ARM assembler

reference manual

ARM Evaluation System

Acorn OEM Products

ARM assembler

Part No 0448,008
Issue No 1.0
4 August 1986

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

ISBN 1 85250 003

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

Contents

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 &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 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 [RETURN]

Press the ESCAPE key [ESCAPE]

Press the DELETE key [DELETE]

Press the COPY key [COPY]

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
#&80000000 ;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 Rn is used as the target address and Rn is modified by the index after the function of the instruction is performed. In this case the syntax is [*Rn*] , *offset* and the operation is known as post-indexing. It follows that for post-indexing to have any value whatsoever, the value generated by [*Rn*] , *offset* must be written back into Rn 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 Rn 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 +&2000004 and backward jumps of up to –&1FFFFF8 to be made. This is sufficient to address the entire memory map, as the calculation 'wraps round' between the top and bottom of memory. The programmer should provide a label from which the assembler will calculate a 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
   xxxxxxxx            xxx
   xxxxxxxx      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.

2.5 The barrel shifter

The ALU has a 32-bit barrel shifter capable of various shift and rotate operations. Data involved in the data processing group of instructions (detailed in section 2.6) may pass through the barrel shifter, either as a direct consequence of the programmer's actions, or in other cases, as a result of the internal computations of the assembler. The barrel shifter also affects the index for the single data transfer instructions (detailed in section 2.7). Because of the importance played by the barrel shifter, its operations are described prior to the formal introduction of the opcodes that use it.

The shift mechanism can produce the following types of operand:

(1) An unshifted register.

Syntax: *register*
For example: `R0`

(2) 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`

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

Syntax: *register*`,RRX`
For example: `R0,RRX`

(4) A register shifted by n bits, where n is the least significant byte of a register.

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

(5) A constant constructed by rotating an 8-bit constant right by n*2 bits, where n is supplied as a 4-bit constant. The shift type is always rotate right.

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

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

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

(6) A constant constructed as in (5), but specified explicitly.

Syntax: `#`*constant*`,`*shift 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.

Note that only forms (1), (2) and (3) are valid for index values in single register transfers.

2.5.1 The shift types

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

The mnemonic `ASL` may be freely interchanged with `LSL`.

2.5.2 Logical shift left

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

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

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

2.5.3 Logical shift right

```
0 - -> bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb - -> C
```

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

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

2.5.4 Arithmetic shift right

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

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

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

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

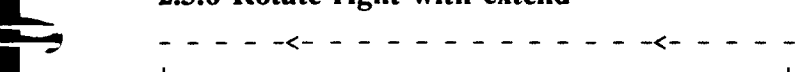
2.5.5 Rotate right

```
- - - - -<- - - - - - - - - - - -<- - - - -
|                                           |
- -> bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb - - ->C
```

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

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

2.5.6 Rotate right with extend



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

2.6 Data processing

There are sixteen data processing instructions.

ADC	ADd with Carry
ADD	ADD
AND	bitwise AND
BIC	BIt Clear
CMN	CoMpare Negated
CMP	CoMPare
EOR	bitwise Exclusive OR
MOV	MOVe
MVN	MoVe Not
ORR	bitwise OR
RSB	Reverse SuBtract
RSC	Reverse Subtract with Carry
SBC	SuBtract with Carry
SUB	SUBtract
TEQ	Test EQuivalence
TST	TeST and mask

Except in the cases of MOV and MVN, the operation is performed between a source register Rn and an operand. In the cases of MOV and MVN, only an operand is needed. The source register can be any one of the 16 registers, and the operand can be any operand that the barrel shifter can produce (see section 2.5 for details). Note that any shifting is done before the operation is performed. Some instructions use the bit held in the ALU's carry flag and add it into the operation. The result of the operation is placed in the destination register, which may be any one of the 16 registers.

The result of the operation affects the N and Z flags, and may also affect the C and V flags. However, the ALU doesn't copy the contents of its flags to the relevant parts of the Processor Status Register (PSR) unless the S bit is set. In the case of the four instructions `CMN`, `CMP`, `TEQ` and `TST` the assembler always sets the S bit since these instructions would be meaningless if their results were not copied to the PSR. In the case of the remaining 12 instructions, the programmer may request that the ALU flags are copied to the PSR, by including the letter `S` in the source line. This forces the PSR update.

For example: `ADDS R2,R0,R1`

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

2.6.1 Instruction syntax

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

(1) `MOV` and `MVN`

opcode{condition}{S} destination, operand

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

{S} is optional: it sets the S bit in the instruction. If `S` is specified, N and Z are set according to the value placed in the destination register, and C is set to the last bit shifted out by the barrel shifter, or is unchanged if no shifting took place. V is unchanged.

destination must be a register.

operand may be any of the operands that the barrel shifter can produce.

`MOV` causes the operand to be placed unchanged in the destination register.

`MVN` causes the operand to be evaluated and its bitwise inverse to be placed in the destination register.

For example: `MOV R0,R1,LSL#2`

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

For example: `MVN R2,R3`

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

(2) `ADD, ADC, SUB, SBC, RSB, RSC, AND, BIC, ORR, EOR`

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

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

{S} is optional. It sets the S bit in the instruction. If `S` is specified, then N and Z are set according to the value placed in the destination register. For `ADC`, `ADD`, `RSB`, `RSC`, `SBC` and `SUB`, C and V are set according to the result of the arithmetic operation. For `AND`, `BIC`, `EOR` and `ORR`, V is left unchanged and C is set to the last bit shifted out by the barrel shifter, or is unchanged if no shifting took place.

destination must be a register.

operand1 must be a register.

operand2 may be any of the operands that the barrel shifter can produce.

`ADD` addition is performed on *operand1* and *operand2*. The result is stored in the destination register.

`ADC` addition is performed on *operand1* and *operand2* and the carry flag. The result is stored in the destination register. This instruction can be used to implement multi-word additions.

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

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

For example:

```
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)
```

The result is stored in the destination register.

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

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

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

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

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

EOR a bitwise Exclusive OR is performed on *operand1* and *operand2*. The result is stored in the destination register. For example:

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

(3) CMN, CMP, TEQ, TST

opcode{condition}{P} operand1,operand2

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

operand1 is a register.

operand2 may be a register or an expression.

There is no need to specify S because for these instructions the assembler will ensure the necessary flags are set. S may be specified in the syntax and it will be accepted provided that P has not also been specified. For example, CMPSP and CMPPS will not be accepted.

CMP *operand2* is subtracted from *operand1*. Flags N, Z, C and V are altered.

CMN *operand2* is added to *operand1*. This compare allows a negative data field to be created for a compare. Flags N, Z, C and V are altered.

TEQ a bitwise exclusive OR is performed between *operand1* and *operand2*.

TST a bitwise AND operation is performed between *operand1* and *operand2*.

In the case of TEQ and TST the N and V flags are altered according to the result, V is unchanged and C is set to the last bit shifted out by the barrel shifter, or is unchanged if no shifting took place. For example:

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

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

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

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

Note the treatment of R15 as the first operand, described in note (2) below.

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

(1) If R15 is the destination register: 24 bits move to R15 if the S bit is not set. These bits become the new program counter (PC). In user mode, 28 bits are moved to R15 if the S bit is set; these are the 24 PC bits and the N, Z, C and V flags. In other modes all 32 bits are moved to R15 if the S bit is set.

(2) If R15 is the first operand in a two operand instruction: R15 is presented to the arithmetic logic unit (ALU) with the PSR bits set to zero.

(3) If the second or only operand is R15 (possibly shifted): R15 is presented to the barrel shifter or ALU with the PSR bits unchanged.

(4) R15 is the rotation register: R15 is presented to the barrel shifter with the PSR bits set to zero.

In the case of an instruction such as `MOV R0,#VALUE`, the assembler will evaluate the expression, and produce a CPU instruction to load the value into the destination register. This may not in fact be the machine level instruction known as `MOV`, but the programmer need not be aware that an alternative instruction has been substituted. An example is `MOV Rn,#-1` which the CPU cannot handle directly. The assembler will accept this syntax, but will convert it and generate object code for `MVN Rn,#0` which results in `Rn` containing –1. Such interconversion also takes place on the `BIC/AND`, `ADD/SUB`, `ADC/SBC` and `CMP/CMN` pairs of instructions.

2.6.2 Data processing instruction summary

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

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

2.6.3 The ADR instruction

Syntax:

ADR *register,expression*

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

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

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

If the expression is program-relative, an ADD *register*,PC,*#constant* or SUB *register*,PC,*#constant* instruction will be produced.

If the expression is numeric, a MOV *register,# constant* or MVN *register,#constant* will be produced.

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

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

2.7 Single data transfer

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

For register to register transfers see the data processing section, and the MOV instruction in particular.

2.7.1 Instruction syntax

Pre-indexed instruction (possibly with write back)

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

Post-indexed instruction

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

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

{cond} may be any of the two-character condition mnemonics listed in section 2.3. If omitted, AL is assumed.

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

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

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

(a) *#{immediate value}* the immediate value must lie in the range −4095 to 4095.

(b) *{-}{index register}{,shift}* the shift may be omitted, in which case no shifting is assumed. The allowed shift types are those listed under (1), (2) and (3) in section 2.5. Register controlled shifts are not allowed. The minus, if specified, means that the index value is to be subtracted.

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

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

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

{!} if present, write back will be done. *register* will assume the value of *register+index*, or *register-index*, as appropriate.

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

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

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

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

LDR R1,[BASE,#17]    ;load R1 from contents of
                     ;BASE+17
                     ;Don't write back

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

2.8 Block data transfer

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

2.8.1 Instruction syntax

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

opcode may be `STM` or `LDM`.

{cond} may be any of the two-character conditional mnemonics listed in section 2.3. If omitted, `AL` is assumed.

type is a two-character mnemonic indicating one of eight instruction types. It may not be omitted. The types are `FD`, `ED`, `FA`, `EA`, `IA`, `IB`, `DA` and `DB` and their description differs depending on whether they are appended to `STM` or `LDM`:

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

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

base, which may be any register, is the base register. It must be specified.

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

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

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

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

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

For `LDM`: if R15 is in the list of registers, only the 24 PC bits are normally loaded. Coding ^ causes the N, Z, C and V flags to be loaded as well as the PC in user mode, or all 32 bits to be loaded in other modes. Thus, return from interrupt or return from `SWI` using `LDM` will normally have the ^ coded. For example:

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

which may also be written:

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

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

which may also be written:

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

Provided that the contents of Rn and of the stack have not been corrupted by another instruction, the LDMIA instruction will reverse the effect of the STMIA instruction and recover the contents of the four registers from memory.

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

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

To recover the register contents would now require:

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

2.8.2 Stacking

Push to stack

Various forms of STM and LDM may be used to save the ARM registers on a stack. The opcodes generated for the various styles of stacking and unstackings are no different from those of the STMDB/DA/IB/IA and LDMDB/DA/IB/IA instructions, but the syntax is different.

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

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

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

Worked examples of STMEA and STMFD will now be given.

(1) STMEA

For example:

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

which may also be written:

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

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

```
     |      |
     |      |
     |      |
Rn->|      |<-which is empty
     | V3   |
     | V2   |
     | V1   | stack base
      - - -
```

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

```
     |      |<-Rn
     | R8   |
     | R7   |
     | R6   |
     | R3   |
     | V3   |
     | V2   |
     | V1   | stack base
      - - -
```

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

(2) STMFD

For example:

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

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

Pop from stack

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

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

Worked examples of LDMEA and LDMFD will now be given.

(1) LDMEA

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

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

```
|      |<-Rn
| V7   |
| V6   |
| V5   |
| V4   |
| V3   |
| V2   |
| V1   | stack base
 - - -
```

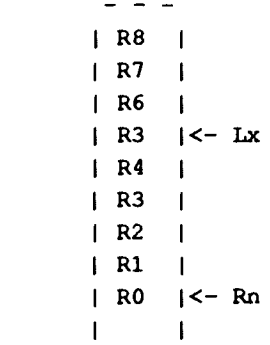
The following transfer would take place:

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

(2) LDMFD

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

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



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

2.8.3 Special points

(1) When the base register is in the list of registers.

The base register may be pushed on to the stack 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:

write lowest-numbered register to memory.
perform the write back.
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 popped from the stack, 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.

(2) When R15, the PC register, is in the list of registers.

When R15 is pushed on to the stack, the PSR is saved as well.

When R15 is popped from the stack, the PSR is only included if the symbol ^ is coded following the register list. The part of the PSR included will in any case only be that which may be modified in the currently selected ARM mode. For example:

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

(3) When the base register is R15.

When the PC is used as the base register, the PSR bits form part of the 32-bit address. Unless all flags are zero and the interrupts enabled, an address greater than &3FFFFFF will be formed. This is an address exception, and will cause the address exception trap in the ARM to be entered.

Write back is switched off when the PC is the base register.

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

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

do not swap the contents of R1 and R2.

(5) In order to force the saving of the user mode registers when executing in a different mode, ^ should be coded following the register list. For example:

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

(6) The operation of placing or removing registers to or from the stack starting with the lowest-numbered register is independent of stack type and exists to ensure that if a data abort occurs during the instruction, the PC is preserved.

2.9 Supervisor calls

These instructions are used extensively in the ARM Development System to communicate with I/O devices.

2.9.1 Instruction syntax

The syntax is: `SWI` *expression*

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

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

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

In the development system, this will send the message "Hello world" to the output terminal.

The significance of the assembler directive `ALIGN` is explained in chapter 3.

3. The assembler

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

{label} {instruction}{; comment}

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

{label} is optional; if present, it defines a symbol which is set equal to the address of the instruction assembled. If *label* is absent, then the address used is the current value of the instruction location pointer. This may not be word-aligned, for example, when the last directive was one of the store-loading directives detailed in section 3.7. However, using a label with an opcode instruction ensures that the address generated is word-aligned.

{instruction} is optional; if present, it defines the instruction to be assembled. See chapter 2 for the syntax of instructions.

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

Two special cases of this syntax, both of which are ignored by the assembler, are worth noting:

(1) A completely blank line is valid, and may be used to make the text more readable.

(2) A line may start with a semi-colon, the entire line being comment.

AAsm will treat a tab character (09) in a source file as a space (20), and will accept both line feeds (0A) and carriage returns (0D) as line terminators. The combinations 0A0D and 0D0A are treated as a single line terminator.

3.1 Symbols

A symbol is a group of alphanumeric characters which represents a number, logical value or string value. A label is a special type of symbol: the number it represents is not always immediately known to either the programmer or the assembler, but is generated as the assembly takes place. Other symbols are assigned values immediately by assembler directives.

The assembler recognises a label by its position on the line. The first character of the label occupies the first column of the line.

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

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

`|Interpret.P$Interpret$|` is one such symbol or label.

3.2 Expressions

Expressions are combinations of simple values, unary and binary operators, and brackets. The order of evaluation of expressions is determined firstly by bracketing, secondly by precedence rules, and in the absence of either, is from left to right. Specifically, any bracketed sub-expression will always be evaluated before being used as an operand to an operator, and any operand with a binary operator on both sides will always be used as an operand to the higher precedence operator, or if they have equal precedence, to the left hand operator.

Every simple value has a type associated with it, as does every operand produced at any stage of the expression evaluation, including the expression result. The types of operand are: numeric values, string values, logical values, program-relative values and register-relative values. For an expression to be syntactically correct, every operator must be provided with operands of the correct types.

Numeric values are unsigned integers in the range 0 to &FFFFFFFF. Overflow is ignored when doing calculations with numeric values (-1 evaluates to &FFFFFFFF, for instance). Comparisons are always unsigned comparisons, which may have counter-intuitive results in some cases (for example, -1>1 evaluates to 'true'). In a few places this manual contains such statements as 'The immediate value must lie in the range -4095 to 4095'. The values are presented this way for clarity, but the accurate interpretation of this example is 'The immediate value must lie in the range 0 to &FFF, or &FFFFF001 to &FFFFFFFF'.

String values are strings of from 0 to 256 bytes, each of which may take any value in the range 0 to 255; the byte values are usually ASCII printable characters in practice. The assembler will convert a string of length 1 into a numeric value if necessary, as described below in section 3.3.

Logical values are 'true' and 'false'; these constants can be input to the assembler as `{TRUE}` and `{FALSE}`.

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

Register-relative values are offsets from a base register, so in particular the difference of two such values is a numeric value. Simple register-relative values can be defined using the two operand form of the ^ directive and the # directive (see section 3.11). Internally within the evaluation of an expression a register-relative value may acquire a base of a signed sum of registers, but by the time the evaluation of the expression is complete this must have collapsed to either a numeric value or an offset from a single register. Register-relative values for which the base register is the PC are always converted into program-relative values.

3.3 Numeric constants

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

```
123456  ;  decimal constants
&A1F40  ;  hexadecimal constants
n_xxxx  ;  number in the form base n
           e.g. 2_101 is binary 101
           n may be between 2 and 9
```

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

3.4 The number equating directive *

Numeric values are assigned to symbols by the * directive or, alternatively, by the EQU directive. Program-relative values can also be assigned by * or EQU.

Syntax:

label * *numeric or program-relative-expression*

For example:

```
LINEFEED  *    &0A                    ;equate LINEFEED as &0A
MASK      EQU  &FF00FFF               ;create a mask
FRAMESIZE *    4*((framebase+3)/4);calculate FRAMESIZE
                                      ;from framebase
LABEL     SWI  16
LABEL2    *    LABEL-4
```

3.5 The register equating directive RN

The directive RN is used to assign a register number 0-15 to a symbol.

Syntax:

label RN *numeric expression*

For example:

```
Reg2        RN   2
TempStore   RN   3
SL          RN   4
sl          RN   SL
```

A register name is taken to be a constant when included in an arbitrary expression, but only register names are valid where a register is required.

All register names must be defined. Many examples in this manual assume that `PC`, `R0`, `R1`, `R2`, and so on are valid register names. To make this the case it is first necessary to use the `RN` directive at the beginning of the source code, thus:

```
R0      RN   0
R1      RN   1
R2      RN   2
:       :    :
:       :    :
:       :    :
R15     RN   15
PC      RN   15
```

3.6 Assembler operators

The ARM assembler provides an extensive set of operators for use in expressions. The syntax of many of these resembles their counterparts in high level languages.

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

For example: `VALUE-2`

```
1:SHL:EXPONENT
```

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

For example: `-VALUE`

```
:LNOT:FLAG
```

3.6.1 The arithmetic operators

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

For the purposes of division and remainder all values are treated as 32-bit unsigned integers. + and - act on numeric, program-relative and register-relative expressions, the others act only on numeric expressions.

3.6.2 Boolean logical operators

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

These act on logical expressions.

3.6.3 Bitwise logical operators

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

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

3.6.4 Shift operators

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

These act on numeric expressions. The first operand is shifted or rotated by an amount given by the second operand. The SHIFTs are logical rather than arithmetic.

3.6.5 Relational operators

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

These act between two operands of the same type. The allowable types are numeric, program-relative, register-relative and string. They produce a logical value. For details of how string comparisons are done, see section 4.3.

3.6.6 String operators

Concatenation (binary)

`:CC:` joins (concatenates) two strings.

expression1 `:CC:` *expression2*

where *expression1* and *expression2* are strings.
For example: `"ABCD":CC:"EFGH"` gives `"ABCDEFGH"`

Slicing (binary)

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

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

"sssss" `:LEFT:`*n* returns the *n* left-most characters from the string *"sssss"*.
"sssss" `:RIGHT:`*n* returns the *n* right-most characters from the string *"sssss"*.

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

Length (unary)

`:LEN:`*expression* returns the length of a string *expression*.

Conversion (unary)

`:CHR:`*expression* returns a string of length 1 having ASCII Code *expression*. The *expression* must be numeric.

`:STR:`*expression* returns an eight digit hexadecimal string corresponding to an *expression* , if the *expression* is numeric.

`:STR:`*expression* returns the string TRUE or FALSE if the *expression* is logical.

3.6.7 Operator summary

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

+	7	+*A*	Unary plus
-	7	-*A*	Unary negate
LNOT	7	:LNOT:*A*	Logical complement of *A*
NOT	7	:NOT:*A*	Bitwise complement of *A*
LEN	7	:LEN:*A*	Length of string *A*
CHR	7	:CHR:*A*	ASCII string of *A*
STR	7	:STR:*A*	Hexadecimal string of *A*
*	6	*A***B*	Multiply
/	6	*A*/*B*	Divide
MOD	6	*A*:MOD:*B*	*A* modulo *B*
LEFT	5	*A*:LEFT:*B*	the left most *B* characters of *A*
RIGHT	5	*A*:RIGHT:*B*	the right most *B* characters of *A*
CC	5	*A*:CC:*B*	*B* concatenated on to the end of *A*
ROL	4	*A*:ROL:*B*	Rotate *A* left *B* bits
ROR	4	*A*:ROR:*B*	Rotate *A* right *B* bits
SHL	4	*A*:SHL:*B*	Shift *A* left *B* bits
SHR	4	*A*:SHR:*B*	Shift *A* right *B* bits
+	3	*A*+*B*	Add *A* and *B*
-	3	*A*-*B*	Subtract *B* from *A*
AND	3	*A*:AND:*B*	Bitwise AND *A* and *B*
OR	3	*A*:OR:*B*	Bitwise OR *A* and *B*
EOR	3	*A*:EOR:*B*	Bitwise exclusive OR *A* and *B*
=	2	*A*=*B*	*A* equal to *B*
>	2	*A*>*B*	*A* greater than *B*
>=	2	*A*>=*B*	*A* greater than or equal to *B*
<	2	*A*<*B*	*A* less than *B*
<=	2	*A*<=*B*	*A* less than or equal to *B*
/=	2	*A*/=*B*	*A* not equal to *B* .blank
LAND	1	*A*:LAND:*B*	Logical AND
LOR	1	*A*:LOR:*B*	Logical OR
LEOR	1	*A*:LEOR:*B*	Logical exclusive OR

3.7 Store-loading directives

The line takes the general form: *{label} directive expression list*
They place data in store at the current instruction location and advance the instruction location pointer.

The possible directives are:

DCD or & which defines one or more words
DCW which defines one or more half-words (16-bit numbers)
DCB or = which defines one or more bytes.

expression list is a list of one or more numeric expressions, separated by commas. In the case of DCB or =, the list may also include string expressions, which causes the characters of the string to be loaded into consecutive bytes in store. For example:

```
TABLE1   DCD     VALUE1,VALUE2;load 2 words into Table1
TABLE2   =       1,2,3,4,5,6 ;load 6 bytes into Table2
MESSAGE  =       "Turn off motor"
ERRORM   =       99,"Error number 99",0
TABLE4   =       """a sentence within quotes"""
TABLE5   =       1,2,3,"a","b",4,5,6
PROMPT   =       ">" ;loads 62 into one byte of memory
PROMPT2  DCW     ">" ;loads 62, and then 0 into 2 bytes
PROMPT3  &       ">" ;loads 62,then 0,0,0 into 4 bytes
```

Loading memory with nulls has its own directive:

Syntax: *{label}* % *numeric expression*

For example:

```
BLANKS   %       &400 ;store 1K of nulls
```

3.7.1 Syntax differences

In AAsm, DCD can only take numeric expressions.

In ObjAsm, DCD can take a program-relative expression, even when the code does not have an absolute origin.

3.8 The ALIGN directive

After using memory filling directives such as:

```
=    "a long string"        ;messages
=    1,2,3,4,5              ;a long list
%    VALUE4/SIZE            ;nulls
```

the program counter doesn't necessarily point to a word boundary, which it must do if the file is to continue with program instructions. The alignment of the PC to a word boundary is automatic if an instruction mnemonic is encountered after the tables. The assembler will insert up to three nulls to achieve automatic alignment. However, there are occasions when an alignment needs to be forced.

```
ALIGN
```

on its own will set the instruction location to the next word boundary. However, `ALIGN` can take two optional parameters:

`ALIGN` *{power-of-two} {,offset-expression}*

4 is the *power-of-two* default and 0 is the *offset-expression* default, so `ALIGN` on its own will increment the PC to the next word boundary. Other values will force the PC to align to any particular boundary needed by the programmer. These extra arguments will only rarely be needed.

3.9 ?label

?label is used to interrogate a label and so find out how many bytes of code were produced on its defining line. For a label on a line containing an opcode mnemonic the length is 4, for a label on an otherwise blank line the length is zero. For `DCD`, `DCW`, `DCB` and `%` directives, the length is the combined length of all the operands.

For example:

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

3.10 Literals

The directive LTORG (literal origin) is used to define an area in which to place literals. Literals are addressed using PC relative addressing so large programs may need several LTORG directives. Literals are intended to enable the programmer to load immediate values into a register which might be out of range as MOV/MVN arguments. The syntax for their use is: LDR *register*, =*expression*

The assembler will then take certain actions. It will, if possible :

(1) Replace the instruction with a MOV or MVN, or

(2) Generate a program-relative instruction and if no such literal already exists within the addressable range, then place the literal in the next literal pool.

Literal values are stored in a literal pool which is either at the end of the file or immediately following the next LTORG directive. Duplication of values in the pool will be avoided, provided that any possible duplicate expressions are evaluable on pass 1.

3.11 Laying out areas of memory

The assembler can lay out areas of memory. The start address of such an area is given by the ^ directive.

Syntax: ^ *expression*

The origin of the storage area is set to *expression*, and a storage-area location counter @ is also set to *expression*. The expression must be fully evaluated on the first pass of the assembly, but may be program-relative. In the absence of a ^ directive, the @ counter is set to zero.

Space is reserved by the # directive.
Syntax: *{label}* # *expression*

For example:

```
LABEL1      #    n      ;reserve n bytes
 .......
 .......code
 .......code
 .......
LABEL2      #    4        ;reserve 4 bytes,
              ;attached to the end of LABEL1's store
```

Every time # is encountered, the label is given the value of @, and then @ is incremented by the number of bytes reserved. The @ counter may be set to another value any number of times by the repeated use of ^ and so storage areas can be easily established anywhere in memory.

A special extension of ^ allows a register to be attached to the base address of a storage area:

^ *expression,register*

The register introduced by this extra parameter is taken to be implicit in all symbols defined by any # directives which follow, until cancelled by another ^ directive. In this case *expression* must be an absolute value. For example:

```
SB                  RN   10     ;SB is register 10
                    ^    0,SB   ;@=0
Start               #    0      ;i.e. [SB,#0]
Frame               #    4      ;i.e. [SB,#0]
StaticBase          #    4      ;i.e. [SB,#4]
StaticBase_Offset   *    StaticBase-Start
```

The subsequent # directives are therefore generating register-relative symbols. This means that later in the source program, it becomes possible to quote any symbol containing an implicit register name in a load or store instruction and the pre-indexed form of opcode will be generated.

For example, the valid line:

```
LDR  [SB,#StaticBase_Offset]
```

can be replaced by the shorter line:

```
LDR  R0,StaticBase
```

and the same code will be generated by the assembler.

3.11.1 Counter values

The current value of the assembler's program location counter is referred to by the dot symbol '.' while the current value of the storage-area location counter is, as has already been noted, the '@' symbol. Since these symbols are not particularly obvious (especially when appearing in expressions) they may, if the programmer wishes, be replaced by {PC} and {VAR} respectively.

3.12 Variables

Symbols have a fixed value attached to them, derived from the first or second pass of the assembly process. It is also possible to define symbols which have a value which changes as the assembly proceeds. Such symbols are called variables, and the ARM assembler has two types, local variables and global variables. Global variables can operate over the entire source file, whereas local variables are accessible within the confines of a macro expansion. Local variables are described in section 4.4.

3.12.1 Global variables

Variables must be declared before they are used. The three types of global variable are arithmetical, logical and string, declared by respectively GBLA, GBLL and GBLS. These symbols may now be used in expressions like normal symbols. The directives SETA, SETL and SETS are provided to alter the values of both global and local variables.

Syntax: GBL*x* *variable name*

Syntax: *variable name* SET*x* *expression*

For example:

```
count       SETA  count+1
message     SETS  "media error"
```

`count` and `message` can be used as required in the source file:

```
space       #     count
string      =     message
```

Any attempt to use them as labels will, quite rightly, cause the syntax checker to issue error messages: they have been declared as global variables and will not therefore be accepted as labels. However, if the $ symbol is prefixed to them, variable substitution will take place before the line is passed to the syntax checker. Logical and arithmetic variables will be replaced by the result of applying `:STR:` to them. String variables will be replaced by their value.

For example:

```
                GBLS    A
                GBLA    B
                GBLL    C
        ;three variable types declared
        A       SETS    "Labname"
        B       SETA    1
        C       SETL    {TRUE}
        ;and duly set
        ;without $ they are rejected as labels
        A       ADD     R0,R0,R1;  syntax error!
        ;with $ they are accepted
        $A      AND     R0,R1,#8
        L$B     AND     R2,R3,#16
        $C      AND     R4,R5,#32
```

After the assembler has performed variable substitution, its own internal conception of the last three lines of source can be considered as:

```
Labname     AND     R0,R1,#8
L00000001   AND     R2,R3,#16
TRUE        AND     R4,R5,#32
```

3.12.2 Other useful variables

The variables {PC} and {VAR} have already been explained, but there are three other useful variables which take the bracketed form of {*name*}. These are {TRUE} and {FALSE}, which are logical constants, and {OPT} which is the value of the currently set printer output option. (The printer option values are shown in section 5.9.4.) A simple but extremely useful way of using {OPT} is to use it to store the currently set printer options, force a temporary change in printing mode, and then, later in the source code, to restore the original value of {OPT}. For example:

```
            GBLA    AS_WAS
AS_WAS      SETA    {OPT}
;start of long section of code
;e.g. a macro
            OPT     2       ;turn off listing!
            .....lots of code......
            OPT     AS_WAS ;restore print option
;end of long section of code
```

3.13 Local labels

Although normal labels may not begin with a digit, there is a special form of local label which bears a number in the range 0-99. Such labels have limited scope; their scope being delimited by ROUT directives.

The syntax to begin a new local label area is:
{label} ROUT
in the label and instruction fields respectively. The start of the source is the start of the first local label area.

The local label definition syntax is:
number{routinename}
in the label field. The number must lie in the range 0–99. *routinename* need not be present, but if it is it will be checked against the label on the last ROUT directive. If no label is present on the last ROUT directive, yet a *routinename* has been provided, an assembly error will be generated.

The syntax for the local label reference syntax is:

% *{x}{y}n{routinename}*

% The % symbol introduces a local label reference. It may be used anywhere where an ordinary label reference is valid.

{x}{y} The optional letters *x* and *y* tell the assembler the direction and/or level for the search of the location of the local label.

The *{x}* character:

absent	look backwards and forwards for the label
B	look backwards for the label
F	look forwards for the label

searches for a local label will never go outside the current local label area – that is, they will never go past a ROUT directive. The same local label may be defined many times. The assembler always uses the first matching local label that it finds in its search.

The *{y}* character:

absent	look at this macro and all levels towards the source
A	look at all macro levels
T	look only at this macro level

The number n is the number given to the local label.

{routinename} is optional but if used, makes the source listing more readable. If present the assembler will check it against the routine's label.

```
NORMLABEL   ROUT      ;The routine is between the ROUTs.
            .......   ;Its name is NORMLABEL, but the
            .......   ;naming of the routine is optional
            .......
00          .......   ;Local label 00
            .......
            BEQ  %00NORMLABEL  ;Branch if equal to 00
            .......
01          .......   ;Local label 01
            .......
NEXTROUTINE ROUT
```

Local labels can be used anywhere in the source file and are particularly useful for the macro label uniqueness problem.

3.14 Error handling directives

As an aid to error trapping, the `ASSERT` directive is provided for use inside and outside macros.

The syntax is: `ASSERT` *logical expression*

For example: `ASSERT TEMP1 < TEMP`

If the *logical expression* returns a true result then nothing happens but a false result will generate an error during the second pass of the assembly. The error message is `"Assert failed at line xxxxxx"`

A similar directive ! is inspected on both passes of the assembler. This time an arithmetic expression is evaluated:

! *arithmetic expression,string expression*

If the arithmetic expression = 0, no action is taken on pass 1 and the string is printed out as a warning on pass 2. No error is generated.

If the arithmetic expression <> , 0 an error is produced and assembly halts after pass 1.

The arithmetic expression is evaluated on pass one, so forward referencing is not permitted. The string expression is printed as a warning or error, if produced.

3.15 The ORG and LEADR directives

The program starting point is determined by the ORG directive.

The syntax is: ORG *absolute-expression*

For example:

```
            ORG     &100      ;or

START       *       &100
            ORG     START
```

At most one ORG is allowed in the entire source and no ARM instructions or assembler store directives can precede the ORG directive. ORG sets the program location counter, the symbol for which is '.'. For AAsm, ORG also sets the load and execute address for the code file. In the absence of an ORG directive, the program is considered to be relocatable; the program location counter is initially set to 0.

AAsm (but not ObjAsm) has a directive called LEADR, the load and execute address. LEADR can be used with or without the ORG directive to indicate the address at which the program should load and run. If ORG is present, then LEADR will override its effect on load and execute addresses; the purpose of the directive is to enable a default run address to be set for relocatable binary output.

The syntax is: LEADR *absolute-expression* , for example:

```
LEADR   &1000
```

3.16 The END directive

Processing of an input file stops on encountering END. If the input file was part of a nested piece of assembly, invoked by a GET directive (see section 5.3), then assembly will continue within the file containing the GET, at the line following the GET directive. Otherwise the current pass will stop. If this was the first pass, and no errors have been generated, then assembly will proceed to the second pass starting again in the original source file. Failing to end a file with an END or LNK (see section 5.3), is an error. Any source after END or LNK will be ignored by the assembler.

4. Conditional assembly, repetitive assembly and macros

4.1 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, but ELSE is also supported, giving the full IF...THEN...ELSE...ENDIF conditional assembly.

The start of the section is

[*logical expression*

and is known as the IF directive.

|

is the ELSE directive and

]

is the ENDIF directive.

If the logical expression yields a false result, the assembler immediately searches for the | or] directive and will only continue assembly when one of these is reached. Lines conditionally skipped by these directives are not listed if -TERSE is given to the command line, or TERSE ON is given to the action prompt, or by default. If -NOTERSE is given to the command line, or TERSE OFF is given to an action prompt, then conditionally skipped code will be listed.

A block being conditionally assembled can itself contain more [|] directives, that is conditional assembly can be nested. It is also valid to place more than one ELSE directive within an IF block. For example: here is a notional data storage routine which can either use a disc or a tape data storage system. To assemble the code for tape operation, the programmer prepares the system by altering just one line of code, the label SWITCH.

```
DISC   *   0
TAPE   *   1
SWITCH *   DISC
       ..........
       ...code...
       ..........
       [ SWITCH=TAPE
       ........................
       ...tape interface code...
       ........................
       ]
       [ SWITCH=DISC
       ........................
       ...disc interface code...
       ........................
       ]
       ...code continues...
       ....................
```

or alternatively,

```
       [ SWITCH=TAPE
       ........................
       ...tape interface code...
       ........................
       |
       ........................
       ...disc interface code...
       ........................
       ]
       ...code continues...
       ....................
```

The IF construction can be used inside macro expansions as easily as it is used in the main program.

4.2 Repetitive assembly

It is often useful for program segments and macros to produce tables and to do this they must be able to have a conditional looping statement. The ARM assembler has the `WHILE...WEND` construction, and its use is much the same as the similar form found in high level languages, taking the syntax:

`WHILE` *logical expression*

to start the repetitive block and

`WEND`

to end it.

For example:

```
            GBLA    counter
counter     SETA    100

            WHILE counter >0
            ..................
            ....do something....
            ..................
counter     SETA    counter-1
            WEND
```

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.

See section 4.1 for details of when conditionally skipped lines are listed.

4.3 Evaluating logical expressions

The ARM assembler provides six relational operators and four Boolean operators which can be combined in various ways to form logical expressions. The relational operators are:

operator	meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<> or /=	not equal to

The operands may be string, numeric, program-relative or register-relative expressions, but must be of the same type. Note that a length 1 string will be coerced to a numeric value if necessary.

When the operands evaluate to numbers, the comparisons are unsigned. When using strings as the operands, the results are not always so straightforward unless the simple case of <> or = is being used:

```
A <  B if and only if A is a leading substring of B
A >  B if and only if B <  A
A <= B if and only if A <  B or A=B
A >= B if and only if B <= A
```

For example:

```
"XY":CC:"Z" = "XYZ"      gives TRUE
"XYZ"       <> "XYZ"     gives FALSE
"XYZ"       <> "XY"      gives TRUE
"B"         =  "C"       gives FALSE
"B"         <  "C"       gives FALSE
"B"         >  "C"       gives FALSE
```

The logical operators `:LOR:`, `:LAND:`, `:LEOR:` and `:LNOT:` perform the normal logical operations.
Thus:

- *expression1* `:LOR:` *expression2* gives TRUE if either expression is TRUE.
- *expression1* `:LEOR:` *expression2* gives TRUE if one of the expressions is TRUE but not both.
- *expression1* `:LAND:` *expression2* gives TRUE if both expressions are TRUE and FALSE otherwise.
- `:LNOT:` *expression* gives TRUE if the expression is FALSE, and vice versa.

4.4 Macros

A macro facility allows similar pieces of code to be repeated throughout the program. It can also be used to generate identical pieces of code in situations where for some reason a call to a subroutine is not the optimum solution. A macro must be able to change the code that it generates in accordance with parameters passed to it.

Syntax: `MACRO`

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

This is immediately followed by a macro prototype statement which takes the form:

{$ label} macroname $ parameter,$ parameter,$ parameter,...

{$ label} is optional. If present, it is treated as an additional parameter.

Parameters are passed to the macro as strings, and substituted before syntax analysis. They are optional and any number of them may be given.

A short example will illustrate the use of a macro:

```
            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, others may have none at all.

The body of the macro now follows, with `$label` being optional, even if it was given in the macro prototype statement.

Note that `$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. 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

MACRONAME 9

MACRONAME ,"disc",
```

The $ symbol may be used in a string, provided that it is marked by a $$ escape sequence, for example:

```
-       "the price is $$$etc"
```

The `$$` will be interpreted as a real `$`, and then `$etc` will be correctly interpreted as the parameter. This illustrates the important point that macro parameters will be substituted wherever they occur in the macro's body.

4.4.1 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 arithmetical, logical and string, declared by respectively `LCLA`, `LCLL` and `LCLS`:

```
LCLA  symbol ;Arithmetic type, initialised to zero.
LCLL  symbol ;Logical type, initialised to FALSE.
LCLS  symbol ;String type, initialised to a 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` . This would be given in the following form:

variable name `SETx` *expression*

4.4.2 The MEXIT directive

Normally macro expansion terminates on encountering the `MEND` directive, at which point in time 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.

4.4.3 Default values

Syntax: *$parameter=default value*

For example: in the macro prototype statement of the MACRONAME macro above, it is possible to write `$num=10` and then when calling the macro a vertical bar symbol '|' causes the default value 10 to be used rather than the value $num, so:

```
MACRONAME |,"disc",7
```

will be equivalent to:

```
MACRONAME 10,"disc",7
```

Quotes are used around the default value if embedded spaces or leading or trailing spaces are needed as part of the default string, for example:

```
$string=" a default string "
```

If the default string needs to be in quotes itself, then a double quote is used to distinguish the substituted quotes from the discarded quotes, for example:

```
$string=""""hello world""""
```

will ultimately be substituted into the macro as "hello world".

4.4.4 The 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, logic variables and arithmetic variables are prefixed by the $ symbol they are substituted by putting in a string equivalent of them. Normal syntax checking is performed upon the line after these substitutions have been performed.

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 `$T33` part of the label to the macro prototype statement and would have accepted `$T33L25` as a label in its own right — which was not the intention.

4.4.5 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. Macros may be nested up to a depth of 255.

5. Assembling, link files, printing

5.1 The command line

The assembler is loaded by entering `aasm` on the ARM command line, which has the effect of bringing up the assembler command level prompt, which is:

```
Action:
```

In assembler command level a number of commands can be used:

`Asm`
`Print`
`Symbol`
`WIdth` *number*
`Length` *number*
`Terse`
`Warning` *filename*
`Quit`
`Xref`
`Help`

Capital letters indicate minimum abbreviations.

It is possible to add keywords to the `aasm` command to duplicate the effects of most of these assembler level commands.

`-FRom` *filename*	specifies source file
`-TO` *filename*	specifies object file
`-Warning` *filename*	duplicates `WARNING` *filename*
`-Print`	duplicates `PRINT ON`
`-NOPrint`	duplicates `PRINT OFF`
`-Quit`	causes `QUIT` after the assembly
`-NOQuit`	does not cause `QUIT` after the assembly
`-Terse`	duplicates `TERSE ON`
`-NOTerse`	duplicates `TERSE OFF`
`-Xref`	duplicates `XREF ON`
`-NOXref`	duplicates `XREF OFF`
`-WIdth` *number*	duplicates `WIDTH` *number*
`-Length` *number*	duplicates `LENGTH` *number*

The defaults are `-NOPRINT -NOQUIT -TERSE -NOXREF -WIDTH 131 -LENGTH 60`.

`-FROM` and `-TO` have no effect unless both are specified. If both are specified, an assembly is performed immediately using the specified files. The parameters belonging to `-FROM` and `-TO` may be specified without specifying the key words.

5.2 Assembling a program

The prompt may be followed specifically as in –

```
Action : ASM
```

```
Source filename : filename
```

Object filename : *object-filename*

or more succinctly –

```
Action : A filename object-filename
```

The assembler will load the file and attempt to assemble it, indicating that it is running a first pass. Any discrepancies in the format or syntax of the source will be detected and the appropriate error message, line number, and the offending line itself will be printed. Linked source files will be fetched if the primary source asks for them to be included.

If no errors are detected, the second pass will be started and an object file will be built up. A complete list of error messages is given in appendix B.

5.3 Linking source files

The GET directive in the source file is used to include a secondary source file within the current assembly. The syntax within the source code is:

GET *filename*

In this case assembly continues in the original source when assembly of the secondary source file is complete. The secondary source file must be terminated by an END or LNK directive, and may include further GET directives.

In the following example the primary file is called `file_a` :

```
SYM1        *   SYM2+100
            .................
            ....file_a code....
            .................
            GET file_b
            .................
            ....more file_a....
            ........code.......
            .................
            END
```

This is the secondary file, `file_b` :

```
SYM2        *   200
            ............
            ............
            END
```

Symbol SYM1 takes the value 300. There are two points to notice in this example:

`file_b` has no ORG statement and so the program counter merely continues its increment as `file_b` is assembled. Had the secondary file been given an ORG of its own, an error would be flagged. `file_b` must have an END directive, whereupon control passes back to `file_a`.

Alternatively, the secondary file can be called using LNK:

LNK *filename*

Now, when the secondary file is assembled, there will be no return to the primary file. LNK is generally used to split large source files into sequences of smaller more manageable ones. GET is generally used for inclusion of standard code such as macro libraries.

5.4 The object file

The object file can be loaded and executed by the command:

*RUN *filename*

5.5 The SYMBOL command

After an assembly all the symbols encountered can be listed if the SYMBOL command is given. Typing SYMBOL brings the prompt Option: onto the screen. The valid options are A, N, S and HELP. Usually, SYMBOL will be typed with the option letter (or HELP) on the same line.

SYMBOL A

gives an alphabetic listing of the symbols and their hex values.

SYMBOL N

gives a numeric listing of the symbols and their hex values.

SYMBOL S

gives no automatic symbol list but instead prompts for a symbol name. The name will then be printed, together with its value. To return from this prompt to the option prompt type a dot '.'

Symbols declared but unused are marked with a '*'

The format of the symbol table is designed, as neatly as possible, to fit the current WIDTH and LENGTH settings.

To return from the option prompt to the action prompt type quit.

5.6 The XREF command

XREF ON is a command to the assembler which causes it to collect (on the next assembly) cross-reference information for all the symbols used. When the SYMBOL command is next used, not only will the actual values of the symbols be supplied but also information on where the symbol was defined and where it was used (line numbers within macros and line numbers within files) will be given. The XREF OFF cancels the effect of XREF ON. Typing XREF on its own prints the prompt Option: onto the screen. The valid options are XREF ON, XREF OFF and HELP.

5.7 The WARNING command

WARNING *filename* specifies a file which will receive all error warnings from the assembler. This file will be closed when the assembly has finished. Typing WARNING on its own prints the words Error file name: onto the screen and so prompts for a *filename* to be entered. Note that typing WARNING HELP creates a filename called 'HELP'.

5.8 The QUIT command

QUIT, if specified, causes the assembler to terminate.

5.9 Assembler print commands

Assembler source code may be viewed directly from the TWIN editor, or from within the assembler using the command PRINT ON. This command turns on the assembler screen listing, and on subsequent assemblies, the source code, object code, memory addresses and reference line numbers will be printed on the screen. PRINT OFF turns this facility off. Typing PRINT on its own brings the prompt Option: onto the screen. The valid options are PRINT ON, PRINT OFF and HELP. The default condition is PRINT OFF.

5.9.1 WIDTH n

The width of the output can be specified using this command. n should be a number between 1 and 254, and for screen output `WIDTH 80` is probably the most suitable choice. If a source line has characters and spaces giving it a length greater than n, the excess is printed on subsequent lines. The default for n is 131, which is a convenient width for many printers. If no number is given, the prompt `Value:` is put on the screen.

5.9.2 LENGTH n

The number of lines output per page can be specified using this command. n should be a number between 1 and 254. After n lines have been output, the assembler gives a form feed command which steps the printer onto the next perforation (assuming that continuous paper is used and has been properly loaded) and clears the screen on the computer. Page lengths of between 60 and 70 are suitable for most sizes of stationery. The default for n is 60. If no number is given, the prompt `Value:` is put on the screen.

5.9.3 TERSE

`TERSE ON`, which is the default, ensures that code which is conditionally not assembled is not printed. `TERSE OFF` allows the printing of unselected code. Typing `TERSE` on its own prints the prompt `Option:` onto the screen. The valid options are `TERSE ON`, `TERSE OFF` and `HELP`.

5.9.4 Dynamic print options

The output of the listing can be controlled at assembly time by directives placed in the source code, provided that the `PRINT ON` command is in force.

`OPT n`

There are 14 options which take effect as they are encountered in the listing. `n` is a numeric expression. It may be any of the following, or any sum of them; for example, `OPT 1025` will turn on both pass one and normal listing.

`OPT 1`

Turn on listing.

`OPT 2`

Turn off listing.

`OPT 4`

Page throw. Issues an immediate form feed and so starts a new page.

`OPT 8`

Reset the line number counter to zero.

`OPT 16`

Turn on the listing of `SET`, `GBL` and `LCL` directives.

`OPT 32`

Turn off the listing of `SET`, `GBL` and `LCL` directives.

`OPT 64`

Turn on the listing of `MACRO` expansions.

`OPT 128`

Turn off the listing of `MACRO` expansions.

`OPT 256`

Turn on the listing of `MACRO` calls.

`OPT 512`

Turn off the listing of `MACRO` calls.

`OPT 1024`

Turn on the pass one listing.

```
OPT 2048
```

Turn off the pass one listing.

```
OPT 4096
```

Turn on the listing of conditional directives.

```
OPT 8192
```

Turn off the listing of conditional directives.

The default settings are:

listing on
`SET, GBL, LCL` on
macro expansion on
macro calls on
pass one listing off
conditional directives on.

5.9.5 TTL

Syntax: `TTL` *string*

This directive creates a title which will be used on all subsequent pages until either the assembly is complete or another `TTL` directive creates adifferent title. `TTL` on its own creates a title consisting of a blank line.

5.9.6 SUBTTL

Syntax: `SUBTTL` *string*

This directive creates a subtitle, which will be printed directly beneath the title. As with `TTL`, its effect remains until a new `SUBTTL` is issued.

If two or more `TTL` or `TTL/SUBTTL` combinations occur before the next page break, only the latest of the combinations will be obeyed. Forcing a new page immediately after a `TTL` will ensure that the title is printed and it may be necessary to do this if different titles separated by only small quantities of source listing are required.

6. Appendix A

6.1 ARM instruction set

There are 16 instructions determined by the bit-pattern in B24-B27, divided into 5 classes.

B27	B26	B25	B24	Mnemonics	Instruction type
1	0	1	0	B	BRANCH
1	0	1	1	BL	BRANCH WITH LINK
0	0	0	X	various	DATA PROCESSING
0	0	1	X	various	GROUP
0	1	0	0	LDR/STR	
0	1	0	1	LDR/STR	SINGLE DATA TRANSFER
0	1	1	0	LDR/STR	GROUP
0	1	1	1	LDR/STR	
1	0	0	0	LDM/STM	BLOCK DATA TRANSFER post inc/dec
1	0	0	1	LDM/STM	BLOCK DATA TRANSFER pre inc/dec
1	1	1	1	SWI	SUPERVISOR CALL
1	1	0	0		reserved for future expansion
1	1	0	1		
1	1	1	0		

Bits 31 to 28 encode the condition. The basic instruction set is expanded by altering the pattern of the remaining 24 bits. The main AAsm mnemonic combinations are as follows:

THE ROOT INSTRUCTIONS
All carry the optional conditional postfix *{cc}*.

Branch Group
B*{cc}* Branch
BL*{cc}* Branch with Link

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

S may follow these mnemonics.
P may follow CMP, CMN, TST or TEQ.
S - - Set Condition Codes.
P - - Make Rd=R15 for CMP, CMN, TEQ and TST.
S is included by the assembler
for CMP, CMN, TEQ and TST.

Single Data Transfer Group
`LDR`*{cc}*
`STR`*{cc}*

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

Block Data Transfer Group
`LDM`*{cc}*
`STM`*{cc}*
One of the suffixes DA, DB, IA, IB, EA, ED, FA, FD must follow.

Supervisor Call
`SWI`*{cc}*

7. Appendix B

7.1 AAsm and ObjAsm error messages

`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 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 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 register name symbol`
A register name is wrong.

`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`
A syntactically incorrect shift name.

`Bad symbol`
Syntax error in a symbol name.

`Bad symbol type`
This will occur after a # or * directive and it means that the symbol being defined is already 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.

`Data transfer offset out of range`
The immediate value in a data transfer opcode must be in the range $-4095 <= e <= +4095$.

`Decimal overflow`
The number exceeds 32 bits.

`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.

`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.

`Global name already exists`
This name has already been used in some other context.

`Hexadecimal overflow`
The number exceeds 32 bits.

`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`
A register to register opcode cannot perform a suitable rotation on the value supplied to bring it into 8-bit range.

`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.

`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 or set 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.

`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.

`Multiply defined symbol`
A symbol has been defined more than once.

`No current macro expansion`
A `MEND`, `MEXIT` or local variable has been encountered but there is no corresponding `MACRO`.

`Numeric overflow`
The number exceeds 32 bits.

`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 which has insufficient characters in it.

`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 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`.

`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 which is neither an opcode nor 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 type.

`Unknown shift name`
Not one of the six legal shift mnemonics.

The acceptable syntax for the various directives is shown in this table:

`[`	no label	an expression is expected
`|`	no label	takes no expression
`]`	no label	takes no expression
`!`	no label	two expressions are expected
`#`	optional label	an expression is expected
`*` (EQU)	label	an expression is expected
`=` (DCB)	optional label	an expression list is expected
`%`	optional label	an expression is expected
DCW	optional label	an expression list is expected
`&` (DCD)	optional label	an expression list is expected
`^`	no label	expression and optional register expected
END	no checking performed	
LNK	no checking performed	
GET	no label	a filename is expected
ORG	no label	an expression is expected
OPT	no label	an expression is expected
TTL	no label	
SUBTTL	no label	
RN	label	an expression is expected

WHILE	no label	an expression is expected
WEND	no label	takes no expression
MACRO	no label	takes no expression
MEXIT	no label	takes no expression
MEND	no label	takes no expression
GBLA	no label	a symbol is expected
GBLL	no label	a symbol is expected
GBLS	no label	a symbol is expected
LCLA	no label	a symbol is expected
LCLL	no label	a symbol is expected
LCLS	no label	a symbol is expected
SETA	label	an expression is expected
SETL	label	an expression is expected
SETS	label	an expression is expected
ASSERT	no label	an expression is expected
ROUT	label	takes no expression
ALIGN	no label	one or two expressions expected
LTORG	no label	takes no expression
LEADR	no label	an expression is expected

8. Appendix C

8.1 Example AAsm file

Switch on the computer.

The `ARM A*` prompt appears.

Load *TWIN* from the filing system by typing:

`TWIN` [RETURN]

Type the following into *TWIN*, placing [RETURN] as necessary at the end of each line.

```
; -> test
 ORG &1000
 SWI 17
 END
```

This source file is now saved using the name taken from the *;* -> line at the top of the *TWIN* file. Note that *filename* follows the -> (dash, greater than symbols), and that -> must be on line one of the *TWIN* document.

Next, obtain an ARM command line and load the assembler.

`*aasm` [RETURN]

The screen will show:

```
ARM stand alone Macro Assembler Version x.xx
Entering interactive mode
Action:
```

To which responses should be made which culminate in the following display being achieved:

```
Action: ASM
Source file name: test
Code file name: testc
Pass 1
Pass 2
```

```
Assembly complete
No errors found
Action:
```

Both passes ran, with no errors reported.

The program may now be executed by leaving AASM:

```
Action: quit
```

And the program can be run:

```
*testc
```

testc will now run, finishing very quickly.

9. Appendix D

9.1 ObjAsm directives

ObjAsm is the ARM Assembler which creates Acorn Object File code. The new directives of ObjAsm listed here should be read in conjunction with the *ARM utilities user guide* explaining Acorn Object File (AOF) code.

9.1.1 AREA

This directive gives a name plus optional attributes and alignment to the area in which the code/data following is to be put. The basic form of the directive is AREA *symbol*. The symbol is the name of an area and as such it is an external symbol which can be used in the link phase of processing: other programs may import the symbol and make use of it. The value of the symbol may be taken to be *Offset zero* from the start of the area.

A list of attributes may follow the symbol thus:

AREA *symbol{,attr}{,attr}....{,*align*=expression}*

The attributes, many of which are self-explanatory, are as follows:

ABS	Absolute : this area has a fixed position in the memory map.
REL	Relocatable : this area may be relocated by the linker.
PIC	Position Independent Code : this code may be loaded anywhere without modification.
CODE	This area contains code (and is therefore read only).
DATA	This area contains read–write data.
READONLY	This area may not be written.
COMDEF	Common area definition
COMMON	A common area

The last two are required by Fortran.

ALIGN=*expression*

The expression should be between 2 and 12. This specifies a power of two, that is a 2 would give an alignment to a 4-byte (word) boundary; a 3 would give an alignment to an 8-byte boundary and so on. The ALIGN parameter may beneeded to make sure the area maintains a sound alignment when it becomes attached to other areas during linking. The default alignment is 2, that is word alignment.

9.1.2 IMPORT

Syntax: IMPORT *symbol*

IMPORT is followed by a symbol which is treated as a program address. It provides the assembler with a name which may be referred to but which is not defined within this assembly. It must therefore be imported at link time from another piece of the AOF code, when its value will be ascertained and used.

9.1.3 EXPORT

Syntax: EXPORT *symbol*

EXPORT is also followed by a symbol. This time the symbol is being declared for use by other AOF files at link time.

9.1.4 ENTRY

Syntax: ENTRY

If the file contains the directive ENTRY it is signalling to the whole program (contained in the various AOF files) that the address computed for ENTRY (that is the value of the program location counter when ENTRY is assembled) is the execute address for the entire program.

9.1.5 KEEP

Syntax: KEEP *symbol*

The linker will not normally keep track of symbols it does not need, and when operating in `-adfs` mode it keeps no symbols at all. However, when `-adfs` is not specified for the link operation, a table of all required external symbols is maintained. To force the linker to retain symbols it would otherwise consider unnecessary, a `-keep` is added to the link operation. ObjAsm's own directive KEEP has the function of declaring a symbol which is not needed by the AOF, but which can be maintained in the AOF symbol table.

In this way (provided `-adfs` mode is 'off' and `-keep` mode is 'on' when the link is performed), symbols of use to a symbolic debugger can be stored and will not be lost.

9.1.6 DCD

DCD in ObjAsm will accept program-relative expressions and imported symbols for its operands, as well as the numeric expressions as used by AAsm.

9.1.7 Literals

Program-relative expressions and imported symbols are also valid literals in ObjAsm.

9.1.8 Branch destinations

Imported symbols and program-relative symbols not defined in the current area are valid operands to the branch and branch and link instructions. Note that imported symbols, and program-relative symbols not defined in the current area are not valid in general expressions.

9.1.9 ObjAsm error messages

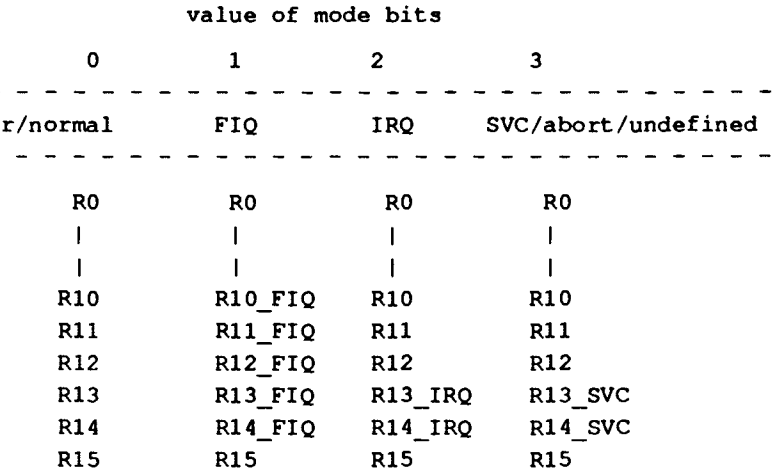
Objasm has 12 error messages of its own, in addition to those of AAsm. (appendix B contains the full list of AAsm and ObjAsm error messages, explained in more detail.)

Area directive missing
Area name missing
Bad area attribute or alignment
Bad exported name
Bad exported symbol type
Bad imported name
Code generated in data area
Entry address already set
External area relocatable symbol used
Externals not valid in expressions
Imported name already exists
Undefined exported symbol

10. Appendix E

Modes and registers

The ARM has four modes of operation, user mode, supervisor mode, interrupt mode and fast interrupt mode. The mode in which the processor runs is determined by the state of bits 0 and 1 in the Processor Status Register. The processor has 25 physical registers, but the state of the mode bits determine which 16 registers, R0-R15, will be seen by the programmer. The four modes available are shown in the diagram which follows.



In each mode the conceptual registers R0-R9 and R15 correspond to the physical registers R0-R9 and R15.

10.1 Mode 0:

User mode is the normal program execution state; registers R0-15 exist directly and in this mode only the N, Z, C and V bits of the PSR may be changed.

10.2 Mode 1:

The FIQ processing state has five private registers mapped to R10-14 (R10_FIQ-R14_FIQ) and a fast interrupt will not destroy anything in R10-R14. Most FIQ programs, particularly those used for data transfer, will not need to use R0-R9, but if they do, then R0-R9 can be saved in memory using a single instruction.

10.3 Mode 2:

The IRQ processing state has two private registers mapped to R13, R14 (R13_IRQ, R14_IRQ). If other registers are needed, their contents should be saved in memory using the single instruction available for this purpose.

10.4 Mode 3:

Supervisor mode (entered on SVC calls and other traps) also has two private registers mapped to R13, R14 (R13_SVC, R14_SVC). If other registers are needed, they too must be saved in memory.

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

10.5 Changing modes.

In the assembler, the action of `{P}` is used to change the PSR; this enables the `TEQ` instruction 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 being used as 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. The only way to pass from user mode to other modes are either to receive an external interrupt, or to make use of the `SWI` instruction.

11. Appendix F

Source code examples

The following examples show ways in which the basic ARM instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they all save some), mostly they just save code.

11.1 Using the conditional instructions

(1) 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
```

(2) Absolute value

```
TEQ   Rn,#0        ;test sign
RSBMI Rn,Rn,#0     ;and 2's complement if necessary
```

(3) Unsigned 32-bit multiply

```
;enter with numbers in Ra, Rb
     MOV   Rm,#0           ;result register
Loop MOVS  Ra,Ra,LSR #1
     ADDCS Rm,Rm,Rb
     ADD   Rb,Rb,Rb
     BNE   Loop ;stops when Ra becomes zero
                ;Rm contains Ra*Rb
                ;(Ra set to zero, Rb junk)
```

(4) 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:="."
```

(5) Division and remainder

```
;enter with numbers in Ra and Rb
     MOV   Rcnt,#1      ;bit to control the division
Div1 CMP   Rb,Ra
     MOVCC Rb,Rb,ASL #1
     MOVCC Rcnt,Rcnt,ASL #1
     BCC   Div1
     MOV   Rc,#0
Div2 CMP Ra,Rb          ;test for possible subtraction
     SUBCS Ra,Ra,Rb     ;subtract if ok
     ADDCS Rc,Rc,Rcnt   ;put relevant bit into result
     MOVS  Rcnt,Rcnt,LSR #1;shift control bit
     MOVNE Rb,Rb,LSR #1    ;halve unless finished
     BNE   Div2
;divide result in Rc
;remainder in Ra
```

11.2 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 exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (that is 2^32-1 cycles before repetition). A 33-bit shift generator with taps at bits 20 and 33 is required. The basic algorithm is newbit:=bit33 eor bit20, shift left the 33 bit number and put in newbit at the bottom. Then do this for all the newbits needed, that is 32 of them. Luckily this can all 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
```

11.3 Multiplication by a constant

(1) Multiplication by 2^n (1,2,4,8,16,32..):

```
MOV   Ra,Ra,LSL #n
```

(2) Multiplication by 2^n+1 (3,5,9,17..):

```
ADD   Ra,Ra,Ra,LSL #n
```

(3) Multiplication by 2^n-1 (3,7,15..):

```
RSB    Ra,Ra,Ra,LSL #n
```

(4) Multiplication by 6:

```
ADD    Ra,Ra,Ra,LSL #1 ;multiply by 3
MOV    Ra,Ra,LSL #1    ;and then by 2
```

(5) Multiply by 10 and add in extra number:

```
ADD    Ra,Ra,Ra,LSL #2 ;multiply by 5
ADD    Ra,Rc,Ra,LSL #1 ;multiply by 2
                       ;and add in next digit
```

(6) General recursive method for Rb := Ra*C, C a constant:

(a) 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
```

(b) 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
```

(c) 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-optimality 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
```

11.4 Loading a word from an unknown alignment

There is no instruction to load a word from an unknown alignment. To do this requires some code (which can be a macro) along the following lines:

```
;enter with 32-bit address in Ra
;uses Rb, Rc; result in Rd
;Note d must be less than c

  BIC    Rb,Ra,#3        ;get word-aligned address
  LDMIA  Rb,{Rd,Rc}      ;get 64 bits containing answer
  AND    Rb,Ra,#3        ;correction factor in bytes
  MOVS   Rb,Rb,LSL #3 ;..now in bits and test if aligned
  MOVNE  Rd,Rd,LSR Rb    ;produce bottom of result word
                         ;if not aligned
  RSBNE  Rb,Rb,#32       ;get other shift amount
  ORRNE  Rd,Rd,Rc,LSL Rb ;combine two halves to get result
```

11.5 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
```

11.6 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 generally applies to non-user mode programming:
; e.g. MOVS PC,R14. MOV PC,R14 is safer!
```

Acorn

The choice of experience.