
ARM assembler

reference manual

ARM Evaluation System

Acorn OEM Products



ARM assembler

Part No 0448,008
Issue No 1.0
4 August 1986

© Copyright Acorn Computers Limited 1986

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act, or for the purpose of review, or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

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

- The manual is provided on an 'as is' basis except for warranties described in the software licence agreement if provided.
- The software and this manual are protected by Trade secret and Copyright laws.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

There are no warranties implied or expressed including but not limited to implied warranties or merchantability or fitness for purpose and all such warranties are expressly and specifically disclaimed.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation, unless notified otherwise, should be notified in writing to the following address:

Customer Services Department,
Acorn Computers Limited,
645 Newmarket Road,
Cambridge
CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers. Acorn Computers can accept no liability whatsoever for any loss, indirect or consequential damages, even if Acorn has been advised of the possibility of such damage or even if caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Econet® and The Tube® are registered trademarks of Acorn Computers Limited.

ISBN 1 85250 003

Published by:

Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, UK

Contents

1. Introduction	1
1.1 Description of the ARM CPU	1
1.2 The assembler	3
1.3 Installing AAsm	4
1.4 Conventions used in this manual	4
2. CPU instruction set	6
2.1 Addressing modes	6
2.1.1 Relative addressing	6
2.1.2 Indexed addressing	6
2.1.3 Register addressing	6
2.1.4 Implied addressing	7
2.1.5 Immediate addressing	7
2.2 Types of indexing	7
2.2.1 Pre-indexing	7
2.2.2 Post-indexing	8
2.2.3 Write back on pre-indexed instructions	8
2.3 Condition testing	9
2.4 Branch instructions	10
2.4.1 Branch	10
2.4.2 Branch with link	11
2.5 The barrel shifter	12
2.5.1 The shift types	13
2.5.2 Logical shift left	13
2.5.3 Logical shift right	14
2.5.4 Arithmetic shift right	14
2.5.5 Rotate right	14
2.5.6 Rotate right with extend	15
2.6 Data processing	15
2.6.1 Instruction syntax	16
2.6.2 Data processing instruction summary	21
2.6.3 The ADR instruction	21
2.7 Single data transfer	22
2.7.1 Instruction syntax	22
2.8 Block data transfer	24
2.8.1 Instruction syntax	25
2.8.2 Stacking	27

2.8.3 Special points	31
2.9 Supervisor calls	33
2.9.1 Instruction syntax	33
3. The assembler	34
3.1 Symbols	35
3.2 Expressions	36
3.3 Numeric constants	37
3.4 The number equating directive *	38
3.5 The register equating directive RN	38
3.6 Assembler operators	39
3.6.1 The arithmetic operators	39
3.6.2 Boolean logical operators	40
3.6.3 Bitwise logical operators	40
3.6.4 Shift operators	40
3.6.5 Relational operators	41
3.6.6 String operators	41
3.6.7 Operator summary	43
3.7 Store-loading directives	44
3.7.1 Syntax differences	44
3.8 The ALIGN directive	45
3.9 ?label	45
3.10 Literals	46
3.11 Laying out areas of memory	46
3.11.1 Counter values	48
3.12 Variables	48
3.12.1 Global variables	48
3.12.2 Other useful variables	50
3.13 Local labels	50
3.14 Error handling directives	53
3.15 The ORG and LEADR directives	54
3.16 The END directive	54
4. Conditional assembly,	55
4.1 Conditional assembly	55
4.2 Repetitive assembly	57
4.3 Evaluating logical expressions	58
4.4 Macros	59
4.4.1 Local variables	61
4.4.2 The MEXIT directive	61
4.4.3 Default values	62
4.4.4 The macro substitution method	62

4.4.5 Nesting macros	63
5. Assembling, link files, printing	64
5.1 The command line	64
5.2 Assembling a program	66
5.3 Linking source files	67
5.4 The object file	68
5.5 The SYMBOL command	68
5.6 The XREF command	69
5.7 The WARNING command	69
5.8 The QUIT command	69
5.9 Assembler print commands	69
5.9.1 WIDTH n	70
5.9.2 LENGTH n	70
5.9.3 TERSE	70
5.9.4 Dynamic print options	71
5.9.5 TTL	73
5.9.6 SUBTTL	73
6. Appendix A	74
6.1 ARM instruction set	74
7. Appendix B	78
7.1 AAsm and ObjAsm error messages	78
8. Appendix C	85
8.1 Example AAsm file	85
9. Appendix D	87
9.1 ObjAsm directives	87
9.1.1 AREA	87
9.1.2 IMPORT	88
9.1.3 EXPORT	88
9.1.4 ENTRY	88
9.1.5 KEEP	89
9.1.6 DCD	89
9.1.7 Literals	89
9.1.8 Branch destinations	89
9.1.9 ObjAsm error messages	90
10. Appendix E	91
Modes and registers	91
10.1 Mode 0:	91
10.2 Mode 1:	92
10.3 Mode 2:	92
10.4 Mode 3:	92

10.5 Changing modes.	93
11. Appendix F	94
Source code examples	94
11.1 Using the conditional instructions	94
11.2 Pseudo-random binary sequence generator	96
11.3 Multiplication by a constant	96
11.4 Loading a word from an unknown alignment	99
11.5 Sign/zero extension of a half word	99
11.6 Return setting condition codes	100



1. Introduction

This document is a reference guide to the assembler for the ACORN RISC Machine (ARM). It is assumed that the reader is familiar with other relevant ARM documentation:

- *ARM system user guide*
- *ARM hardware reference manual*
- *ARM software reference manual*
- *TWIN reference manual*

1.1 Description of the ARM CPU

The ARM is a 32-bit single chip microprocessor which has a reduced instruction set architecture. There are five classes of instruction:

- (1) Branches
- (2) Data operations between registers
- (3) Single register data transfers
- (4) Multiple register data transfers
- (5) Supervisor calls

The ARM has a 32-bit data bus and a 26-bit address bus. An instruction pipeline is used to hold consecutive instructions and fetch, decode and execute phases of instructions occur in parallel. All instructions are designed to fit into one 32-bit word and all instructions have been made conditional.

The processor can access two types of data: bytes (8 bits) and words (32 bits). The program counter PC is 24 bits wide and counts to &FFFFFFF. 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 &3FFFFFFC. The memory capacity of the ARM system is 64 Mbytes, or 16 Mwords.

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

The ARM normally operates in a mode of operation called user mode, and in this environment the programmer sees a bank of sixteen 32-bit registers, R0 to R15. Nine other registers exist and they are used when the ARM is in Interrupt Mode, Fast Interrupt Mode, or Supervisor Mode. A full explanation of the ARM interrupt capability and of its four modes of operation is given in the *ARM Software Reference Manual*, but the register map and an explanation of the ARM Modes is reproduced in Appendix E of this guide.

Of the sixteen registers 0-15, only R14 and R15 are regarded as having specific purpose. R15 contains the Program Counter (PC), and the Processor Status Register (PSR), and R14 is the subroutine link register, which receives a suitably modified copy of R15 on a branch with link instruction. Special bits in the processor's instructions allow the PC and PSR to be treated together or separately. The PSR contains the flag bits N, Z, C and V. These are the Negative flag, the Zero flag, the Carry flag and the oVerflow flag respectively. The *CPU software manual* contains information on the PSR and flags.

1.2 The assembler

The assembler has the following features:

- Full support of the ARM instruction set
- Global and local label capability
- Powerful macro capability
- Comprehensive expression handling
- Conditional assembly
- Repetitive assembly
- Comprehensive symbol table printouts
- Link-file capability
- Pseudo-opcodes to control printout

The ARM assembler AAsm produces object files which can be immediately executed using the **objectfilename* command.

A variant of AAsm, ObjAsm, creates files which can be used by the ARM linker. The purpose of the linker is to take programs which have been written in several portions, resolve all unknown references, and create a single image which can be run. The program parts may be written in a mixture of assembler and compiled languages, and the linker deals with the separate results of all compilations and performs any necessary cross-referencing. ObjAsm object files cannot be executed directly: they must be handled by the linker. Details of ObjAsm are given in appendix D of this guide.

1.3 Installing AAsm

AAsm is supplied on a 5.25 inch floppy disc in Acorn ADFS format. It can be loaded either directly from the ARM A* prompt by typing `aasm` or run as a task under the TWIN editor.

1.4 Conventions used in this manual

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

```
!"#$%&^@  
([{}])!:. , ;  
+ - / * = < > ? _
```

This often makes it difficult for the user to determine precisely which characters on the printed page are explanatory or descriptive, and which (if any) are the ones which AAsm will accept as having the correct syntax. A typewriter-style typeface has been used to indicate both text which appears on the screen and text which can be typed on the keyboard (for example, AAsm source code). This is so that the position of relevant spaces is clearly indicated.

The syntax of AAsm instructions is shown in meta-language form, using an italic typeface for variable items (for example, *filename*, *register*) and a non-italic typeface for fixed items (for example, `ALIGN`, `RRX`). Both general and specific examples of syntax and screen output is given – 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.

Curly brackets { } enclose optional items in the syntax. For example, AAsm accepts a three field source line which may be expressed in the form:

```
{label} {instruction}{;comment}
```

Note that there is a compulsory space between the first two fields.

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

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

The {label} is L321 , the {instruction} portion is ADD Ra,Ra,Ra,LSL #1 and the {;comment} is ;multiply by 3 . (Chapter 2 explains the ARM instruction set and there the instruction field is explained in more detail.)

In actual program examples, curly brackets have a special meaning and do not indicate an optional item.

Function keys (such as f1) and control keys (such as tab) often need to be pressed by themselves or in combination with the shift and ctrl keys. To indicate this, these keys are printed in boxes. This maintains consistency with the *TWIN reference manual*. For example:

Press the RETURN key

Press the ESCAPE key

Press the DELETE key

Press the COPY key

2. CPU instruction set

2.1 Addressing modes

ARM instructions operate on data stored in 32-bit registers and external memory. Addressing refers to the method whereby the address of this data is generated in each instruction. The ARM has three memory addressing modes: program-relative, base-relative (indexed addressing) and implied. However, other modes can be synthesised and there is little difference in speed between the various modes.

2.1.1 Relative addressing

Relative addressing uses a group of bytes within the instruction to specify a displacement from the address of the current instruction to which a program branch is to occur. The programmer supplies the target address, from which a 24-bit displacement value is calculated by the assembler. The offset values permitted are sufficient to allow the entire memory map to be addressed. For example:

```
B LABEL
```

2.1.2 Indexed addressing

This mode of addressing uses a displacement or index which is added to a base register to form a pointer into memory. Any ARM register can be designated as the base register and, being a 32-bit register, can point to any address in the memory map. For example:

```
LDR R0, [R8, #12]
```

The index can be an immediate value, 12 bits in length, or the contents of a 32-bit register (which has possibly passed through the barrel shifter). All bits are taken as the index, with a completely separate bit determining whether the index is added to or subtracted from the value in the base register.

2.1.3 Register addressing

Register to register operations are involved in this type of addressing, with the source and destination registers being specified by bit patterns within the instruction. For example:

```
ADD R0, R0, R1
```

2.1.4 Implied addressing

This is a special case where the instruction automatically generates an address and branches to it. For example:

```
SWI 1
```

2.1.5 Immediate addressing

In this type of addressing, the actual operand is contained in the same word as the instruction. The operand is an 8-bit quantity rotated right by an even amount. For example:

```
MOV R0, #8
```

Examples of other valid immediate constants are:

```
#1
```

```
#&FF
```

```
#&3FC ;This is &FF rotated right by 30
```

```
#&8000000 ;This is 2 rotated right by 2
```

```
#&FC000003 ;This is &FF rotated right by 6
```

Examples of invalid constants are `#&101`, which cannot be obtained by rotating an 8-bit value, and `#&1FE`, which is an 8-bit value rotated by an odd amount but not an 8-bit value rotated by an even amount.

Further details of the operation of the barrel shifter are given in section 2.5.

2.2 Types of indexing

2.2.1 Pre-indexing

In a pre-indexed addressing instruction, the CPU modifies the base address by the index before the function of the instruction is performed. The AAsm syntax for this is `[Rn, offset]` and the calculation within the square brackets is performed first to establish the target address, the offset being either added to or subtracted from the value held in register Rn.

2.2.2 Post-indexing

This is a variant of indexed addressing in which the value held in R_n is used as the target address and R_n is modified by the index after the function of the instruction is performed. In this case the syntax is $[R_n], offset$ and the operation is known as post-indexing. It follows that for post-indexing to have any value whatsoever, the value generated by $[R_n], offset$ must be written back into R_n so that it is available for the following instruction, which may well be another post-indexed instruction. Post-indexing therefore has automatic write back. If the base address is to be preserved, it must be deliberately saved.

2.2.3 Write back on pre-indexed instructions

Write back does not occur implicitly in pre-indexed instructions, but it can be requested by adding an exclamation mark (!) to the assembler syntax. The base address is, of course, lost when the value in R_n is modified in this manner. For example: `STR R1, [R0, #4]!`

Pre-indexing and post-indexing work in conjunction with write back to form the basis of a set of powerful multiple move and stacking operations. These are explained later.

2.3 Condition testing

Every instruction in the ARM repertoire is conditional. The default condition is 'always' but any other condition can be requested by adding the appropriate two character condition mnemonic to the standard form. Because branches which are taken cause breaks in the pipeline they often waste time needlessly, when a suitable conditional instruction sequence would be better.

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 R5,#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 ARM instruction is performed. If it is false, the instruction is skipped and the PC is advanced to the next memory word; this takes one 'S-cycle' of processor time – the first example takes at least twice as long as the second example. (An explanation of S-cycles and other ARM timing details can be found in the *ARM Hardware Reference Manual*.)

The ARM has the ability to test for 16 conditions. These are grouped in pairs of opposites.

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

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

After the instruction is obeyed, the 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.

2.4 Branch instructions

The Branch instruction takes a 26-bit word offset, allowing forward jumps of up to +0x2000004 and backward jumps of up to -0x1FFFFFF8 to be made. This is sufficient to address the entire memory map, as the calculation 'wraps round' between the top and bottom of memory. The programmer should provide a label from which the assembler will calculate a 26-bit offset.

2.4.1 Branch

The instruction syntax is: `B{condition} programrelativeexpression`

For example: `B LABEL ;branch to LABEL`

`BNE LABEL1 ;if not equal goto LABEL1`

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

The branch offset must take account of the prefetch operation, which causes the PC to be two words ahead of the current instruction. The ARM assembler handles this automatically. 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
xxxxxxxxxx		xxx	
xxxxxxxxxx	L2	xxx	

2.4.2 Branch with link

The instruction syntax is: *BL{condition} programrelativeexpression*

Whenever branch with link is specified, 4 is subtracted from the contents of R15 (including the PSR) and the result is written to R14. Thus the value written into the link register is the address of the instruction following the branch and link instruction. Therefore after branching to a subroutine, the program flow can return to the memory address immediately following the branch instruction by writing back the R14 value into R15. Subroutines can be called by a BL instruction. The subroutine should end with a *MOV PC,R14* if the link register has not been saved on a stack or *LDMxx Rn, {PC}* if the link register has been saved on a stack addressed by Rn.

These methods of returning do not restore the original PSR. If the PSR does need to be restored, *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. The last in particular may interfere unintentionally with the interrupt system.