needs.

   Note: `flex_init()` makes the call:

   ```
   _kernel_register_slotextend(flex_dont_budge);
   ```

This means that when the C library attempts to acquire more wimpslot, the extension will fail. This gives you the guarantee that flex store will only be relocated due to `flex_alloc`, `flex_extend`, and `flex_free`. Your wimpslot will grow or shrink to satisfy flex requests, but your malloc-heap will have a bound fixed by the size of your initial wimpslot.

4 If you link with the Acorn Desktop C version of stubs, and use the flex functions in RISC_OSLib, and require malloc to extend the application's wimpslot, you must be prepared to exist in a world where flex store may move as described in the section above.

After calling `flex_init()`, you can make the call:

```
_kernel_register_slotextend(flex_budge);
```

This registers a function which will relocate flex store whenever the C library needs to grow its malloc-heap.

If you choose to do this, then the following guidelines will be of use to you:

- Always pass `flex_ptr`'s (`void **`'s) to your own functions, with an integral offset.

  Avoid passing direct flex block pointers.
- Direct calls to `malloc` may cause the flex store to move in the same way that calls to `flex_alloc`, `flex_extend` and `flex_free` do.
- You can safely make SWI calls which require pointer arguments where these arguments point into flex blocks, by using `_kernel_swi()`, since `_kernel_swi` **cannot** cause stack extension. This state **must** be guaranteed by the C library, since `flex_budge()` uses `_kernel_swi()` and may be called during stack extension.
- Using the Acorn Desktop C version of RISC_OSLib, you can also call any SWI 'veneer' functions, with the knowledge that the stack will not be extended. These functions have been compiled with stack checking turned off. The functions (which are all in RISC_OSLib) are:

  bbc.h
  colourtran.h
  drawmod.h
  font.h
  os.h
  print.h
  sprite.h
  visdelay.h
  wimp.h

- You can turn stack checking off in your own code using pragmas, thus:

```
#pragma no_stack_checks
/* functions defined after here are compiled without stack checks */
#pragma stack_checks
/* functions defined after here are compiled with stack checks */
```

  Or for a whole source file by compiling using the flag `-zps1`

  Note that functions which are compiled with stack checking off have only 512 bytes of stack available to them, and any 'non-stack-check' functions which they call.

- You can toggle whether the malloc-heap is permitted to extend, using calls to `_kernel_register_slotextend()` with arguments `flex_budge or flex_dont_budge`. This can be used to surround critical regions of code, where you may wish to temporarily stop flex blocks moving due to malloc-heap extension.

  You can set the root stack segment size using:

```
int __root_stack_size = 16*1024; /* to get a 16kb stack size */
```

Using heap_alloc and heap_free

Since when malloc heap expansion is inhibited (as it is by default with the Acorn Desktop C version of flex) the bottom flex block is static, it is valid to retain pointers into it, and useful to manage a malloc style heap of fixed blocks within it. The `heap_alloc()` and `heap_free()` functions provide facilities to perform this.

Using the `heap` functions to do memory allocation is similar to `malloc()` in that a pointer to the block allocated is returned to the caller: the routine to do this is called `heap_alloc()`. Memory may be released with `heap_free()`. Before you use heap, you must call `heap_init()`; if `heap_init()` is called with a non-zero parameter, then the heap will be shrunk when it is possible to do so after a call to `heap_free()`. The call to `heap_init()` must be made after flex has been initialised with `flex_init()`. Since the heap functions support a heap in the first flex block allocated, `heap_init` must be called before any calls to flex allocation functions, and you must **not** allow the C heap to extend thus causing all flex blocks to be relocated (ie you must not have registered `flex_budge` with `_kernel_slot_extend()`).

Using memory from relocatable modules

Relocatable modules should use memory from three sources: the supervisor stack; the RMA; and application workspace. Use of pc-relative written data should be avoided as it makes a module unsuitable to ROM, unsuitable for multiple instantiation, and permanently reserves space, possibly only for occasional use.

The supervisor stack is small and not extendable, so care must be taken to use this resource very economically.

The RMA is the standard source of workspace for any of the non-user mode routines contained in a module, as described in the RISC OS *Programmer's Reference Manual*. Care must be taken to deallocate unwanted blocks - the marker word hint described earlier in this chapter may be useful. C malloc uses RMA when called from non-user mode.

Application workspace only belongs to a module when referenced from module user mode code running as the sole current application (with RISC OS desktop multitasking halted) or when running as a RISC OS application having dealt with the `Service_Memory (&11)` service call (sent round by the wimp when your program issues `SWI Wimp_Initialise`) to keep application workspace.

Never access your application's workspace from an interrupt routine. During interrupts, the state of the application area is effectively random. Since your interrupt routine could execute at any time, it could happen while some other application is switched in. If this did happen, and the interrupt routine updated application space, then some other application could be affected. To get around this problem, allocate some RMA space for your interrupt routine to use when it needs to; this memory will be visible when your application is running. Remember to free up the RMA space when you've finished with it.

Part 4 - Appendices

15 Appendix A - Error messages

- `ADRL can't be used with PC`
  The destination register of an ADRL opcode cannot be PC.
- `Area directive missing`
  An attempt has been made to generate code or data before the first AREA directive.
- `Area name missing`
  The name for the area has been omitted from an AREA directive.
- `Bad alignment boundary`
  An alignment has been given which is not a power of two.
- `Bad area attribute or alignment`
  Unknown attribute or alignment not in the range 2-12.
- `Bad based number`
  A digit has been given in a based number which is not less than the base, for example: 7_8.
- `Bad exported name`
  The wording following the EXPORT directive is syntactically not a name.
- `Bad exported symbol type`
  The exported symbol is not a program-relative symbol.
- `Bad expression type`
  For example, a number was expected but a string was encountered.
- `Bad floating point constant`
  The only allowed floating point constants are 0, 1, 2, 3, 5, 10 and 0.5. They must be written in exactly these forms.
- `Bad global name`
  An incorrect character appears in the global name.
- `Bad hexadecimal number`
  The & introducing a hexadecimal number is not followed by a valid hexadecimal digit.
- `Bad imported name`
  The wording following the IMPORT directive is syntactically not a name.
- `Bad local label number`
  A local label number must be in the range 0-99.

- `Bad local name`
  An incorrect character appears in the local name.
- `Bad macro parameter default value`
- `Bad opcode symbol`
  A symbol has been encountered in the opcode field which is not a directive and is syntactically not a label.
- `Bad operand type`
  For example, a logical value was supplied where a string was required.
- `Bad operator`
  The name between colons is not an operator name.
- `Bad or unknown attribute`
  Faulty attribute on an IMPORT directive.
- `Bad register list symbol`
  An expression used as a register set definition (eg in LDM or STM) was not understood or of the wrong type.
- `Bad register name symbol`
  A register name is wrong. Note that all register names must be defined using the RN directive.
- `Bad register range`
  A register range from a higher to a lower register has been given; for example, R4-R2 has been typed.
- `Bad rotator`
  The rotator value supplied must be even and in the range 0-30.
- `Bad shift name`
  Syntax error in shift name.
- `Bad string escape sequence`
  A C style escape character sequence (beginning with '\') within a string was incorrect.
- `Bad symbol`
  Syntax error in a symbol name.
- `Bad symbol type`
  This will occur after a # or * directive and means that the symbol being defined has already been assumed to be of a type which cannot be defined in this way.
- `Branch offset out of range`
  The destination of a branch is not within the ARM address space.
- `Code generated in data area`
  An opcode has been found in an area which is not a code area.
- `Coprocessor number out of range`

- `Coprocessor operation out of range`
- `Coprocessor register number out of range`
- `Data transfer offset out of range`
  The immediate value in a data transfer opcode must be in the range:
  -4095 <= e <= +4095
- `Decimal overflow`
  The number exceeds 32 bits.
- `Division by zero`
- `Entry address already set`
  This is the second or subsequent ENTRY directive.
- `Error in macro parameters`
  The macro parameters do not match the prototype statement in some way.
- `Error on code file`
  An error occurred while writing the output file.
- `External area relocatable symbol used`
  A symbol which is an address in another area has been used in a non-trivial expression.
- `Externals not valid in expressions`
  An imported symbol has been used in a non-trivial expression.
- `Floating point register number out of range`
- `Floating point overflow`
- `Floating point number not found`
- `Global name already exists`
  This name has already been used in some other context.
- `Hexadecimal overflow`
  The number exceeds 32 bits.
- `Illegal combination of code and zero initialised`
  An object file area cannot be declared both to be code and zero initialised data.
- `Illegal label parameter start in macro prototype`
- `Illegal line start should be blank`
  A label has been found at the start of a line with a directive which cannot be labelled.
- `Immediate value out of range`
  An immediate value in a data processing instruction cannot be obtained by rotating an 8-bit value by an even amount.

- `Imported name already exists`
  The name has already been defined or used for something else.
- `Incorrect routine name`
  The optional name following a branch to a local label or on a local label definition does not match the routine's name.
- `Invalid line start`
  A line may only start with a letter character (the first letter of a label), a digit (the first character of a local label), a semi-colon or a space.
- `Invalid operand to branch instruction`
- `Label missing from line start`
  The absence of a label where one is required; for example, in the * directive.
- `Local name already exists`
  A local name has been defined more than once.
- `Locals not allowed outside macros`
  A local variable has been defined in the main body of the source file.
- `MEND not allowed within conditionals`
  A MEND has been found amongst [|] or WHILE/WEND directives.
- `Missing close bracket`
  A missing close bracket or too many opening brackets.
- `Missing close quote`
  No closing quote at the end of a string constant.
- `Missing close square bracket`
  A] is absent.
- `Missing comma`
  Syntax error due to missing comma.
- `Missing hash`
  The hash (#) preceding an immediate value has been forgotten.
- `Missing open bracket`
  A missing open bracket or too many closing brackets.
- `Missing open square bracket`
- `Multiply or incompatibly defined symbol`
  A symbol has been defined more than once.
- `Multiply destination equals first source`
- `No current macro expansion`
  A MEND, MEXIT or local variable has been encountered but there is no corresponding MACRO.
- `Non-zero data within uninitialised area`

- `Numeric overflow`
  The number exceeds 32 bits.
- `Origin illegal for a.out`
  Unix style source or AOUT directive resulted in the assembler producing a.out unix style output, but this does not support fixed origins.
- `Register occurs multiply in LDM/STM list`
- `Register symbol already defined`
  A register symbol has been defined more than once.
- `Register value out of range`
  Register values must be in the range 0-15.
- `Shift option out of range`
  The range permitted is 0-31, 1-32 or 1-31 depending on the shift type.
- `String overflow`
  Concatenation has produced a string of more than 256 characters.
- `String too short for operation`
  An attempt has been made to manipulate a string using :LEFT: or :RIGHT: which has insufficient characters in it.
- `STRONG directive not supported by a.out`
  Unix style source or AOUT directive resulted in the assembler producing a.out unix style output, but this does not support STRONG.
- `Structure mismatch`
  Mismatch of | with | or |, or WEND and WHILE.
- `Substituted line too long`
  During variable and macro parameter substitution the line length has exceeded 256 characters.
- `Symbol missing`
  An attempt has been made to reference the length attribute of a symbol but the symbol was omitted or the name found was not recognised as a symbol.
- `Syntax error following directive`
  An operand has been provided to a directive which cannot take one, for example: the '|' directive.
- `Syntax error following label`
  A label can only be followed by spaces, a semi-colon or the end-of-line symbol.
- `Syntax error following local label definition`
  A space, comment, or end-of-line did not immediately follow the local label.
- `Too late to change output format`
  AOF or AOUT directives incorrectly placed.

- `Too late to define symbol as register list`
  A register list was defined for a symbol already used for another purpose.
- Too late to ban floating point
- `Too late to set origin now`
  The ORG must be set before the Assembler generates code.
- `Too many actual parameters`
  A macro call is trying to pass too many parameters.
- `Too many bss areas for a.out`
- `Too many code areas for a.out`
- `Too many data areas for a.out`
  Unix style source or AOUT directives resulted in the assembler producing a.out unix style output, but this only supports one bss/code/data area.
- `Translate not allowed in pre-indexed form`
  The translate option may not be specified in pre-indexed forms of LDR and STR.
- `Unable to close code file`
- `Unable to open code file`
- `Undefined exported symbol`
  The symbol exported is undefined.
- `Undefined symbol`
  A symbol has not been given a value.
- `Unexpected characters at end of line`
  The line is syntactically complete, but more information is present. The semi-colon prefixing comments may have been omitted.
- `Unexpected operand`
  An operand has been found where a binary operator was expected.
- `Unexpected operator`
  A non-unary operator has been found where an operand was expected.
- `Unexpected unary operator`
  A unary operator has been found where a binary operator was expected.
- `Unknown opcode`
  A name in the opcode field has been found which is not an opcode, a directive, nor a macro.
- `Unknown operand`
  An operand in the bracketed format {PC} {VAR} {OPT} {TRUE} {FALSE} is not of the correct form.

- `Unknown or wrong type of global/local symbol`
  Type mismatch, for example, attempting to set or reset the value of a local or global symbol as logical, where it is a string variable.
- `Unknown shift name`
  Not one of the six legal shift mnemonics.
- `Weak symbols not permitted in a.out`
  Unix style source or AOUT directive resulted in the assembler producing a.out unix style output, but this does not support WEAK.

16 Appendix B - Directives syntax table

The acceptable syntax for the various directives is shown in the following table:

<table>
<tr><td>!</td><td>no label</td><td>two expressions are expected</td></tr>
<tr><td>#</td><td>optional label</td><td>an expression is expected</td></tr>
<tr><td>& (DCD)</td><td>optional label</td><td>an expression list is expected</td></tr>
<tr><td>%</td><td>optional label</td><td>an expression is expected</td></tr>
<tr><td>* (EQU)</td><td>label</td><td>an expression is expected</td></tr>
<tr><td>= (DCB)</td><td>optional label</td><td>an expression list is expected</td></tr>
<tr><td>[</td><td>no label</td><td>an expression is expected</td></tr>
<tr><td>]</td><td>no label</td><td>takes no expression</td></tr>
<tr><td>|</td><td>no label</td><td>takes no expression</td></tr>
<tr><td>^</td><td>no label</td><td>expression and optional register expected</td></tr>
<tr><td>ALIGN</td><td>no label</td><td>one or two expressions are expected</td></tr>
<tr><td>AOF</td><td>no label</td><td>takes no expression</td></tr>
<tr><td>AOUT</td><td>no label</td><td>takes no expression</td></tr>
<tr><td>ASSERT</td><td>no label</td><td>an expression is expected</td></tr>
<tr><td>CN</td><td>label</td><td>an expression is expected</td></tr>
<tr><td>CP</td><td>label</td><td>an expression is expected</td></tr>
<tr><td>DCFD</td><td>label</td><td>floating point expression list</td></tr>
<tr><td>DCFS</td><td>label</td><td>floating point expression list</td></tr>
<tr><td>DCW</td><td>optional label</td><td>an expression list is expected</td></tr>
<tr><td>END</td><td>no checking performed</td><td></td></tr>
<tr><td>FN</td><td>label</td><td>an expression is expected</td></tr>
<tr><td>GBLA</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>GBLL</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>GBLS</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>GET</td><td>no label</td><td>a filename is expected</td></tr>
<tr><td>LCLA</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>LCLL</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>LCLS</td><td>no label</td><td>a symbol is expected</td></tr>
<tr><td>LEADR</td><td>no label</td><td>an expression is expected</td></tr>
<tr><td>LNK</td><td>no checking performed</td><td>a filename is expected</td></tr>
<tr><td>LTORG</td><td>no label</td><td>takes no expression</td></tr>
</table>

MACRO	no label	takes no expression
MEND	no label	takes no expression
MEXIT	no label	takes no expression
NOFP	label	takes no expression
OPT	no label	an expression is expected
ORG	no label	an expression is expected
RLIST	label	a register list expression is expected
RN	label	an expression is expected
ROUT	label	takes no expression
SETA	variable	an expression is expected
SETL	variable	an expression is expected
SETS	variable	an expression is expected
SUBT	no label	takes an optional title
TTL	no label	takes an optional title
WEND	no label	takes no expression
WHILE	no label	an expression is expected

17 Appendix C - Example assembler fragments

The following example assembly language fragments show ways in which the basic ARM instructions can combine to give efficient code. None of the techniques illustrated save a great deal of execution time (although they all save some), mostly they just save code.

Note that, when optimising code for execution speed, consideration to different hardware bases should be given. Some changes which optimise speed on one machine may slow the code on another. An example is unrolling loops (eg divide loops) which speeds execution on an ARM2, but can slow execution on an ARM3, which has a cache.

Using the conditional instructions

Using conditionals for logical OR

```
CMP     Rn,#p       ;IF Rn=p OR Rm=q THEN GOTO Label
BEQ     Label
CMP     Rm,#q
BEQ     Label
```

can be replaced by:

```
CMP     Rn,#p
CMPNE   Rm,#q       ;if condition not satisfied try
BEQ     Label       ;another test.
```

Absolute value

```
TEQ     Rn,#0       ;test sign
RSBMI   Rn,Rn,#0    ;and 2's complement if necessary.
```

Combining discrete and range tests

```
TEQ     Rc,#127     ;discrete test
CMPNE   Rc,#" "-1   ;range test
MOVLS   Rc,#"."     ;IF Rc<#" " OR Rc=CHR$127 THEN Rc:="."
```

Division and remainder

```
;enter with dividend in Ra, divisor in Rb.
;Divisor must not be zero.
      MOV    Rd,Rb         ;Put the divisor in Rd.
      CMP    Rd,Ra,LSR #1  ;Then double it until
Div1  MOVLS  Rd,Rd,LSL #1  ; 2 * Rd > divisor.
      CMP    Rd,Ra,LSR #1
      BLS    Div1
      MOV    Rc,#0         ;Initialise the quotient
Div2  CMP    Ra,Rd         ;Can we subtract Rd?
      SUBCS  Ra,Ra,Rd      ;If we can, do so
      ADC    Rc,Rc,Rc      ;Double quotient and add new bit
      MOV    Rd,Rd,LSR #1  ;Halve Rd.
      CMP    Rd,Rb         ;And loop until we've gone
      BHS    Div2          ; past the original divisor,
;Now Ra holds remainder, Rb holds original divisor,
; Rc holds quotient and Rd holds junk.
```

Pseudo-random binary sequence generator

It is often necessary to generate (pseudo-) random numbers, and the most efficient algorithms are based on shift generators with a feedback rather like a cyclic redundancy check generator. Unfortunately, the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (that is, 2^32-1 cycles before repetition). A 33-bit shift generator with taps at bits 20 and 33 is required.

The basic algorithm is:

- newbit:=bit33 eor bit20
- shift left the 33 bit number
- put in newbit at the bottom.
- Repeat for all the 32 newbits needed.

All this can be done in five S cycles:

```
;enter with seed in Ra (32 bits),Rb (1 bit in Rb lsb)
;uses Rc
      TST    Rb,Rb,LSR #1      ;top bit into carry
      MOVS   Rc,Ra,RRX         ;33 bit rotate right
      ADC    Rb,Rb,Rb          ;carry into lsb of Rb
      EOR    Rc,Rc,Ra,LSL#12   ;(involved!)
      EOR    Ra,Rc,Rc,LSR#20   ;(similarly involved!)
;new seed in Ra, Rb as before
```

Multiplication by a constant

Multiplication by 2^n (1,2,4,8,16,32..)

```
MOV    Ra,Ra,LSL #n;
```

Multiplication by 2^n+1 (3,5,9,17..)

```
ADD    Ra,Ra,Ra,LSL #n.
```

Multiplication by 2^n-1 (3,7,15..)

```
RSB    Ra,Ra,Ra,LSL #n
```

Multiplication by 6

```
ADD    Ra,Ra,Ra,LSL #1  ;multiply by 3
MOV    Ra,Ra,LSL #1     ;and then by 2.
```

Multiply by 10 and add in extra number

```
AD     Ra,Ra,Ra,LSL #2  ;multiply by 5
ADD    Ra,Rc,Ra,LSL #1  ;multiply by 2 and add in next digit
```

General recursive method for Rb := Ra*C, C a constant

If C even, say C = 2^n*D, D odd:

```
D=1 :   MOV    Rb,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        MOV    Rb,Rb,LSL #n
```

If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

```
D=1 :   ADD    Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        ADD    Rb,Ra,Rb,LSL #n.
```

If C MOD 4 = 3, say C = 2^n*D-1, D odd, n>1:

```
D=1 :   RSB    Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        RSB    Rb,Ra,Rb,LSL #n.
```

This is not quite optimal, but close. An example of its non-optimal use is multiply by 45 which is done by:

```
RSB   Rb,Ra,Ra,LSL #2  ;multiply by 3
RSB   Rb,Ra,Rb,LSL #2  ;multiply by 4*3-1 = 11
ADD   Rb,Ra,Rb,LSL #2  ;multiply by 4*11+1 = 45
```

rather than by:

```
ADD   Rb,Ra,Ra,LSL #3  ;multiply by 9
ADD   Rb,Rb,Rb,LSL #2  ;multiply by 5*9 = 45
```

Loading a word from an unknown alignment

There is no instruction to load a word from an unknown alignment. To do this requires some code (which can be a macro) along the following lines:

```
;enter with 32-bit address in Ra
;uses Rb, Rc; result in Rd
;Note d must be less than c

BIC     Rb,Ra,#3           ;get word-aligned address
LDMIA   Rb,{Rd,Rc}         ;get 64 bits containing answer
AND     Rb,Ra,#3           ;correction factor in bytes
MOVS    Rb,Rb,LSL #3       ;..now in bits and test if aligned
MOVNE   Rd,Rd,LSR Rb       ;produce bottom of result
                           ;word if not aligned
RSBNE   Rb,Rb,#32          ;get other shift amount
ORRNE   Rd,Rd,Rc,LSL Rb    ;combine two halves to get result
```

Sign/zero extension of a half word

```
MOV     Ra,Ra,LSL #16      ;move to top
MOV     Ra,Ra,LSR #16      ;and back to bottom
                           ;use ASR to get
                           ;sign extended version
```

Return setting condition codes

```
CFLAG   *       &20000000
        BICS    PC,R14,#CFLAG   ;returns clearing C flag
                                ;from link register
        ORRCCS  PC,R14,#CFLAG   ;conditionally returns
                                ;setting C flag
```

This code should not be used except in user mode, since it will reset the interrupt mode to the state which existed when the R14 was set up. This rule generally applies to non-user mode programming. For example in supervisor mode:

```
MOV     PC,R14
```

is safer than

```
MOVS    PC,R14
```

However, note that MOVS PC,R14 is required by the ARM Procedure Call Standard, used by code compiled from the high level languages C, Fortran 77, ISO-Pascal and so on. Such code, of course, runs in user mode.

Full multiply

The ARM's multiply instruction multiplies two 32-bit numbers together and produces the least significant 32 bits of the result. These 32 bits are the same regardless of whether the numbers are signed or unsigned.

To produce the full 64 bits of a product of two unsigned 32-bit numbers, the following code can be used:

```
;Enter with two unsigned numbers in Ra and Rb.
   MOVS    Rd,Ra,LSR #16        ;Rd is ms 16 bits of Ra
   BIC     Ra,Ra,Rd,LSL #16     ;Ra is ls 16 bits
   MOV     Re,Rb,LSR #16        ;Re is ms 16 bits of Rb
   BIC     Rb,Rb,Re,LSL #16     ;Rb is ls 16 bits
   MUL     Rc,RA,Rb             ;Low partial product
   MUL     Rb,Rd,Rb             ;First middle partial product
   MUL     Ra,Re,Ra             ;Second middle partial product
   MULNE   Rd,Re,Rd             ;High partial product - NE
                                ; condition reduces time taken
                                ; if Rd is zero
   ADDS    Ra,Ra,Rb             ;Add middle partial products -
                                ; could not use MLA because we
                                ; need carry
   ADDCS   Rd,Rd, #&10000       ;Add carry into high partial
                                ; product
   ADDS    Rc,Rc,Ra,LSL #16     ;Add middle partial product
   ADC     Rd,Rd,Ra,LSR #16     ; sum into low and high words
                                ; of result
;Now Rc holds the low word of the product, Rd its high
; word, and Ra, Rb and Re hold junk.
```

18 Appendix D - ARM datasheet

This appendix contains relevant extracts from the Acorn datasheet for the ARM2 microprocessor. It is included as a reference document. The *Programmers' Model* and *Instruction Set* sections are also accurate for the ARM3 microprocessor, except that this chip also supports the SWP instruction.

If further hardware detail is required refer to THE VL86C010 32-BIT RISC MPU AND PERIPHERALS USERS MANUAL (VLSI Technology, Inc., 1989).

The ARM (Advanced RISC Machine) is a general purpose 32-bit single-chip microprocessor. The architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are greatly simplified compared with microprogrammed Complex Instruction Set Computers. This simplification results in a high instruction throughput and a good real-time interrupt response from a small and cost-effective chip.

The instruction set comprises nine basic instruction types. Two of these make use of the on-chip arithmetic logic unit (ALU), barrel shifter and multiplier to perform high-speed operations on the data in a bank of 27 registers, each 32 bits wide. Two instruction types control the transfer of data between main memory and the register bank, one optimised for flexibility of addressing and the other for rapid context switching. Two instructions control the flow and privilege level of execution, and the remaining three types are dedicated to the control of external Co-Processors which allow the functionality of the instruction set to be extended off-chip in an open and uniform way.

The ARM instruction set has proved to be a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors which depend on sophisticated compiler technology to manage complicated instruction interdependencies.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The memory interface has been designed to allow the performance potential to be realised without incurring high costs in the memory system. Speed critical control signals are pipelined to allow system control functions to be implemented in

standard low-power logic, and these control signals facilitate the exploitation of the fast local access modes offered by industry standard dynamic random access memories (DRAMs).

Features:

- 32-bit data bus
- 26-bit address bus giving a 64-MByte uniform address space
- Support for virtual memory systems
- Simple but powerful instruction set
- Co-Processor interface for instruction set extension
- Good high-level language compiler support
- Peak execution rate of 10 million instructions per second (MIPS)
- Fast interrupt response for real-time applications
- Low power consumption (0.1 W typical) with a single +5 V supply
- 84-pin JEDEC B leadless chip carrier or plastic leaded chip carrier

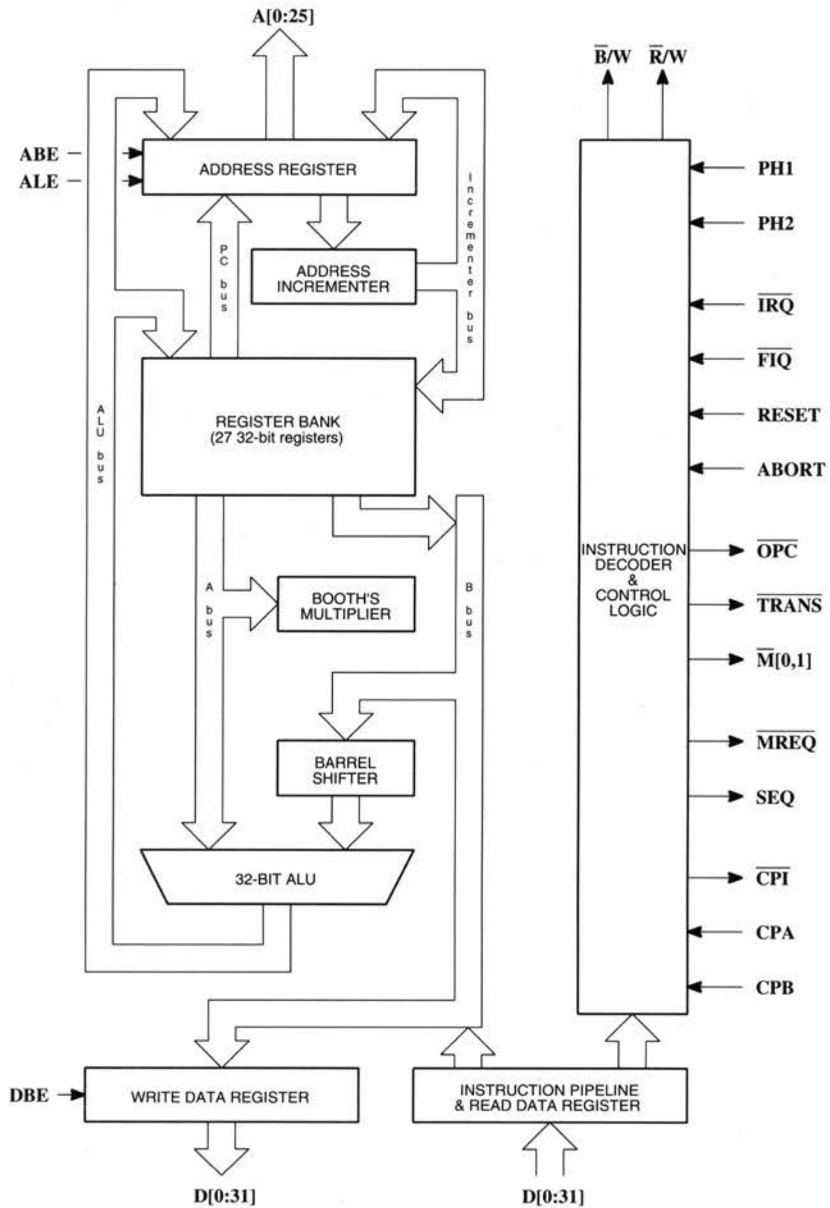


Figure 18.1 Block Diagram

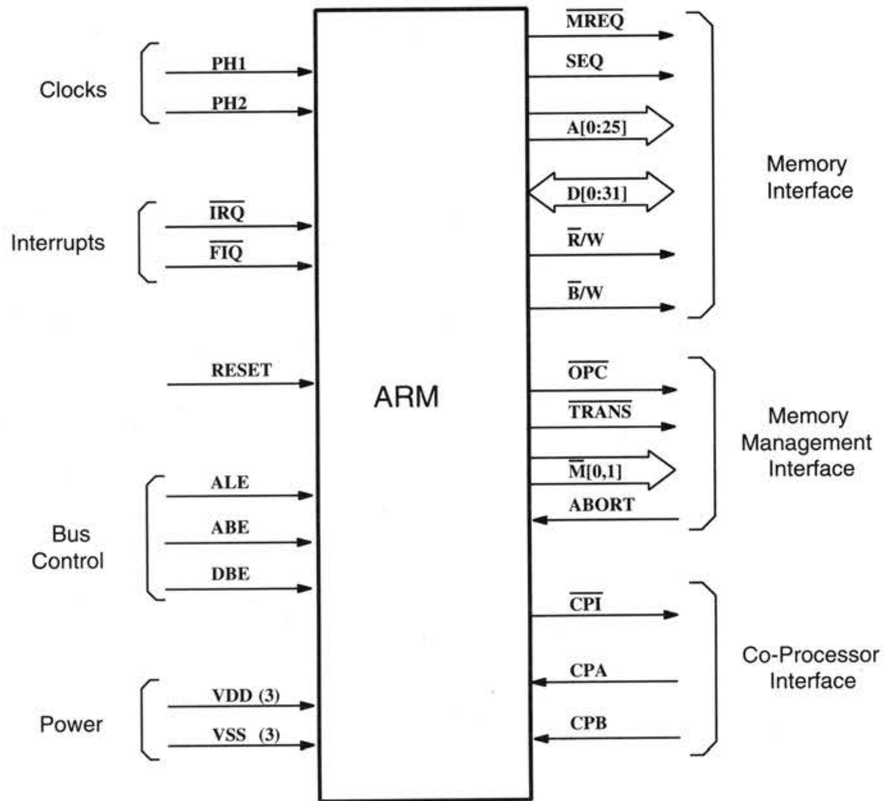


Figure 18.2 *Functional Diagram*

Description of signals

Name	Pin	Type	Description
PH2	1	ICk	Phase two clock.
PH1	2	ICk	Phase one clock.
$\overline{\mathrm{R}}$/W	3	OC	Not read / write. When HIGH this signal indicates a processor write cycle; when LOW, a read cycle. It becomes valid during phase 2 of the cycle before that to which it refers, and remains valid to the end of phase 1 of the referenced cycle.
$\overline{\mathrm{OPC}}$	4	OC	Not op-code fetch. When LOW this signal indicates that the processor is fetching an instruction from memory; when HIGH data (if anything) is being transferred. The signal becomes valid during phase 2 of the previous cycle, remaining valid through phase 1 of the referenced cycle.
$\overline{\mathrm{MREQ}}$	5	OC	Not memory request. This signal, when LOW, indicates that the processor requires memory access during the following cycle. The signal becomes valid during phase 1, remaining valid through phase 2 of the cycle preceding that to which it refers.
ABORT	6	IT	Memory abort. This is an input which allows the memory system to tell the processor that a requested access is not allowed. The signal must be valid before the end of phase 1 of the cycle during which the memory transfer is attempted.
$\overline{\mathrm{IRQ}}$	7	IT	Not interrupt request. This is an asynchronous interrupt request to the processor which causes it to be interrupted if taken LOW when the appropriate enable in the processor is active. The signal is level sensitive and must be held LOW until a suitable response is received from the processor.
$\overline{\mathrm{FIQ}}$	8	IT	Not fast interrupt request. As **$\overline{\mathrm{IRQ}}$**, but with higher priority. May be taken LOW asynchronously to interrupt the processor when the appropriate enable is active.

RESET	9	IT	Reset. This is a level sensitive input signal which is used to start the processor from a known address. A HIGH level will cause the instruction being executed to terminate abnormally. When **RESET** becomes LOW for at least one clock cycle, the processor will re-start from address 0. RESET must remain HIGH for at least two clock cycles, and during the HIGH period the processor will perform dummy instruction fetches with the address incrementing from the point where reset was activated. The address value will overflow to zero if **RESET** is held beyond the maximum address limit.
$\overline{\textbf{TRANS}}$	10	OC	Not memory translate. When this signal is LOW it indicates that the processor is in user mode, or that the supervisor is using a single transfer instruction with the force translate bit active. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator of non-user mode activity.
VDD	11,32,55	PWR	Supply.
VSS	33,54,75	PWR	Supply.
$\overline{\textbf{M}}$**[1,0]**	13,14	OC	Not processor mode. These are output signals which are the inverses of the internal status bits indicating the processor operation mode.
SEQ	15	OC	Sequential address. This is an output signal. It will become HIGH when either: • the address for the next cycle is being generated in the address incrementer, so will be equal to the present address (in bytes) plus 4, *or* • during a cycle which did not use memory ($\overline{\textbf{MREQ}}$ inactive), when the next cycle will use memory and the address is the same as the current address. The signal becomes valid during phase 1 and remains so through phase 2 of the cycle before the cycle whose address it anticipates. It may be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) and/or to by-pass the address translation system.

<table>
<tr><td>ALE</td><td>16</td><td>IT</td><td>Address latch enable. This input to the processor is used to control transparent latches on the address outputs. Normally the addresses change during phase 2 to the value required during the next cycle, but for direct interfacing to ROMs they are required to be stable to the end of phase 2. Taking ALE LOW until the end of phase 2 will ensure that this happens. If the system does not require address lines to be held in this way, ALE may be held permanently HIGH. The ALE latch is dynamic, and ALE should not be held LOW indefinitely.</td></tr>
<tr><td>A[25:0]</td><td>17-31,
34-44</td><td>OCZ</td><td>Addresses. This is the processor address bus. If ALE (address latch enable) is HIGH, the addresses become valid during phase 2 of the cycle before the one to which they refer and remain so during phase 1 of the referenced cycle. Their stable period may be controlled by ALE as described above.</td></tr>
<tr><td>ABE</td><td>45</td><td>IC</td><td>Address bus enable. This is an input signal which, when LOW, puts the address bus drivers into a high impedance state. ABE may be tied HIGH when there is no system requirement to turn off the address drivers.</td></tr>
<tr><td>D[0:31]</td><td>46-53,
56-74,
77-81</td><td>IOTZ</td><td>Data Bus. These are bi-directional signal paths which are used for data transfers between the processor and external memory, as follows:
• during read cycles (when $\overline{\text{R}}$/W = 0), the input data must be valid before the end of phase 2 of the transfer cycle
• during write cycles (when $\overline{\text{R}}$/W = 1), the output data will become valid during phase 1 and remain so throughout phase 2 of the transfer cycle.</td></tr>
<tr><td>DBE</td><td>83</td><td>IT</td><td>Data bus enable. This is an input signal which, when LOW, forces data bus drivers into a high impedance state. (The drivers will always be high impedance except during write cycles, and DBE may be tied HIGH in systems which do not require the data bus for DMA or similar activities.)</td></tr>
</table>

$\overline{\mathbf{B}}$/W	84	OC	Not byte / word. This is an output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. The signal is HIGH for word transfers and LOW for byte transfers and is valid for both read and write cycles. The signal will become valid during phase 2 of the cycle before the one during which the transfer will take place. It will remain stable throughout phase 1 of the transfer cycle.
$\overline{\mathbf{CPI}}$	82	OC	Co-Processor instruction. When ARM executes a Co-Processor instruction, it will take this output LOW and wait for a response from the Co-Processor. The action taken will depend on this response, which the Co-Processor signals on the **CPA** and **CPB** inputs.
CPB	12	IT	Co-Processor busy. A Co-Processor which is capable of performing the operation which ARM is requesting (by asserting **$\overline{\mathbf{CPI}}$**), but cannot commit to starting it immediately, should indicate this by letting **CPB** float HIGH. When the Co-Processor is ready to start it should take **CPB** LOW. ARM samples **CPB** at the end of phase 1 of the cycle when **$\overline{\mathbf{CPI}}$** is LOW.
CPA	76	IT	Co-Processor absent. A Co-Processor which is capable of performing the operation which ARM is requesting (by asserting **$\overline{\mathbf{CPI}}$**) should take **CPA** LOW immediately. If **CPA** is HIGH at the end of phase 1 of the cycle when **$\overline{\mathbf{CPI}}$** is LOW, ARM will abort the Co-Processor handshake and take the undefined instruction trap. If **CPA** is LOW and remains LOW, ARM will busy-wait until **CPB** is LOW and then complete the Co-Processor instruction.

Key to Signal Types

ICk	Unbuffered clock inputs
IT	Input with TTL compatible levels
OC	Output with CMOS compatible levels
OCZ	3-state output with CMOS compatible levels
IOTZ	Bi-directional 3-state input/output with TTL compatible levels
PWR	Power pins

Programmers' Model

Introduction

ARM has a 32 bit data bus and a 26 bit address bus. The data types the processor supports are Bytes (8 bits) and Words (32 bits), where words must be aligned to four byte boundaries. Instructions are exactly one word, and data operations (eg ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

ARM supports four modes of operation, including protected supervisor and interrupt handling modes.

Registers

The processor has 27 32-bit registers, 16 of which are visible to the programmer at any time. The visible subset depends on the processor mode; special registers are switched in to support interrupt and supervisor processing. The register bank organisation is shown in the diagram entitled *Register Organisation* on page 200.

User mode is the normal program execution state; registers R0-15 are directly accessible.

All registers are general purpose and may be used to hold data or address values, except that register R15 contains the Program Counter (PC) and the Processor Status Register (PSR). Special bits in some instructions allow the PC and PSR to be treated together or separately as required. The allocation of bits within R15 is shown in the diagram entitled *The Program Counter* (PC) *and Process Status Register* (PSR) on page 202.

R14 is used as the subroutine Link register, and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. R14_svc, R14_irq and R14_fiq are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within supervisor or interrupt routines.

user mode	svc mode	irq mode	fiq mode
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			
R8			R8_fiq
R9			R9_fiq
R10			R10_fiq
R11			R11_fiq
R12			R12_fiq
R13	R13_svc	R13_irq	R13_fiq
R14	R14_svc	R14_irq	R14_fiq
R15 (PC/PSR)			

Figure 18.3 *Register Organisation*

The FIQ processing state (described in the section entitled *Exceptions* on page 201) has seven private registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ programs will not need to save any registers.

The IRQ processing state has two private registers mapped to R13 and R14 (R13_irq and R14_irq).

Supervisor mode (entered on SWI instructions and other traps) has two private registers mapped to R13 and R14 (R13_svc and R14_svc).

The two private registers allow the IRQ and supervisor modes each to have a private stack pointer and link register. Supervisor and IRQ mode programs are expected to save the User state on their respective stacks and then use the User registers, remembering to restore the User state before returning.

In User mode only the N, Z, C and V bits of the PSR may be changed. The I, F and Mode flags will change only when an exception arises. In supervisor and interrupt modes all flags may be manipulated directly.

Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for instance) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

ARM handles exceptions by making use of the banked registers to save state. The old PC and PSR are copied into the appropriate R14, and the PC and processor mode bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 should be saved onto a stack in main memory before re-enabling the interrupt. When multiple exceptions arise simultaneously a fixed priority determines the order in which they are handled.

FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the $\overline{\textbf{FIQ}}$ pin LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect the processor execution flow. It is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, so that the overhead of context switching is minimised. The FIQ exception may be disabled by setting the F flag in the PSR (but note that this is not possible from user mode). If the F flag is clear ARM checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction. When ARM is FIQed it will:

1. save R15 in R14_fiq;
2. force M0, M1 to FIQ mode and set the F and I bits in the PC word;
3. force the PC to fetch the next instruction from address 1CH.

To return normally from FIQ use `SUBS PC,R14_fiq,#4`. This will resume execution of the interrupted code sequence, and restore the original mode and interrupt enable state.

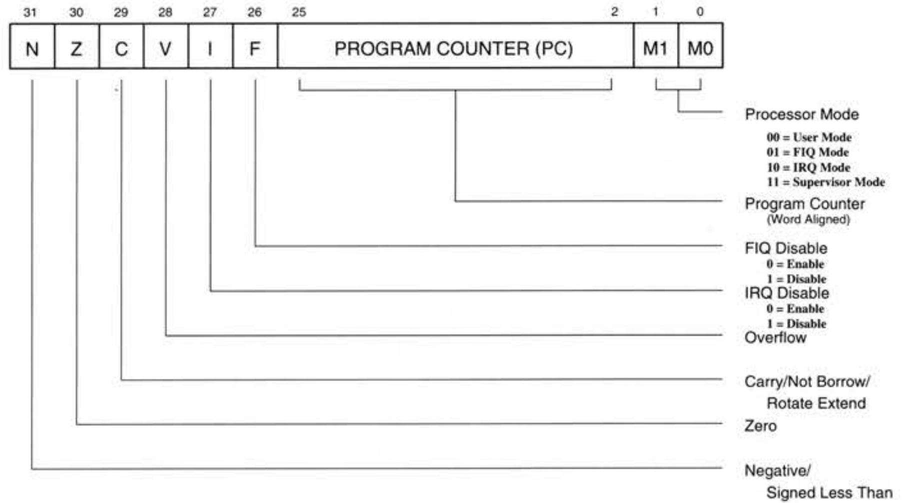


Figure 18.4 *The Program Counter* (PC) *and Process Status Register* (PSR)

IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **$\overline{\text{IRQ}}$** pin. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). If the I flag is clear ARM checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. When successfully IRQed ARM will:

1. save R15 in R14_irq;
2. force M0, M1 to IRQ mode and set the I bit in the PC word;
3. force the PC to fetch the next instruction from address 18H.

To return normally from IRQ use `SUBS PC,R14_irq,#4`. This will restore the original processor state and thereby re-enable IRQ.

Address exception trap

An address exception arises whenever a data transfer is attempted with a calculated address above 3FFFFFFH. The ARM address bus is 26 bits wide, and an address calculation will have a 32-bit result. If this result has a logic "1" in any of the top 6 bits it is assumed that the address overflow is an error, and the address exception trap is taken.

Note that a branch cannot cause an address exception, and a block data transfer instruction which starts in the legal area but increments into the illegal area will not trap. The check is performed only on the address of the first word to be transferred.

When an address exception is seen ARM will:

1. if the data transfer was a store, force it to load. (This protects the memory from spurious writing.)
2. complete the instruction, but prevent internal state changes where possible. The state changes are the same as if the instruction had aborted on the data transfer.
3. save R15 in R14_svc;
4. force M0, M1 to supervisor mode and set the I bit in the PC word;
5. force the PC to fetch the next instruction from address 14H.

Normally an address exception is caused by erroneous code, and it is inappropriate to resume execution. If a return is required from this trap, use `SUBS PC,R14_svc,#4`. This will return to the instruction after the one causing the trap.

Abort

The Abort signal comes from an external Memory Management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. ARM checks for an Abort at the end of the first phase of each bus cycle. When successfully Aborted ARM will respond in one of three ways:

i) if the abort occurred during an instruction prefetch (a *Prefetch Abort*), the prefetched instruction is marked as invalid; when it comes to execution, it is reinterpreted as below. (If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, the abort will have no effect.)

ii) if the abort occurred during a data access (a *Data Abort*), the action depends on the instruction type. Data transfer instructions (LDR, STR) are aborted as though the instruction had not executed. The LDM and STM instructions complete, and if writeback is set, the base is updated. If the instruction would normally have overwritten the base with data (ie LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

iii) if the abort occurred during an internal cycle it is ignored.

Then, in cases (i) and (ii):

1. save R15 in R14_svc;
2. force M0, M1 to supervisor mode and set the I bit in the PC word;
3. force the PC to fetch the next instruction from address 0CH for Prefetch Abort, 10H for Data Abort.

To continue after a Prefetch Abort use `SUBS PC,R14_svc,#4`. This will attempt to re-execute the aborting instruction (which will only be effective if action has been taken to remove the cause of the original abort). A Data Abort requires any auto-indexing to be reversed before returning to re-execute the offending instruction, the return being done by `SUBS PC,R14_svc,#8`.

The abort mechanism allows a 'demand paged virtual memory system' to be implemented when a suitable memory management unit (such as MEMC) is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the memory manager signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

Software interrupt

The software interrupt is used for getting into supervisor mode, usually to request a particular supervisor function. ARM will:

1. save R15 in R14_svc;
2. force M0, M1 to supervisor mode and set the I bit in the PC word;
3. force the PC to fetch the next instruction from address 08H.

To return from a SWI, use `MOVS PC,R14_svc`. This returns to the instruction following the SWI.

Undefined instruction trap

When ARM executes a Co-Processor instruction or an Undefined instruction, it offers it to any Co-Processors which may be present. If a Co-Processor can perform this instruction but is busy at that moment, ARM will wait until the Co-Processor is ready. If no Co-Processor can handle the instruction ARM will take the undefined instruction trap.

The trap may be used for software emulation of a Co-Processor in a system which does not have the Co-Processor hardware, or for general purpose instruction set extension by software emulation.

When the undefined instruction trap is taken ARM will:

1. save R15 in R14_svc;
2. force M0, M1 to supervisor mode and set the I bit in the PC word;
3. force the PC to fetch the next instruction from address 04H.

To return from this trap (after performing a suitable emulation of the required function), use `MOVS PC,R14_svc`. This will return to the instruction following the undefined instruction.

Reset

When Reset goes HIGH ARM will:

1. stop the currently executing instruction and start executing no-ops. When Reset goes LOW again it will:
2. save R15 in R14_svc;
3. force M0, M1 to supervisor mode and set the F and I bits in the PC word;
4. force the PC to fetch the next instruction from address 0H.

Vector Summary

Address	
0000000	Reset
0000004	Undefined instruction
0000008	Software interrupt
000000C	Abort (prefetch)
0000010	Abort (data)
0000014	Address exception
0000018	IRQ
000001C	FIQ

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine. The FIQ routine might reside at 000001CH onwards, and thereby avoid the need for (and execution time of) a branch instruction.

Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

1. Reset (highest priority)
2. Address exception, Data abort
3. FIQ
4. IRQ
5. Prefetch abort
6. Undefined Instruction, Software interrupt (lowest priority)

Note that not all exceptions can occur at once. Address exception and data abort are mutually exclusive, since if an address is illegal the ARM will ignore the **ABORT** input. Undefined instruction and software interrupt are also mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If an address exception or data abort occurs at the same time as a FIQ, and FIQs are enabled (ie the F flag in the PSR is clear), ARM will enter the address exception or data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the address exception or data abort handler to resume execution. Placing address exception and data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection, but the time for this exception entry should be added to worst case FIQ latency calculations.

Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax*), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is load multiple registers), plus the time for address exception or data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). At the end of this time ARM will be executing the instruction at 1CH.

Tsyncmax is 2.5 processor cycles, *Tldm* is 18 cycles, *Texc* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 25.5 processor cycles, which is just over 2.5 microseconds in a system which uses a continuous 10 MHz processor clock. In a DRAM based system running at 4 and 8 MHz, for example using MEMC, this time becomes 4.5 microseconds, and if bus bandwidth is being used to support video or other DMA activity, the time will increase accordingly.

The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time.

The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchroniser (*Tsyncmin*) plus *Tfiq*. This is 3.5 processor cycles.

The condition field

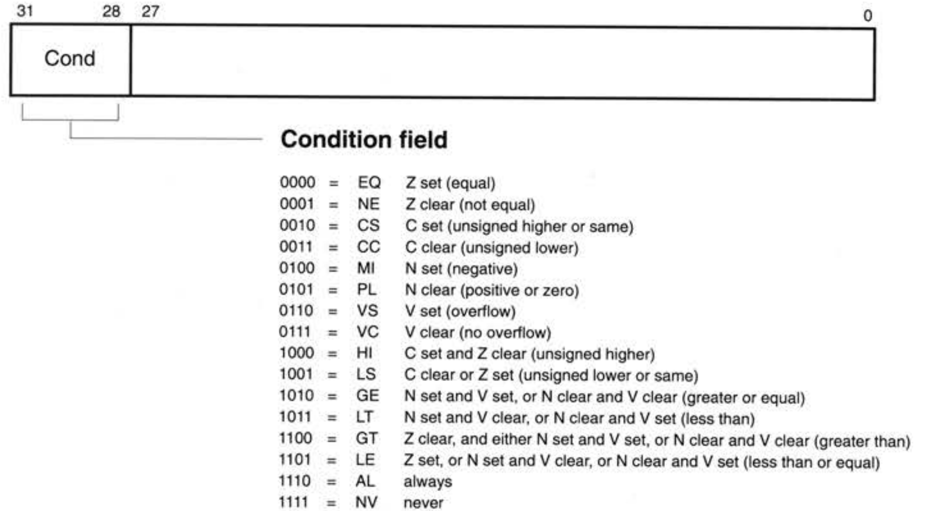


Figure 18.5 The condition field

All ARM instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the PSR at the end of the preceding instruction.

If the ALways condition is specified, the instruction will be executed irrespective of the flags, and likewise the NeVer condition will cause it not to be executed (it will be a no-op, ie take one cycle and have no effect on the processor state).

The other condition codes have meanings as detailed above, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag, and the instruction will not be executed.

Branch and branch with link (B, BL)

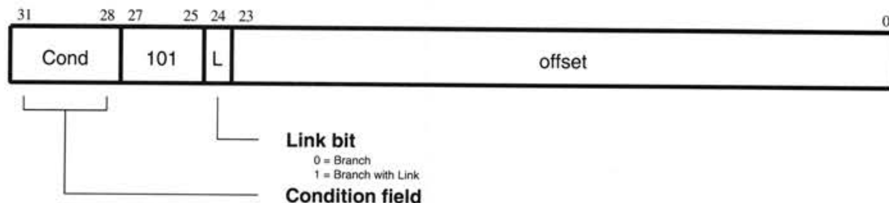


Figure 18.6 *Branch and branch with link* (B, BL)

The instruction is only executed if the condition specified in the condition field is true (see the section entitled *The condition field* on page 208).

All branches take a 24 bit offset. This is shifted left two bits and added to the PC, with any overflow being ignored. The branch can therefore reach any word aligned address within the address space. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words ahead of the current instruction.

The link bit

Branch with Link writes the old PC and PSR into R14 of the current bank. The PC value written into the link register (R14) is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction.

To return and restore the PSR use MOVS PC,R14 if the link register is still valid or LDM Rn!,{PC}^ if the link register has been saved onto a stack. To return without restoring the PSR use MOV PC,R14 if the link register is still valid or LDM Rn!,{PC} if the link register has been saved onto a stack.

Assembler syntax

`B{L}{cond} expression`

`{L}`	is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.
`{cond}`	is a two-char mnemonic as shown in the section entitled *The condition field* on page 208 (EQ, NE, VS etc.). If absent then AL (ALways) will be used.
`expression`	is the destination. The assembler calculates the offset.

Items in `{}` are optional.

Examples

```
here BAL here ; assembles to EAFFFFFE
              ; (note effect of PC offset)

  B there     ; ALways condition used as default

  CMP R1,#0   ; compare register 1 with zero
  BEQ fred    ; branch to fred if register 1 was zero
              ; otherwise continue to next instruction

  BL sub + ROM; unconditionally call subroutine at
              ; computed address

  ADDS R1,#1  ; add 1 to register 1, setting PSR flags on
              ; the result
  BLCC sub    ; call subroutine if the C flag is clear, which
              ; will be the case unless R1 contained FFFFFFFFH
              ; otherwise continue to next instruction

  BLNV sub    ; NeVer call subroutine (this is a NO-OP)
```

Data processing

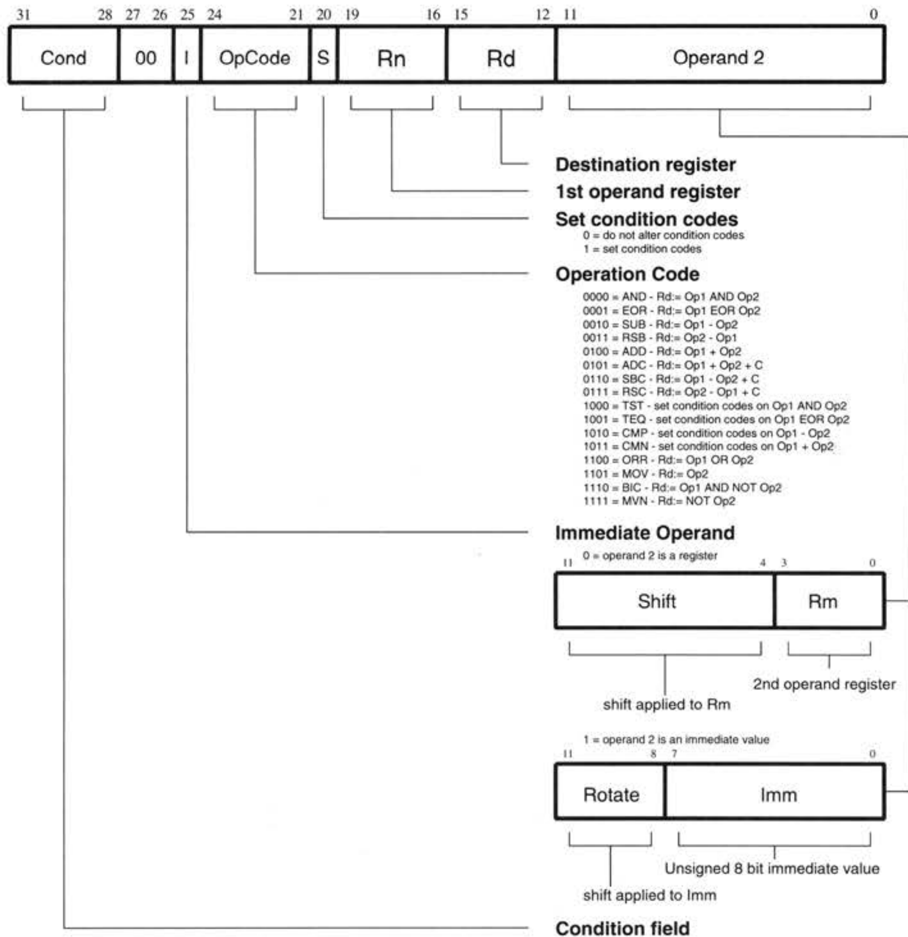


Figure 18.7 Data processing

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the PSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the

condition codes on the result, and therefore should always have the S bit set. (The assembler treats TST, TEQ, CMP and CMN as TSTS, TEQS, CMPS and CMNS by default.)

Operations

The operations supported are:

Assembler Mnemonic	OpCode	Action
AND	0000	Bit-wise logical AND of operands
EOR	0001	Bit-wise logical EOR of operands
SUB	0010	Subtract operand 2 from operand 1
RSB	0011	Subtract operand 1 from operand 2
ADD	0100	Add operands
ADC	0101	Add operands plus carry (PSR C flag)
SBC	0110	Subtract operand 2 from operand 1 plus carry
RSC	0111	Subtract operand 1 from operand 2 plus carry
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	Bit-wise logical OR of operands
MOV	1101	Move operand 2 (operand 1 is ignored)
BIC	1110	Bit clear (bit-wise logical AND of operand 1 and NOT operand 2)
MVN	1111	Move NOT operand 2 (operand 1 is ignored)

PSR flags

The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the PSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeroes, and the N flag will be set to the logical value of bit 31 of the result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the PSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the

Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register:

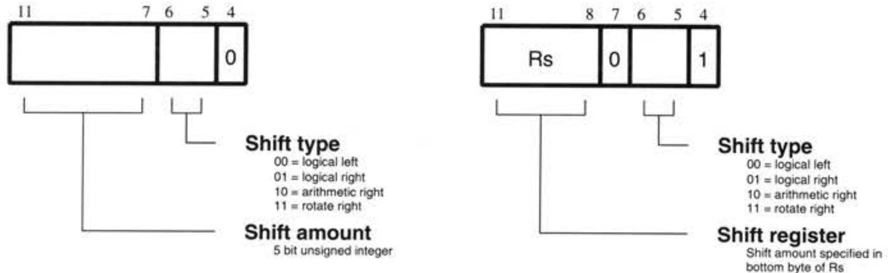
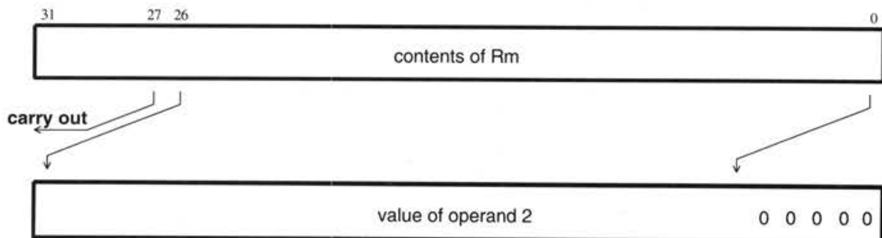


Figure 18.8 *Shifts*

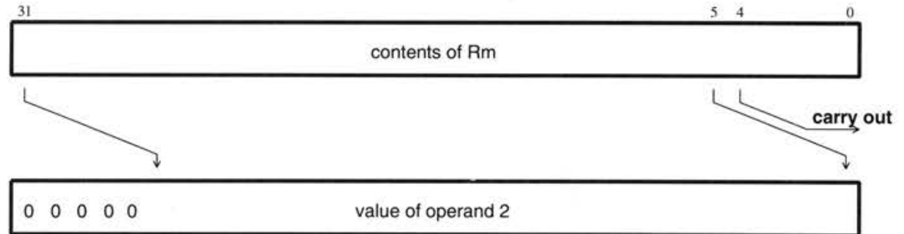
Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeroes, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the PSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is:



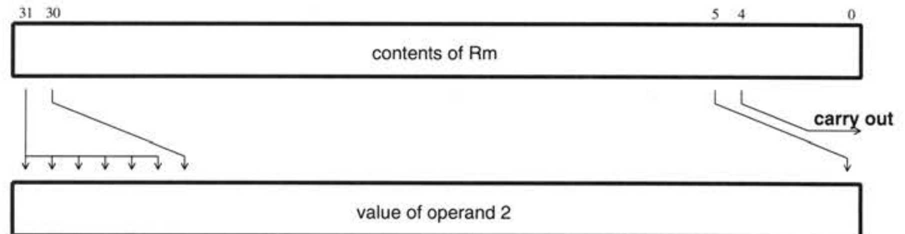
Note that LSL #0 is a special case, where the shifter carry out is the old value of the PSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has this effect:



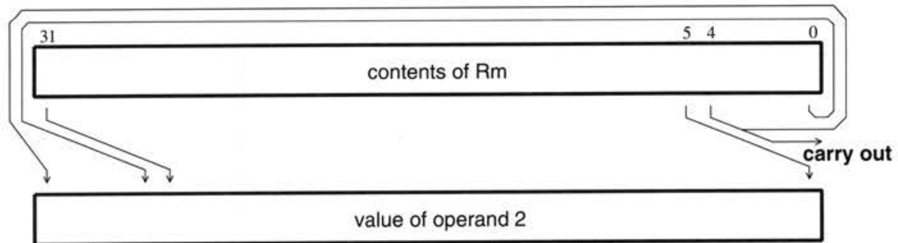
The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeroes. This preserves the sign in 2's complement notation. For example, ASR #5:

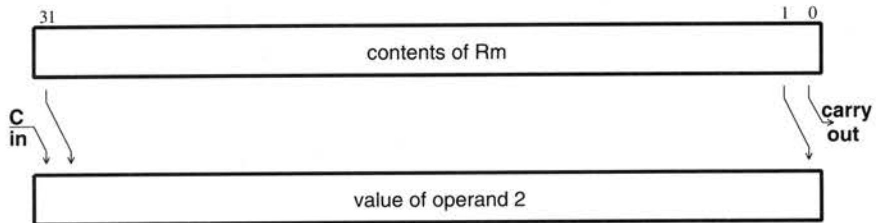


The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeroes, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeroes used to fill the high end in logical right operations. For example, ROR #5:



The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the PSR C flag to the most significant end of the contents of Rm:



Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the PSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shifting processes described above: .numberpars * LSL by 32 has result zero, carry out equal to bit 0 of Rm.

- LSL by more than 32 has result zero, carry out zero.
- LSR by 32 has result zero, carry out equal to bit 31 of Rm.

- LSR by more than 32 has result zero, carry out zero.
- ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.
- ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
- ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or an undefined instruction.

Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2. Another example is that the 8 bit constant may be aligned with the PSR flags (bits 0, 1, and 26 to 31). All the flags can thereby be initialised in one TEQP instruction.

Writing to R15

When Rd is a register other than R15, the condition code flags in the PSR may be updated from the ALU flags as described above. When Rd is R15 and the S flag in the instruction is set, the PSR is overwritten by the corresponding bits in the ALU result, so bit 31 of the result goes to the N flag, bit 30 to the Z flag, bit 29 to the C flag and bit 28 to the V flag. In user mode the other flags (I, F, M1, M0) are protected from direct change, but in non-user modes these will also be affected, accepting copies of bits 27, 26, 1 and 0 of the result respectively.

When one of these instructions is used to change the processor mode (which is only possible in a non-user mode), the following instruction should not access a banked register (R8-R14) during its first cycle. A no-op should be inserted if the next instruction must access a banked register. Accesses to the unbanked registers (R0-R7 and R15) are safe.

If the S flag is clear when Rd is R15, only the 24 PC bits of R15 will be written. Conversely, if the instruction is of a type which does not normally produce a result (CMP, CMN, TST, TEQ) but Rd is R15 and the S bit is set, the result will be used in this case to update those PSR flags which are not protected by virtue of the processor mode.

Using R15 as an operand

If R15 is used as an operand in a data processing instruction it can present different values depending on which operand position it occupies. It will always contain the value of the PC. It may or may not contain the values of the PSR flags as they were at the completion of the previous instruction.

When R15 appears in the Rm position it will give the value of the PC together with the PSR flags to the barrel shifter.

When R15 appears in either of the Rn or Rs positions it will give the value of the PC alone, with the PSR bits replaced by zeroes.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount, the PC will be 8 bytes ahead when used as Rs, and 12 bytes ahead when used as Rn or Rm.

Assembler syntax

- * MOV,MVN - single operand instructions

  `opcode{cond}{S} Rd,Op2`

- CMP,CMN,TEQ,TST - instructions which do not produce a result.

  `opcode{cond}{P} Rn,Op2`

- AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

  `opcode{cond}{S} Rd,Rn,Op2`

where *Op2* is `Rm{,shift}` or `,#expression`

`{cond}` two-character condition mnemonic.

`{S}` set condition codes if S present (implied for CMP, CMN, TEQ, TST).

`{P}` make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the PSR directly from the ALU result.)

`Rd`, `Rn` and `Rm` are expressions evaluating to a register number.

If `#expression` is used, the assembler will attempt to match the expression by generating a shifted immediate 8-bit field. If this is impossible, it will give an error.

`shift` is `shiftname register` or `shiftname #expression`, or `RRX` (rotate right one bit with extend).

`shiftname`s are: `ASL, LSL, LSR, ASR, ROR`.

(ASL is a synonym for LSL, the two assemble to the same code.)

Examples

```
ADDEQ R2,R4,R5          ; if the Z flag is set make R2:=R4+R5

TEQS R4, #3             ; test R4 for equality with 3
                        ; (the S is in fact redundant as
                        ; the assembler inserts it
                        ; automatically)

SUB R4,R5,R7,LSR R2     ; logical right shift R7 by the number
                        ; in the bottom byte of R2, subtract the
                        ; result from R5, and put the answer
                        ; into R4

                        ; assume non-user mode here
TEQP R15, #0            ; Change to user mode and clear
                        ; N,Z,C,V,I,F
                        ; NB R15 is here in the Rn position, so
                        ; it comes without the PSR flags
MOVNV R0,R0             ; no-op to avoid mode change hazard
MOV PC,R14              ; return from subroutine (R14 is a banked
                        ; register)

MOVS PC,R14             ; return from subroutine and restore the PSR
```

Multiply and multiply-accumulate (ML)

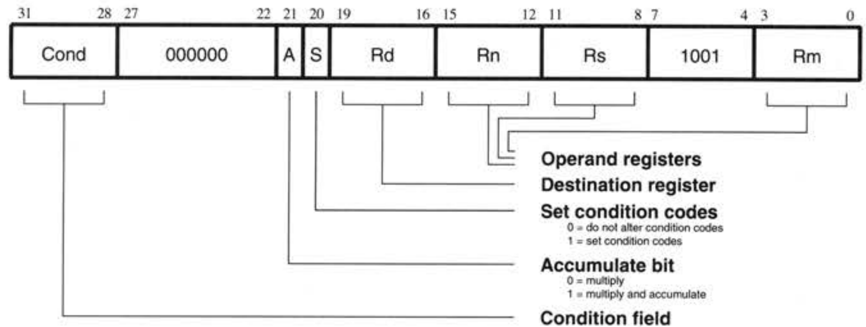


Figure 18.9 *Multiply and multiply-accumulate*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2's complement) or unsigned integers.

Operand restrictions

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result.

The destination register (Rd) should also not be R15. R15 is protected from modification by these instructions, so the instruction will have no effect, except that it will put meaningless values in the PSR flags if the S bit is set.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

PSR flags

Setting the PSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to bit 31 of the result, Z is set if and only if the result is zero), the V flag is unaffected by the instruction (as for logical data processing instructions), and the C flag is set to a meaningless value.

Writing to R15

As mentioned above, R15 must not be used as the destination register (Rd). If it is so used, the instruction will have no effect except possibly to scramble the PSR flags.

Using R15 as an operand

R15 may be used as one or more of the operands, though the result will rarely be useful. When used as Rs the PC bits will be used without the PSR flags, and the PC value will be 8 bytes on from the address of the multiply instruction. When used as Rn, the PC bits will be used along with the PSR flags, and the PC will again be 8 bits

on from the address of the instruction. When used as Rm, the PC bits will be used together with the PSR flags, but the PC will be the address of the instruction plus 12 bytes in this case.

Assembler syntax

`MUL`*`{cond}{S}`* `Rd,Rm,Rs`

`MLA`*`{cond}{S}`* `Rd,Rm,Rs,Rn`

{cond}	two-character condition mnemonic (see the section entitled *The condition field* on page 208).
{S}	set condition codes if S present.

Rd, Rm, Rs and Rn are expressions evaluating to a register number.

(Rd must not be R15 and must not be the same as Rm.)

Examples

```
MUL R1,R2,R3              ;R1:=R2*R3

MLAEQS R1,R2,R3,R4        ;conditionally R1:=R2*R3+R4,
                          ; setting condition codes
```

The multiply instruction may be used to synthesize higher precision multiplications, for instance to multiply two 32 bit integers and generate a 64 bit result:

```
mul64
 MOV    a1,A,LSR #16    ; a1:= top half of A
 MOV    D,B,LSR #16     ; D := top half of B
 BIC    A,A,a1,LSL #16  ; A := bottom half of A
 BIC    B,B,D,LSL #16   ; B := bottom half of B
 MUL    C,A,B           ; low section of result
 MUL    B,a1,B          ; ) middle sections
 MUL    A,D,A           ; ) of result
 MUL    D,a1,D          ; high section of result
 ADDS   A,B,A           ; add middle sections (couldn't use
                        ;   MLA as we need C correct)
 ADDCS  D,D,#&10000     ; carry from above add
 ADDS   C,C,A,LSL #16   ; C is now bottom 32 bits of product
 ADC    D,D,A,LSR #16   ; D is top 32 bits
```

(A, B are registers containing the 32 bit integers; C, D are registers for the 64 bit result; a1 is a temporary register. A and B are overwritten during the multiply.)

Single data transfer (LDR, STR)

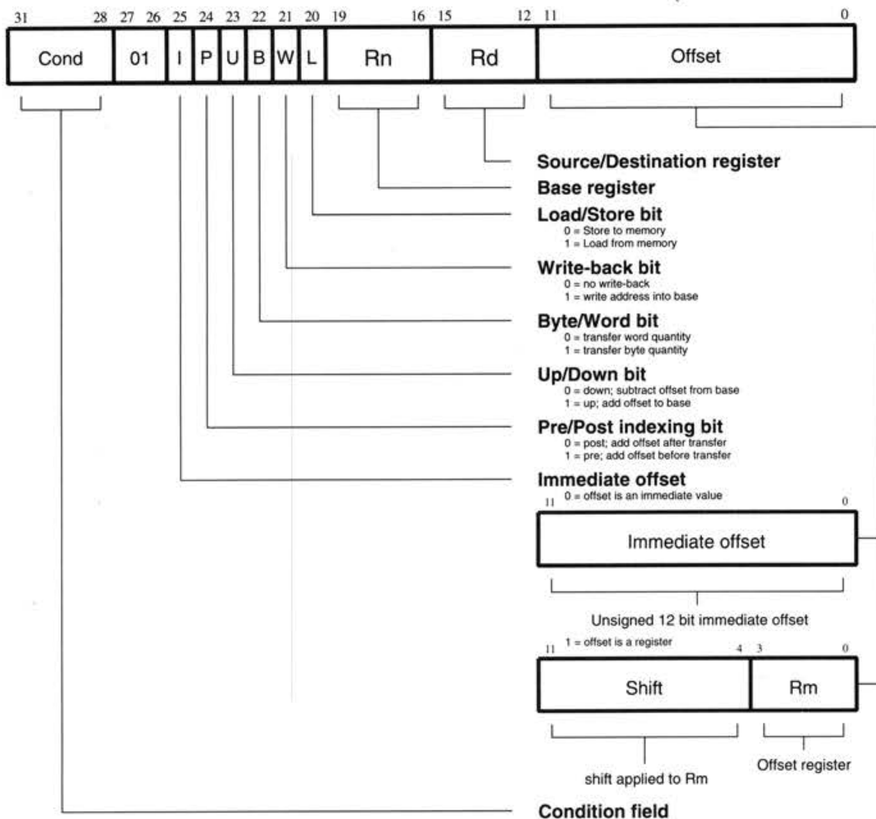


Figure 18.10 *Single data transfer*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if 'auto-indexing' is required.

Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in non-user mode code, where setting the W bit forces the **$\overline{\text{TRANS}}$** pin to go LOW for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this pin.

Shifted register offset

The 8 shift control bits are described in the section entitled *Data processing* on page 211, but the register specified shift amounts are not available in this instruction class.

Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM register and memory.

A byte load (LDRB) expects the data on bits 0 to 7 if the supplied address is on a word boundary, on bits 8 to 15 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeroes.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across the data bus. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. External hardware could perform a double access to memory to allow non-aligned word loads, but existing systems do not support this.

A word store (STR) should generate a word aligned address. The data presented to the data bus are not affected if the address is not word aligned, so if support were required for non-aligned stores external hardware would have to switch bytes around on the bus.

Use of R15

These instructions will never cause the PSR to be modified, even when Rd or Rn is R15.

If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register one must remember that it contains an address 8 bytes on from the address of the current instruction.

If R15 is specified as the register offset (Rm), the value presented will be the PC together with the PSR.

When R15 is the source register (Rd) of a register store (STR) instruction, the value stored will be the PC together with the PSR. The stored value of the PC will be 12 bytes on from the address of the instruction. A load register (LDR) with R15 as Rd will change only the PC, and the PSR will be unchanged.

Address exceptions

If the address used for the transfer (ie the unmodified contents of the base register for post-indexed addressing, or the base modified by the offset for pre-indexed addressing) has a logic one in any of the bits 26 to 31, the transfer will not take place and the address exception trap will be taken.

Note that it is only the address actually used for the transfer which is checked. A base containing an address outside the legal range may be used in a pre-indexed transfer if the offset brings the address within the legal range, and likewise a base within the legal range may be modified by post-indexing to outside the legal range without causing an address exception.

Data Aborts

A transfer to or from a legal address may still cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor **ABORT** pin HIGH, whereupon the data transfer instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Assembler syntax

LDR|STR*{cond}*{B}*{T}* Rd,*Address*

LDR	load from memory into a register.
STR	store from a register into memory.
{*cond*}	two-character condition mnemonic (see the section entitled *The condition field* on page 208).
{B}	if B is present then byte transfer, otherwise word transfer.
{T}	if T is present the W bit will be set in a post-indexed instruction, causing the **TRANS** pin to go LOW for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
Rd	is an expression evaluating to a valid register number.

Address - can be:

- An expression which generates an address:
  expression

  The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.
- A pre-indexed addressing specification:

  [Rn] offset of zero
  [Rn,#*expression*]{!} offset of *expression* bytes
- [Rn,{+/-}Rm{,*shift*}]{!} offset of +/- contents of index register, shifted by *shift*.
- A post-indexed addressing specification:

  [Rn],#*expression* offset of *expression* bytes
  [Rn],{+/-}Rm{,*shift*} offset of +/- contents of index register, shifted as by *shift*.

Rn and Rm are expressions evaluating to a valid register number. Note if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM pipelining.

shift is a general shift operation (see the section entitled *Data processing* on page 211) but note that the shift amount may not be specified by a register.

{!} write back the base register (set the W bit) if ! is present.

Examples

```
STR R1,[BASE,INDEX]!         ;store R1 at BASE+INDEX (both of
                             ; which are registers) and write
                             ; back address to BASE

STR R1,[BASE],INDEX          ;store R1 at BASE and writeback
                             ; BASE+INDEX to BASE

LDR R1,[BASE,#16]            ;load R1 from contents of BASE+16.
                             ; Don't write back

LDR R1,[BASE,INDEX,LSL #2] ;load R1 from contents of
                             ; BASE+INDEX*4

LDREQB R1,[BASE,#5]          ;conditionally load byte at BASE+5
                             ; into R1 bits 0 to 7, filling bits
                             ; 8 to 31 with zeroes

STR R1,PLACE                 ;generate PC relative offset to
                             ; address PLACE
      .
      .
PLACE
```

Block data transfer (LDM, STM)

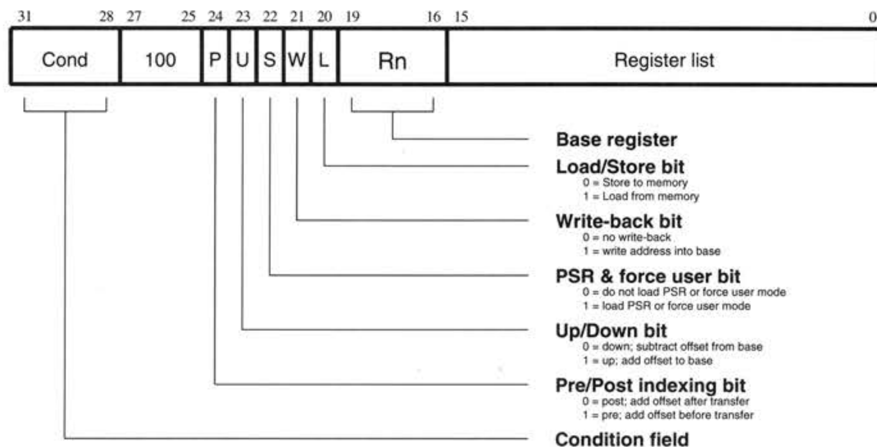


Figure 18.11 *Block data transfer*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of

illustration, consider the transfer of R1, R5 and R7 in the case where Rn=1000H and write back of the modified base is required (W=1). The following figures show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

(In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 1000H unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.)

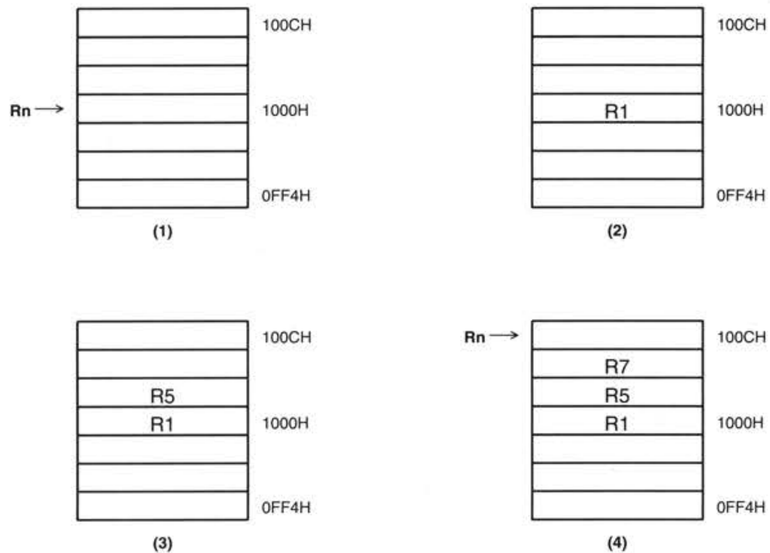


Figure 18.12 *Post-increment addressing*

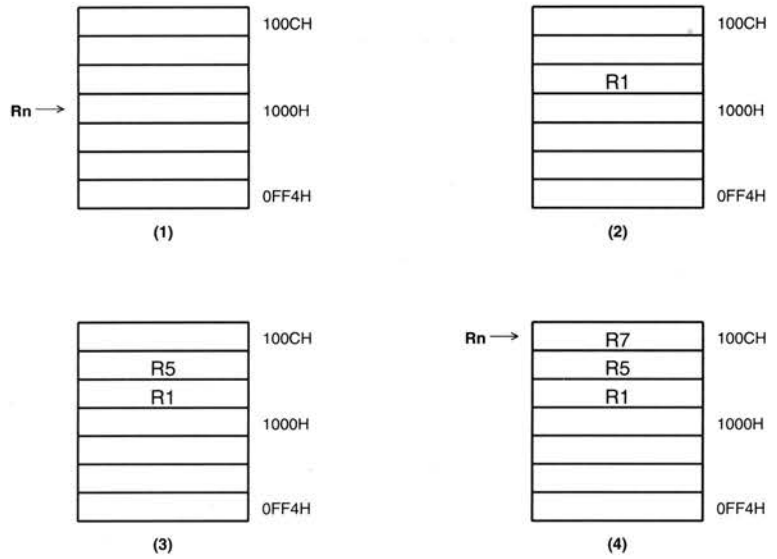


Figure 18.13 Pre-increment addressing

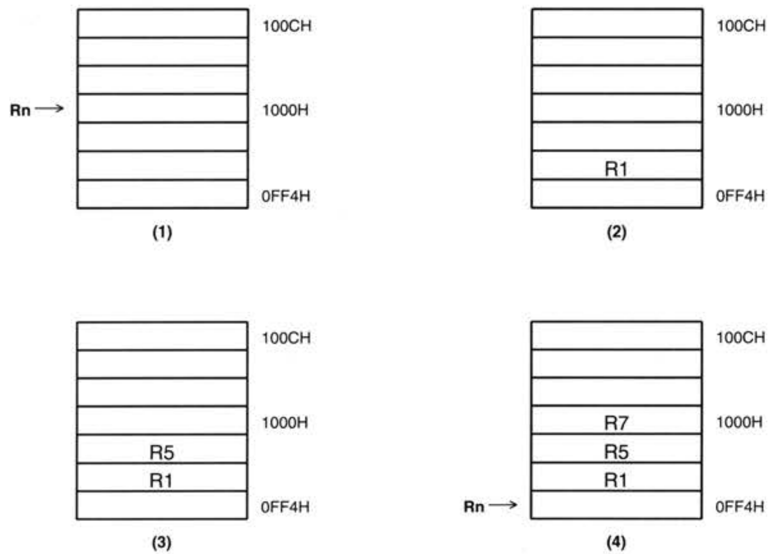


Figure 18.14 Post-decrement addressing

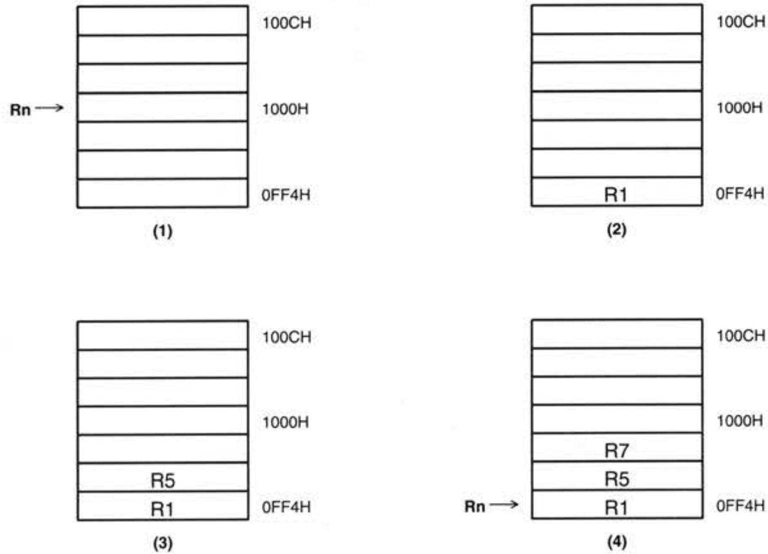


Figure 18.15 *Pre-decrement addressing*

Transfer of R15

Whenever R15 is stored to memory, the value transferred is the PC together with the PSR flags. The stored value of the PC will be 12 bytes on from the address of the STM instruction.

If R15 is in the transfer list of a load multiple (LDM) instruction the PC is overwritten, and the effect on the PSR is controlled by the S bit. If the S bit is 0 the PSR is preserved unchanged, but if the S bit is 1 the PSR will be overwritten by the corresponding bits of the loaded value. In user mode, however, the I, F, M0 and M1 bits are protected from change whatever the value of the S bit. The mode at the start of the instruction determines whether these bits are protected, and the supervisor may return to the user program, re-enabling interrupts and restoring user mode with one LDM instruction.

Forcing transfer of the user bank

For STM instructions the S bit is redundant as the PSR is always stored with the PC whenever R15 is in the transfer list. In user mode programs the S bit is ignored, but in other modes it has a second interpretation. S=1 used to force transfers to take values from the user register bank instead of from the current register bank. This is

useful for saving the user state on process switches. Note that when it is so used, write back of the base will also be to the user bank, though the base will be fetched from the current bank. Therefore don't use write back when forcing user bank.

In LDM instructions the S bit is redundant if R15 is not in the transfer list, and again in user mode programs it is ignored in this case. In non-user mode programs where R15 is not in the transfer list, S=1 is used to force loaded values to go to the user registers instead of the current register bank. When so used, care must be taken not to read from a banked register during the following cycle - if in doubt insert a no-op. Again don't use write back when forcing user bank transfer.

Use of R15 as the base

When the base is the PC, the PSR bits will be used to form the address as well, so unless all interrupts are enabled and all flags are zero an address exception will occur. Also, write back is never allowed when the base is the PC (setting the W bit will have no effect).

Inclusion of the base in the register list

When writeback is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. An LDM will always overwrite the updated base if the base is in the list.

Address exceptions

When the address of the first transfer falls outside the legal address space (ie has a 1 somewhere in bits 26 to 31), an address exception trap will be taken. The instruction will first complete in the usual number of cycles, though an STM will be prevented from writing to memory. The processor state will be the same as if a data abort had occurred on the first transfer cycle (see next section).

Only the address of the first transfer is checked in this way; if subsequent addresses over- or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** pin HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible

- Overwriting of registers stops when the abort happens. The aborting load will not take place, nor will the preceding one, but registers two or more positions ahead of the abort (if any) will be loaded. (This guarantees that the PC will be preserved, since it is always the last register to be overwritten.)
- The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

Assembler syntax

`LDM|STM{cond}FD|ED|FA|EA|IA|IB|DA|DB Rn{!},Rlist{^}`

{cond}	two character condition mnemonic (see the section entitled *The condition field* on page 208).
Rn	is an expression evaluating to a valid register number.
Rlist	can be either a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}), or an expression evaluating to the 16 bit operand.
{!}	if present requests write-back (W=1), otherwise W=0.
{^}	if present set S bit to load the PSR with the PC, or force transfer of user bank when in non-user mode.

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalencies between the names and the values of the bits in the instruction are:

name	stack	other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a 'full' or 'empty' stack, ie whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

Examples

```
LDMFD SP!,{R0,R1,R2}    ; unstack 3 registers

STMIA BASE,{R0-R15}     ; save all registers
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMED SP!,{R0-R3,R14}  ; save R0 to R3 to use as
                       ;  workspace and R14 for returning

BL somewhere           ; this nested call will
                       ;  overwrite R14

LDMED SP!,{R0-R3,R15}^ ; restore workspace and return
                       ;  (also restoring PSR flags)
```

Software interrupt

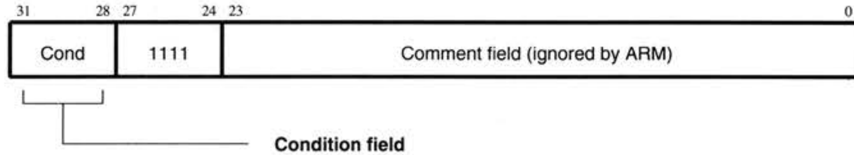


Figure 18.16 *Software interrupt*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

The software interrupt instruction is used to enter supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change but forces the PC to a fixed value (08H). If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

Return from the supervisor

The PC and PSR are saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS R15,R14_svc will return to the user program, restore the user PSR and return the processor to user mode.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address.

Comment field

The bottom 24 bits of the instruction are ignored by ARM, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

Assembler syntax

`SWI{cond} expression`

`{cond}`	two character condition mnemonic (see the section entitled *The condition field* on page 208).
`expression`	is evaluated and placed in the comment field (which is ignored by ARM).

Examples

```
SWI    Read           ;get next character from read stream

SWI    WriteI+"k"     ;output a "k" to the write stream

SWINE  0              ;conditionally call supervisor
                      ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
08H B Supervisor      ;SWI entry point

EntryTable            ;addresses of supervisor routines
  & ZeroRtn
  & ReadCRtn
  & WriteIRtn
  ........

Zero     * 0
ReadC    * 256
WriteI   * 512

Supervisor

; SWI has routine required in bits 8-23,
;  data (if any) in bits 0-7.
; Assumes R13_svc points to a suitable stack

 STMR13,{R0-R2,R14}     ; save work registers and return address
 BICR0,R14,#&FC000003   ; clear PSR bits
 LDRR0,[R0,#-4]         ; get SWI instruction
 BICR0,R0,#&FF000000    ; clear top 8 bits
 MOVR1,R0,LSR #8        ; get routine offset
 ADRR2,EntryTable       ; get start address of entry table
 LDRR15,[R2,R1,LSL #2]  ; branch to appropriate routine

WriteIRtn               ; enter with character in R0 bits 0-7
 .......
 LDM R13,{R0-R2,R15}^   ; restore workspace and return.
```

Co-Processor data operations

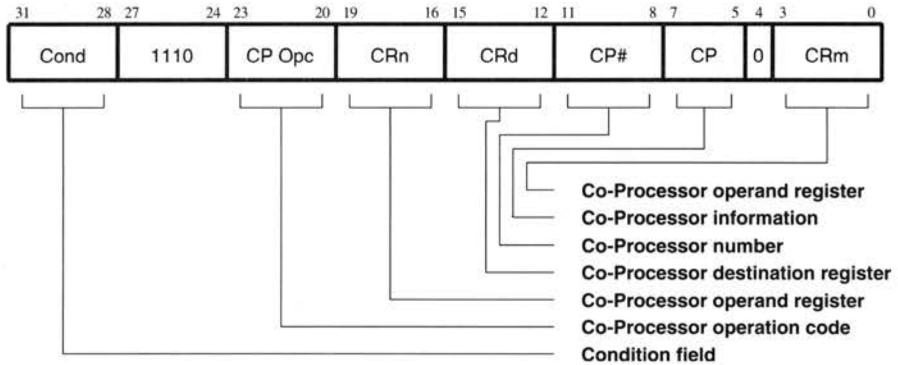


Figure 18.17 *Co-processor data operations*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

This class of instruction is used to tell a Co-Processor to perform some internal operation. No result is communicated back to ARM, and ARM will not wait for the operation to complete. The Co-Processor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM activity allowing the Co-Processor and ARM to perform independent tasks in parallel.

The Co-Processor fields

Only bit 4 and bits 24 to 31 are significant to ARM; the remaining bits are used by Co-Processors. The above field names are used by convention, and particular Co-Processors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each Co-Processor, and a Co-Processor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the Co-Processor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

Assembler syntax

`CDP{cond} CP#,expression1,CRd,CRn,CRm{,expression2}`

`{cond}` two character condition mnemonic (see the section entitled *The condition field* on page 208).

`CP#` the unique number of the required Co-Processor.

expression1 evaluated to a constant and placed in the CP Opc field.

CRd, CRn, CRm are expressions evaluating to a valid Co-Processor register number.

expression2 where present is evaluated to a constant and placed in the CP field.

Examples

```
CDP 1,10,CR1,CR2,CR3     ; request Co-Proc 1 to do operation 10
                         ;  on CR2 and CR3, and put the result
                         ;  in CR1

CDPEQ 2,5,CR1,CR2,CR3,2; if Z flag is set request Co-Proc 2 to
                         ;  do operation 5 (type 2) on CR2 and
                         ;  CR3, and put the result in CR1
```

Warning!

Current ARM chips have a fault in the implementation of CPDO which will cause a Software Interrupt to take the Undefined Instruction trap if the SWI is the next instruction after the CDP. This problem only arises when a hardware Co-Processor is attached to the system, but if it is ever intended to add hardware to support a CDP (rather than trapping to an emulator) the sequence CDP SWI should be avoided.

Co-Processor data transfers

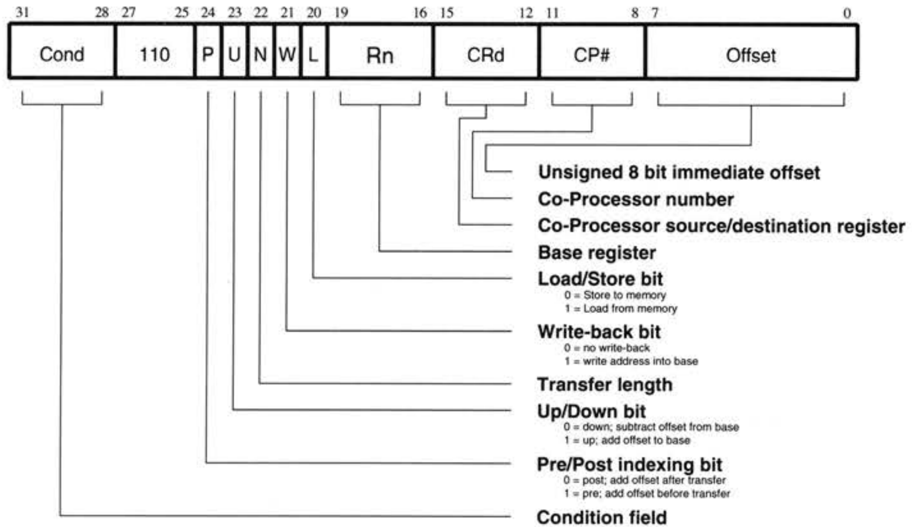


Figure 18.18 *Co-processor data transfers*

The instruction is only executed if the condition is true. The various conditions are defined in the section entitled *The condition field* on page 208.

This class of instruction is used to transfer one or more words of data between the Co-Processor and main memory. ARM is responsible for supplying the memory address, and the Co-Processor supplies or accepts the data and controls the number of words transferred.

The Co-Processor fields

The CP# field is used to identify the Co-Processor which is required to supply or accept the data, and a Co-Processor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the Co-Processor which may be interpreted in different ways by different Co-Processors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

Addressing modes

ARM is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits and specify word offsets here, whereas they are 12 bits and specify byte offsets for single data transfers.

An 8 bit unsigned immediate offset is scaled to words (ie shifted left 2 bits) and added to (U=1) or subtracted from (U=0) a base register (Rn), either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

Use of R15

If Rn is R15, the value used will be the PC without the PSR flags, with the PC being the address of this instruction plus 8 bytes. Write-back to the PC is inhibited, and the W bit will be ignored.

Address exceptions

If the address used for the first transfer is illegal the address exception mechanism will be invoked. Instructions which transfer multiple words will only trap if the first address is illegal; subsequent addresses will wrap around inside the 26 bit address space.

Data aborts

If the address is legal but the memory manager generates an abort the data abort trap will be taken. The writeback of the modified base will take place, but all other processor state will be preserved. The Co-Processor is partly responsible for ensuring restartability, and must either detect the abort or ensure that any actions consequent from this instruction can be repeated when the instruction is retried after the cause of the abort has been resolved.

Assembler syntax

LDC | STC*{cond}{L}* CP#,CRd,*Address*

LDC load from memory to Co-Processor (L=1).

STC store from Co-Processor to memory (L=0).

{L} when present perform long transfer (N=1), otherwise perform short transfer (N=0).

{cond} two character condition mnemonic (.

CP# the unique number of the required Co-Processor.

CRd is an expression evaluating to a valid Co-Processor register number.

Address can be:

- An expression which generates an address:

  expression

  The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

- A pre-indexed addressing specification:

  [Rn] offset of zero

  [Rn,#*expression*]{!} offset of *expression* bytes

- A post-indexed addressing specification:

  [Rn],#*expression* offset of *expression* bytes

Rn is an expression evaluating to a valid ARM register number. Note if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM pipelining.

{!} write back the base register (set the W bit) if ! is present.

Examples

```
LDC 1,CR2,table          ; load CR2 of Co-Proc 1 from address
                         ;  table, using a PC relative address.

STCEQL 2,CR3,[R5,#24]!; conditionally store CR3 of Co-Proc 2
                         ;  into an address 24 bytes up from R5,
                         ;  write this address back into R5, and
                         ;  use long transfer option (probably to
                         ;  store multiple words)
```

Note that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

Co-Processor register transfers

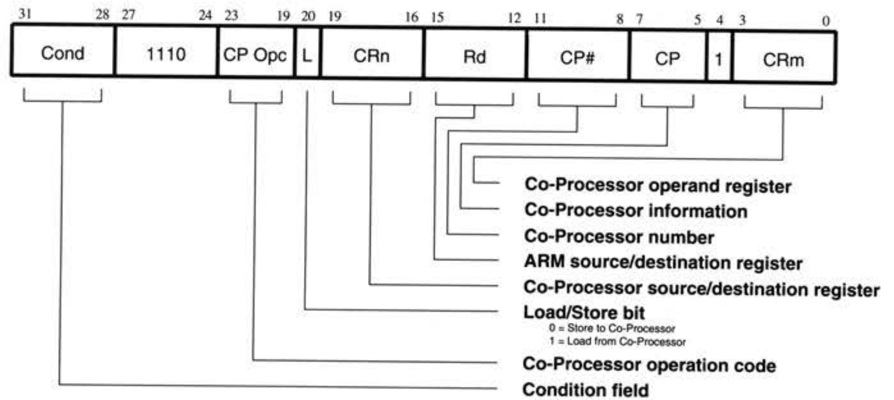


Figure 18.19 *Co-Processor register transfers*

The instruction is only executed if the condition is true. The various conditions are defined in section entitled *The condition field* on page 208.

This class of instruction is used to communicate information directly between ARM and a Co-Processor. An example of an MCR instruction would be a FIX of a floating point value held in a Co-Processor, where the floating point number is converted into a 32 bit integer within the Co-Processor, and the result is then transferred to an ARM register. A FLOAT of a 32 bit value in an ARM register into a floating point value within the Co-Processor illustrates the use of MRC.

An important use of this instruction is to communicate control information directly from the Co-Processor into the ARM PSR flags. As an example, the result of a comparison of two floating point values within a Co-Processor can be moved to the PSR to control the subsequent flow of execution.

The Co-Processor fields

The CP# field is used, as for all Co-Processor instructions, to specify which Co-Processor is being called upon to respond.

The CP Opc, CRn, CP and CRm fields are used only by the Co-Processor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the Co-Processor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the Co-Processor is required to perform, CRn is the Co-Processor register which is the source or destination of the transferred information, and CRm is a second Co-Processor register which may be involved in some way which depends on the particular operation specified.

Transfers to R15

When a Co-Processor register transfer to ARM has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other PSR flags are unaffected by the transfer.

Transfers from R15

A Co-Processor register transfer from ARM with R15 as the source register will store the PC together with the PSR flags.

Assembler syntax

`MCR|MRC{cond} CP#,expression1,Rd,CRn,CRm{,expression2}`

`MCR`	move from Co-Processor to ARM register (L=1).
`MRC`	move from ARM register to Co-Processor (L=0).
`{cond}`	two character condition mnemonic (see the section entitled *The condition field* on page 208).
`CP#`	the unique number of the required Co-Processor.
`expression1`	evaluated to a constant and placed in the CP Opc field.
`Rd`	is an expression evaluating to a valid ARM register number.
`CRn, CRm`	expressions evaluating to a valid Co-Processor register number.
`expression2`	where present is evaluated to a constant and placed in the CP field.

Examples

```
MRC 2,5,R3,CR5,CR6         ; request Co-Proc 2 to perform
                           ;  operation 5 on CR5 and CR6, and
                           ;  transfer the (single 32 bit word)
                           ;  result back to R3

MRCEQ 3,9,R3,CR5,CR6,2     ; conditionally request Co-Proc 2 to
                           ;  perform operation 9 (type 2) on
                           ;  CR5 and CR6, and transfer the
                           ;  result back to R3
```

Undefined instructions

Figure 18.20 *Undefined instructions*

The instruction is only executed if the condition is true. The various conditions are defined in.

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering these instructions to any Co-Processors which may be present, and all Co-Processors must refuse to accept them by letting **CPA** float HIGH.

Assembler syntax

At present the assembler has no mnemonics for generating these instructions. If they are adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, these instructions should not be used.

Instruction set summary

31–28	27–20	19–16	15–12	11–8	7–4	3–0	
Cond	00 I OpCode S	Rn	Rd	Operand 2			Data Processing
Cond	000000 A S	Rd	Rn	Rs	1001	Rm	Multiply
Cond	0001 xxxxxxxxxxxxxxxx				1xx1	xxxx	Undefined
Cond	01 I P U B W L	Rn	Rd	offset			Single Data Transfer
Cond	011 xxxxxxxxxxxxxxxxxxxx				1	xxxx	Undefined
Cond	100 P U S W L	Rn	Register list				Block Transfer
Cond	101 L offset						Branch
Cond	110 P U N W L	Rn	CRd	CP#	offset		Co-Proc Data Transfer
Cond	1110 CP Opc	CRn	CRd	CP#	CP 0	CRm	Co-Proc Data Op
Cond	1110 CP Opc L	CRn	Rd	CP#	CP 1	CRm	Co-Proc Register Transfer
Cond	1111 ignored by ARM						Software Interrupt

Figure 18.21 *Instruction set summary*

(Note that some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 5 or bit 6 changed to a 1. These instructions should be avoided, as their action may change in future ARM implementations.)

Instruction Speeds

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures.

If the condition is met the instructions take:

```
Data Processing     1 S                 + 1 S         for SHIFT(Rs)
                                        + 1 S + 1 N   if R15 written
LDR                 1 S + 1 N + 1 I     + 1 S + 1 N   if R15 loaded
STR                       2 N
LDM                 n S + 1 N + 1 I     + 1 S + 1 N   if R15 loaded
STM             (n-1) S + 2 N
B,BL                2 S + 1 N
SWI,trap            2 S + 1 N
MUL,MLA             1 S       + m I
CDP                 1 S       + b I
LDC,STC         (n-1) S + 2 N + b I
MRC                 1 S       + b I + 1 C
MCR                 1 S    + (b+1)I + 1 C
```

n is the number of words transferred.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between 2^(2m-3) and 2^(2m-1)-1 inclusive takes m cycles for m>1. Multiplication by 0 or 1 takes 1 cycle. The maximum value m can take is 16.

b is the number of cycles spent in the Co-Processor busy-wait loop.

If the condition is not met all instructions take one S cycle.

The four cycle types (N, S, I and C) correspond to data transfer activities:

- Non-sequential cycle. ARM requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.
- Sequential cycle. ARM requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word after the preceding address.
- Internal cycle. ARM does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.
- Co-Processor register transfer. ARM wishes to use the data bus to communicate with a Co-Processor, but does not require any action by the memory system.

I and C cycles are the quickest. S cycles take the same or more time, and N cycles take the same or more time than S cycles: I,C <= S <= N.

For an ARM2 system with MEMC or MEMC1A a typical relationship between the I,C : S : N cycle times is 1 : 1.2 : 2.

Index

Symbols

! 81, 132
126
% 63
& 125
* 123
*Wimpslot 166
= 125
?label 63
^ 126
    extension to 127
    use of 83, 84
|, use of 56

Numerics

09 56
0A 56
0D 56
20 56
32-bit word 45

A

AAsm
    options 21
    starting 20
ABE 197
ABORT 195
abort 203
    during data access 51
    during instruction prefetch 50
    signal 50
ABS 119
absolute value 185
ACS 120
ADC 75, 76
ADD 75
Add 97
    see also ADD
    with Carry 97
        see also ADC
address
    exception 49
    set load and execution address 133
address bus 45
address TRAP 203
ADF 119
ADR 99
ADRL 100
ALE 197
alignment 125
    automatic 125
AND 75, 76
AOF 135, 137
AOUT 137
application
    access of workspace 172
AREA 136
Arithmetic Shift Right, *see* ASR
ARM 45
ASL 70
ASN 120
ASR 70, 71
assembly
    conditional 139
    repetitive 141
ASSERT 132
ATN 120

B

B 72, 80
B/W 198
barrel shifter 68
  shift types 70
BASIC
  routine to search for lost memory blocks 163
BIC 75, 76
binary operations 119
bins (linked lists) 165
Bit Clear 97
  see also BIC
Bitwise And 97
  see also AND
Bitwise Exclusive Or 97
  see also EOR
Bitwise Or 97
  see also ORR
BL 73
blank line, use of 56
block data transfer 81, 226
  force transfer of user bank 229
boolean constants 59
branch 72, 209
  syntax 72
branch with link 73, 209
  syntax 73

C

C storage manager 164
carriage return 56
Carry flag 48
CC 65
chaining memory blocks 165
changing modes 53
check words
  using RMA 162
CMF 120
CMFE 120
CMN 77
CMP 77
CN 183
CNF 120
CNFE 120
CODE 136
COMDEF 136
comment field 233
Compare 97
  see also CMP
Compare Negated 97
  see also CMN
comparisons 58
condition
  default 65
  type 65
condition codes, return setting 188
condition field 208
conditional assembly 139
conditionals for logical OR 185
constant
  boolean 59
  decimal 59
  hexadecimal 59
  immediate 70
  number in the form base n 59
  string 59
conventions used in this manual 3
conversion 99
coprocessor cycles 67
COS 120
CP# 235, 237, 241
CPA 198
CPB 198
CPDO, warning 236
CPI 198
CS 65

D

DA 82
DATA 136

data abort 51
    and block transfer 51
    and data transfer 51
data bus 45
data processing
    instruction summary 79
    operations 211
    syntax 74
data swap 89
    see also SWP
data types 45
DB 82
DBE 197
DCB 63, 125
DCD 63, 125, 137
DCFD 115
DCFS 115
DCW 63, 125
DDT debugger 11
decrement
    after 97
        see also DA
    before 97
        see also DB
directives 123 - 137
    ! 132
    # 126
    % 125
    & 125
    * 123
    = 125
    ^ 126
        extension 127
        use of 83, 84
    align 126
    AREA 136
    ASSERT 132
    DCB 125
    DCD 125, 137
    DCW 125
    ELSE 139
    END 133
    ENDIF 139
    ENTRY 137
    EQU 123
    EXPORT 136
    GBLA 128
    GBLL 128
    GBLS 128
    GET 134
    IF 139
    IMPORT 136
    KEEP 137
    LDR 100
    LEADR 133
    LNK 135
    LTORG 126
    ORG 133
    RN 123
    ROUT 130
    SETA 128
    SETL 128
    SETS 128
    STRONG 137
    syntax 183
discrete and range tests 185
DVF 119

E

EA 82
ED 82
ELSE 139
empty stack
    ascending 97
        see also EA
    descending 98
        see also ED
END 133
ENDIF 139
ENTRY 137
EOR 75, 76
EQU 123
error handling 132
exceptions 201

- abort 203
- address trap 203
- FIQ 201
- interrupt latencies 206
- IRQ 202
- priority system 206
- reset 205
- software interrupt 204
- undefined instruction 205
- vector summary 206

EXP 119
EXPORT 136
expressions 57
- evaluating 57

F

FA 82
{FALSE} 59, 130
Fast Interrupt Mode 46, 53
FD 82
FDV 119
file buffers
- allocation 167

FIQ 48, 195, 201
FIX 118
fixed origin 58
flex 164
- advantages 164
- description 167
- limitations 167
- shifting heaps 167

floating point
- instruction set 115
- literals 116
- number input 114
- store loading directives 115

FLT 118
FML 119
FN 115
Fortran 77 136
fragmentation 166
- of malloc heap 167

FRD 119
FrontEnd 15
full Stack
- descending 98

full stack
- ascending 98
  - *see also* FA
- descending
  - *see also* FD

G

GBLA 128
GBLL 128
GBLS 128
GET 19, 134
guard constant, in memory blocks 164

H

heap
- coalescing 165

HS 65

I

IA 82
IB 82
IF 139, 141
immediate constants 70
IMPORT 136
increment
- after 97
  - *see also* IA
- before 97
  - *see also* IB

installation 1
instruction
- pipeline 45
- timing 66

instruction set summary 243
instruction speeds 243
instructions
- block data transfer 81, 226
- branch 72
- branch with link 73
- conditional 45, 65, 185
- data processing summary 79
- data processing syntax 80
- single data transfer 79, 221
- supervisor calls 92

internal cycles 67
interrupt latencies 206
interrupt mode 46, 53
IRQ 48, 49, 195, 202

K

KEEP 137

L

label 57
- instructions, stand-alone 58
- interrogation of 63
- local 130

language libraries
- recovering memory 161

LCLA 146
LCLL 146
LCLS 146
LDF 115
LDFD 116
LDFS 116
LDM 226
- abort during 231

LDMEA 86, 87
LDMED 83, 86
LDMFA 86
LDMFD 86, 87
LDR 100, 222
LDRB 222
LEADR 133
LFM 116
LGN 119
libraries, making your own 16
line terminators 56
linefeed 56
link register 47
literals 126
LNK 19, 135
LO 65
load and store operations 45
load multiple registers 97
- *see also* LDM

load register from memory location 97
- *see also* LDR

local label
- areas 130
- definition of 131
- referencing 131

location counter
- program 128
- storage area 126, 127

LOG 119
logical shift
- left, *see* LSL
- right, *see* LSR

logical values 59
LSL 70
LSR 70
LTORG 126

M

MACRO 144
Make 7
malloc 164
- deallocation of blocks 166
- use when designing programs 162

malloc heap 164
MCR, example of 240
memory 45

- alignment 164
- allocation in C 163
- allocation of block sizes 164
- allocation of file buffers 167
- allocation with flex and malloc 164
- attaching a base address to a storage area 127
- avoiding permanent loss 162
- avoiding references to deallocated blocks 162
- avoiding wastage 163
- BASIC routine to search for lost blocks 163
- coalescing blocks 165
- efficient use 161
- fragmentation 166
- malloc allocation 164
- reserving storage space 126
- splitting blocks 165

memory management 161
memory, laying out areas of 126
MEND 145
MEXIT 146
MNF 119
MOV 74, 75
Move 97
- *see also* MOV

Move Not 97
- *see also* MVN

MREQ 195
MUF 119
multiplication by a constant 187
multiply 91, 218
multiply-accumulate 90, 218
MVF 119
MVN 74, 75

N

Negative flag 48
NOFP 115
non-sequential cycles 66
non-user modes 53
nulls, loading memory with 125
numeric constants 59
numeric values 58

O

ObjAsm
- directives 135
- using branch destinations 135
- using literals 136

offsets 222
OPC 195
operands produced by barrel shifter
- 8 bit constant 69
- rotate & carry bit one bit right 69
- shifted by a constant amount 69
- shifted by n bits 69
- unshifted 69

operands, type of 57
operating modes 52
operators 60
- arithmetic 60
- Binary 60
- Bitwise logical 61
- boolean logical 60
- Relational 61
- Shift 61
- string (binary) 62
- string BASE (unary) 63
- string conversion (unary) 63
- string INDEX (unary) 63
- String length (unary) 62
- string slicing (binary) 62
- summary of 64
- Unary 60

{OPT} 130
ORG 133
ORR 75, 76
Overflow flag 48

P

P suffix 53, 78
{PC} 128, 130
PC 45, 46
PH1 195
PH2 195
POL 119
pop from stack
    empty stack, ascending 86
    empty stack, descending 86
    full stack, ascending 86
    full stack, descending 86
POW 119
Prefetch abort 50
printer, storing printer options 130
Processor Status Register 46, 53
program counter 45, 46
program design
    for efficient use of memory 161
program-relative values 58
    assigning 123
pseudo-random numbers 186
PSR 46, 48, 66
PSR flags 91, 212
push to stack 84
    empty stack ascending 85
    empty stack, descending 85
    full stack, ascending 85
    full stack, descending 85

R

R/W 195
R14 47
R14_fiq 47, 199
R14_irq 47, 199
R14_svc 47, 199
R15 46
    destination register 78
    operand 78, 91
RDF 119
READONLY 136
register names
    defining 123
    use of in expressions 123
Register R15 46
register-relative values 58
registers 46
REL 136
relocatable binary output 133
relocatable modules 153
    memory usage 172
    using AAsm 154
    using ObjAsm 154
repetitive assembly 141
RESET 196, 205
Reverse Subtract 97
    see also RSB
Reverse Subtract with Carry 97
    see also RSC
RFC 118
RFS 118
RLIST 184
RMA 172
    deallocation 162
    using for storage though SWI calls 162
RMF 119
RN 123
RND 119
ROR 70, 71
Rotate Right 70
    see also ROR
ROUT 130
RPW 119
RRX 72
RSB 75, 76
RSC 75, 76
RSF 119

S

S bit 75
SBC 75, 76

search
 for allocated memory blocks 162
semi-colon, use of 56
SEQ 196
sequential cycles 66
set load and execution address 133
SETA 128
SETL 128
SETS 128
setvbuf 167
SFM 116
shift 213
 field 213
shift types 70
 ASR 70
 LSL 70
 LSR 70
 ROR 70
sign/zero extension of a half word 188
signals 195
 A 197
 ABE 197
 ABORT 195
 ALE 197
 B/W 198
 CPA 198
 CPB 198
 CPI 198
 D 197
 DBE 197
 FIQ 195
 IRQ 195
 M 196
 MREQ 195
 OPC 195
 PH1 195
 PH2 195
 R/W 195
 RESET 196
 SEQ 196
 TRANS 196
 VDD 196
 VSS 196
SIN 119
single data transfer 79, 221
 syntax 80
software interrupt 51, 204, 233
SQT 119
SrcEdit 8
stack
 allocation 167
 extension 164, 169
stack extension 169
stacking 84
 pop from stack 86
 register list 88
 using R15 88
 using the base register 88
 when the base register is R15 88
stacks
 decrement 82
 increment 82
 post-decrement 82
 pre-decrement 82
start address 126
STF 115
STM 84, 226
 abort during 231
STMEA 85
STMED 85
STMFA 85
STMFD 85, 86
storage manager
 description 165
storage-area location counter 126, 127
store
 multiple registers 97
  see also STM
 register from memory location 97
  see also STR
STR 97, 223
STRB 222
string
 $ in 59
 constant 59
 conversion 58

quotes in 59
spaces in 59
values 58
strings, used as operands 62
STRONG 137
SUB 75, 76
Subtract 97
see also SUB
Subtract with Carry 97
see also SBC
SUF 119
supervisor calls 92
syntax 92
supervisor mode 46, 53
SWI 204, 233
XOS_Heap 163
XOS_Module 163
SWP 89, 98
symbol 56
external 136
symbol attributes
common area 136
comon area definition 136
read only area 136
read only code 136
read-write data 136
relocatable 136
symbols
assigning 123
syntax, label, instruction, command 55

T

tab character 56
TAN 120
TEQ 53, 77
TERSE 140
Test and Mask 97
see also TST
Test Equivalence 97
see also TEQ
Texc 206
Tfiq 206
Throwback 8
Tldm 206
TRANS 196
pin 79
T 80
{TRUE} 59, 130
TST 77
Tsyncmax 206

U

unary operations 119
undefined instruction trap 51, 52, 205, 242
unsigned integers 58
user mode 46, 53

V

{VAR} 128, 130
variable types 128
variables 128
declaring 128
global 128
substitution using $ 129
VDD 196
vectors
address and definitions 52
summary 206
VSS 196

W

W bit 222
WEAK Linker option 136
WEND 141
WFC 118
WFS 118
WHILE condition 142
wimp slot
contents 166

Word, loading from an unknown alignment 188
write back, stacking application 85

Z

Zero flag 48

Reader's Comment Form

Acorn Assembler Release 2

We would greatly appreciate your comments about this Manual, which will be taken into account for the next issue:

Did you find the information you wanted?

Do you like the way the information is presented?

General comments:

If there is not enough room for your comments, please continue overleaf

How would you classify your experience with computers?

☐ **Used computers before** ☐ **Experienced User** ☐ **Programmer** ☐ **Experienced Programmer**

Cut out (or photocopy) and post to:

Dept RC, Technical Publications
Acorn Computers Limited
645 Newmarket Road
Cambridge CB5 8PB
England

Your name and address:

This information will only be used to get in touch with you in case we wish to explore your comments further

Acorn