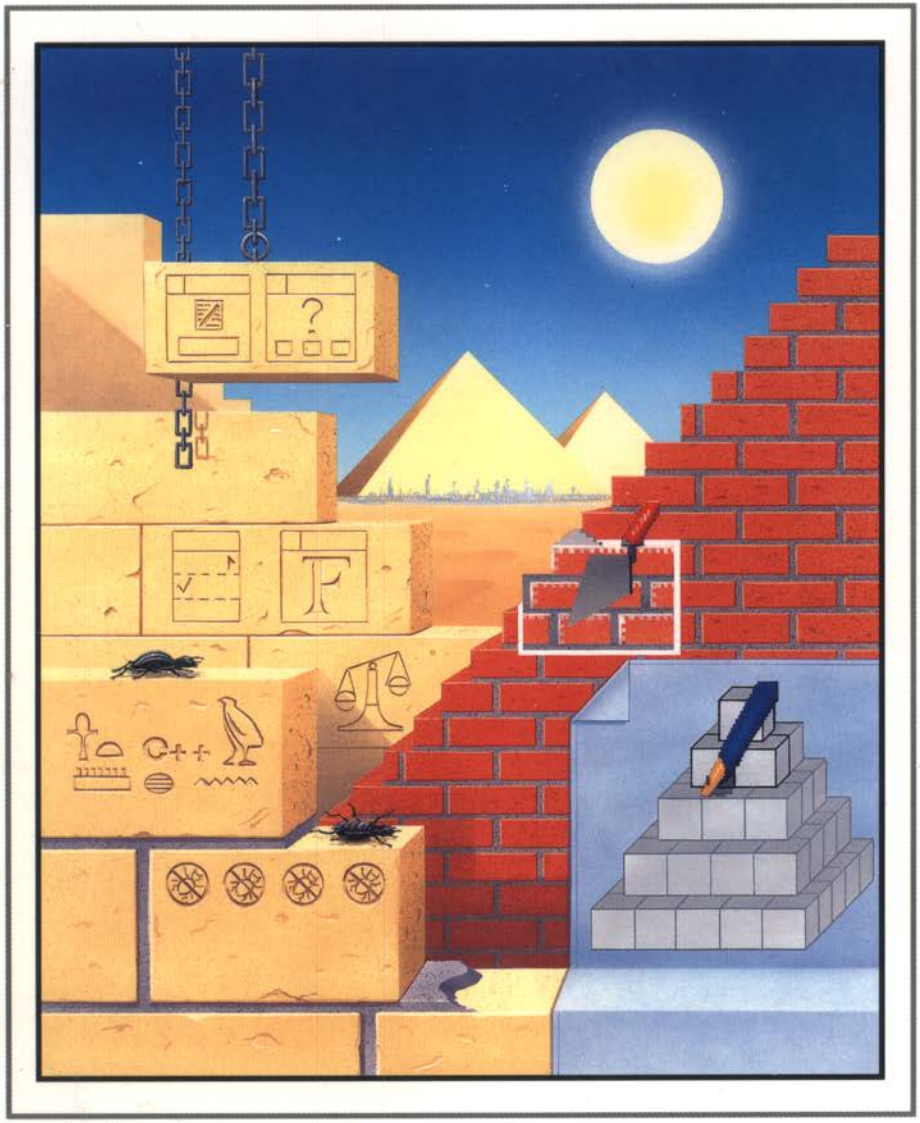
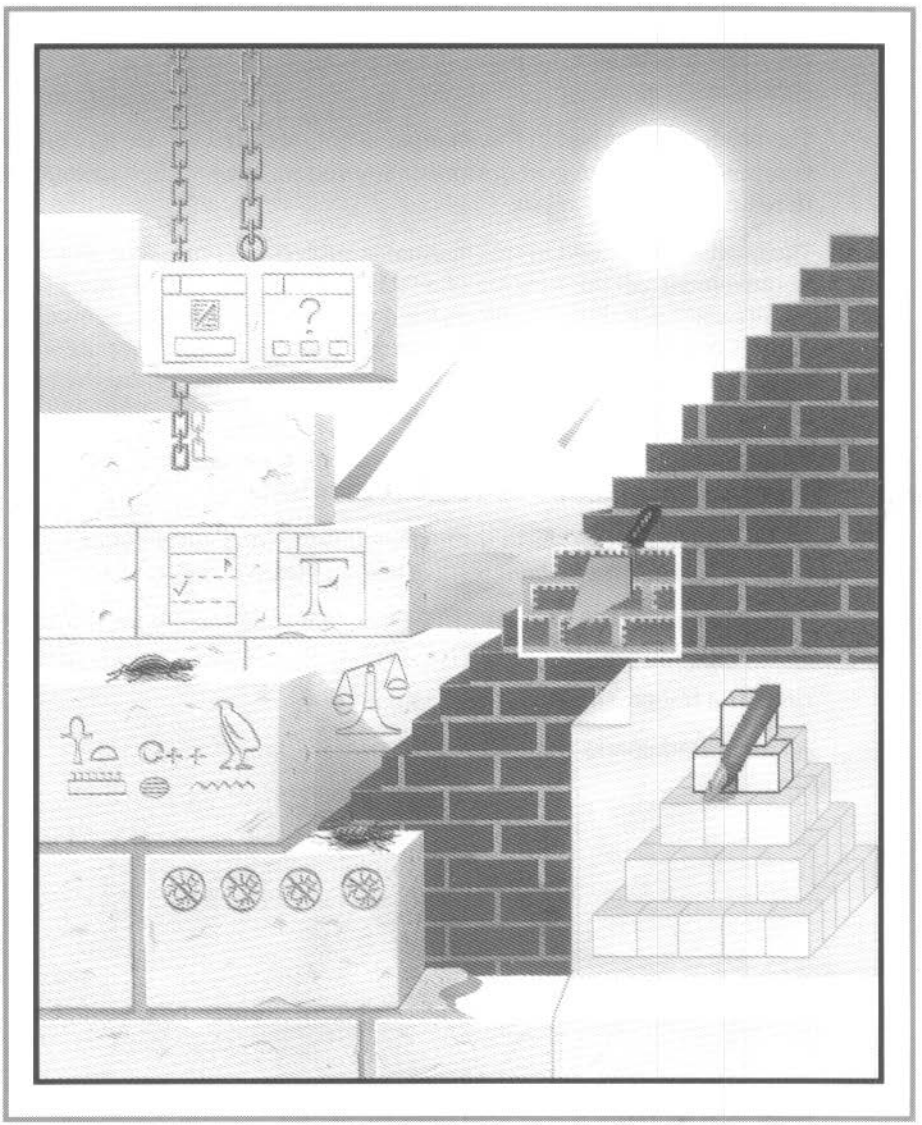
Acorn Assembler



Acorn

Acorn Assembler



Acorn

Published by Acorn Computers Technical Publications Department.

The product described in this manual is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

The product described in this manual is 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 distribution network. Your supplier is available to help resolve any queries you might have.

Published by Acorn Computers Limited
ISBN 1 85250 167 7
Part number 0484,233
Issue 1, December 1994

Contents

1 Introduction

Acorn 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 Assembler may be used with Acorn C/C++ (a part of this product) to provide an environment for mixed C, C++ and assembler development.

This product 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
- design RISC OS desktop interfaces and test their functionality
- use the Toolbox to interact with those interfaces.

Most of the tools in this product are also of general use for constructing applications in other programming languages, such as C and C++. These non-language-specific tools are described in the accompanying *Desktop Tools* guide.

Installation

Installation of Acorn Assembler is described in the chapter *Installing Acorn C/C++* on page 7 of the accompanying *Desktop Tools* guide.

Assembler tools

The assembler provided includes the following features:

- full support of the ARM instruction set, for all versions up to and including the ARM7M core
- global and local label capability
- powerful macro processing
- comprehensive expression handling
- conditional assembly
- repetitive assembly
- comprehensive symbol table printouts
- pseudo-opcodes to control printout.

Objasm

The Assembler 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 chapter *Link* on page 137 of the accompanying *Desktop Tools* guide.

This user guide

This document is a reference guide to ObjAsm, which is the only tool in this product which is not used for programming in other languages. The others are described in the accompanying *Acorn* C/C++ and *Desktop Tools* guides. It is assumed that you are familiar with other relevant Archimedes documentation, such as the:

- *Welcome Guide* supplied with your computer
- RISC OS 3 *User Guide*
- RISC OS 3 *Programmer's Reference Manual*.

You may also find useful one or more of the following books:

- ARM *Assembly Language Programming* / P.J. Cockerell – Computer Concepts/MTC, 1987.
- *Archimedes Assembly Language: A Dabhand Guide* / M. Ginns – Manchester, UK: Dabs Press, 1988.
- *The* ARM RISC *Chip – A Programmer's Guide* / A. van Someren and C. Atack – Wokingham, UK: Addison-Wesley, 1993.

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»`	Italic guillemots « » enclose optional items in the syntax. For example, the Assembler ObjAsm 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`

Part 1 – Using the assembler

2 ObjAsm

ObjAsm is the ARM assembler forming part of the Acorn C/C++ 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.

The sources for large programs can be split into several files, each of which only need be re-assembled to an object file when you have altered it.

An example use of ObjAsm would be 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 Desktop tools, with the common features described in the chapter *General features* on page 103 of the accompanying *Desktop Tools* 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.

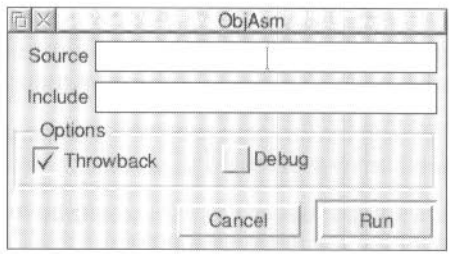
There is no file type to double click on to start ObjAsm – it owns no file type unlike, for example, Draw.

Starting ObjAsm

Like other non-interactive Desktop 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 `AcornC_C++.Tools` directory, then double click Select on !ObjAsm. The ObjAsm main icon appears on the icon bar:

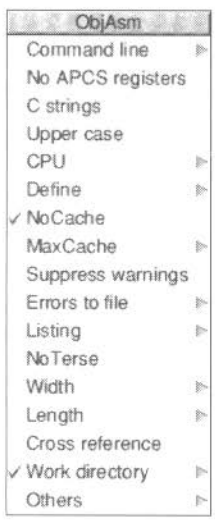
Clicking Select on this icon or dragging an assembly language source file from a directory display to this icon brings up the ObjAsm **SetUp** dialogue box:



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.

The SetUp dialogue box

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.

Include

The **Include** SetUp dialogue box icon adds directories to the source file search path so that arguments to GET/INCLUDE directives (see page 142) do not need to be fully qualified. The search rule used is similar to the ANSI C search rule – the current place being the directory in which the current file was found.

The directories are searched in the order in which they are given in the **Include** icon.

Options

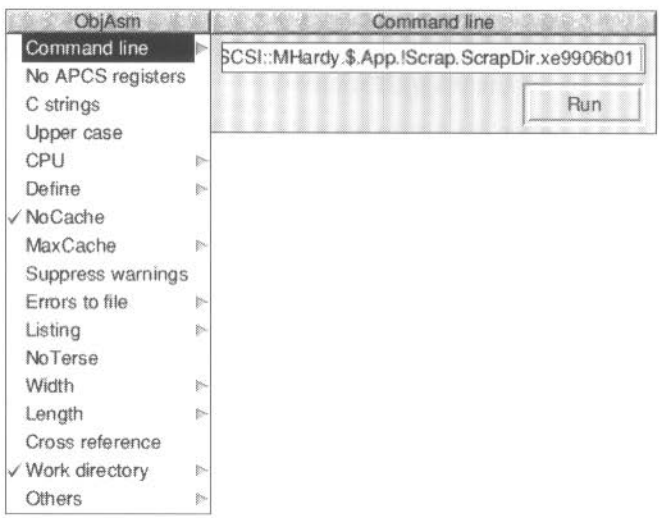
The **Throwback** option switches editor throwback on (the default) or off. When enabled, if the DDEUtils module and SrcEdit are loaded, any assembly errors cause the editor to display an error browser. Double clicking Select on an error line in this browser makes the editor display the source file containing the error, with the offending line highlighted. For more details, see the chapter *SrcEdit* on page 73 of the accompanying *Desktop Tools* guide.

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 Link's **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.

The SetUp menu

The command line

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.

Controlling syntax

The next few entries in the SetUp menu all control the acceptable syntax for the Assembler:

No APCS registers specifies whether the variant of the ARM Procedure Call Standard used by RISC OS is in use, or the APCS is not in use at all. By default the APCS is in use, and ObjAsm pre-declares extra register names and variables, and also specifies some attributes of *code areas*:

- The following extra register names are pre-declared: `a1-a4`, `v1-v6`, `sl`, `fp`, and `ip`. (This is in addition to the default pre-declared register names `R0-R15`, `r0-r15`, `sp`, `SP`, `lr`, `LR`, `pc` and `PC`.)
- The ObjAsm built-in variable `{CONFIG}` is set to 26. This does not generate particular ARM-specific code, but allows the Linker to warn of any mismatch between files being linked, and also allows programs to use the standard built-in variable `{CONFIG}` to determine what code to produce.
- *Code areas* are marked as using `sl` for the stack limit register, following the APCS.

When this menu option is chosen (i.e. it has a tick beside it), the APCS is not in use, and so the above points no longer hold.

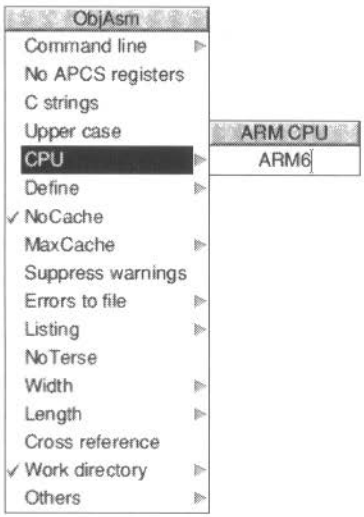
You can specify other APCS variants using the `-APCS` option in the **Others** writable field at the bottom of the menu; see *Specifying other command line options* on page 18, and *Command line options not available from the desktop* on page 23.

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.

Upper case, when chosen, makes ObjAsm recognise instruction mnemonics only if they are entirely in upper case. By default, **Upper case** is not chosen, and ObjAsm recognises mnemonics that are entirely in upper or lower case (but not a mixture of both).

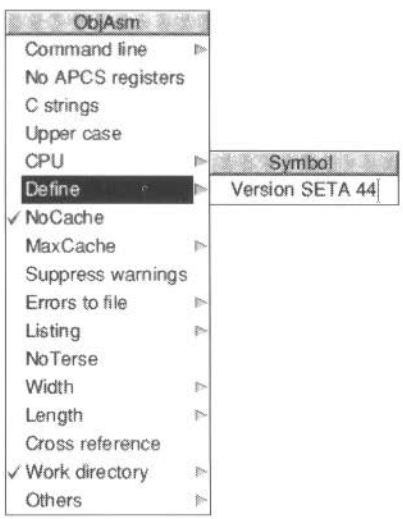
This option is provided mainly to support old code that might have used lower case versions of instruction mnemonics as macro names; it allows the macros to still be recognised as such.

CPU sets the target ARM core. Currently this can take the values ARM6, ARM7 and ARM7M, and defaults to ARM6. Some processor specific instructions will produce warnings if assembled for the wrong ARM core:



Predefining a variable

The next entry – **Define** – allows you to set an initial value for an assembler global variable:



You must give a valid variable name, followed by a **SETL**, **SETA** or **SETS** directive, followed by a value. The value may be a simple constant or a constant expression (in ObjAsm syntax) of appropriate type – logical, arithmetic or string for **SETL**, **SETA** and **SETS** respectively – provided that its value can be computed at the start of assembly. The variable is set as if the directive occurs before the start of the source; an implicit **GBLL**, **GBLA** or **GBLS** directive is also executed. In the case of **SETS**, quotation marks are usually necessary around the value, since it is a string expression.; these must be escaped by preceding each with a backslash ('\').

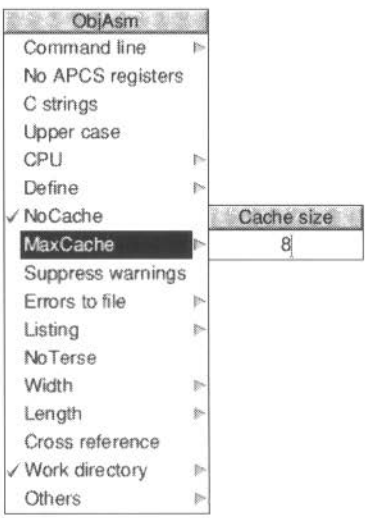
Controlling cacheing

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. This makes very heavy use of memory, and so is unsuitable for smaller machines.

The next two menu options control this cacheing:

NoCache disables cacheing when chosen, which is the default. When **NoCache** is switched off, cacheing is enabled.

MaxCache allows you to specify the maximum amount of RAM to be used for cacheing source files, provided that **NoCache** is off. The maximum cache is specified in megabytes; the default is 8MB:

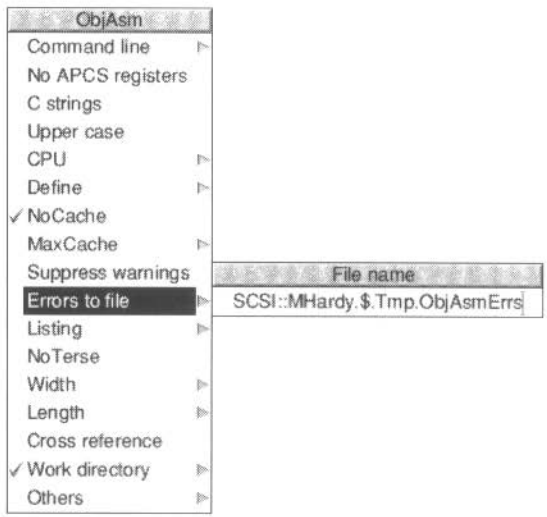


Handling warnings and errors

The next menu options control handling of warnings and errors:

Suppress warnings, when chosen, turns off the warning messages that ObjAsm generates. It is off by default (i.e. warning messages are generated).

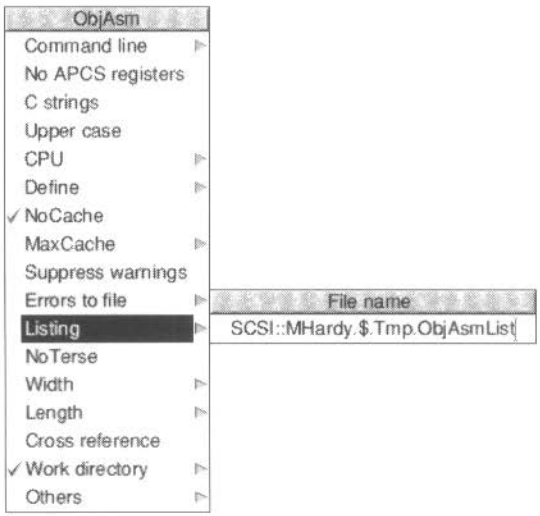
Errors to file allows you to specify a file to which error messages are output for later inspection:



Listings

The next options control whether or not a listing is produced, and its format:

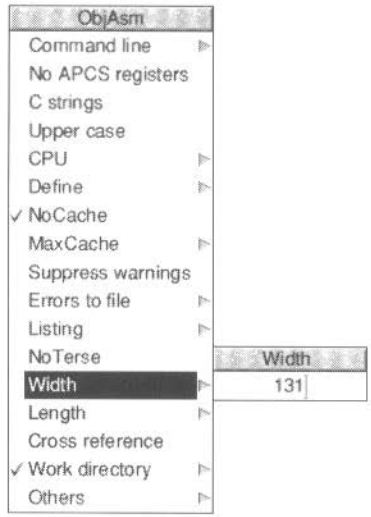
The **Listing** option enables assembler source code to be sent to a file:



This option turns on the Assembler listing, and during assembly the source code, object code, memory addresses and reference line numbers will be sent to the named file. **Listing** is off by default.

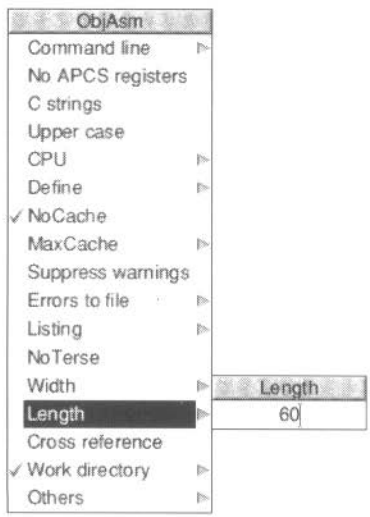
NoTerse modifies the listing that is output, which normally only includes the conditionally assembled parts of your program. If you choose **NoTerse**, conditionally non-assembled parts are listed as well. **NoTerse** is off by default.

Width sets the width, in characters, of the listing that is output:



This should be between 1 and 254. The default width is 131; a width of 76 is suitable for a Mode 12 RISC OS window.

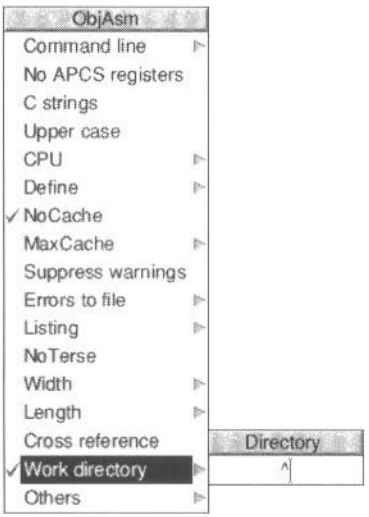
Length sets the number of lines per page for printer output. At the end of each page ObjAsm inserts a form feed character. The default length is 60:



If you choose **Cross reference**, then after assembly ObjAsm outputs an alphabetically sorted cross reference of all symbols encountered. 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.

Choosing your work directory

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::HardDisc4.$.Source.s.foo
```

containing the line:

```
GET    s.macros
```

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

```
adfs::HardDisc4.$.Source.s.^.s.macros
```

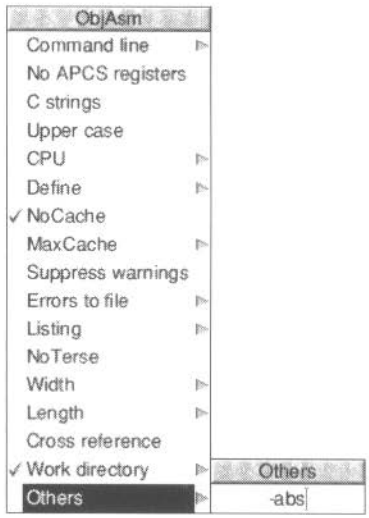
(i.e. `adfs::HardDisc4.$.Source.s.macros`)

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 ^.

Specifying other command line options

The **Others** option on the SetUp menu leads to a writable icon in which you can add an arbitrary extra section of text to the command line to be passed to ObjAsm:

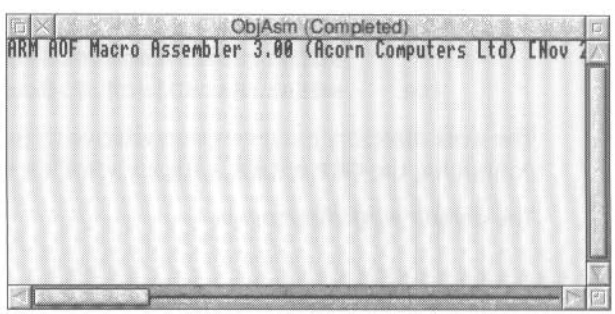


This facility is useful if you wish to use any feature which is not supported by any of the other entries on the SetUp dialogue box and menu. This may be because the feature is used very little, or because it may not be supported in the future.

For a full description of command line options, see *ObjAsm command lines* on page 22.

ObjAsm output

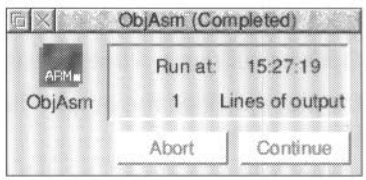
ObjAsm outputs text messages as it proceeds. These include source listings and symbol cross references (as described in the previous sections). By default any such text is directed into a scrollable output window:



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.

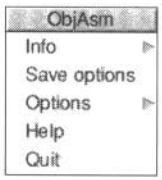
Both the above ObjAsm output displays follow the standard pattern of those of all the non-interactive Desktop tools. The common features of the non-interactive Desktop tools are covered in more detail in the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide. Both ObjAsm output displays and the menus brought up by clicking Menu on them offer the standard features, which allow you to abort, pause or continue execution (if the execution hasn't completed), to save output text to a file, or to repeat execution.

ObjAsm error messages appear in the output viewer, with copies in the editor error browser when throwback is working. The appendix *Error messages* on page 183 of this manual contains a list of common 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. 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 Desktop tools:

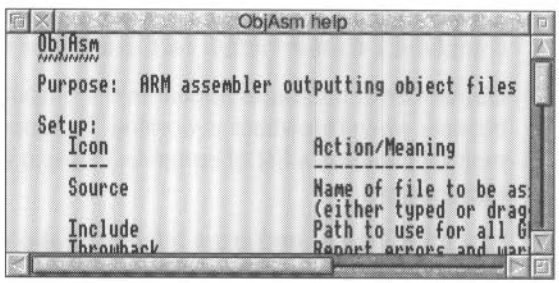


Save options saves all the current ObjAsm options, including both those set from the SetUp dialogue box and from the **Options** item on this menu. When ObjAsm is restarted it is initialised with these options rather than the defaults.

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. **Auto run** is off by default.
- 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. **Auto save** is 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:

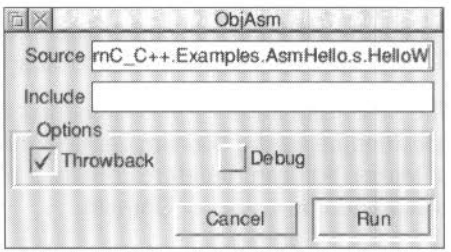


Example ObjAsm session

The programming example `AcornC_C++.Examples.AsmHello` is a non-desktop free standing command line program written in assembly language. It outputs the text 'Hello World'.

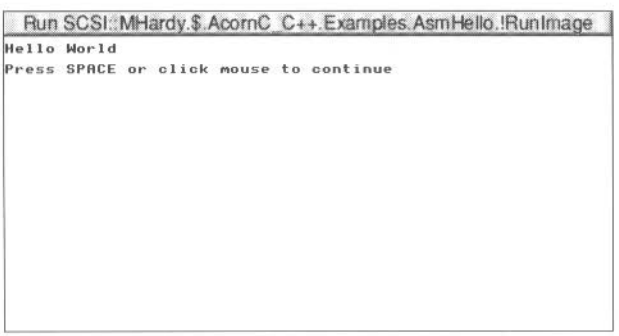
The assembly language source is held in the `s` subdirectory, in the file `HelloW`. The code demonstrates the ObjAsm directives needed for a free standing program;

To assemble `HelloW`, first run !Objasm and !Link by double clicking on them. Drag the `HelloW` 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 the `o` subdirectory. Drag the object file to the Link icon, and **Run** Link to produce an AIF executable image file, the link having the `HelloW` object file as its only input file. Save the image file in `AcornC_C++.Examples.AsmHello.!RunImage`. The command line program is now ready for use.

To run the program under the desktop, double click on it. A window appears with the text 'Hello World':



As the window instructs you to do, press the space bar or click on your mouse. The window disappears.

ObjAsm command lines

ObjAsm, in common with the other non-interactive Desktop 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 Acorn Desktop Assembler product. However, as all the useful ObjAsm features can 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.

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 to invoke ObjAsm takes either of the forms:

```
ObjAsm «options» sourcefile objectfile
ObjAsm «options» -o objectfile sourcefile
```

The options are listed below, split into two sections: those for which there is a direct equivalent in the SetUp dialogue box or menu, and those others for which there is no equivalent. Upper case is used to show the allowable abbreviations. Note that to understand what many of these options do it may be necessary to refer to some of the documentation above.

Command line options available from the desktop

The table below shows the various command line options that correspond to the options available from the SetUp dialogue box and menu, together with a reference to the desktop equivalent, which you should see for full details of the option:

Command line option	Desktop equivalent	Page
`-I dir«,dir»`	**Include** writable icon in dialogue box	9
`-ThrowBack`	**Throwback** option icon in dialogue box	9
`-G`	**Debug** option icon in dialogue box	10
(*See* `-Apcs` *below*)	**No APCS registers** in menu	11
`-Esc`	**C strings** in menu	11
`-UpperCase`	**Upper case** in menu	11

Command line option	Desktop equivalent	Page
`-CPU` *ARMcore*	**CPU** in menu	12
`-PreDefine` *directive*	**Define** in menu	12
`-NOCache`	**NoCache** in menu	13
`-MaxCache` *n*	**MaxCache** in menu	13
`-NOWarn`	**Suppress warnings** in menu	14
`-ERRors` *errorfile*	**Errors to file** in menu	14
`-LIST` *listingfile*	**Listing** in menu	15
`-NOTerse`	**NoTerse** in menu	15
`-WIdth` *n*	**Width** in menu	16
`-Length` *n*	**Length** in menu	16
`-Xref`	**Cross reference** in menu	17
`-Desktop` *dirname*	**Work directory** in menu	17

Command line options not available from the desktop

The table below shows those command line options for which there is no direct equivalent in the SetUp dialogue box or menu. Should you need to use any of these more esoteric options from the desktop, you can add them to the SetUp menu's **Others** option (see *Specifying other command line options* on page 18).

Command line option	Description
`-Help`	Outputs a summary of the command line options.
`-VIA` *filename*	Reads in extra command line arguments from the given *filename*.
`-LIttleend`	Assemble code suitable for a little-endian ARM, by setting the built-in variable `{ENDIAN}` to "`little`".
`-BIgend`	Assemble code suitable for a big-endian ARM, by setting the built-in variable `{ENDIAN}` to "`big`".

Command line option	Description
`-Apcs` *`option«/qualifier»«/qualifier...»`*	
	Specifies whether the ARM Procedure Call Standard is in use, and also specifies some attributes of CODE AREAs. By default the register names `R0-R15`, `r0-r15`, `sp`, `SP`, `lr`, `LR`, `pc`, and `PC` are pre-declared. If the APCS is in use the following register names are also pre-declared: `a1-a4`, `v1-v6`, `sl`, `fp`, and `ip`.
	There are two APCS options: `NONE` and `3`. The SetUp menu's **No APCS registers** option (page 11) – when chosen – declares the APCS in use as `NONE`. The default behaviour is to use the `3/26bit/SWStackcheck` APCS variant used by RISC OS
	The qualifiers – which should only be used with option `3` – are as follows:
`/REENTrant`	Sets the reentrant attribute for any code AREAs, and predeclares `sb` (static base) in place of `v6`.
`/32bit`	Is the default setting and informs the Linker that the code being generated is written for 32 bit ARMs. The built-in variable `{CONFIG}` is also set to `32`.
`/26bit`	Tells the Linker that the code is intended for 26 bit ARMs. The built-in variable `{CONFIG}` is also set to `26`.
	Note that these options do not of themselves generate particular ARM-specific code, but allow the Linker to warn of any mismatch between files being linked, and also allow programs to use the standard built-in variable `{CONFIG}` to determine what code to produce.
`/SWSTackcheck`	Marks CODE AREAs as using sl for the stack limit register, following the APCS (the default setting).
`/NOSWstackcheck`	Marks CODE AREAs as not using software stack-limit checking, and predeclares an additional v-register, `v6` if reentrant, `v7` if not.
`-Depend` *`dependfile`*	Saves source file dependency lists, which are suitable for use with 'make' utilities.

Command line option	Description
`-ABSolute`	Accepts AAsm source code to provide some backwards compatibility in this release. See the appendix *Support for AAsm source* on page 211.
`-FRom` *`filename`*	Supported, for backward compatibility with previous release.
`-TO` *`filename`*	Supported, for backward compatibility with previous release.
`-Print`	Supported, for backward compatibility with previous release.
`-Quit`	Recognised but ignored, for backward compatibility with previous release.

Part 2 – Assembly language details

3 The ARM CPU

The ARM (Advanced Risc Machine) is a general purpose32 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.

Introduction

Bus widths

The ARM2 and ARM3 have a 32 bit data bus and a 26 bit address bus. On later versions of the ARM, both the data bus and the address bus are a full 32 bits wide.

Instruction set

All instructions fit into one 32 bit word, and they can all be made conditional.

The ARM instruction set comprises ten basic classes of instruction:

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

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 the 32 bit registers. Three instruction classes control the transfer of data between main memory and the register bank, one optimised for flexibility of addressing, another for rapid

context switching, and the third for swapping data. Two instruction classes control the flow and privilege level of execution. The remaining three classes are dedicated to the control of external coprocessors, 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.

The instruction set is detailed in the chapter CPU *instruction set* on page 53.

Pipelining

Pipelining is employed so that all parts of the processing and memory systems can operate continuously.

The ARM uses a 3-stage instruction pipeline. This allows it to execute one instruction, and at the same time both to decode the following instruction, and to fetch the one after that from memory.

Memory interface

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).

Data types

The processor can access two types of data:

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

where words must be aligned to four byte boundaries.

Instructions are fetched as words, and so must be aligned to four byte boundaries. Data operations (eg ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words, and can put a full 26 or 32 bit address (depending on the processor variant) – with bits 0 and 1 set as required – on to the address bus.

Block diagram of core

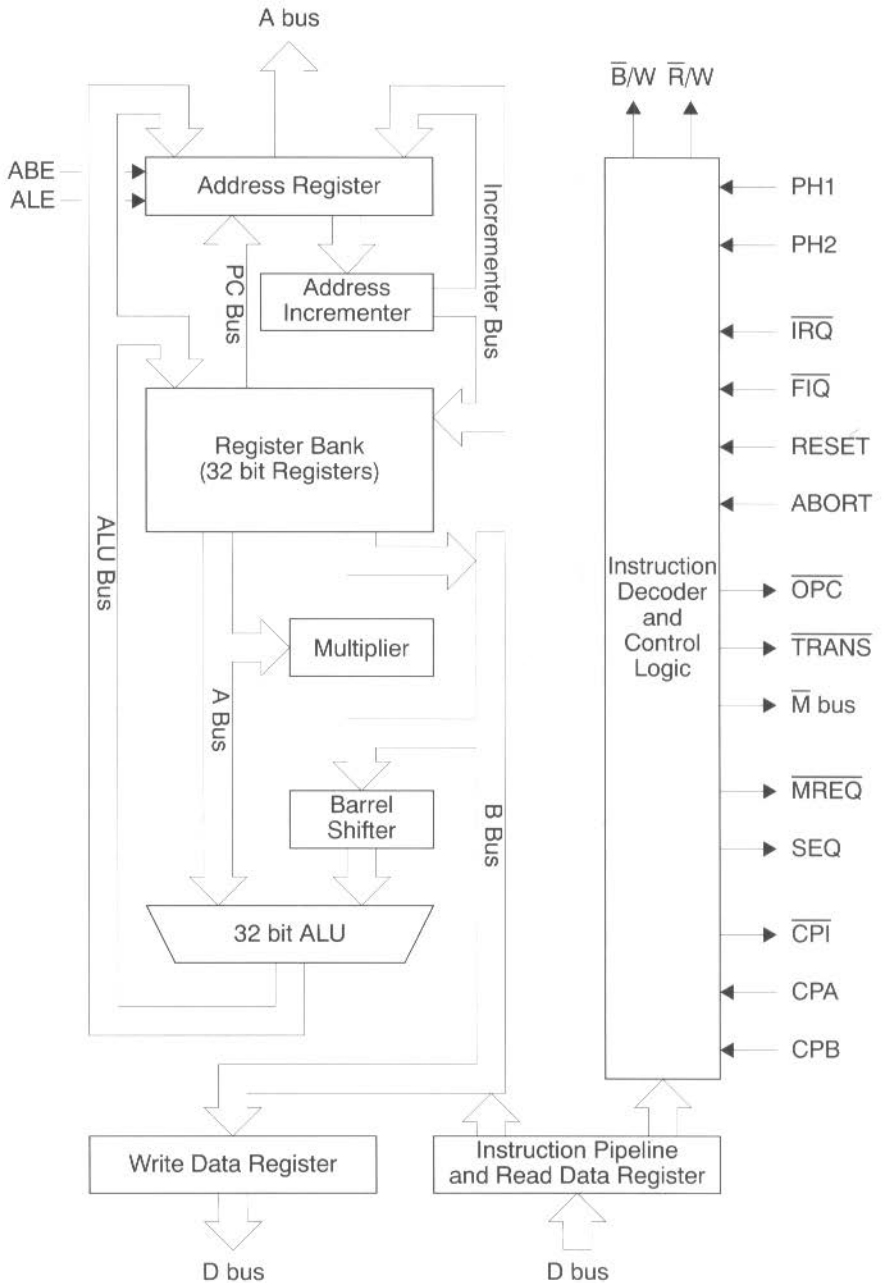


Figure 3.1 ARM Core block diagram

26 bit architecture

This section describes the architecture of the ARM2 and ARM3 series, which only supported a 26 bit address space. However, as we shall see in the section 32 *bit architecture* on page 36, much of this is also relevant to later series of ARM when used so as to provide backward-compatibility with the earlier 26 bit processors.

Processor modes

These older ARM series support four modes of operation:

- User mode: the normal program execution state
- Fast Interrupt mode (abbreviated to FIQ mode): designed to support a data transfer or channel process
- Interrupt mode (abbreviated to IRQ mode): used for general purpose interrupt handling
- Supervisor mode (abbreviated to SVC mode): a protected mode for the operating system, also entered after a data or instruction prefetch abort, or when an undefined instruction is executed.

Mode changes may be made under software control or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as privileged modes, will be entered to service interrupts or exceptions or to access protected resources.

Registers

The ARM has a number of 32 bit registers, 16 of which are visible to the programmer at any time. This subset depends on the processor mode:

- Normally the ARM operates in User mode, with registers R0 to R15 visible.
- When in the other privileged modes (see the section *Processor modes* on page 32) special private registers are switched in. If code running in these modes needs to use any of the shared registers, it should save their contents in memory using one of the block data transfer instructions available for this purpose; see *Block data transfer* (LDM, STM) on page 88.

The IRQ and SVC modes have two private registers mapped to R13 and R14 (R13_irq and R14_irq, and R13_svc and R14_svc respectively).

The FIQ mode has more private registers so that FIQ code – which needs to respond quickly – is less likely to need to use any of the shared registers, and so will be spared the overhead of saving them to a stack. Its seven private registers are mapped to R8-R14 (R8_fiq-R14_fiq).

The register bank organisation is shown in the figure 26 *bit register organisation* below:

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 3.2 26 bit register organisation

All registers are general purpose and may be used to hold data or address values, except for R15 and R14:

- R15 contains the Program Counter (PC) and the Processor Status Register (PSR). See the section *Register* R15 below.
- 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. See the section *Register* R14 below.

R13 is also often used for a special purpose:

- R13 is, **by convention only**, often used as a private stack pointer for a processor mode.

The private copies of R13 and R14 allow each mode to have a private stack pointer and link register. SVC 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.

Register R15

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

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 four byte boundaries.

Special bits in some instructions allow the PC and PSR to be treated together, or separately, as required. The allocation of the bits within the register R15 is shown in the figure *The Program Counter* (PC) *and Process Status Register* (PSR) below.

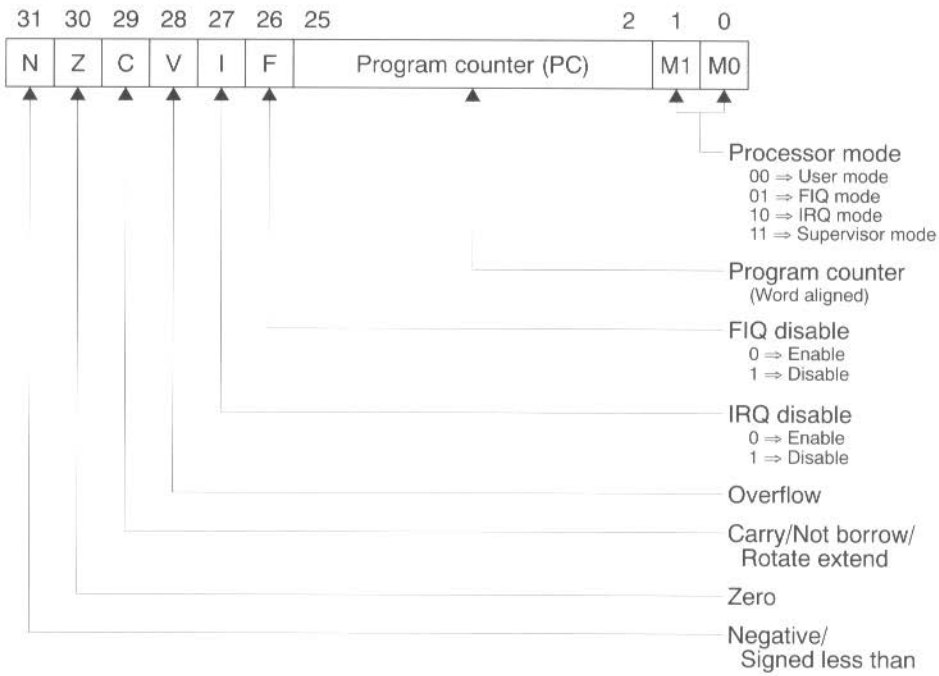


Figure 3.3 The Program Counter (PC) *and Process Status Register* (PSR)

The mnemonics for the four condition flags are derived as follows:

N	**N**egative flag
Z	**Z**ero flag
C	**C**arry flag
V	O**v**erflow flag

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

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 (see page 63). 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.

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 sections *Forcing transfer of the user bank* on page 93 and *Using* R15 *as the destination* on page 71.

32 bit architecture

The ARM architecture changed significantly with the introduction of the ARM6 series. This section describes the **differences** in behaviour of more recent ARM processors.

New features in ARM6

The most notable change made in the ARM6 series was to extend the program counter to a full 32 bits. As a result:

- The PSR had to be separated from the PC into its own register, the CPSR (*Current Program Status Register*).
- The PSR can no longer be saved with the PC when changing processor modes; instead, each privileged mode now has an extra register – the SPSR (*Saved Program Status Register*) – to hold the previous mode's PSR.
- Instructions have been added to use these new status registers.

A further change was the addition of extra privileged processor modes, allowed by the PSR now having a full 32 bits to use. These modes are used to handle Undefined instruction and Abort exceptions. Consequently:

- Undefined instructions, aborts, and supervisor code no longer have to share the same mode. This has removed restrictions on Supervisor mode programs which existed on earlier ARMs.

Processor configuration

The availability of these features in the ARM6 series (and other later compatible chips) is set by one of several on-chip control registers. One of three *processor configurations* can be selected:

- **26 bit program and data space**. This configuration forces ARM to operate with a 26 bit address space. In this configuration only the four 26 bit modes are available (see *Processor modes* below); it is impossible to select a 32 bit mode.

  This configuration is set at reset on all current ARM6 and 7 series processors.
- **26 bit program space and 32 bit data space.** This is the same as the 26 bit program and data space configuration, except that address exceptions are disabled to allow data transfer operations to access the full 32 bit address space.
- **32 bit program and data space**. This configuration extends the address space to 32 bits, and introduces major changes to the programmer's model. In this configuration you can select any of the 26 bit and the 32 bit processor modes (see *Processor modes* below).

Processor modes

When configured for a 32 bit program and data space, the ARM6 and ARM7 series support ten overlapping *processor modes* of operation:

- User mode: the normal program execution state – or
  User26 mode: a 26 bit version of the above
- FIQ mode: designed to support a data transfer or channel process – or
  FIQ26 mode: a 26 bit version of the above
- IRQ mode: used for general purpose interrupt handling – or
  IRQ26 mode: a 26 bit version of the above
- SVC mode: a protected mode for the operating system – or
  SVC26 mode: a 26 bit version of the above
- Abort mode (abbreviated to ABT mode): entered after a data or instruction prefetch abort
- Undefined mode (abbreviated to UND mode): entered when an undefined instruction is executed.

The distinction between processor **modes** and **configurations** is important, and will be rigidly adhered to in the rest of this manual.

The 26 bit processor modes

When in a 26 bit processor mode, the programmer's model reverts to that of earlier 26 bit ARM processors. The behaviour is the same as that of the ARM2aS macrocell with the following alterations:

- Address exceptions are only generated by ARM when it is configured for 26 bit program and data space.
  In other configurations the OS may still simulate the behaviour of address exception, using external logic such as a memory management unit to generate an abort if the 64Mbyte range is exceeded, and converting that abort into an 'address exception trap' for the application.
- The new instructions to transfer data between general registers and the program status registers remain operative. The new instructions can be used by the operating system to return to a 32 bit mode after calling a binary containing code written for a 26 bit ARM.
- When in a 32 bit program and data space configuration, all exceptions (including Undefined Instruction and Software Interrupt) return the processor to a 32 bit mode, so the operating system must be modified to handle them.
- If the processor attempts to write to a location between &0 and &1F inclusive (i.e. the exception vectors), hardware prevents the write operation and generates a data abort. This allows the operating system to intercept all

changes to the exception vectors and redirect the vector to some veneer code. The veneer code should place the processor in a 26 bit mode before calling the 26 bit exception handler.

In all other respects, when operating in a 26 bit mode the ARM behaves as like a 26 bit ARM. (See the section 26 *bit architecture* on page 32.) The relevant bits of the CPSR appear to be incorporated back into R15 to form the PC/PSR with the I and F bits in bits 27 and 26. The instruction set behaves like that of the ARM2aS macrocell, with the addition of the MRS and MSR instructions.

RISC OS processor configuration and modes

For details, see the section RISC OS *processor configuration and modes* on page 167.

Registers

The registers available in the ARM6 and ARM7 series are:

User and User26 mode	SVC and SVC26 mode	IRQ and IRQ26 mode	ABT mode	UND mode	FIQ and FIQ26 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_abt	R13_und	R13_fiq
R14	R14_svc	R14_irq	R14_abt	R14_und	R14_fiq
R15 (PC)					

CPSR					
	SPSR_svc	SPSR_irq	SPSR_abt	SPSR_und	SPSR_fiq

Figure 3.4 32 bit register organisation

These are similar to those available in the ARM2 and ARM3 series registers. The key differences are:

- the PC is a full 32 bits wide
- the PSR is held in its own register, the CPSR (see the section *The CPSR and SPSR registers* below)
- each privileged mode has a private SPSR register in which to save the CPSR
- there are two new privileged modes, each of which has private copies of R13 and R14.

The CPSR and SPSR registers

The allocation of the bits within the CPSR (and the SPSR registers to which it is saved) is shown in the figure *The Current Process Status Register* (CPSR) below.

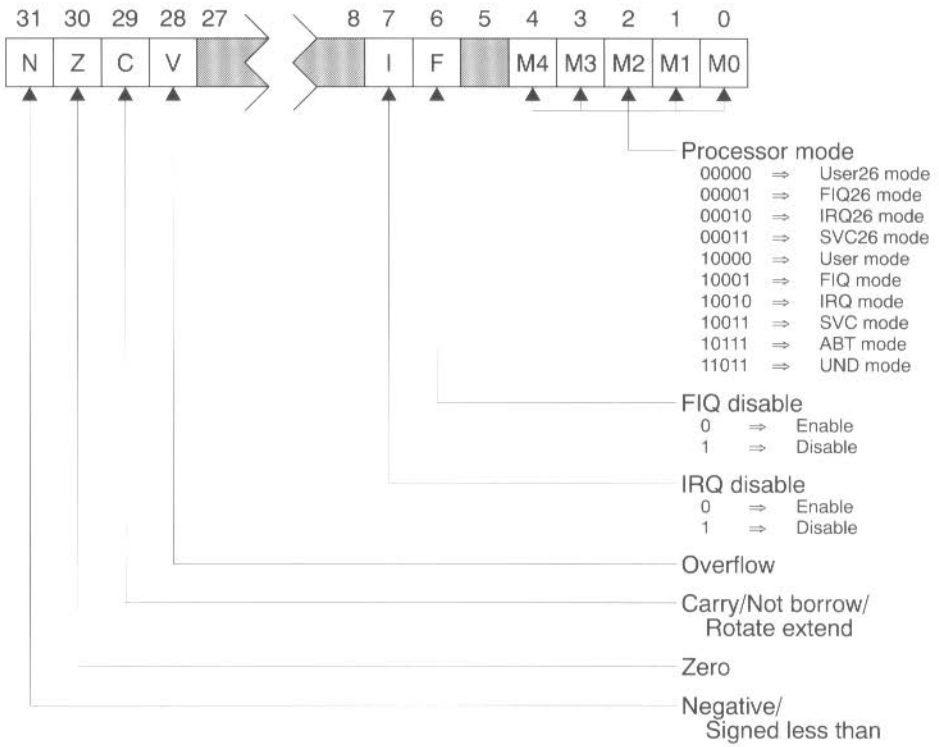


Figure 3.5 The Current Process Status Register (CPSR)

Exceptions

This last section of the chapter is 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 ARM *assembly language* on page 47.

This section describes the general behaviour of the ARM, rather than its behaviour under RISC OS. For details specific to RISC OS you **must** also see the chapter *Exception handling* on page 167.

Introduction

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, in a 26 bit configuration to the appropriate R14, or in a 32 bit configuration to the appropriate R14 and SPSR. 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 (Fast interrupt request)

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the $\overline{\text{FIQ}}$ 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 successfully FIQed it will:

1 Save R15 in R14_fiq, and (for 32 bit configuration ARMs) save the CPSR in SPSR_fiq.

2 Force the mode bits to FIQ mode and set the F and I bits in the PSR.

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 IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the $\overline{\text{IRQ}}$ pin. 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 ARM is successfully IRQed it will:

1 Save R15 in R14_irq, and (for 32 bit configuration ARMs) save the CPSR in SPSR_irq.

2 Force the mode bits to IRQ mode and set the I bit in the PSR.

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

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

On a 32 bit configuration processor, address exceptions are **never** generated, and you may therefore ignore this section for such processors.

On a 26 bit configuration processor, 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 the mode bits to SVC mode and set the I bit in the PSR.

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 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, or (for 32 bit configuration ARMs) save R15 in R14_abt and save the CPSR in SPSR_abt.

2 Force the mode bits to SVC mode or (for 32 bit configuration ARMs) ABT mode and set the I bit in the PSR.

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

To continue after a Prefetch abort use `SUBS PC,R14,#4` (where `R14` is `R14_svc` or `R14_abt` depending on the processor configuration). 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 and STR) are aborted as though the instruction had not executed.
- Block data transfer instructions (LDM and STM) 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, or (for 32 bit configuration ARMs) save R15 in R14_abt and save the CPSR in SPSR_abt.

2 Force the mode bits to SVC mode or (for 32 bit configuration ARMs) ABT mode and set the I bit in the PSR.

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,#8` (where `R14` is `R14_svc` or `R14_abt` depending on the processor configuration).

Abort during an internal cycle

The ARM ignores aborts signalled during internal cycles.

Using aborts to implement virtual memory systems

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

1. Save R15 in R14_svc, and (for 32 bit configuration ARMs) save the CPSR in SPSR_svc.
2. Force the mode bits to SVC mode and set the I bit in the PSR.
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

When ARM executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that moment, ARM will wait until the coprocessor is ready. If no coprocessor can handle the instruction ARM will take the undefined instruction trap.

When the undefined instruction trap is taken ARM will:

1. Save R15 in R14_svc, or (for 32 bit configuration ARMs) save R15 in R14_und and save the CPSR in SPSR_und.
2. Force the mode bits to SVC mode or (for 32 bit configuration ARMs) UND mode and set the I bit in the PSR.
3. Force the PC to fetch the next instruction from address &4.

The undefined instruction trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware; or for general purpose instruction set extension by software emulation (the floating point instruction set is implemented in software this way).

To return from this trap (after performing a suitable emulation of the required function), use `MOVS PC,R14` (where `R14` is `R14_svc` or `R14_und` depending on the processor configuration). 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 stop the currently executing instruction and start executing no-ops. When RESET goes LOW again, it will:

1. Save R15 in R14_svc, and (for 32 bit configuration ARMs) save the CPSR in SPSR_svc.
2. Force the mode bits to SVC mode and set the F and I bits in the PSR.
3. 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 exception
&0000018	IRQ
&000001C	FIQ

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, plus the time for the longest instruction (typically load multiple registers) to complete, plus the time for address exception or data abort entry, plus the time for FIQ entry. At the end of this time ARM will be executing the instruction at 1CH.

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, plus the time for FIQ or IRQ entry.

The above times can vary considerably between different versions of the ARM, and obviously also depend on clock speeds. For more information you should see the relevant datasheets.

4 ARM assembly language

ARM Assembly Language is the language which ObjAsm parses and compiles to produce object code in ARM Object Format. Information on ObjAsm command line options are detailed in *ObjAsm command lines* on page 22. This chapter details ARM Assembly Language, but does not give examples of its use.

General

Instruction mnemonics and register names may be written in upper or lower case (but not mixed case). Directives must be written in upper case.

Input lines

The general form of assembler input lines is:

```
«label» «instruction» «;comment»
```

A space or tab should separate the label, where one is used, and the instruction. If no label is used the line must begin with a space or tab. Any combination of these three items will produce a valid line; empty lines are also accepted by the assembler and can be used to improve the clarity of source code.

Assembler source lines are allowed to be up to 255 characters long. To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character, '\', at the end of a line. The backslash must not be followed by any other characters (including spaces or tabs). The backslash + end of line sequence is treated by ObjAsm as white space. Note that the backslash + end of line sequence should not be used within quoted strings.

AREAs

AREAs are the independent, named, indivisible chunks of code and data manipulated by the Linker. The Linker places each AREA in a program image according to the AREA placement rules (i.e. not necessarily adjacent to the AREAs with which it was assembled or compiled).

Conventionally, an assembly, or the output of a compilation, consists of two AREAs, one for the code (usually marked read-only), and one for the data which may be written to. A reentrant object will generally have a third AREA marked

BASED sb (see below), which will contain relocatable address constants. This allows the code area to be read-only, position-independent and reentrant, making it easily ROM-able.

In ARM assembly language, each AREA begins with an **AREA** directive. If the **AREA** directive is missing the assembler will generate an AREA with an unlikely name (|$$$$$$$|) and produce a diagnostic message to this effect. This will limit the number of spurious errors caused by the missing directive, but will not lead to a successful assembly.

The syntax of the **AREA** directive is:

AREA *name«,attr»«,attr»...*

You may choose any name for your AREAs, but certain choices are conventional. For example, **|C$$code|** is used for code AREAs produced by the C compiler, or for code AREAs otherwise associated with the C library.

Area attributes

AREA attributes are as follows:

ABS	Absolute: rooted at a fixed address.
REL	Relocatable: may be relocated by the Linker (the default).
PIC	Position Independent Code: will execute where loaded without modification.
CODE	Contains machine instructions.
DATA	Contains data, not instructions.
READONLY	This area will not be written to.
COMDEF	Common area definition.
COMMON	Common area.
NOINIT	Data AREA initialised to zero: contains only space reservation directives, with no initialised values.
REENTRANT	The code AREA is reentrant.
BASED Rn	Static base data AREA containing tables of address constants locating static data items. R*n* is a register, conventionally R9. Any label defined within this AREA becomes a register-relative expression which can be used with **LDR** and **STR** instructions. For full details see the appendix ARM *procedure call standard* on page 249 of the *Desktop Tools* guide.

`ALIGN=`*`expression`*

The **ALIGN** sub-directive forces the start of the area to be aligned on a power-of-two byte-address boundary. By default AREAs are aligned on a 4-byte word boundary, but the expression can have any value between 2 and 12 inclusive.

ORG and ABS

`ORG` *`base-address`*

The **ORG** (origin) directive is used to set the base address and the **ABS** (absolute) attribute of the containing AREA, or of the following AREA if there is no containing AREA. In some circumstances this will create objects which cannot be linked. In general it only makes sense to use **ORG** in programs consisting of one AREA, which need to map fixed hardware addresses such as trap vector locations. Otherwise **ORG** should be avoided.

Symbols

Numbers, logical values, string values and addresses may be represented by symbols. Symbols representing numbers or addresses, logical values and strings are declared using the **GBL** and **LCL** directives, and values are assigned immediately by **SETA**, **SETL** and **SETS** directives respectively (see *Local and global variables* – GBL, LCL *and* SET on page 146). Addresses are assigned by the Assembler as assembly proceeds, some remaining in symbolic, relocatable form until link time.

Symbols must start with a letter in either upper or lower case; the assembler is case-sensitive and treats the two forms as distinct. Numeric characters and the underscore character may be part of the symbol name. All characters are significant.

Symbols should not use the same name as instruction mnemonics or directives. While the assembler can distinguish between the uses of the term through their relative positions in the input line, a programmer may not always be able to do so.

Symbol length is limited by the 255 character line length limit.

If there is a need to use a wider range of characters in symbols, for instance when working with other compilers, use enclosing bars to delimit the symbol name; for example, **|C$$code|**. The bars are not part of the symbol.

Labels

Labels are a special form of symbol, distinguished by their position at the start of lines. The address represented by a label is not explicitly stated but is calculated during assembly.

Local labels

The local label, a subclass of label, begins with a number in the range 0-99. Local labels work in conjunction with the **ROUT** directive and are most useful for solving the problem of macro-generated labels. Unlike global labels, a local label may be defined many times; the assembler uses the definition closest to the point of reference. To begin a local label area use:

«label» ROUT

The label area will start with the next line of source, and will end with the next **ROUT** directive or the end of the program.

Local labels are defined as:

number«routinename»

although *routinename* need not be used; if omitted, it is assumed to match the label of the last **ROUT** directive. It is an error to give a routine name when no label has been attached to the preceding **ROUT** directive.

References to local labels

A reference to a local label has the following syntax:

%*«x»«y»n«routinename»*

% introduces the reference and may be used anywhere where an ordinary label reference is valid.

x tells the assembler where to search for the label; use **B** for backward or **F** for forward. If no direction is specified the assembler looks both forward and backward. However searches will never go outside the local label area (i.e. beyond the nearest **ROUT** directives).

y provides the following options: **A** to look at all macro levels, **T** to look only at this macro level, or, if *y* is absent, to look at all macro from the current level to the top level.

n is the number of the local label.

routinename is optional, but if present it will be checked against the enclosing **ROUT**'s label.

Comments

The first semi-colon on a line marks the beginning of a comment, except where the semi-colon appears inside a string constant. A comment alone is a valid line. All comments are ignored by the assembler.

Constants

Numbers

Numeric constants are accepted in three forms: decimal (e.g. `123`), hexadecimal (e.g. `&7B`), and *`n_xxx`*, where *`n`* is a base between 2 and 9, and *`xxx`* is a number in that base.

Strings

Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they should be represented by a pair of the appropriate character; e.g. `$$` for `$`.

Boolean

The Boolean constants 'true' and 'false' should be written as `{TRUE}` and `{FALSE}`.

The END directive

Every assembly language source must end with:

```
END
```

on a line by itself.

5 CPU instruction set

This chapter describes the CPU instructions available in ObjAsm. It includes instruction formats, assembler syntax, and a synopsis of each instruction.

The condition field

All ARM instructions are conditionally executed, which means that they will only be executed if the N, Z, C and V flags in the PSR are in the correct state at the end of the preceding instruction. The condition is encoded in a four bit condition field, held in bits 28 - 31 of an instruction. By default ObjAsm encodes the 'always execute' condition; other conditions can be requested by appending a two-character condition mnemonic to ObjAsm's mnemonic for an instruction.

The figure below shows the condition codes, their mnemonics, and the corresponding conditions under which the instruction is executed:

Condition field

0000	⇒	EQ	Z set (equal)
0001	⇒	NE	Z clear (not equal)
0010	⇒	CS	C set (unsigned higher or same)
0011	⇒	CC	C clear (unsigned lower)
0100	⇒	MI	N set (minus – i.e. negative)
0101	⇒	PL	N clear (plus – i.e. positive or zero)
0110	⇒	VS	V set (overflow)
0111	⇒	VC	V clear (no overflow)
1000	⇒	HI	C set and Z clear (unsigned higher)
1001	⇒	LS	C clear or Z set (unsigned lower or same)
1010	⇒	GE	N set and V set, or N clear and V clear (greater or equal)
1011	⇒	LT	N set and V clear, or N clear and V set (less than)
1100	⇒	GT	Z clear, and either N set and V set, or N clear and V clear (greater than)
1101	⇒	LE	Z set, or N set and V clear, or N clear and V set (less than or equal)
1110	⇒	AL	always execute (ignore flags)
1111			reserved

Figure 5.1 The condition field

Note that ObjAsm implements HS (Higher or Same) and LO (LOwer than) as synonymous with CS and CC respectively, giving it a total of 17 condition mnemonics.

For example, suppose you had a CMP (compare) instruction followed by an instruction with the EQ condition (so it is executed only if the Z flag is set):

- If the CMP instruction's two operands were equal, it would set the Z flag, and so your conditional instruction would be executed.
- If the CMP instruction's two operands were different, it would clear the Z flag, and so your conditional instruction would not be executed.

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, which takes little 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 timings

Instruction timings can vary between versions of the ARM processor, and so we do not detail them here. For code that is timing dependent, we advise that you consult the datasheets for all ARM versions on which your code may run.

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 *Data processing* on page 66) may pass through the barrel shifter, either as a direct consequence of the programmer's actions, or as a result of the internal computations of ObjAsm. The barrel shifter also affects the index for the single data transfer instructions (detailed in the section *Single data transfer* (LDR, STR) on page 83).

The barrel shifter has a carry in, which takes its input from the C flag of the PSR; and a carry out, which may be latched back into the C bit of the PSR for logical data operations (see *The S bit* on page 69).

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 known as rotate right extended.

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 2*n* bits

A constant constructed by rotating an 8-bit constant right by 2*n* 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.

ObjAsm 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, ObjAsm 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 2*n* 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

Various instructions use the barrel shifter to shift register operands. The effects of such shifts are detailed in this section, rather than being repeated for each instruction.

Mnemonics

There are six assembler mnemonics for shift types, used to control the barrel shifter. These are:

`LSL`	Logical Shift Left
`ASL`	Arithmetic Shift Left
`LSR`	Logical Shift Right
`ASR`	Arithmetic Shift Right
`ROR`	Rotate Right
`RRX`	Rotate Right with Extend

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

Specification of the shift amount

The shift amount may either be specified in the instruction, or in a register specified by the instruction.

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.

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 R*m* 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 process. This is detailed for each mnemonic described below.

Logical shift left, or arithmetic shift left

`Rm,LSL #n` or `Rm,ASL #n`	Shift contents of R*m* left by *n* bits, where *n* is 0 to 31.
`Rm,LSL Rs` or `Rm,ASL Rs`	Shift contents of R*m* left by the least significant byte of R*s*.

A logical shift left (LSL) takes the contents of R*m* and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeroes. The high bits of R*m* which do not map into the result are discarded – except that the least significant discarded bit becomes the barrel shifter's carry out.

For example, the effect of LSL #5 is:

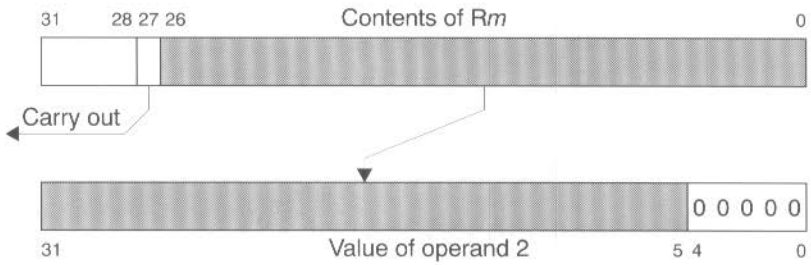


Figure 5.2 A logical or arithmetic shift left by 5

Special cases

- LSL #0 or ASL #0, and LSL R*s* or ASL R*s* where R*s* is 0:
  The barrel shifter's result is the unchanged contents of R*m*, and its carry out is the old value of the PSR C flag.
- LSL R*s* or ASL R*s* where R*s* is 32:
  The result is zero, and the carry out is bit 0 of R*m*.
- LSL R*s* or ASL R*s* where R*s* is greater than 32:
  Both the result and the carry out are zero.

Logical shift right

`Rm,LSR #n` Shift contents of R*m* right by *n* bits, where *n* is 1 to 32.

`Rm,LSR Rs` Shift contents of R*m* right by the least significant byte of R*s*.

A logical shift right (LSR) is similar to a logical shift left, but the contents of R*m* are moved to less significant positions in the result. LSR #5 has this effect:

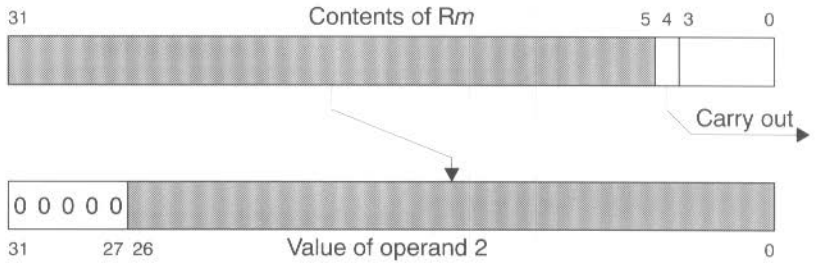


Figure 5.3 A logical shift right by 5

Logical shift right zero is redundant as it is the same as logical shift left zero. The form of the shift field which might be expected to correspond to LSR #0 is therefore used to encode LSR #32. ObjAsm assembles LSR #0 (and ASR #0 and ROR #0) as LSL #0, and allows you to specify LSR #32.

Special cases

- LSR #0:

  This is assembled as LSL #0 (see page 58), which has the same effect as LSR #0.

- LSR R*s* where R*s* is 0:

  The barrel shifter's result is the unchanged contents of R*m*, and its carry out is the old value of the PSR C flag.

- LSR #32, or LSR R*s* where R*s* is 32:

  The result is zero, and the carry out is bit 31 of R*m*. (LSR #32 is encoded in the format you would expect to correspond to LSR #0.)

- LSR R*s* where R*s* is greater than 32:

  Both the result and the carry out are zero.

Arithmetic shift right

`Rm,ASR #n` Shift contents of R*m* right by *n* bits, where *n* is 1 to 32.

`Rm,ASR Rs` Shift contents of R*m* right by the least significant byte of R*s*.

An arithmetic shift right (ASR) is similar to a logical shift right, except that the high bits are filled with bit 31 of R*m* instead of zeroes. This preserves the sign in 2's complement notation. For example, ASR #5:

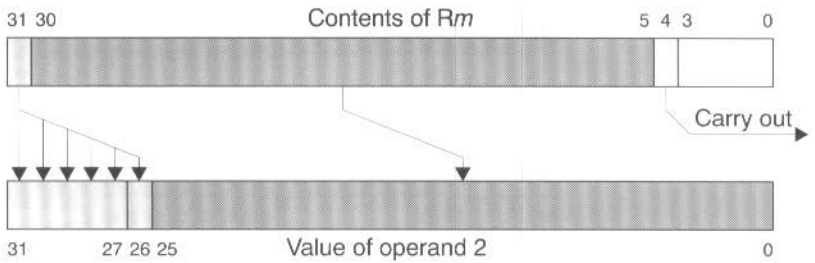


Figure 5.4 An arithmetic shift right by 5

Arithmetic shift right zero is redundant as it is the same as logical shift left zero. The form of the shift field which might be expected to correspond to ASR #0 is therefore used to encode ASR #32. ObjAsm assembles ASR #0 (and LSR #0 and ROR #0) as LSL #0, and allows you to specify ASR #32.

Special cases

- ASR #0:

  This is assembled as LSL #0 (see page 58), which has the same effect as ASR #0.

- ASR R*s* where R*s* is 0:

  The barrel shifter's result is the unchanged contents of R*m*, and its carry out is the old value of the PSR C flag.

- ASR #32, or ASR R*s* where R*s* is 32 or more:

  Each bit of the result is equal to bit 31 of R*m*; the result is therefore all ones or all zeroes. The carry out is also bit 31 of R*m*. (ASR #32 is encoded in the format you would expect to correspond to ASR #0.)

Rotate right

`Rm,ROR #n` Rotate contents of R*m* right by *n* bits, where *n* is 1 to 31.

`Rm,ROR Rs` Rotate contents of R*m* right by the least significant byte of R*s*.

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:

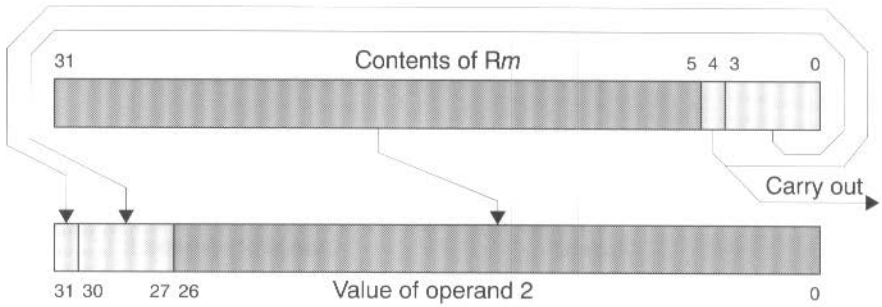


Figure 5.5 A rotate right by 5

Rotate right zero is redundant as it is the same as logical shift left zero. The form of the shift field which might be expected to correspond to ROR #0 is therefore used to encode rotate right extended (see the next section). ObjAsm assembles ROR #0 (and LSR #0 and ASR #0) as LSL #0.

Special cases

- ROR #0:

  This is assembled as LSL #0 (see page 58), which has the same effect as ROR #0.

- ROR Rs where Rs is 0:

  The barrel shifter's result is the unchanged contents of R*m*, and its carry out is the old value of the PSR C flag.

- ROR Rs where Rs is 32:

  The result is equal to R*m*, and the carry out is bit 31 of R*m*.

- ROR Rs where Rs is greater than 32:

  The result and carry out are the same as for ROR ((Rs – 1) MOD 32 + 1); therefore repeatedly subtract 32 from Rs until its value is in the range 1 to 32, and then see above.

Rotate right with extend

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

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 R*m*:

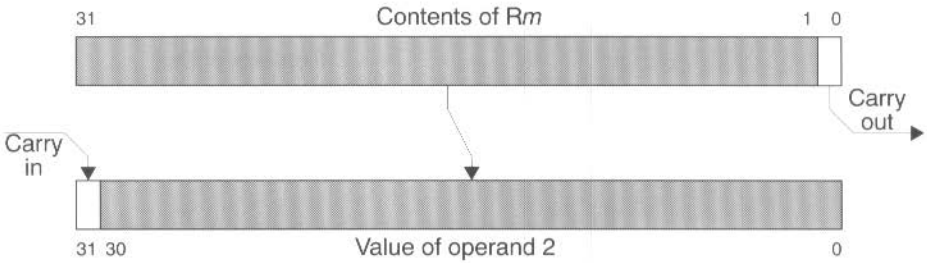


Figure 5.6 A rotate right with extend

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.)

The floating point coprocessor uses coprocessor numbers 1 and 2. If it's absent, the floating point emulator traps the resulting undefined instructions and emulates them. The coprocessor 15 instructions are used by ARM as instructions to control its own operation (such as cache control, and 26/32 bit configuration).

ObjAsm provides support for coprocessors at two levels. Firstly, it provides a set of generic coprocessor instructions, detailed below. Secondly, it recognises a standard set of floating point instructions and translates them into the appropriate coprocessor instructions; see the chapter *Floating point instructions* on page 117 for details.

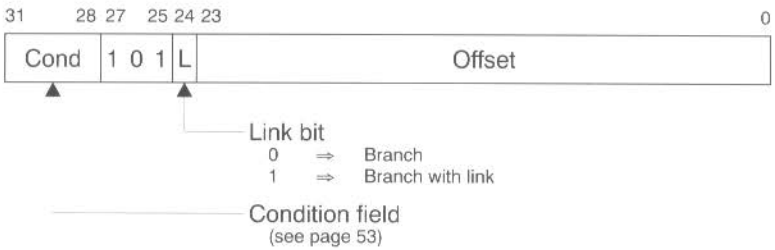
All the generic coprocessor operations include a coprocessor number symbol and one or more coprocessor register symbols. These should be defined using the CP and CN directives respectively. (See the chapter *Directives* on page 139.)

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

Branch, Branch with Link (B, BL)

Instructions for branching to an instruction other than the next one

Instruction format



Assembler syntax

`B«L»«cond» expression`

where:

`«L»`	requests the Branch with Link form of the instruction (see *The link bit* below). If absent, R14 will not be affected by the instruction.
`«cond»`	is a two-character condition mnemonic; see the section *The condition field* on page 53.
`expression`	is a program-relative expression describing the branch destination, from which ObjAsm calculates the offset.

Synopsis

These instructions branch to an instruction other than the next one, by altering the value of the program counter (R15). The Branch with Link form of the instruction also stores a return address in the link register (R14), so that program flow can branch to a subroutine, and then return to the instruction immediately following the Branch with Link instruction; for more details see *The link bit* below.

All branches take a signed 2's complement 24 bit word offset. This is shifted left two bits, and added to the program counter, with any overflow being ignored, giving an offset of ±32Mbytes. The branch can therefore reach any word aligned address within a 26 bit address space, since the calculation 'wraps round' between the top and bottom of memory.

When using this instruction with ObjAsm you should provide a label, from which ObjAsm will calculate the 24 bit offset.

The encoded offset must take account of the effects of pipelining and prefetching within the CPU, which causes the PC to be two words ahead of the current instruction. ObjAsm automatically handles this for you. 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	

The instruction is only executed if the condition specified in the condition field is true (see the section *The condition field* on page 53).

The link bit

Branch with Link works in the same way as Branch, but it also writes the old PC and PSR into the link register (R14) of the current bank. The PC value written is first adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction.

This form of the instruction is often used for branching to subroutines. At the end of the subroutine the program flow can return to the instruction immediately following the Branch with Link instruction by writing the link register (R14) value back into the program counter (R15). To do so, the subroutine should end with:

```
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 R*n*. (*xx* is the stack type; see the section *Block data transfer* (LDM, STM) on page 88.)

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.

32 bit operation

Calculating the offset

In 32 bit operation, the offset is sign extended to 32 bits before it is added to the program counter.

Branches beyond ±32Mbytes must use an offset or an absolute destination which has previously been loaded into a register. In this case you should manually save the PC in R14 if you require a Branch with Link type operation.

The link bit

Branch with Link does not save the CPSR with the PC. If you need to preserve the CPSR over a subroutine, it is your responsibility to explicitly save and restore it, either on entry to and exit from (respectively) the subroutine, or in the calling part of the program.

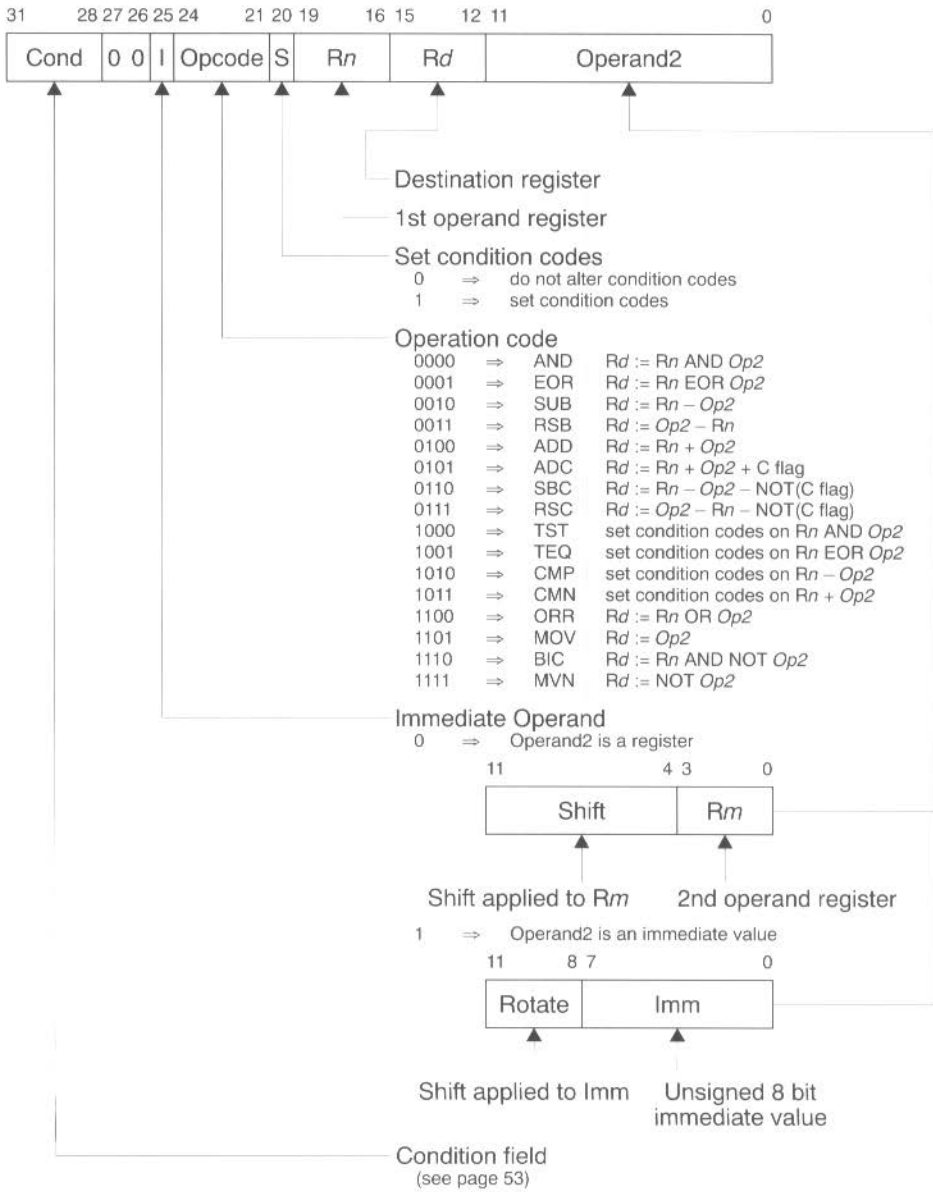
Examples

```
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
```

Data processing

Instructions for performing arithmetic or logical operation on one or two operands

Instruction format



Assembler syntax

The data processing instructions use three different types of syntax, depending on whether the opcode being used takes one or two operands, and whether or not it writes the result into a destination register:

MOV and MVN – single operand

opcode«*cond*»«S» Rd,*op2*

CMN, CMP, TEQ and TST – no result written

opcode«*cond*»«P» Rn,*op2*

ADC, ADD, AND, BIC, OR, ORR, RSB, RSC, SBC, SUB – two operands

opcode«*cond*»«S» Rd,Rn,*op2*

Parameters

opcode	is a mnemonic for the data processing operation to be performed; see *Opcodes* below
«*cond*»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
«S»	means to set the PSR's condition codes from the operation. ObjAsm forces this for CMN, CMP, TEQ and TST, provided the P flag is not specified. See *Opcodes* below for a summary of the flags affected by each opcode, and *The S bit* on page 69 for more detail.
«P»	means to take the result of a CMN, CMP, TEQ or TST operation, and move it to the bits of R15 that hold the PSR – even though the instruction has no destination register. Bits corresponding to the PC are masked out, as are (in User mode) the I, F, and mode bits.
Rd, Rn & Rm	are expressions evaluating to a valid ARM register number.
op2	may be any of the operands that the barrel shifter can produce. The syntax is Rm«,*shift*» or *#expression* If *#expression* is used, ObjAsm 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* Rs or *shiftname #expression*, or RRX (rotate right one bit with extend). *shiftname*s are: ASL, LSL, LSR, ASR, and ROR. (ASL is a synonym for LSL, and the two assemble to the same code.) See *Shift types* on page 57.

Opcodes

The *opcodes* supported are:

Assembler Mnemonic	Meaning	Operation	Flags affected
ADC	Add with Carry	Rd:=Rn + op2 + C flag	N,Z,C,V
ADD	Add	Rd:=Rn + op2	N,Z,C,V
AND	And	Rd:=Rn AND op2	N,Z,C
BIC	Bit Clear	Rd:=Rn AND (NOT(op2))	N,Z,C
CMN	Compare Negated	Rn + op2	N,Z,C,V
CMP	Compare	Rn – op2	N,Z,C,V
EOR	Exclusive Or	Rd:=Rn EOR op2	N,Z,C
MOV	Move	Rd:=op2	N,Z,C
MVN	Move Not	Rd:=NOT op2	N,Z,C
ORR	Logical Or	Rd:=Rn OR op2	N,Z,C
RSB	Reverse Subtract	Rd:=op2 – Rn	N,Z,C,V
RSC	Reverse Subtract with Carry	Rd:=op2 – Rn – NOT(C flag)	N,Z,C,V
SBC	Subtract with Carry	Rd:=Rn – op2 – NOT(C flag)	N,Z,C,V
SUB	Subtract	Rd:=Rn – op2	N,Z,C,V
TEQ	Test Equivalence	Rn EOR op2	N,Z,C
TST	TeST and mask	Rn AND op2	N,Z,C

Synopsis

These instructions produce a result by performing a specified arithmetic or logical operation on one or two operands.

The operation is performed between a source register Rn and an operand op2 – except for MOV and MVN, where only the operand is needed (and for which the assembler sets Rn to R0). The source register can be any one of the 16 registers. The operand can be any operand that the barrel shifter can produce: i.e. a shifted register Rm, or a rotated 8 bit immediate value Imm, according to the value of the I bit in the instruction. (See *The barrel shifter* on page 55 and *Shift types* on page 57.) Note that any shifting is done before the operation is performed.

The logical operations (AND, BIC, EOR, MOV, MVN, ORR, TEQ, TST) perform the logical action on all corresponding bits of the operand or operands to produce the result. The arithmetic operations (ADC, ADD, CMP, CMN, RSB, RSC, SBC, SUB) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). Some add the bit held in the ALU's carry flag into the operation.

The result of the operation is placed in the destination register Rd – except for CMN, CMP, TEQ and TST, which are used only to perform tests and to set the condition codes on the result (and for which the assembler sets Rd to R0). The destination register may be any one of the 16 registers.

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; see *The S bit* below.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

The S bit

The instruction contains a one bit field called the S bit, standing for 'set condition codes'. The result of the operation in the ALU affects its N and Z flags, and may also affect its C and V flags. However, the ALU doesn't copy its flags to the relevant parts of the PSR unless the S bit is set. ObjAsm always sets the S bit for the four instructions CMN, CMP, TEQ and TST, since they would be meaningless unless their results were copied to the PSR. In the case of the remaining 12 instructions, you may request that the S bit be set by appending the letter S to the instruction mnemonic.

The way the PSR flags are altered differs for logical and arithmetic operations:

Logical operations (AND, BIC, EOR, MOV, MVN, ORR, TEQ, TST)

- The V flag in the PSR will be unaffected.
- The C flag will be set to the last bit shifted out by the barrel shifter, or is unchanged if no shifting took place.
- The Z flag will be set if and only if the result is all zeroes.
- The N flag will be set to the logical value of bit 31 of the result.

Arithmetic operations (ADC, ADD, CMP, CMN, RSB, RSC, SBC, SUB)

- The V flag in the PSR will be set if signed overflow occurs (i.e. if you regard the operands as signed 32 bit integers, the signed result does not fit in a 32 bit integer); 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 destination register is set to the bottom 32 bits of the correct unsigned result).
- The C flag will be set to the carry out of bit 31 of the ALU, which for addition indicates that 32 bit overflow occurred, and for subtraction indicates that 32 bit underflow did not occur.
- The Z flag will be set if and only if the result was zero.
- 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.

The P flag

The P flag invokes a special form of the CMN, CMP, TEQ and TST operations, used to update the PSR. The operation is carried out, and then 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.

This is typically used to change modes. For example:

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

Note the treatment of R15 as the first operand, described in *Using R15 as an operand* on page 71.

This form is encoded by setting the S bit, and setting the destination register to R15.

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:

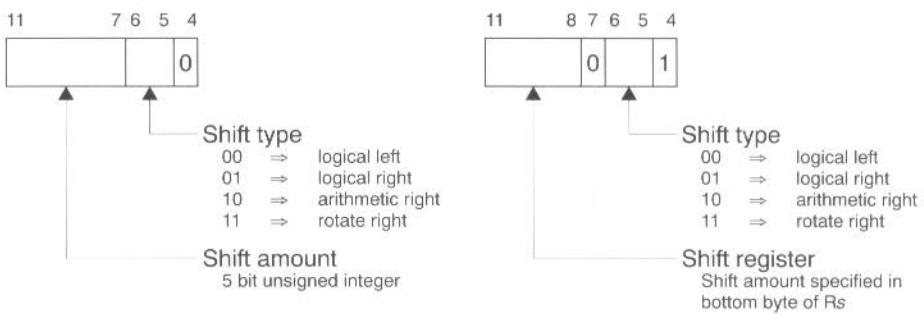


Figure 5.7 Shifts

Shifts are detailed in the section *Shift types* on page 57.

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.

Immediate operand rotates are detailed in the section *The barrel shifter* on page 55.

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:

Using R15 as the destination

If R15 is the destination register, and the S bit is not set, the PC is overwritten, but not the PSR.

If the S bit is set, then the PC is overwritten, and also all bits of the PSR that are unprotected in the current mode; thus in User mode the N, Z, C and V flags are overwritten, whereas in other modes the entire PSR is overwritten.

Using R15 as an operand

R15 will always contain the value of the PC, which 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 R*s*, and 12 bytes ahead when used as R*n* or R*m*.

R15 may or may not contain the values of the PSR flags as they were at the completion of the previous instruction, depending on which operand position it occupies:

- If R15 is the first operand in a two operand instruction, it 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), it is presented to the barrel shifter or ALU with the PSR bits unchanged.
- If R15 is the shift register, it is presented to the barrel shifter with the PSR bits set to zero.

32 bit operation

TEQP, TSTP, CMPP and CMNP

These opcodes should not be used in 32 bit modes. You should instead use the new PSR transfer functions. When used in a privileged mode, TEQP moves the SPSR for the current mode to the CPSR.

Using R15 as the shift register

You must not use R15 as the shift register.

Using R15 as the destination

If R15 is the destination register, and the S bit is not set, the PC is overwritten, but not the CPSR. This is what you would expect as an extension of the 26 bit behaviour.

If the destination register is R15 and the S bit is set, then as well as writing the result to the PC, the SPSR for the current mode is moved to the CPSR. This is again what you would expect as an extension of the 26 bit behaviour.

Examples

```
; Simple use of a one operand instruction:
        MVN     R2,R3           ; R2 is set to the bitwise inverse of the
                                ; contents of R3.

; Simple uses of instructions that does not write a result:
        CMP     R0,R1           ; Compare the contents of R0 with R1

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

        TEQS    R4,#3           ; Test R4 for equality with 3
                                ; (The S is in fact redundant as the assembler
                                ; inserts it automatically)

; Simple use of a two operand instruction:
        ADD     R0,R1,R2        ; R0=R1+R2

; Conditional execution of an instruction:
        ADDEQ   R2,R4,R5        ; If the Z flag is set make R2:=R4+R5

; Use of the S bit to alter the PSR:
        ADDS    R0,R1,#1        ; R0=R1+1, and set N,Z,C,V

; Use of a register specified shift:
        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

; Use of an immediate shift:
        MOV     R0,R1,LSL#2     ; The contents of R1 are shifted left by
                                ; 2 bits and transferred to R0.
```

```
; Using ADC 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

; Using SBC to implement multi-word subtractions. For example:
        SUBS    R4,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)

; Changing to user mode and returning from a subroutine:
        ; 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
        MOV     R0,R0           ; No-op to avoid mode change hazard
        MOV     PC,R14          ; Return from subroutine
                                ; (R14 is a banked register)

; Returning from a subroutine and restoring the PSR:
        MOVS    PC,R14          ; return from subroutine and restore the PSR
```

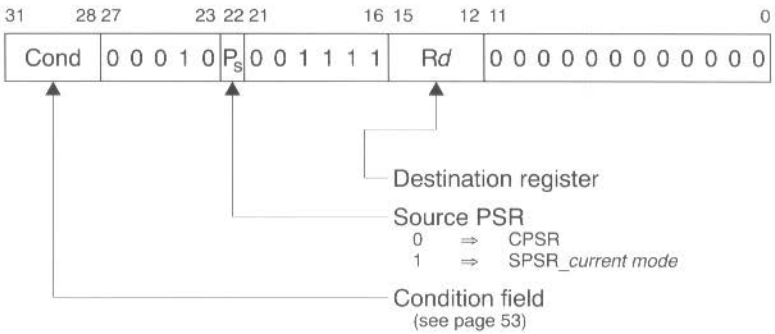
PSR transfer (MRS, MSR)

Instructions for accessing the CPSR and SPSR registers

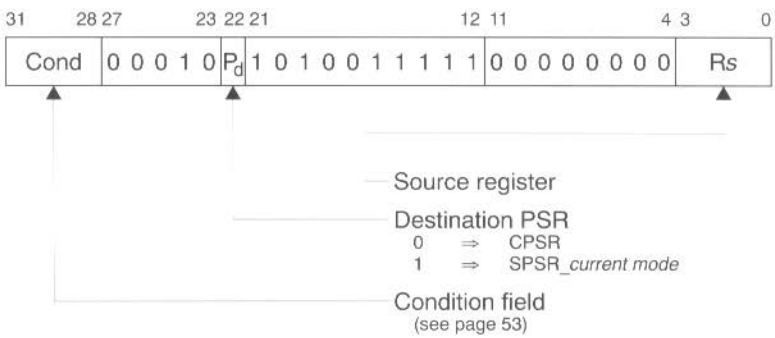
These instructions are not available on ARM2 and ARM3 series processors

Instruction format

MRS (transfer PSR contents to a register)



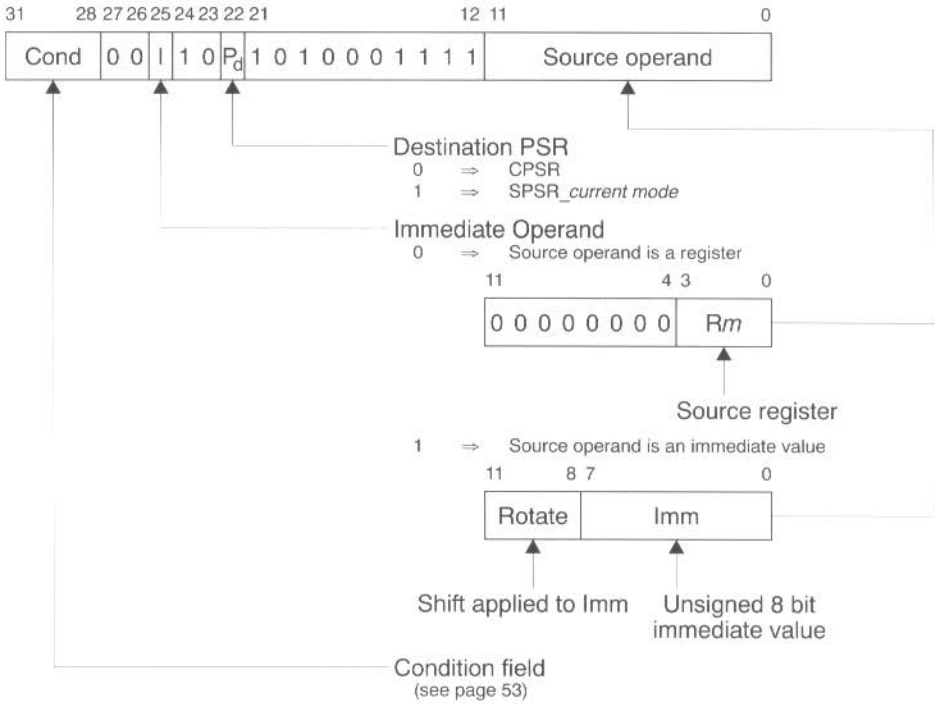
MSR (transfer register contents to PSR)



Assembler syntax

```
MRS«cond» Rd,psr
MSR«cond» psr,Rm
MSR«cond» psrf,Rm
MSR«cond» psrf,#expression
```

MSR (transfer register contents or immediate value to PSR flag bits only)



where:

«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
Rd & Rm	are expressions evaluating to a valid ARM register number other than R15.
psr	is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms, as are SPSR and SPSR_all.)
psrf	is CPSR_flg or SPSR_flg. The most significant four bits of Rm or *#expression* are written to the N, Z, C and V flags respectively.
#expression	is an expression symbolising a 32 bit value. If *#expression* is used, ObjAsm will attempt to match the expression by generating a shifted immediate 8-bit field. If this is impossible, it will give an error.

Synopsis

These instructions allow access to the CPSR and SPSR registers:

- The MRS instruction moves the contents of the CPSR or SPSR_*current mode* register to a general register.
- The MSR instruction moves the contents of a general register to the CPSR or SPSR_*current mode* register.

Alternatively, the MSR instruction can write to the condition code flags of the CPSR or SPSR_*current mode* register without affecting its control bits:

- In this case the source may be either the contents of a general register or an immediate value, and only its top four bits are used.

The instructions are encoded using the CMN, CMP, TEQ and TST instructions without the S flag set.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

These instructions are not available on ARM2 and ARM3 series processors.

On ARM6 series processors and later, they are available in all modes and configurations. However, we recommend that you avoid using these instructions, as you will lose backwards compatibility with older ARMs. Indeed, in the 26 bit modes used by RISC OS (except when handling FIQs), you can access the PSR just as you always have – for example, with TEQP.

Operand restrictions

In user mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

R15 must not be specified as the source or destination register.

You must not attempt to access the SPSR in user mode, as no such register exists.

Reserved bits

Not all bits of the PSR are defined (e.g. only N, Z, C, V, I, F and M[4:0] are defined for the ARM 6 and 7 series). The remaining ones (bits 27-8 and 5 in the ARM 6 and 7 series) are reserved for use in future versions of the ARM. The ensure future compatibility, the following rules should be observed:

- You must preserve the reserved bits when changing the value in a PSR.
- When you are checking the PSR status, you must not rely on specific values from the reserved bits, since they may read as one or zero in future processors.

You should therefore use a read-modify-write strategy when altering the control bits of any PSR register. This involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits, and then transferring the modified value back to the PSR register using the MSR instruction.

For example, to perform a mode change:

```
MRS     R0,CPSR                 ; Take a copy of the PSR
BIC     R0,R),#0x1F             ; Clear the mode bits
ORR     R0,R0,#new_mode         ; Set bits for new mode
MSR     CPSR,R0                 ; Write back the modified CPSR,
                                ; changing mode
```

When you wish simply to change the condition flags in a PSR, you can write an immediate value directly to the flag bits without disturbing the control bits. For example, the following instruction sets the N, Z, C and V flags:

```
MSR     CPSR_flg,#0xF0000000    ; Set all the flags regardless
                                ; of their previous state
                                ; (does not affect any control bits)
```

You must not attempt to write an 8 bit immediate value into the whole PSR, since such an operation cannot preserve the reserved bits.

Examples

In user mode the instructions behave as follows:

```
MSR     CPSR_all,Rm             ; CPSR[31:28] <- Rm[31:28]
MSR     CPSR_flg,Rm             ; CPSR[31:28] <- Rm[31:28]

MSR     CPSR_flg,#0xA0000000    ; CPSR[31:28] <- 0xA
                                ;       (i.e. set N,C; clear Z,V)

MRS     Rd,CPSR                 ; Rd[31:0] <- CPSR[31:0]
```

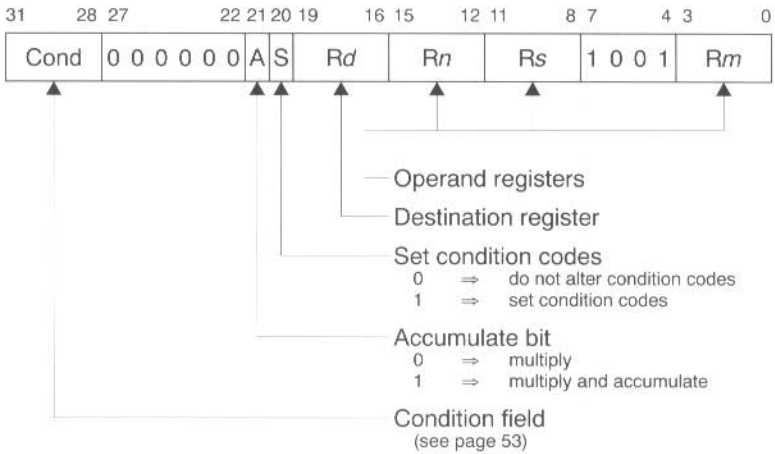
In privileged modes the instructions behave as follows:

```
MSR     CPSR_all,Rm             ; CPSR[31:0]  <- Rm[31:0]
MSR     CPSR_flg,Rm             ; CPSR[31:28] <- Rm[31:28]

MSR     CPSR_flg,#0x50000000    ; CPSR[31:28] <- 0x5
                                ;       (i.e. set Z,V; clear N,C)

MRS     Rd,SPSR                 ; Rd[31:0] <- SPSR[31:0]

MSR     SPSR_all,Rm             ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR     SPSR_flg,Rm             ; SPSR_<mode>[31:28] <- Rm[31:28]

MSR     SPSR_flg,#0xC0000000    ; SPSR_<mode>[31:28] <- 0xC
                                ;       (i.e. set N,Z; clear C,V)

MRS     Rd,SPSR                 ; Rd[31:0] <- SPSR_<mode>[31:0]
```

Multiply and Multiply-Accumulate (MUL, MLA)

Instructions for performing integer multiplication, giving a 32 bit result

Instruction format



Assembler syntax

```
MUL«cond»«S» Rd,Rm,Rs
MLA«cond»«S» Rd,Rm,Rs,Rn
```

«cond» is a two-character condition mnemonic; see the section *The condition field* on page 53.

«S» means to set the PSR's condition codes from the operation.

Rd, Rm, Rs & Rn are expressions evaluating to a valid ARM register number. (*Rd* must not be R15 and must not be the same as *Rm*.)

Synopsis

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.

The results of a signed multiply and of an unsigned multiply of 32 bit operands differ only in the upper 32 bits; the low 32 bits are identical. As these instructions only produce those low 32 bits, they can be used with operands which may be considered as either signed (2's complement) or unsigned integers.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

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, and the C flag is set to a meaningless value.

Operand restrictions

Because of the way the Booth's algorithm has been implemented, you should avoid certain combinations of operand registers. (ObjAsm will issue a warning if you overlook these restrictions.)

The destination register Rd must 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.

The destination register Rd must also not be R15.

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

32 bit operation

R15 must not be used as any of Rd, Rm, Rn or Rs.

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.)

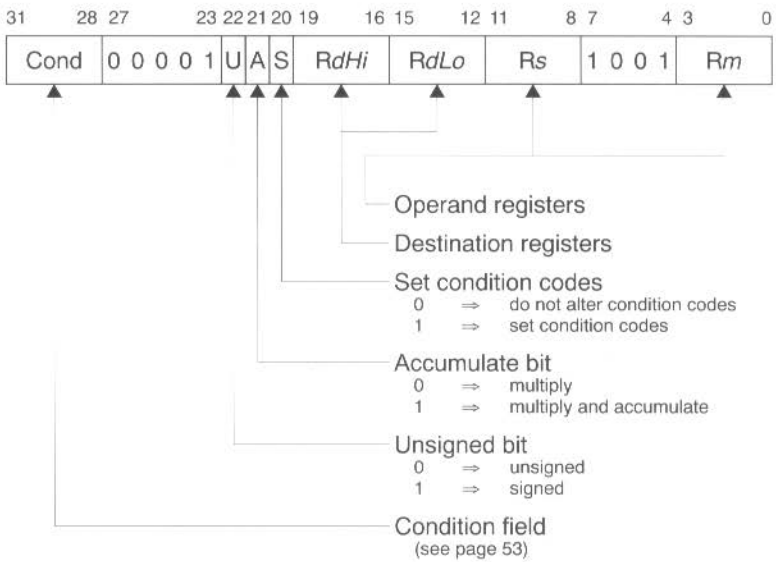
Note that more recent ARM processors have a single instruction to do just this; see the next section.

Multiply Long and Multiply-Accumulate Long (UMULL, SMULL, UMLAL, SMLAL)

Instructions for performing integer multiplication, giving a 64 bit result

This instruction is only available in 32 bit mode on the ARM7M series or later

Instruction format



Assembler syntax

```
UMULL«cond»«S» RdLo,RdHi,Rm,Rs
SMULL«cond»«S» RdLo,RdHi,Rm,Rs
UMLAL«cond»«S» RdLo,RdHi,Rm,Rs
SMLAL«cond»«S» RdLo,RdHi,Rm,Rs
```

«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
«S»	means to set the PSR's condition codes from the operation.
RdLo, RdHi, Rm & Rs	are expressions evaluating to a valid ARM register number other than R15.

Synopsis

The multiply long instructions perform integer multiplication on two 32 bit operands, and produce a 64 bit result. The multiplication can be signed or unsigned, which – with optional accumulate – gives rise to four variations.

The multiply forms of the instruction (UMULL and SMULL) give a 64 bit result of the form *RdHi,RdLo*:=*Rm*×*Rs*.

The multiply-accumulate forms (UMLAL and SMLAL) give *Rd*:=*Rm*×*Rs*+*Rn*, which can save an explicit ADD instruction in some circumstances.

The lower 32 bits of the result and of the accumulator (where used) are held in *RdLo*, and the upper 32 bits in *RdHi*.

The unsigned forms of the instruction (UMULL and UMLAL) treat all four registers as unsigned numbers. The signed forms (SMULL and SMLAL) treat the two operand registers as 2's complement signed 32 bit numbers, and the two destination registers as a 2's complement signed 64 bit number.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

This instruction was first introduced on the ARM7M series of processor, and is only available in 32 bit modes. This instruction is therefore unlikely to be of use under RISC OS.

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), and the V and C flags are set to a meaningless value.

Operand restrictions

R15 must not be used as any of *RdHi*, *RdLo*, *Rm* or *Rs*.

RdHi, *RdLo* and *Rm* must all specify different registers.

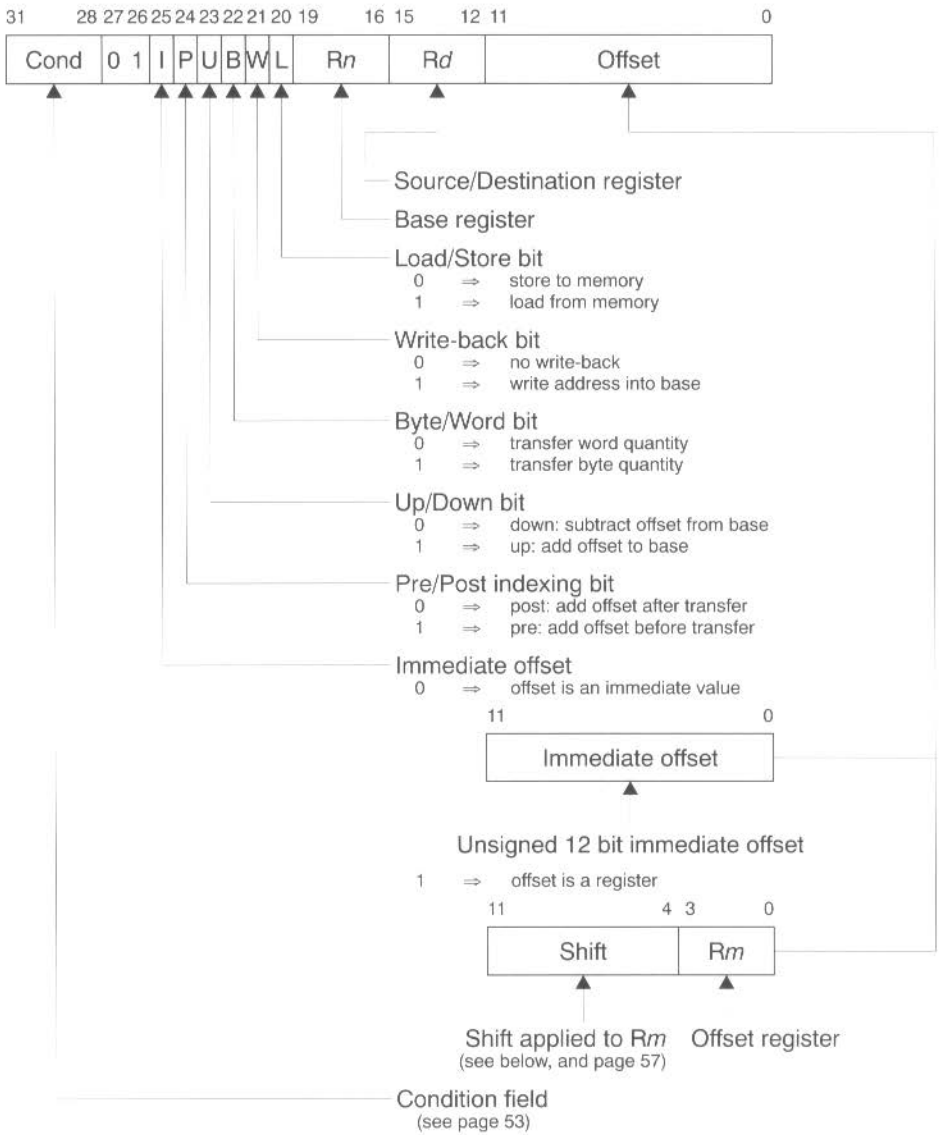
Examples

```
UMULL   R1,R4,R2,R3     ; R1,R4:=R2*R3

UMLALS  R1,R5,R2,R3     ; R1,R5:=R2*R3+R1,R5,
                        ; also setting condition codes
```

Single data transfer (LDR, STR)

Instructions for loading or storing single bytes or words of data

Instruction format



Assembler syntax

`LDR|STR«cond»«B»«T» Rd,address`

`LDR`	loads from memory into a register.
`STR`	stores from a register into memory.
`«cond»`	is a two-character condition mnemonic; see the section *The condition field* on page 53.
`«B»`	means to transfer a byte, otherwise a word is transferred.
`«T»`	means to set the W bit in a post-indexed instruction, forcing non-privileged mode 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 ARM register number.
`address`	can be:

- An expression which generates an address:

  `expression`

  ObjAsm 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 by `shift`.

`Rn` and `Rm` are expressions evaluating to a valid ARM register number. Note if `Rn` is R15 then ObjAsm will subtract 8 from the offset value to allow for ARM pipelining.

`shift` is a general shift operation (see the section *Shift types* on page 57), but note that the shift amount may not be specified by a register.

`«!»` if present sets the W bit to write-back the base register.

Synopsis

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. If the contents of the 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.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

For register to register transfers, see the section *Data processing* on page 66, particularly the MOV instruction.

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 R*n*. 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 privileged mode code; depending on the processor, setting the W bit either forces the $\overline{\text{TRANS}}$ pin to go LOW or forces non-privileged mode 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 hardware.

Shifted register offset

The 8 shift control bits are described in the section *Data processing* on page 66, 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) or word store (STR) should generate a word aligned address. Using a non-word-aligned addresses has non-obvious and unspecified results.

Use of R15

These instructions will never cause the PSR to be modified, even when R*d* or R*n* is R15.

If R15 is specified as the base register (R*n*), the PC is used without the PSR flags. When using the PC as the base register you 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 (R*m*), the value presented will be the PC together with the PSR.

When R15 is the source register (R*d*) 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 R*d* will change only the PC, and the PSR will be unchanged.

Address exceptions

On an ARM2 or ARM3 processor, 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.

Later versions of the ARM do not generate address exceptions when in a 32 bit configuration (as used by RISC OS from very soon after reset), even when running in 26 bit modes.

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.

32 bit operation

R15 must not be used as the register offset (R*m*).

If R15 is specified as the base register (R*n*), you must not use write-back – including post indexing.

For a post-indexed LDR or STR, R*m* and R*n* must not be the same register.

When using write-back – including post indexing –R*d* and R*n* must not be the same register.

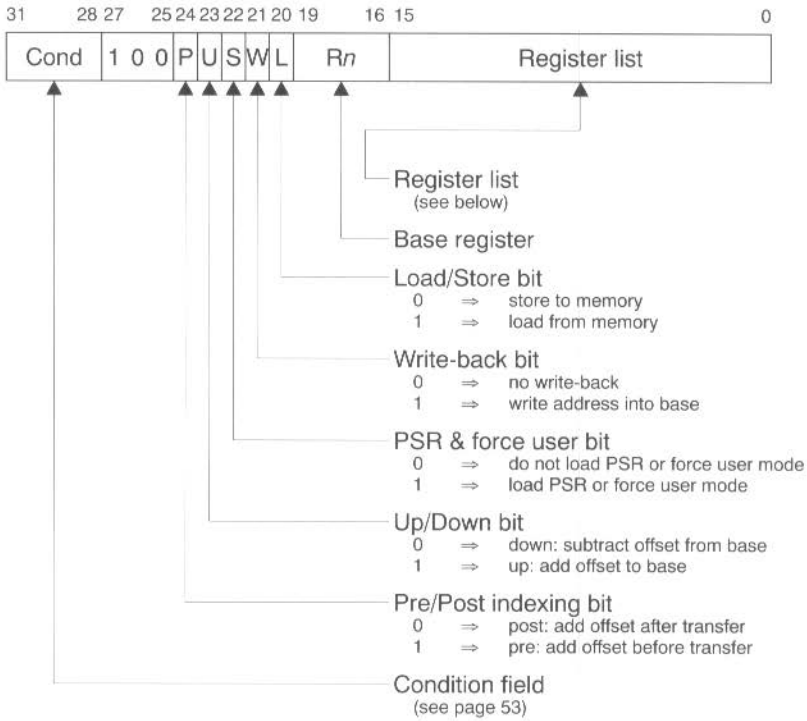
Examples

```
        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
        More instructions
PLACE
```

Block data transfer (LDM, STM)

Instructions for loading or storing any subset of the currently visible registers

Instruction format



Assembler syntax

`LDM|STM«cond»FD|ED|FA|EA|IA|IB|DA|DB Rn«!»,Rlist«^»`

`LDM`	loads from memory into register(s).
`STR`	stores from register(s) into memory.
«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
`Rn`	is an expression evaluating to a valid ARM register number.

Rlist is either a comma-separated list of registers and/or of register ranges indicated by hyphens, all enclosed in `{}` (e.g. `{R0,R2-R7,R10}`); or an expression evaluating to the 16 bit operand.

« ! » if present sets the W bit to write-back the base register.

« ^ » if present sets the S bit to load the PSR with the PC, or forces storing of user bank registers when in a non-user mode.

Addressing mode names

There are different assembler mnemonics for each of the addressing modes. There are alternative forms for each mnemonic: one form is intended for use with stacks, and describes the type of stack the addressing mode supports; the other form merely describes the instructions functionality. 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

In the stacking forms of the mnemonics (FD, ED, FA and EA), the F and E refer to a *full* or *empty* stack, and the A and D refer to an *ascending* or *descending* stack:

- A full stack is one in which the stack pointer points to the last data item written, whereas an empty stack is one in which the stack pointer points to the first free slot.
- A descending stack is one which grows from high memory addresses to low ones, whereas an ascending stack is one which grows from low memory addresses to high ones.

The other forms of the mnemonics (IA, IB, DA and DB) simply mean Increment After, Increment Before, Decrement After, and Decrement Before.

Synopsis

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers from or to memory. 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 instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

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 must not be empty.

Addressing modes

The transfer addresses are determined by the contents of the base register (R*n*), the pre/post bit (P) and the up/down bit (U). The registers are stored such that the lowest register is always at the lowermost address in memory, the highest register is always at the uppermost address, and the others are stored in numerical order between them.

(As an aside, this means that instruction sequences such as:

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

do not swap the contents of R1 and R2.)

By way of illustration, consider the transfer of R1, R5 and R7 in the case where R*n*=1000H and write back of the modified base is required (W=1). The figures below show the sequence of register transfers, the addresses used, and the value of R*n* after the instruction has completed.

(In all cases, had write back of the modified base not been required (W=0), R*n* 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.)

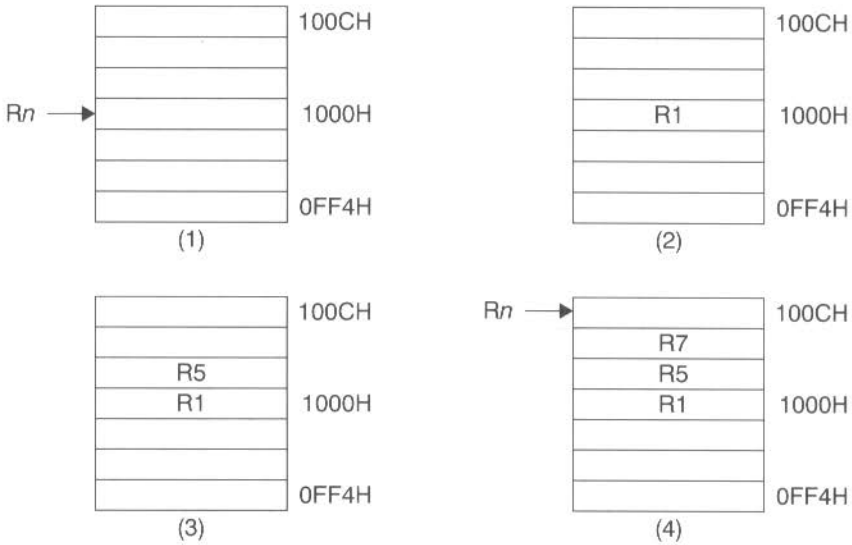


Figure 5.8 Post-increment addressing

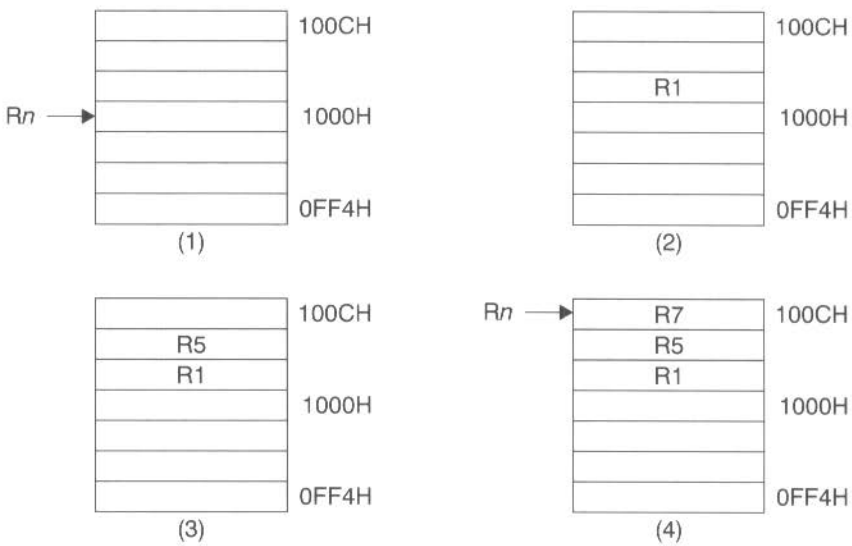


Figure 5.9 Pre-increment addressing

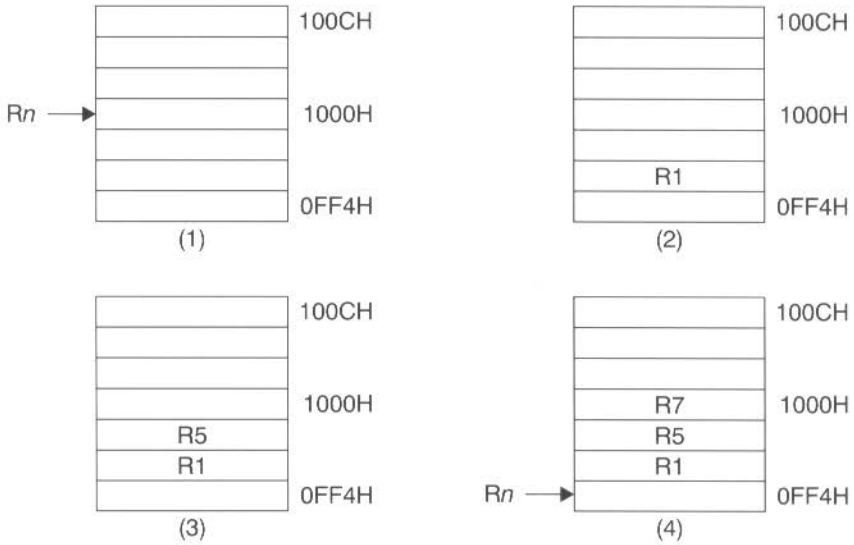


Figure 5.10 Post-decrement addressing

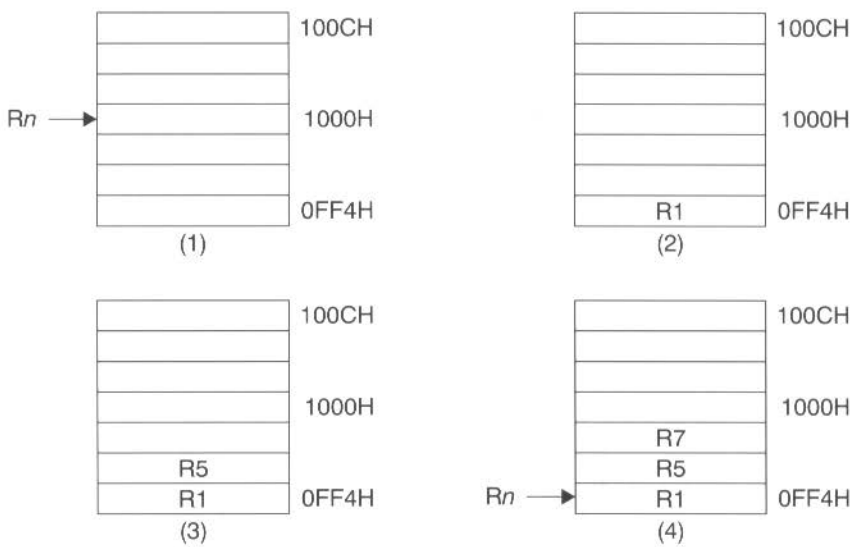


Figure 5.11 Pre-decrement addressing

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. For LDM instructions the S bit is redundant if R15 is not in the transfer list.

In both the above cases, the S bit is instead used to force transfers in non-user modes to use the user register bank instead of the current register bank. This is useful for saving and restoring the user state on process switches. You must not use write back of the base when forcing user bank transfer.

For an LDM instruction, you must take care not to read from a banked register during the following cycle; if in doubt insert a no-op.

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.

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.

Address exceptions

On an ARM2 or ARM3 processor, if 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.

Later versions of the ARM do not generate address exceptions when in a 32 bit configuration (as used by RISC OS from very soon after reset), even when running in 26 bit modes.

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 signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if the 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, but earlier ones may have overwritten registers. The PC is always the last register to be written, and so will always be preserved.
- 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.

32 bit operation

For an STM instruction where R15 is in the transfer list, the PC is stored, but the CPSR is not stored to the current mode's SPSR. (The intuitive extension of the 26 bit behaviour would be for the CPSR to be stored.)

For an LDM instruction where R15 is in the transfer list, if the S bit is set then as well as overwriting the PC, the SPSR for the current mode is moved to the CPSR. This is what you would expect as an extension of the 26 bit behaviour.

The S bit must not be set for instructions that are to be executed in user mode.

You must not use R15 as the base register.

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:

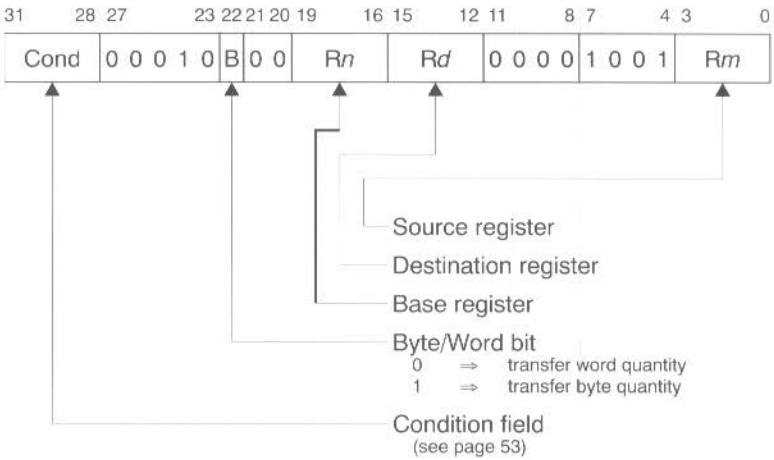
```
STMED   SP!,{R0-R3,R14} ; save R0 to R3 to use as workspace
                        ; and R14 for returning

BL      somewhere       ; this nested call will overwrite R14

LDMED   SP!,{R0-R3,R15}^; restore workspace and return
                        ; (also restoring PSR flags)
```

Single data swap (SWP)

Instruction for swapping atomically between a register and external memory

This instruction is not available on the ARM2 processor

Instruction format



Assembler syntax

`SWP«cond»«B» Rd,Rm,[Rn]`

«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
«B»	means to transfer a byte, otherwise a word is transferred.
Rd, Rm & Rn	are expressions evaluating to a valid ARM register number.

Synopsis

The data swap instruction is used to swap atomically a byte or word quantity between a register and external memory. It is implemented as a memory read followed by a memory write to the same address, which are 'locked' together. The processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable. This instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (R*n*). The processor first reads the contents of the swap address. It then writes the contents of the source register (R*m*) to the swap address, and stores the old memory contents in the destination register (R*d*). The same register may be specified as both the source and destination; its contents are correctly swapped with memory.

The LOCK output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multi-processor systems, where the swap instruction is the only indivisible instruction which may be used to implement semaphores. Control of the memory must not be removed from a processor while it is performing a locked operation.

The SWP instruction is not supported by the ARM2 processor, but is available in the ARM3, in the ARM2aS macrocell (as used for the ARM250 chip in the Acorn A3010, 3020 and A4000), and on the ARM6 series and later.

Bytes and words

This instruction may be used to swap a byte (B=1) or a word (B=0) between a register and memory. The SWP instruction is implemented as a LDR followed by a STR, and the action of these is as described in *Single data transfer* (LDR, STR) on page 83.

Use of R15

You must not use R15 as an operand (R*d*, R*n* or R*m* in a SWP instruction.

Data aborts

If the address used for the swap is unacceptable to a memory management system, the internal MMU or external memory manager can flag the problem by driving ABORT HIGH. This can happen on either the read or the write cycle (or both), and in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem. Once this has been done, the instruction can be restarted and the original program continued.

Examples

```
SWP     R0,R1,[R2]      ; load R0 with the word addressed by R2,
                        ; and then store R1 at the same address

SWPB    R2,R3,[R4]      ; load R2 with the byte addressed by R4,
                        ; and then store bits 0 to 7 of R3 at the
                        ; same address

SWPEQ   R0,R0,[R1]      ; conditionally swap the word addressed
                        ; by R1 with the contents of R0
```

Software interrupt (SWI)

Instruction for entering supervisor mode in a controlled manner

Instruction format

Assembler syntax

`SWI«cond» expression`

«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
expression	is evaluated and placed in the comment field as a SWI number (which is ignored by ARM).

Synopsis

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. The PC is then forced to the SWI vector. If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

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 calling program, and restore the PSR.

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 (as in RISC OS).

32 bit operation

The CPSR is saved in SPSR_svc. The **MOVS R15,R14_svc** instruction used to return to the supervisor restores the CPSR from SPSR_svc. This is what you would expect as an extension of the 26 bit behaviour.

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
```

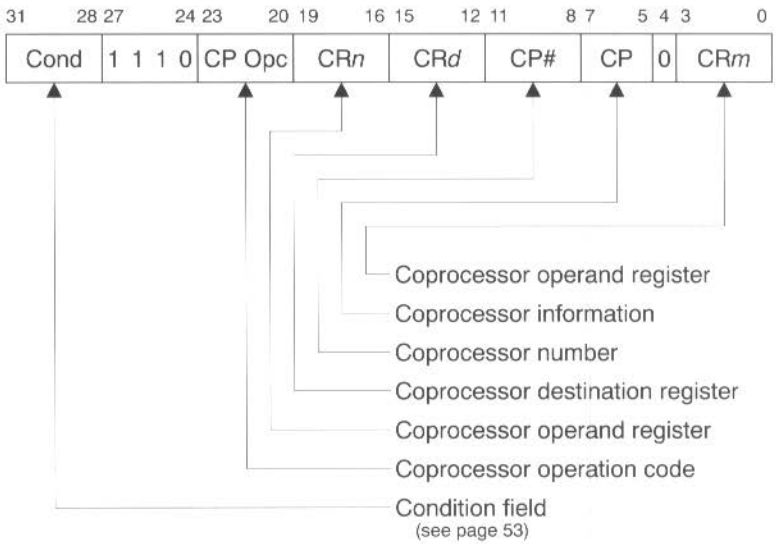
The above examples assume that suitable supervisor code exists at the SWI vector address, for instance:

```
        B       Supervisor              ; SWI entry point

EntryTable                              ; addresses of supervisor routines
        DCD     ZeroRtn
        DCD     ReadCRtn
        DCD     WriteIRtn
        ...

Zero    EQU     0
ReadC   EQU     256
WriteI  EQU     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.

        STM     R13,{R0-R2,R14}         ; Save work registers and return
                                        ; address
        BIC     R0,R14,#&FC000003       ; Clear PSR bits
        LDR     R0,[R0,#-4]             ; Get SWI instruction
        BIC     R0,R0,#&FF000000        ; Clear top 8 bits
        MOV     R1,R0,LSR #8            ; Get routine offset
        ADR     R2,EntryTable           ; Get start address of entry table
        LDR     R15,[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.
```

Coprocessor data operations (CDP)

Instruction for telling a coprocessor to perform some internal operation

Instruction format



Assembler syntax

`CDP«cond» CP#,operation,CRd,CRn,CRm«,info»`

`«cond»`	is a two-character condition mnemonic; see the section *The condition field* on page 53.
`CP#`	is the unique number of the required coprocessor, which must be a symbol defined via the CP directive.
`operation`	is evaluated to a constant and placed in the `CP Opc` field.
`CRd`, `CRn`, & `CRm`	are expressions evaluating to a valid coprocessor register number, which must be a symbol defined via the CN directive.
`info`	where present is evaluated to a constant and placed in the `CP` field.

Synopsis

This instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM activity, allowing the coprocessor and ARM to perform independent tasks in parallel.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

The coprocessor fields

Only bit 4 and bits 24 to 31 are significant to ARM; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors 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 coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CR*n* and CR*m*, and place the result in CR*d*.

Restriction

Current ARM chips have a fault in the implementation of CDP 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 coprocessor 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.

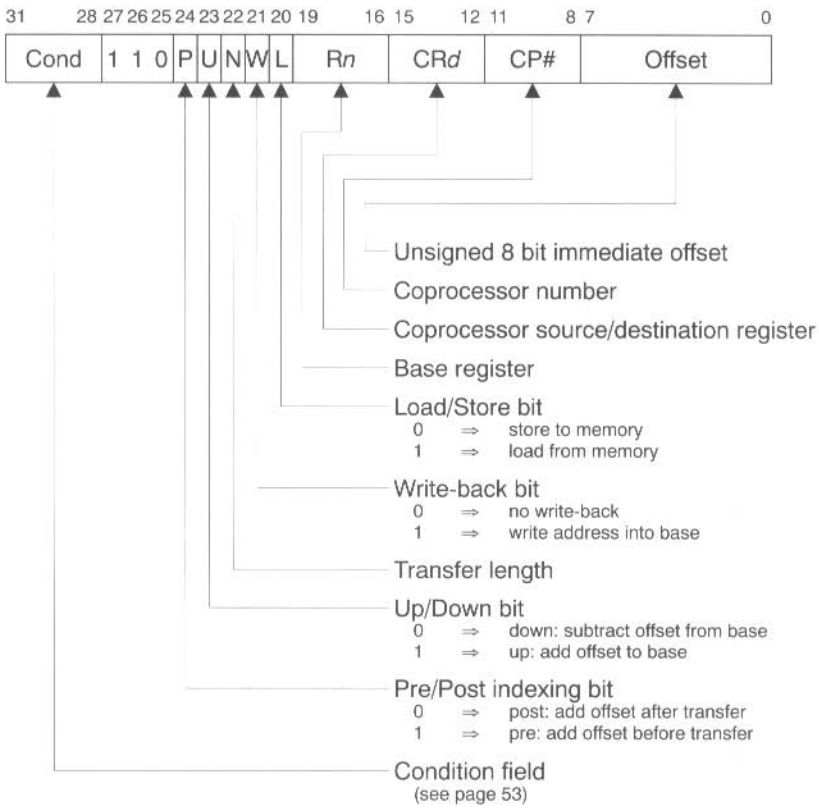
Examples

```
CDP     p1,10,CR1,CR2,CR3       ; Request coprocessor 1 to do
                                ; operation 10 on CR2 and CR3,
                                ; and put the result in CR1.

CDPEQ   p2,5,CR1,CR2,CR3,2      ; If Z flag is set, request
                                ; coprocessor 2 to do operation 5
                                ; (type 2) on CR2 and CR3,
                                ; and put the result in CR1.
```

Coprocessor data transfers (LDC, STC)

Instructions for transferring data between the coprocessor and main memory

Instruction format



Assembler syntax

`LDC|STC«cond»«L» CP#,CRd,address`

`LDC` loads from memory to coprocessor (L=1).

`STC` stores from coprocessor to memory (L=0).

`«L»` when present perform long transfer (N=1), otherwise perform short transfer (N=0).

`«cond»` is a two-character condition mnemonic; see the section *The condition field* on page 53.

`CP#` is the unique number of the required coprocessor, which must be a symbol defined via the CP directive.

`CRd` is an expression evaluating to a valid coprocessor register number, which must be a symbol defined via the CN directive.

`address` can be:

- An expression which generates an address:

  `expression`

  ObjAsm 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 ObjAsm will subtract 8 from the offset value to allow for ARM pipelining.

`«!»` if present sets the W bit to write-back the base register.

Synopsis

These instructions are used to load (LDC) or store (STC) a subset of the coprocessor's registers directly to memory. ARM is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

The instruction is only executed if the condition is true. The various conditions are defined in the section *The condition field* on page 53.

The coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CR*d* field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CR*d* 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 wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and added to (U=1) or subtracted from (U=0) a base register (R*n*); this calculation may be performed 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.

Address alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on A[1:0] and might be interpreted by the memory system.

Use of R15

If R*n* 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 coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

32 bit operation

If R15 is specified as the base register (R*n*), you must not use write-back.

Examples

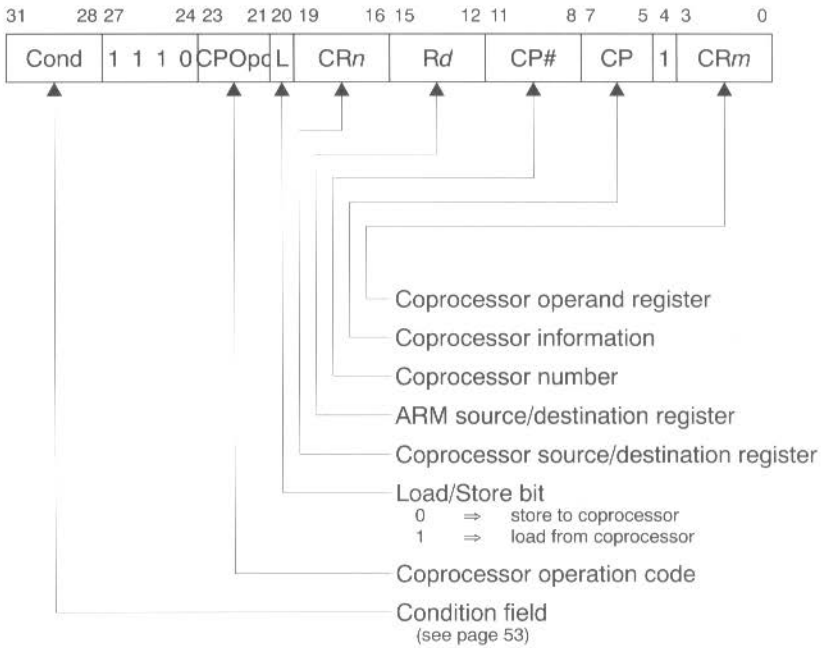
```
LDC     p1,CR2,table            ; Load CR2 of coprocessor 1 from
                                ; address table, using a PC relative
                                ; address.

STCEQL  p2,CR3,[R5,#24]!        ; Conditionally store CR3 of
                                ; coprocessor 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. ObjAsm will adjust the offset appropriately.

Coprocessor register transfers (MCR, MRC)

Instructions for communicating information between ARM and a coprocessor

Instruction format



Assembler syntax

MCR|MRC«*cond*» CP#,*operation*,Rd,CRn,CRm«,*info*»

MCR	moves from coprocessor to ARM register (L=1).
MRC	moves from ARM register to coprocessor (L=0).
«*cond*»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
CP#	is the unique number of the required coprocessor, which must be a symbol defined via the CP directive.
operation	is evaluated to a constant and placed in the CP Opc field.
Rd	is an expression evaluating to a valid ARM register number.

`CRn` & `CRm`	are expressions evaluating to a valid coprocessor register number, which must be a symbol defined via the CN directive.
`info`	where present is evaluated to a constant and placed in the `CP` field.

Synopsis

These instructions are used to communicate information directly between ARM and a coprocessor. An example of a coprocessor to ARM register transfer (MCR) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, 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 coprocessor illustrates the use of an ARM register to coprocessor transfer (MRC).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM PSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the PSR to control the subsequent flow of execution.

Note that the ARM6 series and later have an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed during coprocessor register transfers.

The instruction is only executed if the condition is true. The various conditions are defined in section *The condition field* on page 53.

The coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon to respond.

The CP Opc, CR*n*, CP and CR*m* fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CR*n* is the coprocessor register which is the source or destination of the transferred information, and CR*m* is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

Transfers to R15

When a coprocessor 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) of the PSR. 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 coprocessor register transfer from ARM with R15 as the source register will store the PC together with the PSR flags.

32 bit operation

Transfers to R15

When a coprocessor 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) of the CPSR. The other bits of the transferred word are ignored, and the PC and other PSR flags are unaffected by the transfer. This is what you would expect as an extension of the 26 bit behaviour.

Transfers from R15

A coprocessor register transfer from ARM with R15 as the source register will store the PC+12. Unlike the 26 bit behaviour, it does not store the CPSR to the coprocessor.

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

Undefined instructions

Instruction format

Assembler syntax

At present ObjAsm has no mnemonics for generating these instructions. If they are adopted in the future for some specified use, suitable mnemonics will be added to ObjAsm. Until such time, these instructions should not be used.

Synopsis

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering these instructions to any coprocessors which may be present, and all coprocessors must refuse to accept them by letting CPA float HIGH.

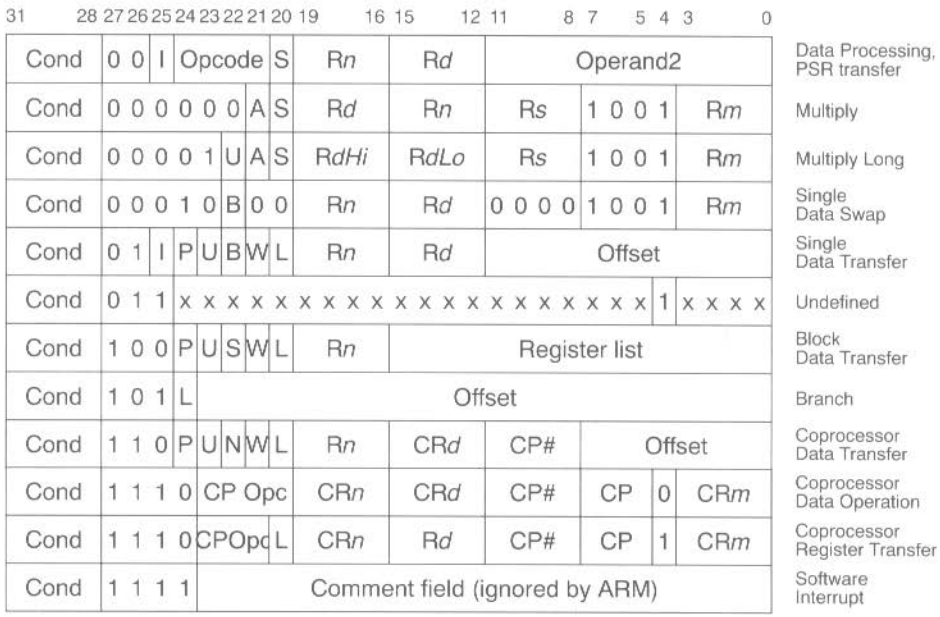
(Note that some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 5 or bit 6 changed to a 1. These instructions should be avoided, as their action may change in future ARM implementations.)

The instruction is only executed if the condition is true. The various conditions are defined in section *The condition field* on page 53.

Instruction set summary

Instructions available on ARM, briefly summarised

Instruction formats



Assembler syntax

```
B|BL«cond» expression

MOV|MVN«cond»«S» Rd,op2
CMN|CMP|TEQ|TST«cond»«P» Rn,op2
ADC|ADD|AND|BIC|OR|ORR|RSB|RSC|SBC|SUB«cond»«S» Rd,Rn,op2

MRS«cond» Rd,psr
MSR«cond» psr,Rm
MSR«cond» psrf,Rm
MSR«cond» psrf,#expression

MUL«cond»«S» Rd,Rm,Rs
MLA«cond»«S» Rd,Rm,Rs,Rn

UMULL|SMULL|UMLAL|SMLAL«cond»«S» RdLo,RdHi,Rm,Rs

LDR|STR«cond»«B»«T» Rd,address
```

```
LDM|STM«cond»FD|ED|FA|EA|IA|IB|DA|DB Rn«!»,Rlist«^»
SWP«cond»«B» Rd,Rm,[Rn]
SWI«cond» expression
CDP«cond» CP#,operation,CRd,CRn,CRm«,info»
LDC|STC«cond»«L» CP#,CRd,address
MCR|MRC«cond» CP#,operation,Rd,CRn,CRm«,info»
```

Parameters for the above, alphabetically sorted

address can be:

- An expression which generates an address:

  expression

  ObjAsm 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* (not available for LDC/STC).

- A post-indexed addressing specification:

`[Rn],#expression`	offset of *expression* bytes
`[Rn],«+|-»Rm«,shift»`	offset of ±contents of index register, shifted by *shift* (not available for LDC/STC).

Rn and Rm are expressions evaluating to a valid ARM register number. Note if Rn is R15 then ObjAsm will subtract 8 from the offset value to allow for ARM pipelining.

shift is a general shift operation (see the section *Shift types* on page 57), but note that the shift amount may not be specified by a register.

«!» if present sets the W bit to write-back the base register.

«B» means to transfer a byte, otherwise a word is transferred.

«cond»	is a two-character condition mnemonic; see the section *The condition field* on page 53.
CP#	is the unique number of the required coprocessor, which must be a symbol defined via the CP directive.
CR*d*, CR*n*, & CR*m*	are expressions evaluating to a valid coprocessor register number, which must be a symbol defined via the CN directive.
expression	for B and BL is a program-relative expression describing the branch destination, from which ObjAsm calculates the offset; for SWI, it is evaluated and placed in the comment field as a SWI number (which is ignored by ARM).
#expression	is an expression symbolising a 32 bit value. If *#expression* is used, ObjAsm will attempt to match the expression by generating a shifted immediate 8-bit field. If this is impossible, it will give an error.
info	where present is evaluated to a constant and placed in the **CP** field.
«L»	when present perform long transfer (N=1), otherwise perform short transfer (N=0).
op2	may be any of the operands that the barrel shifter can produce. The syntax is **Rm**«**,***shift*» or *#expression* If *#expression* is used, ObjAsm 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* **Rs** or *shiftname #expression*, or **RRX** (rotate right one bit with extend). *shiftname*s are: **ASL**, **LSL**, **LSR**, **ASR**, and **ROR**. (ASL is a synonym for LSL, and the two assemble to the same code.) See *Shift types* on page 57.
operation	is evaluated to a constant and placed in the **CP Opc** field.
«P»	means to take the result of a CMN, CMP, TEQ or TST operation, and move it to the bits of R15 that hold the PSR – even though the instruction has no destination register. Bits corresponding to the PC are masked out, as are (in User mode) the I, F, and mode bits.
psr	is **CPSR**, **CPSR_all**, **SPSR** or **SPSR_all**. (**CPSR** and **CPSR_all** are synonyms, as are **SPSR** and **SPSR_all**.)

psrf	is `CPSR_flg` or `SPSR_flg`. The most significant four bits of `Rm` or `#expression` are written to the N, Z, C and V flags respectively.
`Rd`, `RdLo`, `RdHi`, `Rm`, `Rn` & `Rs`	are expressions evaluating to a valid ARM register number.
Rlist	is either a comma-separated list of registers and/or of register ranges indicated by hyphens, all enclosed in `{}` (e.g. `{R0,R2-R7,R10}`); or an expression evaluating to the 16 bit operand.
«S»	means to set the PSR's condition codes from the operation. ObjAsm forces this for CMN, CMP, TEQ and TST, provided the P flag is not specified.
«T»	means to set the W bit in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
«!»	if present sets the W bit to write-back the base register.
«^»	if present sets the S bit to load the PSR with the PC, or forces storing of user bank registers when in a non-user mode.

Synopsis

For a detailed synopsis of the various instructions, see the following sections:

Further instructions

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

Extended range immediate constants

Synopsis

In the case of an instruction such as

```
MOV     R0,#VALUE
```

ObjAsm 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). ObjAsm will accept this syntax, but will convert it and generate object code for

```
MVN     Rn,#0
```

which results in R*n* containing –1. Such conversions also takes place between the following pairs of instructions:

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

The ADR instruction

Assembler syntax

`ADR«cond» register,expression`

Synopsis

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, ObjAsm provides an ADR instruction.

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

- **Register-relative:** `ADD|SUB register,register2,#constant`
  will be produced, where *register2* is the register to which the expression is relative.
- **Program-relative:** `ADD|SUB register,PC,#constant`
  will be produced.
- **Numeric:** `MOV|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, ObjAsm 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.

The ADRL instruction

Assembler syntax

`ADR«cond»L register,expression`

Synopsis

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

Assembler syntax

```
LDR register,=expression
```

Synopsis

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

ObjAsm will take certain actions with literals. 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. See the section *Organisational directives* – END, ORG, LTORG *and* KEEP on page 141 for further information.

6 Floating point instructions

The ARM 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.

Generally, programs do not need to know whether a coprocessor is fitted; the only effective difference is in the speed of execution. Note that there may be slight variations in accuracy between hardware and software – refer to the instructions supplied with the coprocessor for details of these variations.

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:

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
FPE 400	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

The old FPE has two different variants. Versions up to (but not including) 3.40 do not provide any hardware support, whereas versions 3.40 to 3.99 inclusive provide support for the FPPC hardware – if it is fitted. All versions of the FPE 400 provide support for the FPA hardware.

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 6.1 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 $\pm 0.fraction \times 2^{-126}$.
- If the exponent is in the range 1 to 254, the number represented is $\pm 1.fraction \times 2^{exponent-127}$.
- 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)

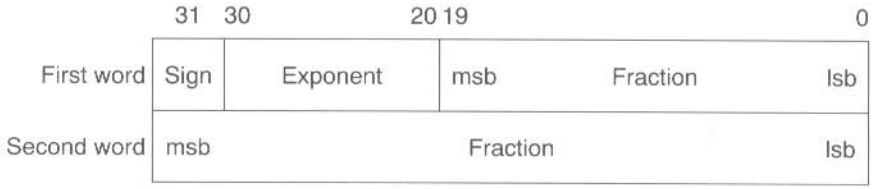


Figure 6.2 Double precision format

- 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 $\pm 0.fraction \times 2^{-1022}$.
- If the exponent is in the range 1 to 2046, the number represented is $\pm 1.fraction \times 2^{exponent-1023}$.
- 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)

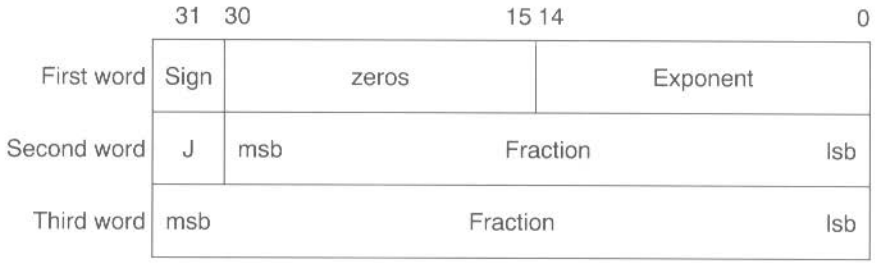


Figure 6.3 Double extended precision format

- 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 $\pm\infty$.
- 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, to have then been 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 FPE 400, 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 6.4 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 d, and e3 of the exponent e. The significand has an assumed decimal point between d18 and d17, and is normalised so that for a normal number $1 \le d18 \le 9$. 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. (By comparison, an S format number has 9 digits of significand and a maximum exponent of 53; a D format 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 $\pm\infty$ 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 $\pm\infty$.
- 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 6.5 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 d, and e6 of the exponent e. The significand has an assumed decimal point between d23 and d22, and is normalised so that for a normal number $1 \le d23 \le 9$. 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. (By comparison, an S format number has 9 digits of significand and a maximum exponent of 53; a D format 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 $\pm\infty$ 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 $\pm\infty$.
- 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.

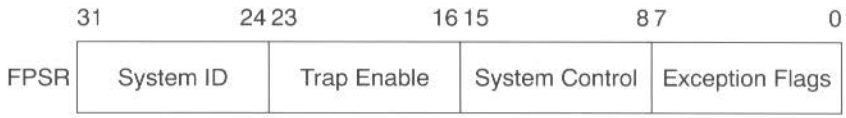


Figure 6.6 Floating point status register byte usage

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 System IDs are currently defined:

System	System ID
Old FPE	&00
FPPC	&80
FPE 400	&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 *Cumulative Exception Flags Byte* on page 126.

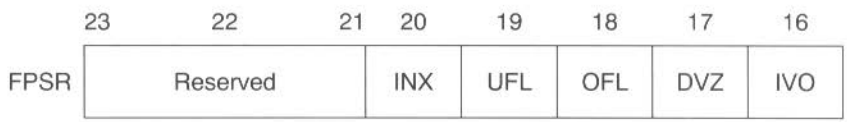


Figure 6.7 Exception trap enable byte

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.

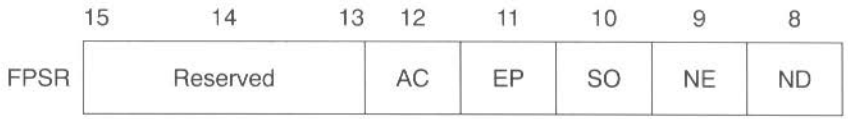


Figure 6.8 System control byte

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 FPE 400:

ND	**N**o **D**enormalised numbers
NE	**N**aN **E**xception
SO	**S**elect synchronous **O**peration of FPA
EP	Use **E**xpanded **P**acked decimal format
AC	Use **A**lternative 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 6.9 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 6.10 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:

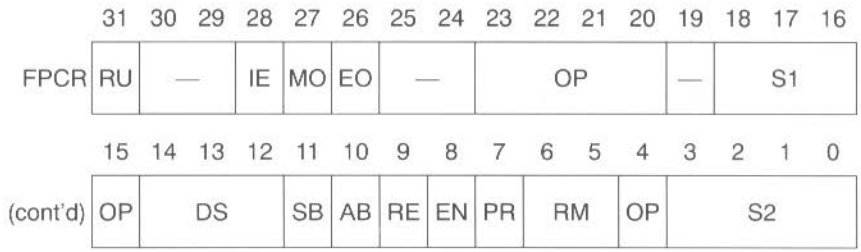


Figure 6.11 FPCR bit allocation in the FPA system

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 *Floating point number formats* on page 119.

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* on page 132 and *Floating point literals* on page 133.

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* DCF*x 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 writeback 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 writeback to add the offset to the base register; but in post-indexed mode the assembler forces writeback for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if writeback will take place.

Examples:

```
LDFS    F0,[R0]         ; load F0 from address held in R0
                        ; (single precision)
STFP    F1,[R2]         ; store number held in F1 at R2
                        ; as a packed decimal number
```

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`

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 FPPC system, or by some versions of the old FPE:

- versions 2.80 - 2.84 do not support them
- versions 2.85 - 3.39 do support them
- version 3.40 – which is effectively a version of 2.80 that also provides FPPC hardware support – does not support these instructions.

Attempting to execute these instructions on systems that do not provide them will cause undefined instruction traps, so you should only use these instructions in software intended for machines you are confident are using an appropriate version of the old FPE, or the FPE 400, 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 three registers 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 writeback 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 writeback to add the offset to the base register; but in post-indexed mode the assembler forces writeback for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if writeback 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 := *Rd*	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	Subtract	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 FPE 400.

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 F*n* less than F*m* (or –F*m*)
Z	Equal	
C	Greater than or equal	ie F*n* greater than or equal to F*m* (or –F*m*)
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.

7 Directives

This chapter describes the directives available in the assembler, which provide a powerful range of extra features.

Storage reservation and initialisation – DCB, DCW and DCD

`DCB`	Defines one or more bytes: can be replaced by =
`DCW`	Defines one or more half-words (16-bit numbers)
`DCD`	Defines one or more words: can be replaced by &
`%`	Reserves a zeroed area of store

The syntax of the first three directives is:

```
«label» directive expression-list
```

DCD can take program-relative and external expressions as well as numeric ones. In the case of DCB, the *expression-list* can include string expressions, the characters of which are loaded into consecutive bytes in store. Unlike C-strings, ObjAsm strings do not contain an implicit trailing NUL, so a C-string has to be fabricated thus:

```
C_string DCB "C_string",0
```

The syntax of `%` is:

```
«label» % numeric-expression
```

This directive will initialise to zero the number of bytes specified by the *numeric expression*.

Note that an *external expression* consists of an external symbol followed optionally by a constant expression. The external symbol **must** come first.

Floating point store initialisation – DCFS and DCFD

DCFS	Defines single precision floating point values
DCFD	Defines double precision floating point values

The syntax of these directives is:

```
«label» directive fp-constant«,fp-constant»
```

Single precision numbers occupy one word, and double precision numbers occupy two; both should be word aligned. An *fp-constant* takes one of the following forms:

`«-»integer E«-»integer`	e.g. `1E3`, `-4E-9`
`«-»«integer».integer«E«-»integer»`	e.g. `1.0`, `-.1`, `3.1E6`

`E` may also be written in lower case.

Describing the layout of store – ^ and

^	Sets the origin of a storage map
#	Reserves space within a storage map

The syntax of these directives is:

```
        ^ expression«,base-register»
«label» # expression
```

The ^ directive sets the origin of a storage map at the address specified by the *expression*. A storage map location counter, @, is also set to the same address. The *expression* must be fully evaluable in the first pass of the assembly, but may be program-relative. If no ^ directive is used, the @ counter is set to zero. @ can be reset any number of times using ^ to allow many storage maps to be established.

Space within a storage map is described by the # directive. Every time # is used its *label* (if any) is given the value of the storage location counter @, and @ is then incremented by the number of bytes reserved.

In a ^ directive with a *base register*, the register becomes implicit in all symbols defined by # directives which follow, until cancelled by a subsequent ^ directive. These register-relative symbols can later be quoted in load and store instructions. For example:

```
        ^ 0,r9
        # 4
Lab     # 4
        LDR r0,Lab
```

is equivalent to:

```
        LDR r0,[r9,#4]
```

Organisational directives – END, ORG, LTORG and KEEP

`END`

The assembler stops processing a source file when it reaches the END directive. If assembly of the file was invoked by a GET directive, the assembler returns and continues after the GET directive (see *Links to other source files* – GET/INCLUDE on page 142). If END is reached in the top-level source file during the first pass without any errors, the second pass will begin. Failing to end a file with END is an error.

`ORG` *`numeric-expression`*

A program's origin is determined by the ORG directive, which sets the initial value of the program location counter. Only one ORG is allowed in an assembly and no ARM instructions or store initialisation directives may precede it. If there is no ORG, the program is relocatable and the program counter is initialised to 0.

`LTORG`

LTORG directs that the current literal pool be assembled immediately following it. A default LTORG is executed at every END directive which is not part of a nested assembly, but large programs may need several literal pools, each closer to where their literals are used to avoid violating LDR's 4KB offset limit.

`KEEP` *`«symbol»`*

The assembler does not by default describe local symbols (i.e. *non-exported* symbols; see *Links to other object files* – IMPORT *and* EXPORT on page 142) in its output object file. However, they can be retained in the object file's symbol table by using the KEEP directive. If the directive is used alone all symbols are kept; if only a specific symbol needs to be kept it can be specified by name.

Links to other object files – IMPORT and EXPORT

```
IMPORT symbol«[FPREGARGS]»«,WEAK»
EXPORT symbol«[FPREGARGS,DATA,LEAF]»
```

IMPORT provides the assembler with a name (symbol) which is not defined in this assembly, but will be resolved at link time to a symbol defined in another, separate object file. The symbol is treated as a program address; if the WEAK attribute is given the Linker will not fault an unresolved reference to this symbol, but will zero the location referring to it. If [FPREGARGS] is present, the symbol defines a function which expects floating point arguments passed to it in floating point registers.

EXPORT declares a symbol for use at link time by other, separate object files. FPREGARGS signifies that the symbol defines a function which expects floating point arguments to be passed to it in floating point registers. DATA denotes that the symbol defines a code-segment datum rather than a function or a procedure entry point, and LEAF that it is a leaf function which calls no other functions.

Links to other source files – GET/INCLUDE

```
GET filename
INCLUDE filename
```

GET includes a file within the file being assembled. This file may in turn use GET directives to include further files. Once assembly of the included file is complete, assembly continues in the including file at the line following the GET directive. INCLUDE is a synonym for GET.

Diagnostic generation – ASSERT and !

```
ASSERT logical-expression
. ! arithmetic-expression, string-expression
```

ASSERT supports diagnostic generation. If the *logical expression* returns {FALSE}, a diagnostic is generated during the second pass of the assembly. ASSERT can be used both inside and outside macros.

! is related to ASSERT but is inspected on both passes of the assembly, providing a more flexible means for creating custom error messages. The arithmetic expression is evaluated; if it equals zero, no action is taken during pass one, but the string is printed as a warning during pass two; if the expression does not equal zero, the string is printed as a diagnostic and the assembly halts after pass one.

Dynamic listing options – OPT

The OPT directive is used to set listing options from within the source code, providing that listing is turned on. The default setting is to produce a normal listing including the declaration of variables, macro expansions, call-conditioned directives and MEND directives, but without producing a pass one listing. These settings can be altered by adding the appropriate values from the list below, and using them with the OPT directive as follows:

OPT *n*	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw: issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on the listing of SET, GBL and LCL directives.
32	Turns off the listing of SET, GBL and LCL directives.
64	Turns on the listing of macro expansions.
128	Turns off the listing of macro expansions.
256	Turns on the listing of macro calls.
512	Turns off the listing of macro calls.
1024	Turns on the pass one listing.
2048	Turns off the pass one listing.
4096	Turns on the listing of conditional directives.
8192	Turns off the listing of conditional directives.
16384	Turns on the listing of MEND directives.
32768	Turns off the listing of MEND directives.

Titles – TTL and SUBT

Titles can be specified within the code using the TTL (title) and SUBT (subtitle) directives. Each is used on all pages until a new title or subtitle is called. If more than one appears on a page, only the latest will be used: the directives alone create blank lines at the top of the page. The syntax is:

```
TTL title
SUBT subtitle
```

Miscellaneous directives – ALIGN, NOFP, RLIST and ENTRY

`ALIGN «power-of-two«,offset-expression»»`

After store-loading directives have been used, the program counter (PC) will not necessarily point to a word boundary. If an instruction mnemonic is then encountered, the assembler will insert up to three bytes of zeros to achieve alignment. However, an intervening label may not then address the following instruction. If this label is required, ALIGN should be used. On its own, ALIGN sets the instruction location to the next word boundary. The optional *power-of-two* parameter – which is given in bytes – can be used to align with a coarser byte boundary, and the *offset expression* parameter to define a byte offset from that boundary.

`NOFP`

In some circumstances there will be no support in either target hardware or software for floating point instructions. In these cases the NOFP directive can be used to ensure that no floating point instructions or directives are allowed in the code.

`RLIST`

The syntax of this directive is:

`label` `RLIST` *`list-of-registers`*

The RLIST (register list) directive can be used to give a name to a set of registers to be transferred by LDM or STM. *List-of-registers* is a list of register names or ranges enclosed in `{}` (see *Block data transfer* (LDM, STM) on page 88).

`ENTRY`

The ENTRY directive declares its offset in its containing AREA to be the unique entry point to any program containing this AREA.

8 Symbolic capabilities

The assembler also has a range of symbolic capabilities, with which you can set up symbols as constants or as variables. These are described below.

Setting constants

The **EQU** and * directives are used to give a symbolic name to a fixed or program-relative value. The syntax is:

```
label EQU expression
label * expression
```

RN defines register names. Registers can only be referred to by name. The names `R0-R15`, `r0-r15`, `PC`, `pc`, `LR`, `lr`, `SP` and `sp` are predefined. Names may also be defined for the registers used by the ARM Procedure Call Standard; see *Controlling syntax* on page 11.

FN defines the names of floating point registers. The names `F0-F7` and `f0-f7` are built in. The syntax is:

```
label RN numeric-expression
label FN numeric-expression
```

CP gives a name to a coprocessor number, which must be within the range 0 to 15. The names `p0-p15` are pre-defined.

CN names a coprocessor register number; `c0-c15` are pre-defined. The syntax is:

```
label CP numeric-expression
label CN numeric-expression
```

Local and global variables – GBL, LCL and SET

While most symbols have fixed values determined during assembly, variables have values which may change as assembly proceeds. The assembler supports both global and local variables. The scope of global variables extends across the entire source file while that of local variables is restricted to a particular instantiation of a macro (see the chapter *Macros* on page 157). Variables must be declared before use with one of these directives.

`GBLA`	Declares a global arithmetic variable. Values of arithmetic variables are 32-bit unsigned integers.
`GBLL`	Declares a global logical variable
`GBLS`	Declares a global string variable
`LCLA`	Declares and initialises a local arithmetic variable (initial state zero)
`LCLL`	Declares and initialises a local logical variable (initial state false)
`LCLS`	Declares and initialises a local string variable (initial state null string)

The syntax of these directives is:

```
directive variable-name
```

The value of a variable can be altered using the relevant one of the following three directives:

`SETA`	Sets the value of an arithmetic variable
`SETL`	Sets the value of a logical variable
`SETS`	Sets the value of a string variable

The syntax of these directives is:

```
variable-name directive expression
```

where *expression* evaluates to the value being assigned to the variable named.

(You can also declare and set the value of global variables at assembly time; see *Predefining a variable* on page 12.)

Variable substitution – $

Once a variable has been declared its name cannot be used for any other purpose, and any attempt to do so will result in an error. However, if the $ character is prefixed to the name, the variable's value will be substituted before the assembler checks the line's syntax. Logical and arithmetic variables are replaced by the result of performing a `:STR:` operation on them (see *Unary operators* on page 149); string variables are replaced by their value.

Built-in variables

There are seven built-in variables. They are:

`{PC}` or `.`	Current value of the program location counter.
`{VAR}` or `@`	Current value of the storage-area location counter.
`{TRUE}`	Logical constant true.
`{FALSE}`	Logical constant false.
`{OPT}`	Value of the currently set listing option. The OPT directive can be used to save the current listing option, force a change in it or restore its original value.
`{CONFIG}`	Has the value `32` if the assembler is in 32-bit program counter mode, and the value `26` if it is in 26-bit mode.
`{ENDIAN}`	Has the value "`big`" if the assembler is in big-endian mode, and the value "`little`" if it is in little-endian mode.

9 Expressions and operators

Expressions are combinations of simple values, unary and binary operators, and brackets. There is a strict order of precedence in their evaluation: expressions in brackets are evaluated first, then operators are applied in precedence order. Adjacent unary operators evaluate from right to left; binary operators of equal precedence are evaluated from left to right.

The assembler includes an extensive set of operators for use in expressions, many of which resemble their counterparts in high-level languages.

Unary operators

Unary operators have the highest precedence (bind most tightly) so are evaluated first. A unary operator precedes its operand, and adjacent operators are evaluated from right to left.

Operator	Usage	Explanation
`?`	`?A`	Number of bytes generated by line defining label `A`.
`BASE` `INDEX`	`:BASE:A` `:INDEX:A`	If `A` is a PC-relative or register-relative expression then `BASE` returns the number of its register component, and `INDEX` the offset from that base register. `BASE` and `INDEX` are most likely to be of use within macros.
`LEN`	`:LEN:A`	Length of string `A`
`CHR`	`:CHR:A`	ASCII string of `A`
`STR`	`:STR:A`	Hexadecimal string of `A`. `STR` returns an eight-digit hexadecimal string corresponding to a numeric expression, or the string `T` or `F` if used on a logical expression.
`+`	`+A`	Unary plus
`-`	`-A`	Unary negate. `+` and `-` can act on numeric, program-relative and string expressions.
`NOT`	`:NOT:A`	Bitwise complement of `A`

Operator	Usage	Explanation
LNOT	:LNOT:A	Logical complement of A
DEF	:DEF:A	{TRUE} if A is defined, otherwise {FALSE}

Binary operators

Binary operators are written between the pair of sub-expressions on which they operate. Operators of equal precedence are evaluated in left to right order. The binary operators are presented below in groups of equal precedence, in decreasing precedence order.

Multiplicative operators

These are the binary operators which bind most tightly and have the highest precedence:

Operator	Usage	Explanation
*	A*B	Multiply
/	A/B	Divide
MOD	A:MOD:B	A modulo B

These operators act only on numeric expressions.

String manipulation operators

Operator	Usage	Explanation
LEFT	A:LEFT:B	The leftmost B characters of A
RIGHT	A:RIGHT:B	The rightmost B characters of A
CC	A:CC:B	B concatenated on to the end of A

In the two slicing operators LEFT and RIGHT, A must be a string and B must be a numeric expression.

Shift operators

Operator	Usage	Explanation
ROL	A:ROL:B	Rotate A left B bits
ROR	A:ROR:B	Rotate A right B bits
SHL	A:SHL:B	Shift A left B bits
SHR	A:SHR:B	Shift A right B bits

The shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second. Note that SHR is a logical shift and does not propagate the sign bit.

Addition and logical operators

Operator	Usage	Explanation
AND	A:AND:B	Bitwise AND of A and B
OR	A:OR:B	Bitwise OR of A and B
EOR	A:EOR:B	Bitwise Exclusive OR of A and B
+	A+B	Add A to B
–	A–B	Subtract B from A

The bitwise operators act on numeric expressions. The operation is performed independently on each bit of the operands to produce the result.

Relational operators

Operator	Usage	Explanation
=	A=B	A equal to B
>	A>B	A greater than B
>=	A>=B	A greater than or equal to B
<	A<B	A less than B
<=	A<=B	A less than or equal to B
/=	A/=B	A not equal to B
<>	A<>B	A not equal to B

The relational operators act upon two operands of the same type to produce a logical value. Allowable types of operand are numeric, program-relative, register-relative, and strings. Strings are sorted using ASCII ordering. String A will be less than string B if it is either a leading substring of string B, or if the left-most character of A in which the two strings differ is less than the corresponding character in string B. Note that arithmetic values are unsigned, so the value of 0>-1 is {FALSE}.

Boolean operators

These are the weakest binding operators with the lowest precedence.

Operator	Usage	Explanation
LAND	A:LAND:B	Logical AND of A and B
LOR	A:LOR:B	Logical OR of A and B
LEOR	A:LEOR:B	Logical Exclusive OR of A and B

The Boolean operators perform the standard logical operations on their operands, which should evaluate to {TRUE} or {FALSE}.

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 known as the IF directive:

`[ logical_expression` or `IF logical_expression`

This is the ELSE directive:

`|` or `ELSE`

and this is the ENDIF directive:

`]` or `ENDIF`

A block which is being conditionally assembled can contain several `[ | ]` directives; that is, conditional assembly can be nested.

Simple use of the IF and ENDIF directives

You can use the IF and ENDIF directives (without the ELSE directive) like this:

```
[ 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.

Simple use of the IF, ELSE and ENDIF directives

Alternatively you can use all three directives, thus:

```
[ 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 NoTerse option

Lines conditionally skipped by these directives are not listed unless ObjAsm is switched from its default terse mode. For desktop assembly, you must choose **NoTerse** from ObjAsm's menu (see *Listings* on page 15); for command line usage, you must specify the `-NoTerse` command line option (see page 22).

An example

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:

```
DISC    *       0
TAPE    *       1
SWITCH  *       DISC
        ...code...
        [ 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.

The syntax is:

`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 you a means of placing a single instruction in your source which will be expanded at assembly time to several assembler instructions and directives, just as if you'd written those instructions and directives within the source at that point.

As an example, we will 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,$cc    ; 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

The fact that a macro is about to be defined is given by the directive **MACRO** in the instruction field:

```
MACRO
```

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:

```
MACRONAME       9,"disc",7
```

or:

```
MACRONAME       9
```

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, using the syntax:

$parameter=default_value

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
```

you could instead 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**. 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 symbols and labels (see the section *Symbols* on page 49 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 onentry. 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 Exception handling

This chapter describes the processor configuration and modes used by RISC OS when running on 32 bit architecture ARMs (i.e. ARM6 series and later), and the ways in which this affects exception handling. If you are writing any exception handler that you wish to run on such a processor, you must read both this chapter and the chapter *The* ARM CPU on page 29, especially the section *Exceptions*.

RISC OS processor configuration and modes

Early in its startup code, RISC OS writes to the ARM's control register to change it into the 32 bit program and data space configuration, where it remains. You must not alter the processor's configuration yourself when writing code for RISC OS.

Although RISC OS runs under a 32 bit configuration, it remains in 26 bit modes for normal operation, providing a high degree of backward compatibility with code written to run on earlier 26 bit processors.

However, because the processor is in a 32 bit configuration, all exceptions (including Undefined Instruction and Software Interrupt) force the processor to a privileged 32 bit mode appropriate to the exception. There are therefore some differences in exception handling between 26 and 32 bit architecture ARM chips, although RISC OS provides a considerable degree of backward compatibility by faking 26 bit behaviour on 32 bit architecture chips in most circumstances. For full details, see the next section.

The pre-veneers

To ensure easy backward compatibility, 32 bit aware versions of RISC OS install a pre-veneer on all hardware vectors apart from FIQ (see the section *Writing to the* FIQ *vector* on page 168) and address exception (which is never generated by a 32 bit configured ARM). Each pre-veneer first sets up R14 to contain a combined PC and PSR that will return the processor to the 26 bit mode it was in when the exception arose. It then places the processor in the privileged 26 bit mode used by the earlier 26 bit chips for that exception. It thus effectively fakes the earlier chips' behaviour. The pre-veneer is called before any exception handlers that are installed with software interfaces such as OS_ChangeEnvironment, so you can usually use such exception handlers unchanged on all versions of RISC OS (hardware dependencies excepted).

Entering 32 bit modes

One consequence of this is that **you may not enter a 32 bit mode with IRQs enabled**. Were you to do so, and an IRQ were to occur, the IRQ pre-veneer would be entered; then the IRQ handler would return you to a 26 bit mode, rather than the 32 bit mode you were in at the time of the IRQ.

Claiming the hardware vectors

Under earlier versions of RISC OS, you could also claim the hardware vectors directly, by overwriting the existing instruction on the vector, and replacing it afterwards. It was your responsibility to do any housekeeping, in particular checking for subsequent claimants before restoring the original instruction.

Under 32 bit aware versions of RISC OS, if you attempt to write to any hardware vector other than FIQ a data abort is generated. You must instead call the new SWI OS_ClaimProcessorVector (page 5-46 of the RISC OS 3 *Programmer's Reference Manual*), passing it the address of your exception handler. The handler is installed on the vector, and is called directly, before the pre-veneers. Such handlers are therefore entered in a 32 bit mode.

For handlers installed directly on the vector to work across all versions of RISC OS, you must therefore change the method of claiming and releasing the vector depending on the version of RISC OS:

- On versions up to RISC OS 3.1, you must write directly to the vector, doing any appropriate housekeeping yourself
- On later versions you must call OS_ClaimProcessorVector.

Your handler must also cope with running in both 26 bit and 32 bit modes.

Writing to the FIQ vector

On a 32 bit architecture ARM, the FIQ vector is entered in FIQ mode (i.e. the 32 bit form of the mode). There are no pre-veneers to simulate 26 bit behaviour. To install a FIQ handler, you must write directly to the FIQ vector, just as always.

The sample code below is the recommended way to write to the FIQ vector on both 26 and 32 bit configured processors – you can use the same code on all versions of RISC OS. Obviously the handler you install must cope with running in both 26 bit and 32 bit FIQ modes. In practice this is unlikely to be a problem, and most existing handlers will run unchanged once installed.

In the code, comments that are prefixed by '`32:`' apply to a 32 bit configured processor, and comments that are prefixed by '`26:`' apply to a 26 bit configured processor.

```
; We assume that at this point, you are already in a privileged 26 bit mode.

        ; 26: Does not alter processor mode. Reads as follows:
        ; NOP                           ; 26: Encodes a NOP (TST Ra,R0)
        ; Push  Ra                      ; 26: Pushes entry Ra onto stack
        ; ORR   Ra, Ra, #2_11000000     ; 26: Corrupts Ra
        ; NOP                           ; 26: Encodes a NOP (TEQ R9,Ra)
        ; ORR   Ra, Ra, #2_10000        ; 26: Corrupts Ra
        ; NOP                           ; 26: Encodes a NOP (TEQ R9,Rb)

        ; 32: Switch to _32 mode with IRQs and FIQs off.
        ; 32: Must switch interrupts off before switching mode as there can be
        ; 32: an interrupt after the MSR instruction but before the next one.
        MRS     Ra, CPSR_all            ; 32: Read privileged 26 bit mode,
        Push    Ra                      ; 32: and push it onto the stack
        ORR     Ra, Ra, #2_11000000     ; 32: Set IRQ and FIQ disable bits
        MSR     CPSR_all, Ra            ; 32: Disable IRQs and FIQs
        ORR     Ra, Ra, #2_10000        ; 32: Set M4 bit (for 32 bit mode)
        MSR     CPSR_all, Ra            ; 32: Change to 32 bit mode

        ; Now do a NOP, to let things settle down:
        NOP                             ; e.g. MOV R0,R0

; Now in a suitable mode to write FIQ handler code to FIQ vector
; (&1C-&FC incl.), whatever the processor configuration.
; Code written should be able to run in both fiq_32 and fiq_26 modes,
; and should end with a SUBS PC,R14,#4 to return normally.
; For example we might write the handler code like this:

        ; Assume Rb already points to location from which to copy the handler.

        MOV     LR, #FIQVector          ; Get address of FIQ vector

40      LDR     Ra, [Rb], #4            ; Get opcode.
        TEQS    Ra, #0                  ; All done?
        STRNE   Ra, [LR], #4            ; No, so copy to FIQ area...
        BNE     %BT40                   ; ...and repeat for next opcode.

        ; The above may not be optimal, and is for illustration only.

; Having written FIQ vector, now need to restore the original
; privileged 26 bit mode.

        ; 26: Does not alter processor mode. Reads as follows:
        ; PULL  Ra                      ; 26: Pull entry Ra from stack
        ; NOP                           ; 26: Encodes a NOP (TST Ra,R0)

        PULL    Ra                      ; 32: Pull saved CPSR, and
        MSR     CPSR_all, Ra            ; 32: Restore privileged 26 bit mode

; Now back where we started, except Ra and Rb should be treated as corrupted.
; (We must assume neither is preserved, because we don't know the processor
; configuration.)
```

13 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.

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 3 *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, you should again see the RISC OS 3 *Programmer's Reference Manual*.

Unlike the construction of relocatable modules in high level languages, no tools are provided to generate substantial standard portions of code. 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

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.

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
```

Example

This product is supplied with the source for an example relocatable module that provides an extra soft screen mode: Mode 63. This has to be done via service call handling, and to be useful must be persistent, so providing a typical use of relocatable modules.

There are two source files held in `AcornC_C++.Examples.AsmModule.s`:

- The `ModeExHdr` file produces the module header, and may be useful for you to copy and edit to form headers for your own modules.
- The other file, `ModeExBody`, is the source for the main module body.

To build the module, use ObjAsm to assemble the source. Then link the resultant object files using Link, remembering first to set the **Module** option on its Setup dialogue box.

The module is specific to VIDC1 and VIDC1a, and so will not work on Acorn computers that are fitted with later versions of VIDC – such as the Risc PC.

14 Interworking assembler with C

Interworking assembly language and C – writing programs with both assembly language and C parts – requires using both ObjAsm and C/C++.

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 the appendix ARM *procedure call standard* on page 249 of the accompanying *Desktop Tools* guide.

Examples

The following examples are provided to demonstrate how to write programs combining assembly language and C.

PrintLib

The directory **AcornC_C++.Examples.PrintLib.s** contains three source files from which you can build a library: **PrintStr**, **PrintHex** and **PrintDble**. These are the assembly language sources for three screen printing routines: **print_string**, **print_hex** and **print_double**. These respectively print null terminated strings, integers in hexadecimal, and double precision floating point numbers in scientific format.

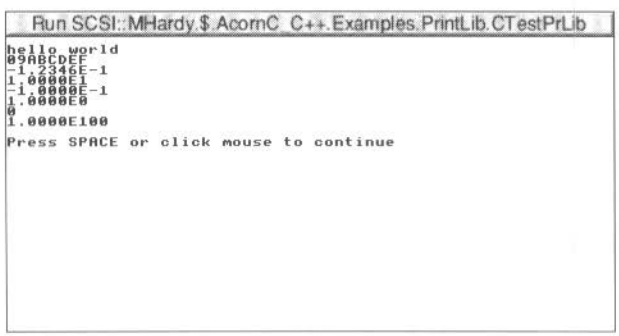
Each routine is written to obey the APCS, so it can be called from assembler, C, or any other high level language obeying the APCS. The sources for PrintLib illustrate several aspects of the APCS, such as the distinction between leaf and non-leaf procedures, and how floating point arguments are passed into a procedure.

Compiling the CTestPrLib example

To show you that you can call the routines in PrintLib from C, we've supplied a small C program in `AcornC_C++.Examples.PrintLib.c.CTestPrLib`. To build this example, you must:

1. Build the PrintLib library; you'll find instructions for this in the section *Assembler example* on page 134 of the *Desktop Tools* guide.
2. Start CC if you've not already got it loaded.
3. Drag the `CTestPrLib` file to the CC icon, which will display its **Setup** dialogue box with `CTestPrLib` already entered as the source to compile.
4. Add the full pathname of the PrintLib library to the list of **Libraries** on the Setup menu.
5. Click on Run to compile and link the program.
6. Save the program to disc.

To run the program, double click on its icon in the directory display to which you saved it. 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:



Compiling and linking CTestPrLib in separate stages

If you prefer, you can instead use the **Compile only** option of CC to compile `CTestPrLib` to an object file.

You can then use Link to link this object file with the libraries it uses. As well as the PrintLib library, it also uses the C library, so you must link three files: the object code for `CTestPrLib`, the library built from the PrintLib source, and the C library stubs held in `AcornC_C++.Libraries.clib.o.stubs`.

(In the above section *Compiling the CTestPrLib example*, the C library stubs were linked in because they were already in the Setup menu's default list of **Libraries**.)

CStatics

The directory `AcornC_C++.Examples.CStatics` gives an example of accessing C static variables from both assembler and C source code. The example builds to form a relocatable module providing a single * Command: `*CStatics`.

The files in the directory are as follows:

- `c.CInit` is the C source code. It declares two variables: `extern int var1`, which is provided by and initialised to 0 in `s.AsmInit` (see below), and `int var2`, which it initialises as 0. It prints the values of the two variables. It then calls the routine `Asm_Change_Vars` provided by `s.AsmInit` (see below), which changes the values of the two variables. Finally it prints the new values.
- `cmhg.Header` is the CMHG description file for the module. `hdr.CVars` is an assembler source file that contains a series of macros used by `s.AsmInit`. You will find these useful if you too ever need to share static data between assembler and C.
- `MakeFile` is the make file for the CStatics module.
- `o` is an empty directory used to hold the object files created when making the CStatics module.
- `s.AsmInit` is the assembler source code. It initialises the variable var1 to 0 and exports it; it also imports the variable var2. It also provides an APCS conformant routine `Asm_Change_Vars` which adds 10 to `var1` and subtracts 10 from `var2`. All this code makes heavy use of the macros in `hdr.CVars`.

To build the CStatics module, simply double click on the MakeFile.

When Make has completed, you can see the example in use. Load the resultant **CStatics** module by double clicking on it, then type **CStatics** at the command line. You will get this output:

```
var1 = 0
var2 = 0
var1 = 10
var1 = -10
```

If you repeat the ***CStatics** command you will see the variables change again:

```
var1 = 10
var2 = -10
var1 = 20
var1 = -20
```

and so on, every time you repeat the command.

Part 4 – Appendixes

Appendix A: Changes to the assembler

This release of the assembler replaces the product *Acorn Assembler Release* 2. It has seen the following major changes:

- The product has been merged with the C compiler.
- ObjAsm has added support for the ARM6, ARM7 and ARM7M versions of the processor. All new instructions are implemented, and there is also support for other new features such as big- and little-endian memory systems.
- ObjAsm now accepts instruction mnemonics in lower case; this feature can be disabled for backward compatibility.
- ObjAsm now supports many more options through its **Setup** menu.
- The AAsm tool is no longer supplied, but has been replaced – at least for this release – by a backward compatible mode.
- The Toolbox has been added to the product, to facilitate the design and coding of consistent user interfaces for RISC OS desktop applications. See the accompanying *User Interface Toolbox* guide.

Appendix B: Error messages

This appendix lists most of the common error messages that you may get when using the assembler, and gives an explanation for each one of the circumstances that may provoke the error.

- `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:
  $-4095 \leq e \leq +4095$
- `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.
- `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.
- `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 'I' 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 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.
- `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.

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:

- *new bit* := bit 33 EOR bit 20
- shift left the 33 bit number
- put in *new bit* at the bottom.
- Repeat for all the 32 *new bits* 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            ; If not aligned, produce bottom
                                        ;  of result word
        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 language C. 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.
```

Of course, the ARM7M core provides the Multiply Long class of instructions to perform a 64 bit signed or unsigned multiply or multiply-accumulate (see *Multiply Long and Multiply-Accumulate Long* (UMULL, SMULL, UMLAL, SMLAL) on page 81).

Appendix D: Warnings on the use of ARM assembler

The ARM processor family uses Reduced Instruction Set (RISC) techniques to maximise performance; as such, the instruction set allows some instructions and code sequences to be constructed that will give rise to unexpected (and potentially erroneous) results. These cases must be avoided by all machine code writers and generators if correct program operation across the whole range of ARM processors is to be obtained.

In order to be upwards compatible with future versions of the ARM processor family **never** use any of the undefined instruction formats:

- those shown in the section *Undefined instructions* on page 109, which the processor traps;
- those which are not shown in the manual and which don't trap (for example, a Multiply instruction where bit 5 or 6 of the instruction is set).

In addition the condition code 1111 (which was given the mnemonic 'NV', i.e. never) should not be used. We recommend that you use the instruction 'MOV R0,R0' as a general purpose no-op.

This appendix lists the instructions and code sequences to be avoided. It is **strongly** recommended that you take the time to familiarise yourself with these cases because some will only fail under particular circumstances which may not arise during testing.

For more details on the ARM chip and its instruction set see the chapters *The* ARM CPU on page 29 and CPU *instruction set* on page 53, and the datasheets for the different versions of the ARM chip.

Restrictions to the ARM instruction set

There are three main reasons for restricting the use of certain parts of the instruction set:

- **Dangerous instructions**

  Such instructions can cause a program to fail unexpectedly, for example:

  ```
  LDM R15,Rlist
  ```

  uses PC+PSR as the base and so can cause an unexpected address exception.

- **Useless instructions**

  It is better to reserve the instruction space occupied by existing 'useless' instructions for instruction expansion in future processors. For example:

  ```
  MUL R15,Rm,Rs
  ```

  only serves to scramble the PSR.

  This category also includes ineffective instructions, such as a PC relative LDC/STC with writeback; the PC cannot be written back in these instructions, so the writeback bit is ineffective (and an attempt to use it should be flagged as an error).

- **Instructions with undesirable side-effects**

  It is hard to guarantee the side-effects of instructions across different processors. If, for example, the following is used:

  ```
  LDR Rd,[R15,#expression]!
  ```

  the PC writeback will produce different results on different types of processor.

Instructions and code sequences to avoid

The instructions and code sequences are split into a number of categories. Each category starts with an indication of which of the two main ARM variants (ARM2, ARM3) it applies to, and is followed by a recommendation or warning. The text then goes on to explain the conditions in more detail and to supply examples where appropriate.

Unless a program is being targeted **specifically** for a single version of the ARM processor family, all of these recommendations should be adhered to.

TSTP/TEQP/CMPP/CMNP: Changing mode

Applicability: ARM2

> When the processor's mode is changed by altering the mode bits in the PSR using a data processing operation, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7, R15) are safe.

The following instructions are affected, but note that mode changes can only be made when the processor is in a non-user mode:

```
TSTP Rn,Op2
TEQP Rn,Op2
CMPP  Rn,Op2
CMNP Rn,Op2
```

These are the only operations that change all the bits in the PSR (including the mode bits) without affecting the PC (thereby forcing a pipeline refill during which time the register bank select logic settles).

The following examples assume the processor starts in Supervisor mode:

a)
```
TEQP PC,#0
MOV  R0,R0
ADD  R0,R1,R13_usr
```
Safe: NOP added between mode change and access to a banked register (R13_usr)

b)
```
TEQP PC,#0
ADD  R0,R1,R2
```
Safe: No access made to a banked register

c)
```
TEQP PC,#0
ADD  R0,R1,R13_usr
```
Fails: Data **not** read from Register R13_usr!

The safest default is always to add a NOP (e.g. MOV R0,R0) after a mode changing instruction; this will guarantee correct operation regardless of the code sequence following it.

LDM/STM: Forcing transfer of the user bank (Part 1)

Applicability: ARM2, ARM3

Do not use writeback when forcing user bank transfer in LDM/STM.

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 is 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.

Similarly, in LDM instructions the S bit is redundant if R15 is not in the transfer list. In user mode programs, the S bit is ignored, but 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.

In both cases where the processor is in a non-user mode and transfer to or from the user bank is forced by setting the S bit, writeback of the base will also be to the user bank though the base will be fetched from the current bank. Therefore don't use writeback when forcing user bank transfer in LDM/STM.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

`STMxx Rn!,Rlist`	**Safe:** Storing non-user registers with write back to the non-user base register
`LDMxx Rn!,Rlist`	**Safe:** Loading non-user registers with write back to the non-user base register
`STMxx Rn,Rlist^`	**Safe:** Storing user registers, but no base write-back
`STMxx Rn!,Rlist^`	**Fails:** Base fetched from non-user register, but written back into user register
`LDMxx Rn!,Rlist^`	**Fails:** Base fetched from non-user register, but written back into user register

LDM: Forcing transfer of the user bank (Part 2)

Applicability: ARM2, ARM3

When loading user bank registers with an LDM in a non-user mode, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7,R15) are safe.

Because the register bank switches from user mode to non-user mode during the first cycle of the instruction following an `LDM Rn,Rlist^`, an attempt to access a banked register in that cycle may cause the wrong register to be accessed.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

```
LDM Rn Rlist^
ADD R0,R1,R2
```

Safe: Access to unbanked registers after LDM^

```
LDM Rn,Rlist^
MOV R0,R0
ADD R0,R1,R13_svc
```

Safe: NOP inserted before banked register used following an LDM^

```
LDM Rn,Rlist^
ADD R0,R1,R13_svc
```

Fails: Accessing a banked register immediately after an LDM^ returns the wrong data

```
ADR   R14_svc, saveblock
LDMIA R14_svc, {R0 - R14_usr}^
LDR   R14_svc, [R14_svc,#15*4]
MOVS  PC, R14_svc (R14_svc)
```

Fails: Banked base register used immediately after the LDM^

```
ADR   R14_svc, saveblock
LDMIA R14_svc, {R0 - R14_usr}^
MOV   R0,R0
LDR   R14_svc, [R14_svc,#15*4]
MOVS  PC, R14_svc
```

Safe: NOP inserted before banked register (R14_svc) used

Note: The ARM2 and ARM3 processors **usually** give the expected result, but cannot be guaranteed to do so under all circumstances, therefore this code sequence should be avoided in future.

SWI/Undefined Instruction trap interaction

Applicability: ARM2

Care must be taken when writing an undefined instruction handler to allow for an unexpected call from a SWI instruction. The erroneous SWI call should be intercepted and redirected to the software interrupt handler.

The implementation of the CDP instruction on ARM2 may cause – under certain circumstances – a Software Interrupt (SWI) to take the Undefined Instruction trap if the SWI was the next instruction after the CDP. For example:

```
SIN F0
SWI &11
```

Fails: ARM2 may take the undefined instruction trap instead of software interrupt trap.

All Undefined Instruction handler code should check the failed instruction to see if it is a SWI, and if so pass it over to the software interrupt handler by branching to the SWI hardware vector at address 8.

Undefined instruction/Prefetch abort trap interaction

Applicability: ARM2, ARM3

Care must be taken when writing the Prefetch abort trap handler to allow for an unexpected call due to an undefined instruction.

When an undefined instruction is fetched from the last word of a page, where the next page is absent from memory, the undefined instruction will cause the undefined instruction trap to be taken, and the following (aborted) instructions will cause a prefetch abort trap. One might expect the undefined instruction trap to be taken first, then the return to the succeeding code will cause the abort trap. In fact the prefetch abort has a higher priority than the undefined instruction trap, so the prefetch abort handler is entered before the undefined instruction trap, indicating a fault at the address of the undefined instruction (which is in a page which is actually present). A normal return from the prefetch abort handler (after loading the absent page) will cause the undefined instruction to execute and take the trap correctly. However the indicated page is already present, so the prefetch abort handler may simply return control, causing an infinite loop to be entered.

Therefore, the prefetch abort handler should check whether the indicated fault is in a page which is actually present, and if so it should suspect the above condition and pass control to the undefined instruction handler. This will restore the expected sequential nature of the execution sequence. A normal return from the undefined instruction handler will cause the next instruction to be fetched (which will abort), the prefetch abort handler will be re-entered (with an address pointing to the absent page), and execution can proceed normally.

Single instructions to avoid

Applicability: ARM2, ARM3

The following single instructions and code sequences should be avoided in writing any ARM code.

Any instruction that uses the 1111 condition code

Avoid using the condition code 1111 (which was given the mnemonic 'NV', i.e. never):

```
opcodeNV ...
```

i.e. any operation where «*cond*» = NV

By avoiding the use of the 'NV' condition code, 2^{28} instructions become free for future expansion.

Note: It is recommended that the instruction `MOV R0,R0` be used as a general purpose NOP.

Data processing

Avoid using R15 in the Rs position of a data processing instruction:

```
MOV|MVN«cond»«S» Rd,Rm,shiftname R15
CMP|CMN|TEQ|TST«cond»«P» Rn,Rm,shiftname R15
ADC|ADD|SBC...|EOR«cond»«S» Rd,Rn,shiftname R15
```

Shifting a register by an amount dependent upon the code position should be avoided.

Multiply and multiply-accumulate

Do not specify R15 as the destination register as only the PSR will be affected by the result of the operation:

```
MUL«cond»«S» R15,Rm,Rs
MLA«cond»«S» R15,Rm,Rs,Rn
```

Do not use the same register in the Rd and Rm positions, as the result of the operation will be incorrect:

```
MUL«cond»«S» Rd,Rd,Rs
MLA«cond»«S» Rd,Rd,Rs
```

Single data transfer

Do not use a PC relative load or store with base writeback as the effects may vary in future processors:

```
LDR|STR«cond»«B»«T»  Rd,[R15,#expression]!
LDR|STR«cond»«B»«T»  Rd,[R15,«-»Rm«,shift»]!

LDR|STR«cond»«B»«T»  Rd,[R15],#expression
LDR|STR«cond»«B»«T»  Rd,[R15],«-»Rm«,shift»
```

Note: It is safe to use pre-indexed PC relative loads and stores **without** base writeback.

Avoid using R15 as the register offset (R*m*) in single data transfers as the value used will be PC+PSR which can lead to address exceptions:

```
LDR|STR«cond»«B»«T»  Rd,[Rn,«-»R15«,shift»]«!»
LDR|STR«cond»«B»«T»  Rd,[Rn],«-»R15«,shift»
```

A byte load or store operation on R15 must not be specified, as R15 contains the PC, and should always be treated as a 32 bit quantity:

```
LDR|STR«cond»B«T»  R15,address
```

A post-indexed LDR|STR where R*m*=R*n* must not be used (this instruction is very difficult for the abort handler to unwind when late aborts are configured – which do not prevent base writeback):

```
LDR|STR«cond»«B»«T»  Rd,[Rn],«-»Rn«,shift»
```

Do not use the same register in the R*d* and R*m* positions of an LDR which specifies (or implies) base writeback; such an instruction is ambiguous, as it is not clear whether the end value in the register should be the loaded data or the updated base:

```
LDR«cond»«B»«T»  Rn,[Rn,#expression]!
LDR«cond»«B»«T»  Rn,[Rn,«-»Rm«,shift»]!

LDR«cond»«B»«T»  Rn,[Rn],#expression
LDR«cond»«B»«T»  Rn,[Rn],«-»Rm«,shift»
```

Block data transfer

Do not specify base writeback when forcing user mode block data transfer as the writeback may target the wrong register:

```
STM«cond»<FD|ED...|DB> Rn!,Rlist^
LDM«cond»<FD|ED...|DB> Rn!,Rlist^
```

where *Rlist* does not include R15.

Note: The instruction:

```
LDM«cond»<FD|ED...|DB> Rn!,<Rlist,R15>^
```

does **not** force user mode data transfer, and can be used safely.

Do not perform a PC relative block data transfer, as the PC+PSR is used to form the base address which can lead to address exceptions:

```
LDM|STM«cond»<FD|ED...|DB> R15«!»,Rlist«^»
```

Single data swap

Do not perform a PC relative swap as its behaviour may change in the future:

```
SWP«cond»«B» Rd,Rm,[R15]
```

Avoid specifying R15 as the source or destination register:

```
SWP«cond»«B» R15,Rm,[Rn]
SWP«cond»«B» Rd,R15,[Rn]
```

Coprocessor data transfers

When performing a PC relative coprocessor data transfer, writeback to R15 is prevented so the W bit should not be set:

```
LDC|STC«cond»«L» CP#,CRd,[R15]!
LDC|STC«cond»«L» CP#,CRd,[R15,#expression]!
LDC|STC«cond»«L» CP#,CRd,[R15]#expression!
```

Undefined instructions

ARM2 has two undefined instructions, and ARM3 has only one (the other ARM2 undefined instruction has been defined as the Single data swap operation).

Undefined instructions should not be used in programs, as they may be defined as a new operation in future ARM variants.

Register access after an in-line mode change

Care must be taken not to access a banked register (R8-R14) in the cycle following an in-line mode change. Thus the following code sequences should be avoided:

1 `TSTP | TEQP | CMPP | CMNP«cond» Rn,Op2`

2 any instruction that uses R8-R14 in its first cycle.

Register access after an LDM that forces user mode data transfer

The banked registers (R8-R14) should not be accessed in the cycle immediately after an LDM that forces user mode data transfer. Thus the following code sequence should be avoided:

1 `LDM«cond»<FD | ED... | DB> Rn,Rlist^`
where `Rlist` does **not** include R15

2 any instruction that uses R8-R14 in its first cycle.

Other points to note

This section highlights some obscure cases of ARM operation which should be borne in mind when writing code.

Use of R15

Applicability: ARM2, ARM3

Warning: When the PC is used as a destination, operand, base or shift register, different results will be obtained depending on the instruction and the exact usage of R15.

Full details of the value derived from or written into R15+PSR for each instruction class is given in the chapter CPU *instruction set* on page 53. Care must be taken when using R15 because small changes in the instruction can yield significantly different results. For example, consider data operations of the type:

```
    opcode«cond»«S» Rd,Rn,Rm
or  opcode«cond»«S» Rd,Rn,Rm,shiftname Rs
```

- When R15 is used in the R*m* position, it will give the value of the PC together with the PSR flags.
- When R15 is used in the R*n* or R*s* positions, it will give the value of the PC without the PSR flags (PSR bits replaced by zeros).

```
MOV R0,#0
ORR R1,R0,R15      ; R1:=PC+PSR    (bits 31:26,1:0 reflect PSR flags)
ORR R2,R15,R0      ; R2:=PC        (bits 31:26,1:0 set to zero)
```

Note: The relevant instruction description in the chapter CPU *instruction set* on page 53 should be consulted for full details of the behaviour of R15.

STM: Inclusion of the base in the register list

Applicability: ARM2, ARM3

Warning: In the case of a STM with writeback that includes the base register in the register list, the value of the base register stored depends upon its position in the register list.

During an STM, the first register is written out at the start of the second cycle of the instruction. When writeback is specified, the base is written back at the end of the second cycle. An 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, it will store the modified value.

For example:

```
MOV    R5,#&1000
STMIA  R5!,{R5-R6}     ; Stores value of R5=&1000

MOV    R5,#&1000
STMIA  R5!,{R4-R5}     ; Stores value of R5=&1008
```

MUL/MLA: Register restrictions

Applicability: ARM2, ARM3

Given	`MUL Rd,Rm,Rs`
or	`MLA Rd,Rm,Rs,Rn`
Then	*Rd* & R*m* must be different registers
	Rd must not be R15

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

The destination register (R*d*) 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 R*d*, R*n* and R*s* may use the same register when required.

LDM/STM: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDM or STM operation will not cause an address exception.

Only the address of the first transfer of a LDM or STM is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

The following examples assume the processor is in a non-user mode and MEMC is being accessed:

```
MOV   R0,#&04000000 ; R0=&04000000
STMIA R0,{R1-R2}    ; Address exception reported
                    :    (base address illegal)

MOV   R0,#&04000000
SUB   R0,R0,#4      ; R0=&03FFFFFC
STMIA R0,{R1-R2}    ; No address exception reported
                    :    (base address legal)
                    ; code will overwrite data at address &00000000
```

Note: The exact behaviour of the system depends upon the memory manager to which the processor is attached; in some cases, the wraparound may be detected and the instruction aborted.

LDC/STC: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDC or STC operation will not cause an address exception (affects LDF/STF).

The coprocessor data transfer operations act like STM and LDM with the processor generating the addresses and the coprocessor supplying/reading the data. As with LDM/STM, only the address of the first transfer of a LDC or STC is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

Note that the floating point LDF/STF instructions are forms of LDC and STC.

The following examples assume the processor is in a non-user mode and MEMC is being accessed:

```
MOV   R0,#&04000000     ; R0=&04000000
STC   CP1,CR0,[R0]      ; Address exception reported
                        ;    (base address illegal)

MOV   R0,#&04000000
SUB   R0,R0,#4          ; R0=&03FFFFFC
STFD  F0,[R0]           ; No address exception reported
                        ;    (base address legal)
                        ; code will overwrite data at address &00000000
```

Note: The exact behaviour of the system depends upon the memory manager to which the processor is attached; in some cases, the wraparound may be detected and the instruction aborted.

LDC: Data transfers to a coprocessor fetch more data than expected

Applicability: ARM3

Data to be transferred to a coprocessor with the LDC instruction should never be placed in the last word of an addressable chunk of memory, nor in the word of memory immediately preceding a read-sensitive memory location.

Due to the pipelining introduced into the ARM3 coprocessor interface, an LDC operation will cause one extra word of data to be fetched from the internal cache or external memory by ARM3 and then discarded; if the extra data is fetched from an area of external memory marked as cacheable, a whole line of data will be fetched and placed in the cache.

A particular case in point is that an LDC whose data ends at the last word of a memory page will load and then discard the first word (and hence the first cache line) of the next page. A minor effect of this is that it may occasionally cause an unnecessary page swap in a virtual memory system. The major effect of it is that (whether in a virtual memory system or not), the data for an LDC should never be placed in the last word of an addressable chunk of memory: the LDC will attempt to read the immediately following non-existent location and thus produce a memory fault.

The following example assumes the processor is in a non-user mode, FPU hardware is attached and MEMC is being accessed:

```
MOV  R13,#&03000000 ; R13=Address of I/O space
STFD F0,[R13,#-8]!  ; Store F.P. register 0 at top of physical memory
                    ;  (two words of data transferred)
LDFD F1,[R13],#8    ; Load F.P. register 1 from top of physical
                    ;  memory, but three words of data are
                    ;  transferred, and the third access will read
                    ;  from I/O space which may be read sensitive
```

Static ARM problems

The static ARM is a variant of the ARM processor designed for low power consumption, that is built using static CMOS technology. (The difference between it and the standard ARM is similar to that between SRAM and DRAM.)

The static ARM exhibits different behaviour to ARM2 and ARM3 when executing a PC relative LDR with base writeback. This class of instruction has very limited application, so the discrepancy should not be a problem, but if you wish to use any of the following instructions in your code you are advised to contact Acorn Computers.

```
LDR Rd,[PC,#expression]!
LDR Rd,[PC],#expression
LDR Rd,[PC,{-}Rm{,shift}]!
LDR Rd,[PC],{-}Rm{,shift}
```

Note: A PC relative LDR **without** writeback works exactly as expected.

Provided that this instruction class is unused, it is likely that writeback to the PC on LDR and STR will be disabled completely in the future. The fewer incidental ways there are to modify the PC the better.

Unexpected Static ARM2 behaviour when executing a PC relative LDR with writeback

The instructions affected are:-

- `LDR Rd,[PC,#expression]!`
- `LDR Rd,[PC],#expression`

Case 1: LDR R*d*,[PC,*#expression*]!

Expected result:
R*d* ← (PC+8+expression)
PC ← PC+8+expression
...so execution continues from PC+8+expression

Actual ARM2 result:
R*d* ← R*d* (no change)
PC ← PC+8+expression+4
...so execution continues from PC+12+expression

Case 2: LDR R*d*,[PC],*#expression*

Expected result:
R*d* ← (PC+8)
PC ← PC+8+expression
...so execution continues from PC+8+expression

Actual ARM2 result:
R*d* ← R*d* (no change)
PC ← PC+8+expression+4
...so execution continues from PC+12+expression

Appendix E: Support for AAsm source

AAsm was an alternative variant of the assembler supplied with previous releases of this product. It has been removed from this product, but to ease porting source code written for AAsm, some limited support has been added to ObjAsm. **This support for AAsm may be removed in future releases of Acorn Assembler.**

To enable this support you must pass the new **-ABSolute** option to ObjAsm. There is no option on the Setup menu directly corresponding to this option; the best way to pass the option from the desktop is to include it in the Setup menu's **Others** option (see *Specifying other command line options* on page 18).

The -ABSolute option

The new **-ABSolute** option makes ObjAsm accept AAsm source code. This option is provided to simplify the use of code originally developed using AAsm. Unlike AAsm, the output format produced is AOF, as for any ordinary assembly operation, and this must be linked by the linker as usual, in order to create an absolute image. However, the contents of the AOF file will be marked as having an absolute address (if either the **ORG** or **LEADR** directive is used), and the linker, given suitable options, can produce an image file equivalent to that previously generated directly by AAsm. The following changes to normal ObjAsm input syntax apply:

- There is an implicit **AREA** declaration before the start of the source. The normal rule that there must be an **AREA** directive in the source before use of any instruction or data generating statements does not apply. The implicitly declared area is called **ABS$$BLOCK**, and has the new **ABS** attribute (see *Area attributes* on page 48) implying that it must be loaded at a fixed absolute base address.
- The directive **LEADR** is accepted. (Previously only AAsm implemented this; ObjAsm did not.)
- The **ORG** directive, if used within the source file, will apply to the implicitly declared current area.
- The following directives are not recognised (since they were not available with AAsm), and may be used for any other purpose, in particular as macro names: **AREA**, **IMPORT**, **EXPORT**, **STRONG**, **ENTRY**, **KEEP**, **AOF**, **AOUT**.

  This change is important, since ObjAsm recognises directives before it does macro names.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z

Reader's Comment Form

Acorn Assembler, Issue 1
0484,233

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
Acorn House, Vision Park
Histon, Cambridge CB4 4AE
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