6. Data Structures

We have already encountered some of the ways in which data is passed between parts of a program: the argument and result passing techniques of the previous chapter.

In this chapter we concentrate more on the ways in which global data structures are stored, and give example routines showing typical data manipulation techniques.

Data may be classed as internal or external. For our purposes, we will regard internal data as values stored in registers or 'within' the program itself. External data is stored in memory allocated explicitly by a call to an operating system memory management routine, or on the stack.

6.1 Writing for ROM

A program's use of internal memory data may have to be restricted to read-only values. If you are writing a program which might one day be stored in a ROM, rather than being loaded into RAM, you must bear in mind that performing an instruction such as:

```
STR R0, label
```

will not have the desired effect if the program is executing in ROM. So, you must limit internal references to look-up tables etc. if you wish your code to be ROMmable. For example, the BBC BASIC interpreter only accesses locations internal to the program when performing tasks such as reading the tables of keywords or help information.

A related restriction on ROM code is that it should not contain any self-modifying instructions. Self-modifying code is sometimes used to alter an instruction just before it is executed, for example to perform some complex branch operation. Such techniques are regarded as bad practice, and something to be avoided, even in RAM programs. Obviously if you are tempted to write self-modifying code, you will have to cope with some pretty obscure bugs if the program is ever ROMmed.

Finally, the need for position-independence is an important consideration when you write code for ROM. A ROM chip may be fitted at any address in the ROM address space of the machine, and should still be expected to work.

The only time it is safe to write to the program area is in programs which will always, always, be RAM-based, e.g. small utilities to be loaded from disc. In fact, even RAM-based programs aren't entirely immune from this problem. The MEMC memory controller chip which is used in many ARM systems has the ability to make an area of memory 'read-only'. This is to protect the program from over-writing itself, or other programs in a multi-tasking system. Attempting to write to such a region will lead to an *abort*, as described in

Chapter Seven.

It is a good idea, then, to only use RAM which has been allocated explicitly as workspace by the operating system, and treat the program area as 'read-only'.

6.2 Types of data

The interpretation of a sequence of bits in memory is entirely up to the programmer. The only assumption the processor makes is that when it loads a word from the memory addressed by the program counter, the word is a valid ARM instruction.

In this section we discuss the common types of data used in programs, and how they might be stored.

6.3 Characters

This is probably the most common data type, as communication between programs and people is usually character oriented. A character is a small integer whose value is used to stand for a particular symbol. Some characters are used to represent control information instead of symbols, and are called control codes.

By far the most common character representation is ASCII - American Standard Code for Information Interchange. We will only be concerned with ASCII in this book.

Standard ASCII codes are seven bits - representing 128 different values. Those in the range 32..126 stand for printable symbols: the letters, digits, punctuation symbols etc. An example is 65 (&41), which stands for the upper-case letter A. The rest 0..31 and 127 are control codes. These codes don't represent physical characters, but are used to control output devices. For example, the code 13 (&0D) is called carriage return, and causes an output device to move to the start of the current line.

Now, the standard width for a byte is eight bits, so when a byte is used to store an ASCII character, there is one spare bit. Previously (i.e. in the days of punched tape) this has been used to store a parity bit of the character. This is used to make the number of 1 bits in the code an even (or odd) number. This is called even (or odd) parity. For example, the binary of the code for the letter A is 1000001. This has an even number of one bits, so the parity bit would be 0. Thus the code including parity for A is 01000001. On the other hand, the code for C is 1000011, which has an odd number of 1s. To make this even, we would store C with parity as 11000011. Parity gives a simple form of checking that characters have been sent without error over transmission lines.

As output devices have become more sophisticated and able to display more than the limited 95 characters of pure ASCII, the eighth bit of character codes has changed in use.

Instead of this bit storing parity, it usually denotes another 128 characters, the codes for which lie in the range 128..255. Such codes are often called 'top-bit-set' characters, and represent symbols such as foreign letters, the Greek alphabet, symbol 'box-drawing' characters and mathematical symbols.

There is a standard (laid down by ISO, the International Standards Organisation) for top-bit-set codes in the range 160..255. In fact there are several sets of characters, designed for different uses. It is expected that many new machines, including ARM-based ones will adopt this standard.

The use of the top bit of a byte to denote a second set of character codes does not preclude the use of parity. Characters are simply sent over transmission lines as eight bits plus parity, but only stored in memory as eight bits.

When stored in memory, characters are usually packed four to each ARM word. The first character is held in the least significant byte of the word, the second in the next byte, and so on. This scheme makes for efficient processing of individual characters using the **`LDRB`** and **`STRB`** instructions.

In registers, characters are usually stored in the least significant byte, the other three bytes being set to zero. This is clearly wise as **`LDRB`** zeroes bits 8..31 of its destination register, and **`STRB`** uses bits 0..7 of the source register as its data.

Common operations on registers are translation and type testing. We cover translation below using strings of characters. Type testing involves discovering if a character is a member of a given set. For example, you might want to ascertain if a character is a letter. In programs which perform a lot of character manipulation, it is common to find a set of functions which return the type of the character in a standard register, e.g. R0.

These type-testing functions, or predicates, are usually given names like **`isLower`** (case) or **`isDigit`**, and return a flag indicating whether the character is a member of that type. We will adopt the convention that the character is in R0 on entry, and on exit all registers are preserved, and the carry flag is cleared if the character is in the named set, or set if it isn't. Below are a couple of examples: **`isLower`** and **`isDigit`**:

```
DIM org 100
sp = 1
link =14
WriteI = &100
NewLine = 3
Cflag = &20000000 : REM Mask for carry flag
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;Character type-testing predicates.
;On entry, R0 contains the character to be tested
;On exit C=0 if character in the set, C=1 otherwise
;All registers preserved
;
.isLower
CMP R0, #ASC"a" ;Check lower limit
BLT predFail ;Less than so return failure
CMP R0, #ASC"z"+1 ;Check upper limit
MOV pc, link ;Return with appropriate Carry
.predFail
TEQP pc, #Cflag ;Set the carry flag
MOV pc, link ;and return
;
.isDigit
CMP R0, #ASC"0" ;Check lower limit
BLT predFail ;Lower so fail
CMP R0, #ASC"9"+1 ;Check upper limit
MOV pc, link ;Return with appropriate Carry
;
;Test for isLower and isDigit
;If R0 is digit, 0 printed; if lower case, a printed
;
.testPred
STMFD (sp)!,{link}
BL isDigit
SWICC WriteI+ASC"0"
BL isLower
SWICC WriteI+ASC"a"
SWI NewLine
LDMFD (sp)!,{pc}
;
]
NEXT pass
REPEAT
INPUT"Character ",c$
A%=ASCc$
CALL testPred
UNTIL FALSE
```

The program uses two different methods to set the carry flag to the required state. The first is to use **TEQP**. Recall from Chapter Three that this can be used to directly set bits of the status register from the right hand operand. The variable **Cflag** is set to &20000000, which is bit mask for the carry flag in the status register. Thus the instruction

```
TEQP pc, #Cflag
```

will set the carry flag and reset the rest of the result flags. The second method uses the fact that the **CMP** instruction sets the carry flag when the **<lhs>** is greater than or equal to its **<rhs>**. So, when testing for lower case letters, the comparison

```
CMP R0,#ASC"z"+1
```

will set the carry flag if R0 is greater than or equal to the ASCII code of z plus 1. That is, if R0 is greater than the code for z, the carry will be set, and if it is less than or equal to it (and is therefore a lower case letter), the carry will be clear. This is exactly the way we want it to be set-up to indicate whether R0 contains a lower case letter or not.

Strings of characters

When a set of characters is stored contiguously in memory, the sequence is usually called a string. There are various representations for strings, differentiated by how they indicate the number of characters used. A common technique is to terminate the string by a pre-defined character. BBC BASIC uses the carriage return character &0D to mark the end of its **`$`** indirection operator strings. For example, the string **`"ARMENIA"`** would be stored as the bytes

```
A &41
R &52
M &4D
E &45
N &4E
I &49
A &41
cr &0D
```

An obvious restriction of this type of string is that it can't contain the delimiter character.

The other common technique is to store the length of the string immediately before the characters - the language BCPL adopts this technique. The length may occupy one or more bytes, depending on how long a string has to be represented. By limiting it to a single byte (lengths between 0 and 255 characters) you can retain the byte-aligned property of characters. If, say, a whole word is used to store the length, then the whole string must be word aligned if the length is to be accessed conveniently. Below is an example of how the string **`"ARMAMENT"`** would be stored using a one-byte length:

```
len &08
A &41
R &52
M &4D
A &41
M &4D
E &45
N &4E
T &54
```

Clearly strings stored with their lengths may contain any character.

Common operations on strings are: copying a string from one place to another, counting the length of a string, performing a translation on the characters of a string, finding a sub-string of a string, comparing two strings, concatenating two strings. We shall cover some of these in this section. Two other common operations are converting from the binary to ASCII representation of a number, and vice versa. These are described in the next section.

Character translation

Translation involves changing some or all of the characters in a string. A common translation is the conversion of lower case letters to upper case, or vice versa. This is used, for example, to force filenames into upper case. Another form of translation is converting between different character codes, e.g. ASCII and the less popular EBCDIC.

Overleaf is a routine which converts the string at `strPtr` into upper case. The string is assumed to be terminated by CR.

```
DIM org 100, buff 100
cr = &0D
strPtr = 0
sp = 13
link = 14
carryBit = &20000000
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;toUpper. Converts the letters in the string at strPtr
;to upper case. All other characters are unchanged.
;All registers preserved
;R1 used as temporary for characters
;
toUpper
STMFD (sp)!,{R1,strPtr,link};Preserve registers
.toUpLp
LDRB R1, [strPtr], #1 ;Get byte and inc ptr
CMP R1, #cr ;End of string?
LDMEQFD (sp)!,{R1,strPtr,pc} ;Yes, so return
BL isLower ;Check lower case
BCS toUpLp ;Isn't, so loop
SUB R1,R1,#ASC"a"-ASC"A" ;Convert the case
STRB R1,[strPtr,#-1] ;Save char back
B toUpLp ;Next char
;
.isLower
CMP R1, #ASC"a"
BLT notLower
CMP R1, #ASC"z"+1
MOV pc,link
.notLower
TEQP pc,#carryBit
MOV pc,link
]
NEXT
REPEAT
INPUT"String ",$buff
A%=buff
CALL toUpper
PRINT"Becomes "$buff
UNTIL FALSE
```

The program uses the fact that the upper and lower case letters have a constant difference in their codes under the ASCII character set. In particular, each lower case letter has a code which is 32 higher than its upper case equivalent. This means that once it has been determined that a character is indeed a letter, it can be changed to the other case by adding or subtracting 32. You can also swap the case by using this operation:

```
EOR R0, R0, #ASC"a"-ASC"A" ;Swap case
```

The `EOR` instruction inverts the bit in the ASCII code which determines the case of the letter.

Comparing strings

The example routine in this section compares two strings. String comparison works as follows. If the strings are the same in length and in every character, they are equal. If they are the same up to the end of the shorter string, then that is the lesser string. If they are the same until a certain character, the relationship between the strings is the same as that between the corresponding characters at that position.

`strCmp` below compares the two byte-count strings at `str1` and `str2`, and returns with the flags set according to the relationship between them. That is, the zero flag is set if they are equal, and the carry flag is set if `str1` is greater than or equal to `str2`.

```
DIM org 200, buff1 100, buff2 100
REM str1 to char2 should be contiguous registers
str1 = 0
str2 = 1
len1 = 2
len2 = 3
index = 4
flags = len2
char1 = 5
char2 = 6
sp = 13
link = 14
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;strCmp. Compare the strings at str1 and str2. On exit,
;all registers preserved, flags set to the reflect the
;relationship between the strings.
;Registers used:
;len1, len2 - the string lengths. len1 is the shorter one
;flags - a copy of the flags from the length comparison
;index - the current character in the string
;char1, char2 - characters from each string
;NB len2 and flags can be the same register
;
.strCmp
;Save all registers
STMFD (sp)!, {str1-char2,link}
LDRB len1, [str1], #1 ;Get the two lengths
LDRB len2, [str2], #1 ;and move pointers on
CMP len1, len2 ;Find the shorter
MOVGT len1, len2 ;Get shorter in len1
MOV flags, pc ;Save result
MOV index, #0 ;Init index
.strCmpLp
CMP index, len1 ;End of shorter string?
BEQ strCmpEnd ;Yes so result on lengths
LDRB char1, [str1, index] ;Get a character from each
LDRB char2, [str2, index]
ADD index, index, #1 ;Next index
CMP char1, char2 ;Compare the chars
BEQ strCmpLp ;If equal, next char
;
;Return with result of last character compare
;Store flags so BASIC can read them
;
STR pc,theFlags
LDMFD (sp)!,{str1-char2,pc}
;
;Shorter string exhausted so return with result of
;the comparison between the lengths
;
.strCmpEnd
TEQP flags, #0 ;Get flags from register
;
;Store flags so BASIC can read them
;
STR pc,theFlags
LDMFD (sp)!, {str1-char2,pc}
;
.theFlags
EQUD 0
]
NEXT pass
carryBit = &20000000
zeroBit = &40000000
REPEAT
INPUT'"String 1 ",s1$,"String 2 ",s2$
?buff1=LENs1$ : ?buff2=LENs2$
$(buff1+1)=s1$
$(buff2+1)=s2$
A%=buff1
B%=buff2
CALL strCmp
res = !theFlags
PRINT "String 1 "
IF res AND carryBit THEN PRINT">= "; ELSE PRINT "< ";
PRINT "String 2"
PRINT "String 1 ";
IF res AND zeroBit THEN PRINT"= "; ELSE PRINT"<> ";
PRINT "String 2"
UNTIL FALSE
```

Finding a sub-string

In text-handling applications, we sometimes need to find the occurrence of one string in another. The BASIC function **INSTR** encapsulates this idea.

The call

```
INSTR("STRING WITH PATTERN","PATTERN")
```

will return the integer 13, as the sub-string "`PATTERN`" occurs at character 13 of the first argument.

The routine listed below performs a function analogous to `INSTR`. It takes two arguments - byte-count string pointers - and returns the position at which the second string occurs in the first one. The first character of the string is character 1 (as in BASIC). If the sub-string does not appear in the main string, 0 is returned.

For a change, we use the stack to pass the parameters and return the result. It is up to the caller to reserve space for the result under the arguments, and to 'tidy up' the stack on return.

```
DIM org 400,mainString 40, subString 40
str1 = 0
str2 = 1
result = 2
len1 = 3
len2 = 4
char1 = 5
char2 = 6
index = 7
work = 8
sp = 13
link = 14
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;instr. Finds the occurence of str2 in str1. Arguments on
;the stack. On entry and exit, the stack contains:
;
; result word 2
; str1 word 1
; str2 <-- sp word 0 plus 10 pushed words
;
;str1 is the main string, str2 the substring
;All registers are preserved. Result is 0 for no match
;
.instr
;Save work registers
STMFD (sp)!,{str1-work,link}
LDR str1, [sp, #(work-str1+2+0)*4] ;Get str1 pointer
LDR str2, [sp, #(work-str1+2+1)*4] ;and str2 pointer
MOV work, str1 ;Save for offset calculation
LDRB len1, [str1], #1 ;Get lengths and inc pointers
LDRB len2, [str2], #1
.inLp1
CMP len1, len2 ;Quick test for failure
BLT inFail ;Substr longer than main string
MOV index, #0 ;Index into strings
.inLp2
CMP index, len2 ;End of substring?
BEQ inSucc ;Yes, so return with str2
CMP index, len1
BEQ inNext ;End of main string so next try
LDRB char1, [str1, index] ;Compare characters
LDRB char2, [str2, index]
ADD index, index, #1 ;Inc index
CMP char1, char2 ;Are they equal?
BEQ inLp2 ;Yes, so next char
.inNext
ADD str1, str1, #1 ;Move onto next start in str2
SUB len1, len1, #1 ;It's one shorter now
B inLp1
.inFail
MOV work, str1 ;Make SUB below give 0
.inSucc
SUB str1, str1, work ;Calc. pos. of sub string
STR str1,[sp,#(work-str1+2+2)*4] ;Save it in result
;Restore everything and return
LDMFD (sp)!,{str1-work,pc}
;
;Example of calling instr.
;Note that in order that the STM pushes the
;registers in the order expected by instr, the following
;relationship must exist. str2 < str1 < result
;
.testInstr
ADR str1,mainString ;Address of main string
ADR str2,subString ;Address of substring
STMFD (sp)!, {str1,str2,result,link} ;Push strings and
BL instr ;room for the result. Call instr.
LDMFD (sp)!, {str1,str2,result} ;Load strings & result
MOV R0,result ;Result in r0 for USR function
LDMFD (sp)!,{pc}
;
]
NEXT
REPEAT
INPUT"Main string 1 ",s1$ , "Substring 2 ",s2$
?mainString = LEN s1$
?subString = LEN s2$
$(mainString+1) = s1$
$(subString+1) = s2$
pos = USR testInstr
PRINT "INSTR("""s1$""","""s2$""") =";pos;
PRINT " (";INSTR(s1$,s2$)")"
UNTIL FALSE
```

The Note in the comments is to act as a reminder of the way in which multiple registers are stored. `STM` always saves lower numbered registers in memory before higher numbered ones. Thus if the correct ordering on the stack is to be obtained, register `str2` must be lower than `str1`, which must be lower than `result`. Of course, if this weren't true, correct ordering on the stack could still be achieved by pushing and pulling the registers one at a time.

6.4 Integers

The storage and manipulation of numbers comes high on the list of things that computers are good at. For most purposes, integer (as opposed to floating point or 'real') numbers suffice, and we shall discuss their representation and operations on them in this section.

Integers come in varying widths. As the ARM is a 32-bit machine, and the group one instructions operate on 32-bit operands, the most convenient size is obviously 32-bits. When interpreted as signed quantities, 32-bit integers represent a range of -2,147,483,648 to +2,147,483,647. Unsigned integers give a corresponding range of 0 to 4,294,967,295.

When stored in memory, integers are usually placed on word boundaries. This enables them to be loaded and stored in a single operation. Non word-aligned integer require two **LDR**s or **STR**s to move them in and out of the processor, in addition to some masking operations to 'join up the bits'.

It is somewhat wasteful of memory to use four bytes to store quantities which need only one or two bytes. We have already seen that characters use single bytes to hold an eight-bit ASCII code, and string lengths of up to 255 characters may be stored in a single byte. An example of two-byte quantities is BASIC line numbers (which may be in the range 0..65279 and so require 16 bits).

LDRB and **STRB** enable unsigned bytes to to transferred between the ARM and memory efficiently. There may be occasions, though, when you want to store a signed number in a single byte, i.e. -128 to 127, instead of more usual 0..255. Now **LDRB** performs a zero-extension on the byte, i.e. bits 8..31 of the destination are set to 0 automatically. This means that when loaded, a signed byte will have its range changed to 0..255. To sign extend a byte loaded from memory, preserving its signed range, this sequence may be used:

```
LDRB R0, <address> ;Load the byte
MOV R0, R0, LSL #24 ;Move to bits 24..31
MOV R0, R0, ASR #24 ;Move back with sign
```

It works by shifting the byte to the most significant byte of the register, so that the sign bit of the byte (bit 7) is at the sign bit of the word (bit 31). The arithmetic shift right then moves the byte back again, extending the sign as it does so. After this, normal 32-bit ARM instructions may be performed on the word.

(If you are sceptical about this technique giving the correct signed result, consider eight-bit and 32-bit two's complement representation of numbers. If you examine a negative number, zero and a positive number, you will see that in all cases, bit 7 of the eight-bit version is the same as bits 8..31 of the 32-bit representation.)

The store operation doesn't need any special attention: **STRB** will just store bits 0..7 of the word, and bit 7 will be the sign bit (assuming, of course, that the signed 32-bit number being stored is in the range -128..+127 which a single byte can represent).

Double-byte (16-bit) operands are best accessed using a couple of `LDRB`s or `STRB`s. To load an unsigned 16-bit operand from an byte-aligned address use:

```
LDRB R0, <address>
LDRB R1, <address>+1
ORR R0, R0, R1, LSL #8
```

The calculation of `<address>+1` might require an extra instruction, but if the address of the two-byte value is stored in a base register, pre- or post-indexing with an immediate offset could be used:

```
LDRB R0, [addr, #0]
LDRB R1, [addr, #1]
ORR R0, R0, R1, LSL #8
```

Extending the sign of a two-byte value is similar to the method given for single bytes shown above, but the shifts are only by 16 bits.

To store a sixteen-bit quantity at an arbitrary byte position also requires three instructions:

```
STRB R0, <address>
MOV R0, R0, ROR #8
STRB R0, <address>+1
```

We use `ROR #8` to obtain bits 8..15 in the least significant byte of R0. The number can then be restored if necessary using:

```
MOV R0, R0, ROR #24
```

Multiplication and division

Operations on integers are many and varied. The group one instructions cover a good set of them, but an obvious omission is division. Also, although there is a `MUL` instruction, it is limited to results which fit in a single 32-bit register. Sometimes a 'double precision' multiply, with a 64-bit result, is needed.

Below we present a 64-bit multiplication routine and a division procedure. First, though, let's look at the special case of multiplying a register by a constant. There are several simple cases we can spot immediately. Multiplication by a power of two is simply a matter of shifting the register left by that number of places. For example, to obtain R0*16, we would use:

```
MOV R0, R0, ASL #4
```

as $16=2^4$. This will work just as well for a negative number as a positive one, as long as the result can be represented in 32-bit two's complement. Multiplication by 2^n-1 or 2^n+1 is just as straightforward:

```
RSB R0, R0, R0, ASL #n ;R0=R0*(2^n-1)
ADD R0, R0, R0, ASL #n ;R0=R0*(2^n+1)
```

So, to multiply R0 by 31 (=2^5-1) and again by 5 (=2^2+1) we would use:

```
RSB R0, R0, R0, ASL #5
ADD R0, R0, R0, ASL #2
```

Other numbers can be obtained by factorising the multiplier and performing several shift operations. For example, to multiply by 10 we would multiply by 2 then by 5:

```
MOV R0, R0, R0, ASL #1
ADD R0, R0, R0, ASL #2
```

You can usually spot by inspection the optimum sequence of shift instructions to multiply by a small constant.

Now we present a routine which multiplies one register by another and produces a 64-bit result in two other registers. The registers **lhs** and **rhs** are the two source operands and **dest** and **dest+1** are the destination registers. We also require a register **tmp** for storing temporary results.

The routine works by dividing the task into four separate multiplies. The biggest numbers that **MUL** can handle without overflow are two 16-bit operands. Thus if we split each of our 32-bit registers into two halves, we have to perform:-

```
lhs (low) * rhs (low)
lhs (low) * rhs (high)
lhs (high) * rhs (low)
lhs (high) * rhs (high)
```

These four products then have to be combined in the correct way to produce the final result. Here is the routine, with thanks to Acorn for permission to reproduce it.

```
;
; 32 X 32 bit multiply.
; Source operands in lhs, rhs
; result in dest, dest+1
; tmp is a working register
;
.mul64
MOV tmp, lhs, LSR #16 ;Get top 16 bits of lhs
MOV dest+1, rhs, LSR #16 ;Get top 16 bits of rhs
BIC lhs,lhs,tmp,LSL #16 ;Clear top 16 bits of lhs
BIC rhs,rhs,dest+1,LSL#16 ;Clear top 16 bits of rhs
MUL dest, lhs, rhs ;Bits 0-15 and 16-31
MUL rhs, tmp, rhs ;Bits 16-47, part 1
MUL lhs, dest+1, lhs ;Bits 16-47, part 2
MUL dest+1, tmp, dest+1 ;Bits 32-63
ADDS lhs, rhs, lhs ;Add the two bits 16-47
ADDCS dest+1, dest+1, #&10000 ;Add in carry from above
ADDS dest, dest, lhs, LSL #16 ;Final bottom 32 bits
ADC dest+1,dest+1,lhs,LSR#16 ;Final top 32 bits
```

The worst times for the four `MUL`s are 8 s-cycles each. This leads to an overal worst-case timing of 40 s-cycles for the whole routine, or 5us on an 8MHz ARM.

The division routine we give is a 32-bit by 32-bit signed divide, leaving a 32-bit result and a 32-bit remainder. It uses an unsigned division routine to do most of the work. The algorithm for the unsigned divide works as follows. The quotient (`div`) and remainder (`mod`) are set to zero, and a count initialised to 32. The `lhs` is shifted until its first 1 bit occupies bit 31, or the count reaches zero. In the latter case, `lhs` was zero, so the routine returns straightaway.

For the remaining iterations, the following occurs. The top bit of `lhs` is shifted into the bottom of `mod`. This forms a value from which a 'trial subtract' of the `rhs` is done. If this subtract would yield a negative result, `mod` is too small, so the next bit of `lhs` is shifted in and a 0 is shifted into the quotient. Otherwise, the subtraction is performed, and the remainder from this left in `mod`, and a 1 is shifted into the quotient. When the count is exhausted, the remainder from the division will be left in `mod`, and the quotient will be in `div`.

In the signed routine, the sign of the result is the product of the signs of the operands (i.e. plus for same sign, minus for different) and the sign of the remainder is the sign of the left hand side. This ensures that the remainder always complies with the formula:

a MOD b = a - b*(a DIV b)

The routine is listed below:

```
DIM org 200
lhs = 0
rhs = 1
div = 2
mod = 3
divSgn = 4
modSgn = 5
count = 6
sp = 13
link = 14
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;sDiv32. 32/32 bit signed division/remainder
;Arguments in lhs and rhs. Uses the following registers:
;divSgn, modSgn - The signs of the results
;count - bit count for main loop
;div - holds lhs / rhs on exit, truncated result
;mod - hold lhs mod rhs on exit
;
.sDiv32
STMFD (sp)!, {link}
EORS divSgn, lhs, rhs ;Get sign of div
MOVS modSgn, lhs ;and of mod
RSBMI lhs, lhs, #0 ;Make positive
TEQ rhs, #0 ;Make rhs positive
RSBMI rhs, rhs, #0
BL uDiv32 ;Do the unsigned div
TEQ divSgn, #0 ;Get correct signs
RSBMI div, div, #0
TEQ modSgn, #0 ;and of mod
RSBMI mod, mod, #0
;
;This is just so the BASIC program can
;read the results after the call
;
ADR count, result
STMIA count, {div,mod}
LDMFD (sp)!,{pc} ;Return
;
.uDiv32
TEQ rhs, #0 ;Trap div by zero
BEQ divErr
MOV mod, #0 ;Init remainder
MOV div, #0 ;and result
MOV count, #32 ;Set up count
.divLp1
SUBS count, count, #1 ;Get first 1 bit of lhs
MOVEQ pc, link ;into bit 31. Return if 0
MOVS lhs, lhs, ASL #1
BPL divLp1
.divLp2
MOVS lhs, lhs, ASL #1 ;Get next bit into...
ADC mod, mod, mod ;mod for trial subtract
CMP mod, rhs ;Can we subtract?
SUBCS mod, mod, rhs ;Yes, so do
ADC div, div, div ;Shift carry into result
SUBS count, count, #1 ;Next loop
BNE divLp2
.divErr
MOV pc, link ;Return
;
.result
EQUD 0
EQUD 0
]
NEXT pass
@%=&0A0A
FOR i%=1 TO 6
A%=RND : B%=RND
CALL sDiv32
d%=!result : m%=result!4
PRINTA%" DIV ";B%" = ";d%" (";A% DIV B%")"
PRINTA%" MOD ";B%" = ";m%" (";A% MOD B%")"
PRINT
NEXT i%
```

ASCII to binary conversion

Numbers are represented as printable characters for the benefit of us humans, and stored in binary for efficiency in the computer. Obviously routines are needed to convert between these representations. The two subroutines listed in this section perform conversion of an ASCII string of decimal digits to 32-bit signed binary, and vice versa.

The ASCII-to-binary routine takes a pointer to a string and returns the number represented by the string, with the pointer pointing at the first non-decimal digit.

```
DIM org 200
REM Register assignments
bin = 0
sgn = 1
ptr = 3
ch = 4
sp = 13
link = 14
cr = &0D
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
.testAscToBin
;Test routine for ascToBin
;
STMFD (sp)!,{link} ;Save return address
ADR ptr,digits ;Set up pointer to the string
BL ascToBin ;Convert it to binary in R0
LDMFD (sp)!,{PC} ;Return with result
;
.digits
EQUS "-123456"
EQUB cr
;
;ascToBin. Read a string of ASCII digits at ptr,
;optionally preceded by a + or - sign. Return the
;signed binary number corresponding to this in bin.
;
.ascToBin
STMFD (sp)!,{sgn,ch,link}
MOV bin,#0 ;Init result
MOV sgn,#0 ;Init sign to pos.
LDRB ch,[ptr,#0] ;Get possible + or -
CMP ch,#ASC"+" ;If +,just skip
BEQ ascSkp
CMP ch,#ASC"-" ;If -,negate sign and skip
MVNEQ sgn,#0
.ascSkp
ADDEQ ptr,ptr,#1 ;Inc ptr if + or -
.ascLp
LDRB ch,[ptr,#0] ;Read digit
SUB ch,ch,#ASC"0" ;Convert to binary
CMP ch,#9 ;Make sure it is a digit
BHI ascEnd ;If not,finish
ADD bin,bin,bin ;Get bin*10. bin=bin*2
ADD bin,bin,bin,ASL #2 ;bin=bin*5
ADD bin,bin,ch ;Add in this digit
ADD ptr,ptr,#1 ;Next character
B ascLp
.ascEnd
TEQ sgn,#0 ;If there was - sign
RSBMI bin,bin,#0 ;Negate the result
LDMFD (sp)!,{sgn,ch,pc}
]
NEXT pass
PRINT "These should print the same:"
PRINT $digits ' ;USRtestAscToBin
```

Notice that we do not use a general purpose multiply to obtain **`bin`***10. As this is **`bin`***2*5, we can obtain the desired result using just a couple of **`ADD`**s. As with many of the routines in this book, the example above illustrates a technique rather than providing a fully-fledged solution. It could be improved in a couple of ways, for example catching the situation where the number is too big, or no digits are read at all.

To convert a number from binary into a string of ASCII characters, we can use the common divide and remainder method. At each stage the number is divided by 10. The remainder after the division is the next digit to print, and this is repeated until the quotient is zero.

Using this method, the digits are obtained from the right, i.e. the least significant digit is calculated first. Generally we want them in the opposite order - the most significant digit first. To reverse the order of the digits, they are pushed on the stack as they are obtained. When conversion is complete, they are pulled off the stack. Because of the stack's 'last-in, first-out' property, the last digit pushed (the leftmost one) is the first one pulled back.

```
buffSize=12
DIM org 200,buffer buffSize
REM Register allocations
bin = 0
ptr = 1
sgn = 2
lhs = 3
rhs = 4
div = 5
mod = 6
count = 7
len = 8
sp =13
link = 14
cr=&0D
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;binToAscii - convert 32-bit two's complement
;number into an ASCII string.
;On entry,ptr holds the address of a buffer
;area in which the ASCII is to be stored.
;bin contains the binary number.
;On exit,ptr points to the first digit (or -
;sign) of the ASCII string. bin = 0
;
.binToAscii
STMFD (sp)!,{ptr,sgn,lhs,rhs,div,mod,link}
MOV len,#0 ;Init number of digits
MOV mod,#ASC"-"
TEQ bin,#0 ;If -ve,record sign and negate
STRMIB mod,[ptr],#1
RSBMI bin,bin,#0
.b2aLp
MOV lhs,bin ;Get lhs and rhs for uDiv32
MOV rhs,#10
BL uDiv32 ;Get digit in mod,rest in div
ADD mod,mod,#ASC"0" ;Convert digit to ASCII
STMFD (sp)!,{mod} ;Save digit on the stack
ADD len,len,#1 ;Inc string length
MOVS bin,div ;If any more,get next digit
BNE b2aLp
;
.b2aLp2
LDMFD (sp)!,{mod} ;Get a digit
STRB mod,[ptr],#1 ;Store it in the string
SUBS len,len,#1 ;Decrement count
BNE b2aLp2
MOV mod,#cr ;End with a CR
STRB mod,[ptr],#1
LDMFD (sp)!,{ptr,sgn,lhs,rhs,div,mod,pc}
;
;
.uDiv32
STMFD (sp)!,{count,link}
TEQ rhs,#0 ;Trap div by zero
BEQ divErr
MOV mod,#0 ;Init remainder
MOV div,#0 ;and result
MOV count,#32 ;Set up count
.divLp1
SUBS count,count,#1 ;Get first 1 bit of lhs
MOVEQ pc,link ;into bit 31. Return if 0
MOVS lhs,lhs,ASL #1
BPL divLp1
.divLp2
MOVS lhs,lhs,ASL #1 ;Get next bit into...
ADC mod,mod,mod ;mod for trial subtract
CMP mod,rhs ;Can we subtract?
SUBCS mod,mod,rhs ;Yes,so do
ADC div,div,div ;Shift carry into result
SUBS count,count,#1 ;Next loop
BNE divLp2
.divErr
LDMFD (sp)!,{count,pc}
]
NEXT pass
A%=-12345678
B%=buffer
CALL binToAscii
PRINT"These should be the same:"
PRINT;A% ' $buffer
```

As there is no quick way of doing a divide by 10, we use the **`uDiv32`** routine given earlier, with **`lhs`** and **`rhs`** set-up appropriately.

6.5 Floating point

Many real-life quantities cannot be stored accurately in integers. Such quantities have fractional parts, which are lost in integer representations, or are simply too great in magnitude to be stored in an integer of 32 (or even 64) bits.

Floating point representation is used to overcome these limitations of integers. Floating point, or FP, numbers are expressed in ASCII as, for example, 1.23, which has a fractional part of 0.23, or 2.345E6, which has a fractional part and an exponent. The exponent, the number after the E, is the power of ten by which the other part (2.345 in this example) must be multiplied to obtain the desired number. The 'other part' is called the mantissa. In this example, the number is $2.345*10^6$ or 2345000.

In binary, floating point numbers are also split into the mantissa and exponent. There are several possible formats of floating point number. For example, the size of the mantissa, which determines how many digits may be stored accurately, and the size of the exponent, determining the range of magnitudes which may be represented, both vary.

Operations on floating point numbers tend to be quite involved. Even simple additions require several steps. For this reason, it is often just as efficient to write in a high-level language when many FP calculations are performed, and the advantage of using assembler is somewhat diminished. Also, most machines provide a library of floating point routines which is available to assembly language programs, so there is little point in duplicating them here.

We will, however, describe a typical floating point format. In particular, the way in which BBC BASIC stores its floating point values is described.

An FP number in BBC BASIC is represented as five bytes. Four bytes are the mantissa, and these contain the significant digits of the number. The mantissa has an imaginary binary point just before its most significant bit. This acts like a decimal point, and digits after the point represents successive negative powers of 2. For example, the number 0.101 represents 1/2 + 0/4 + 1/8 or 5/8 or 0.625 in decimal.

When stored, FP numbers are in normalised form. This means that the digit immediately after the point is a 1. A normalised 32-bit mantissa can therefore represent numbers in the range:

0.10000000000000000000000000000000 to

0.11111111111111111111111111111111

in binary which is 0.5 to 0.9999999998 in decimal.

To represent numbers outside this range, a single byte exponent is used. This can be viewed as a shift count. It gives a count of how many places the point should be moved to the right to obtain the desired value. For example, to represent 1.5 in binary floating point, we would start with the binary value 1.1, i.e. 1 + 1/2. In normalised form, this is .11. To obtain the original value, we must move the point one place to the right. Thus the exponent is 1.

We must be able to represent left movements of the point too, so that numbers smaller than 0.5 can be represented. Negative exponents represent left shifts of the point. For example, the binary of 0.25 (i.e. a quarter) is 0.01. In normalised form this is 0.1. To obtain this, the point is moved one place to the left, so the exponent is -1.

Two's complement could be used to represent the exponent as a signed number, but is is more usual to employ an excess-128 format. In this format, 128 is added to the actual exponent. So, if the exponent was zero, representing no shift of the point from the normalised form, it would be stored as 128+0, or just 128. A negative exponent, e.g. -2, would be stored as 128-2, or 126.

Using the excess-128 method, we can represent exponents in the range -128 (exponent stored as zero) to +127 (exponent stored as 255). Thus the smallest magnitude we can represent is 0.5/(2^{128}), or 1.46936794E-39. The largest number 0.9999999998*(2^{127}), or 1.701411834E38

So far, we have not mentioned negative mantissas. Obviously we need to represent negative numbers as well as positive ones. A common 'trick', and one which BBC BASIC uses, is to assume that the most significant bit is 1 (as numbers are always in normalised form) and use that bit position to store the sign bit: a zero for positive numbers, and 1 for negative numbers.

We can sum up floating point representation by considering the contents of the five bytes used to store them in memory.

byte 0	LS byte of mantissa
byte 1	Second LSB of mantissa
byte 2	Second MSB of mantissa
byte 3	MS byte of mantissa. Binary point just to the left of bit 7
byte 4	Exponent, excess-128 form

Consider the number 1032.45. First, we find the exponent, i.e. by what power of two the number must be divided to obtain a result between 0.5 and 0.9999999. This is 11, as $1032.45/(2^{11})=0.504125976$. The mantissa, in binary, is: 0.10000001 00001110 01100110 01100110 or, in hex 81 0E 66 66. So, we would store the number as:

byte 0	LSB = &66
byte 1	2rd LSB = &66
byte 2	2nd MSB = &0E
byte 3	MSB = &81 AND &7F = &01
byte 4	exponent = 11+128 = &8B

This are the five bytes you would see if you executed the following in BASIC:

```
DIM val 4 :REM Get five bytes
|val=1032.45 :REM Poke the floating point value
FOR i=0 TO 4 :REM Print the five bytes
PRINT ~val?i
NEXT i
```

Having described BBC BASIC's floating point format in some detail, we now have to confess that it is not the same as that used by the ARM floating point instructions. It is, however, the easiest to 'play' with and understand.

The ARM floating point instructions are extensions to the set described in Chapter Three. They follow the IEEE standard for floating point. The implementation of the instructions is initially by software emulation, but eventually a much faster hardware unit will be available to execute them. The full ARM FP instruction set and formats are described in Appendix B.

6.6 Structured types

Sometimes, we want to deal with a group of values instead of just single items. We have already seen one example of this - strings are groups, or arrays, of characters. Parameter blocks may also be considered a structured type. These correspond to records in Pascal, or structures in C.

Array basics

We define an array as a sequence of objects of the same type which may be accessed individually. An index or subscript is used to denote which item in an array is of interest to us. You have probably come across arrays in BASIC. The statement:

```
DIM value%(100)
```

allocates space for 101 integers, which are referred to as **`value%(0)`** to **`value%(100)`**. The number in brackets is the subscript. In assembler, we use a similar technique. In one register, we hold the base address of the array. This is the address of the first item. In another register is the index. The ARM provides two operations on array items: you can load one into the processor, or store one in memory from the processor.

Let's look at a concrete example. Suppose register R0 holds the base address of an array of four-byte integers, and R1 contains the subscript of the one we want to load. This instruction would be used:

```
LDR R2, [R0, R1, LSL #2]
```

Note that as R1 holds the index, or subscript, of the element, we need to multiply this by four (using the **`LSL #2`**) to obtain the actual offset. This is then added to the base address in R0, and the word at this address is loaded into R2. There is no **`!`** at the end, so R0 is not affected by write-back.

If the array was of byte-sized objects, the corresponding load operation would be:

```
LDRB R2, [R0, R1]
```

This time there is no scaling of the index, as each item in the array occupies only one byte.

If you are accessing an array of objects which are some inconvenient size, you will need to scale the index into a byte offset before loading the item. Moreover, further adjustment might be needed to ensure that the load takes place on word boundaries.

To illustrate the problem of loading odd-sized objects from arbitrary alignments, we give a routine below to load a five byte floating point value into two registers, mant (the mantissa) and exp (exponent). The number is stored in memory as a four-byte mantissa followed by a single-byte exponent. An array of these objects could use two words each, the first word holding the mantissa and the LSB of the second word storing the mantissa. It would then be a simple job to load two words from a word-aligned address, and mask out the unused part of the exponent word.

Using two whole words to store five bytes is wasteful when many elements are used (e.g. an array of 5000 numbers would waste 15000 bytes), so we obviously have to store the numbers contiguously. It is quite likely, therefore, that the mantissa and exponent will be aligned in a way which makes simple **`LDR`** instructions insufficient to load the number into registers.

Consider the value stored starting at address &4001:

&4000	****************
&4001	LSB of mantissa
&4002	2nd LSB of mantissa
&4003	2nd MSB of mantissa
&4004	MSB of mantissa
&4005	Exponent
&4006	****************
&4007	****************

Three bytes of the number are held in the three most significant bytes of one word; the last two bytes are stored at the start of the next word.

The technique we will use is to load the two words which the value straddles, then shift the registers appropriately so that **mant** contains the four bytes of the mantissa in the correct order, and the LSB of **exp** contains the exponent byte.

On entry to the code, **base** contains the base address of the array, and **off** holds the index (in elements rather than bytes).

```
ADD off, off, off, LSL #2 ;offset=5*offset
ADD base, base, off ;base=base+5*n
AND off, base, #3 ;Get offset in bytes
BIC base, base, #3 ;Get lower word address
LDMIA base, {mant,exp} ;Load the two words
MOVS off, off, LSL #3 ;Get offset in bits
MOVNE mant, mant, LSR off ;Shift mantissa right
RSBNE base, off, #32 ;Get shift for exponent
ORRNE mant, mant, exp,LSL base ;OR in mantissa
.part
MOVNE exp, exp, LSR off ;Get exponent is LSB
AND exp, exp, #&FF ;Zero high bytes of exp
```

Notice we use **LDMIA** to load the two word. The code assumes that the register number of **mant** is lower than **exp**, so that the words are loaded in the correct order.

The last four instructions are all conditional on the byte offset being non-zero. If it is zero, the value was on a word boundary, and no shifting is required.

Arrays of strings

We have already noted that a string is an array of characters. Sometimes, we want an array of strings, i.e. an array of character arrays. For example, the BASIC declaration:

```
DIM name$(10)
```

gives us an array of 11 strings, **name$(0)** to **name$(10)**. How do we organise such a structure in assembly language? There are two solutions. If each of the strings is to have a fixed length, the easiest way is to store the strings contiguously in memory. Suppose we wanted ten strings of ten characters each. This would obviously occupy 100 bytes. If the base address of these bytes is stored in **base**, and the subscript of the string we want in **sub**, then this code would be used to calculate the address of the start of string sub:

```
ADD base,base,sub,LSL #3 ;Add sub*8
ADD base,base,sub,LSL #1 ;Add sub*2
```

After these instructions, **base** would point to (old) **base+10*sub**, i.e. the start character of string number **sub**.

Storing all the characters of every string can be wasteful if there are many strings and they can have widely varying lengths. For example, if a lot of the strings in an array contain no characters for a lot of the time, the storage used for them is wasted. The solution we present is to use an array of string information blocks instead of the actual characters.

A string information block (SIB) is a structure which describes the address and length of a string. Unlike the string itself, it is a fixed length, so is more amenable to storing in an array. BASIC uses SIBs to describe its strings. When you **DIM** a string array in BASIC, the only storage allocated is for the appropriate number of SIBs. No character storage is allocated until you start assigning to the strings.

The format of a BASIC SIB is:

bytes 0 to 3 address of the string

byte 4 current length of the string

When you **DIM** an array, all entries have their length bytes set to zero. As soon as you assign something to the string, BASIC allocates some storage for it and fills in the address part. The way in which BASIC allocates storage for strings is interesting in its own right, and is described in sectionÊ6.7

To illustrate how we might use an array of SIBs, the routine below takes a base address and a subscript, and prints the contents of that string.

```
DIM org 200
sub = 0
base = 1
len = 2
p1 = 3
p2 = 4
argPtr = 9 : REM BASIC's pointer to CALL arguments
WriteC = 0
sp = 13
link = 14
```

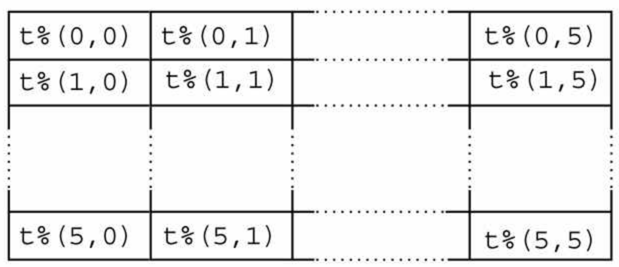
```
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;Print base$(sub) where base points to the start of
;an array of five-byte block of the format:
; 0..3 pointer to string
; 4 length of string (see section 4.7)
;and sub is the subscript 0..n of the desired
;string. The SIB may be at any byte alignment.
;
.pStr
;Get address of SIB = base+sub*5
ADD base,base,sub,LSL #2 ;base=base+sub*4
ADD base,base,sub ;base=base+sub*1
LDRB len,[base,#4] ;Get string len
TEQ len,#0 ;If zero,nothing to do
MOVEQ pc,link
;
;Arbitrary alignment load of four-byte pointer into
;p1. Address of pointer in base
;
AND sub,base,#3 ;Get offset in bytes
BIC base,base,#3 ;Get lower word address
LDMFD base,{p1,p2} ;Load the two words
MOVS sub,sub,LSL #3 ;Get offset in bits
MOVNE p1,p1,LSR sub ;Shift lower word
RSBNE sub,sub,#32 ;Get shift for high word
ORRNE p1,p1,p2,LSL sub ;ORR in the high word
;
;Now print the string. NB len > 0 so we can test at the
;end of the loop
;
.pStrLp
LDRB R0,[p1],#1 ;Load a character into R0
SWI WriteC ;Print it
SUBS len,len,#1 ;Decrememt the count
BNE pStrLp ;Loop if more
.endPStr
MOV pc,link ;Return
;
;testPstr. This takes a subscipt in sub (r0)
;and a BASIC string array CALL parameter
;and prints the appropriate string
;
.testPstr
STMFD (sp)!,{link}
LDR base,[argPtr] ;Load the address of the
BL pStr ;CALL parm. Print the string
LDMFD (sp)!,{pc}
]
NEXT pass
DIM a$(10)
FOR i%=0 TO 10
a$(i%)=STR$RND
NEXT i%
FOR i%=0 TO 10
A%=i%
PRINT"This should say "a$(i%)": ";
CALL testPstr,a$(0)
```

```
PRINT
NEXT i%
```

Multi-dimensional arrays

A single list of items is not always sufficient. It may be more natural to store items as a table of two or even more dimensions. A BASIC array declaration of two dimensions is:

```
DIM  t%(5,5)
```



This allocates space for a matrix of 36 integers:

We can use such arrays in assembly language by imagining the rows laid out end to end in memory. Thus the first six words of the array would hold **t%(0,0)** to **t%(0,5)**. The next six would store **t%(1,0)** to **t%(1,5)** and so on. To calculate the address of **t%(a,b)** we use **base+a*lim+b**, where **lim** is the limit of the second subscript, which is 6 in this case.

The routine below takes **base**, **sub1** and **sub2**. It calculates the address of the required element, assuming each element is 4 bytes (e.g. an integer) and that there are 16 elements per row.

```
ADD sub1, sub2, sub1, LSL #4 ;sub1=sub2+16*sub1
ADD base, base, sub1, LSL #2 ;base=base+sub1*4
```

Once **base** has been calculated, the usual instruction could be used to load the integer at that address. In the more general case, the number of elements per row would be stored in a register, and a general multiply used to multiply **sub1** by this limit.

Bit fields

We end this discussion of the basic types of data by reverting to the lowest level of representation: the bit. We have seen how sequences of bits, usually in multiples of eight, may be used to represent characters, integers, pointers and floating point numbers. However, single bits may usefully store information too.

One binary digit can represent two values: 0 or 1. Often this is all we need to distinguish between two events. A bit used to represent a choice such as yes/no, true/false, success/failure is called a flag. We already know of several useful flags: the result flags in the status register. The V flag for example represents overflow/no overflow.

It is common to find in many programs a set of flags which could be grouped together and stored in a single byte or word. Consider a text editor. There might be flags to indicate insert/overtype mode, justify/no justify mode, help displayed/no help, case sensitive/insensitive searches. Each of these would be assigned a bit in the flag byte. The value given to a flag is that of the bit in that position. Thus we might define the examples above as:

```
insMode = &01
juMode = &02
hlpMode = &04
snsMode = &08
```

There are four main operations on flags: set, clear, invert and test. To set a flag, the **`ORR`** operation is used. For example, to set the **`insMode`** flag of register **`flags`**:

```
ORR flags, flags, #insMode
```

Clearing a flag is achieved using **`BIC`**, bit clear. To clear both **`hlpMode`** and **`snsMode`**:

```
BIC flags, flags, #hlpMode OR snsMode
```

To invert a flag we use the **`EOR`** operation. This is often called 'toggling a flag', because applying the operation repeatedly has the effect of switching the flag between the two values.

To invert the **`juMode`** flag:

```
EOR flags, flags, #juMode
```

Finally, to test the state of a flag, we use **`TST`**. This performs an **`AND`** operation, and the result is zero if the flag is cleared, and non-zero if it is set:

```
TST flags, #insMode
```

tests the insert mode flag. If you test more than one flag in a single **`TST`**, the result is non-zero if any of the flags are set, and zero if all of them are cleared. You can also use **`TEQ`** to see if all of a set of flags are set and the rest are cleared. For example,

```
TEQ flags, #insMode OR juMode
```

sets the Z flag if **`insMode`** and **`juMode`** are set and **`hlpMode`** and **`snsModer`** are cleared. Otherwise Z is cleared.

As 32 bits are held in a single word, arrays of flags can be stored very efficiently. To illustrate this, we show *Byte* magazine's well-known Sieve of Eratosthenes program. This benchmark is often used to test the speed of a few simple types of operation, for example when various compilers for a language are being compared. The purpose of the program is to find prime numbers using a technique attributed to the eponymous Greek mathematician.

The Sieve technique works as follows. Start with an array of flags, one for each of the integers from 2 to the maximum prime to be found. All of the flags are set to 'true' initially, indicating that any number could be a prime. Go through the array looking for the next 'true' bit. The number corresponding to this bit is prime, so output it. Then go through all of the multiples of this number, setting their bits to 'false' to eliminate them from the enquiry. Repeat this process till the end of the array.

Byte's version of the Sieve algorithm is slightly different as it starts from the supposition that all even numbers (except for two) are non-prime, so they are not even included. For comparison, we give a BBC BASIC version of the algorithm first, then the ARM assembler one. The BASIC version is shown overleaf.

Each flag is stored in a byte, and the storage for the array of **size%** bytes is obtained using a **DIM** statement. Notice that the program doesn't actually print the primes it discovers, because the idea of the benchmark is to test the speed of things like array accesses, not printing. The place in the program where the prime would be printed is shown in a **REM**.

```
size% = 8190
DIM flags% size%
count% = 0
FOR i% = 0 TO size%
flags%?i% = TRUE
NEXT i%
FOR i% = 0 TO size%
IF flags%?i% THEN
prime% = i%+i%+3 : REM PRINT prime%
k% = prime%+i%
WHILE k% <= size%
flags%?k% = FALSE
k% += prime%
ENDWHILE
count% += 1
ENDIF
NEXT i%

PRINT count%
```

In the ARM assembler version, we are eight times more efficient in the storage of the flags, and use a single bit for each one. Thus 32 flags can be stored in each word of memory. The ARM version is shown below:

```
DIM org 2000
REM Register allocations
count = 0
ptr = 1
i = 2
mask = 3
base = 4
prime = 5
k = 6
tmp = 7
size = 8
iter = 9
link = 14
SIZE = 8190
iterations = 10
FOR pass=0 TO 2 STEP 2
P%=org
[opt pass
;Sieve of Eratosthenes in ARM assembler
;The array of SIZE flags is stored 32 per word from
;address 'theArray'. The zeroth element is stored at bit
;0 of word 0,the 32nd element at bit 0 of word 1, and so
;on. 'Base' is word-aligned
;
;Registers:
; count holds the number of primes found
; mask used as a bit mask to isolate the required flag
; ptr used as a general pointer/offset into the array
; i used as a counting register
; size holds the value SIZE for comparisons
; base holds the address of the start of the array
; prime holds the current prime number
; k holds the current entry being 'crossed out'
; tmp is a temporary
; iter holds the count of iterations
;
.sieve
MOV iter,#iterations
.mainLoop
ADR base,theArray
MVN mask,#0 ;Get &FFFFFFFF, ie all bits set
MOV size,#SIZE AND &FF;Load size with SIZE in 2 steps
ORR size,size,#SIZE AND &FF00
;
;Initialise the array to all 'true'. First store the
;complete words (SIZE DIV 32 of them), then the partial
;word at the end
;
MOV i,#SIZE DIV 32 ;Loop counter = number of words
MOV ptr,base ;Start address for initing array
.initLp
STR mask,[ptr],#4 ;Store a word and update pointer
SUBS i,i,#1 ;Next word
BNE initLp
LDR tmp,[ptr] ;Get last, incomplete word
MOV mask,mask,LSR #32-SIZE MOD 32 ;Clear top bits
ORR tmp,tmp,mask ;Set the bottom bits
STR tmp,[ptr] ;Store it back
MOV i,#0 ;Init count for main loop
MOV count,#0
.lp
MOV ptr,i,LSR #5 ;Get word offset for this bit
MOV mask,#1 ;Get mask for this bit
AND tmp,i,#31 ;Bit no. = i MOD 32
MOV mask,mask,LSL tmp
LDR tmp,[base,ptr,LSL #2] ;Get the word
ANDS tmp,tmp,mask ;See if bit is set
BEQ nextLp ;No so skip
ADD prime,i,i ;Get prime
ADD prime,prime,#3
ADD k,i,prime ;Get intial k
ADD count,count,#1 ;Increment count
.while
CMP k,size ;While k<=size
BGT nextLp
MOV ptr,k,LSR #5 ;Get word for flags[k]
MOV mask,#1
AND tmp,k,#31
MOV mask,mask,LSL tmp
LDR tmp,[base,ptr,LSL #2]
BIC tmp,tmp,mask ;Clear this bit
STR tmp,[base,ptr,LSL #2] ;Store it back
ADD k,k,prime ;Do next one
B while
.nextLp
ADD i,i,#1 ;Next i
CMP i,size
BLE lp
SUBS iter,iter,#1
BNE mainLoop
MOV pc,link ;Return after iter iterations.
;
.theArray
]
REM Reserve the bytes for the array
P%=P%+SIZE/8
NEXT
REM Time 10 iterations, as in Byte
TIME=0
primes = USR sieve
T%=TIME
PRINT"It took ";T%/100" seconds for ";iterations" loops."
```

Notice the sequence which obtains the mask and offset for a given bit in the array occurs twice. The first step is to find the word offset of the word which contains the desired element. There are 32 flags in a word, so the word containing flag `i` is `i DIV 32` words from the start of the array. The division by 32 is performed using a right shift by five bits. Next, the position of the desired flag within the word is needed. This is simply `i MOD 32`, which is obtained using `i AND 31` in the assembler. A mask, which starts off at bit 0, is shifted by (`i MOD 32`) places to the left to obtain the correct mask. Finally, to load the word containing the flag, a scaled indexed load of the form:

```
LDR reg,[base,offset,LSL #2]
```

is used, the `LSL #2` scaling the word offset which we calculated into a (word-aligned) byte address.

The difference in the speed of the BASIC and assembler versions is quite dramatic. BASIC takes 6.72 seconds to perform one iteration of the program. Assembler takes 0.73 seconds to perform *ten* of them, which makes it over 90 times faster. A version in assembler which more closely mirrors the BASIC version, with one byte per flag, takes 0.44 seconds for ten iterations.

6.7 Memory allocation

Some of the examples of ARM assembler we have already given rely on memory being available to store data. For example, strings are generally referenced using a pointer to the characters in memory, and arrays are treated in the same way. In a program that manipulates a lot of data, some way of managing memory must be provided. For example, a text editor needs to be able to allocate space to hold the text that the user types, and the BASIC interpreter needs to allocate space for program lines, variables etc.

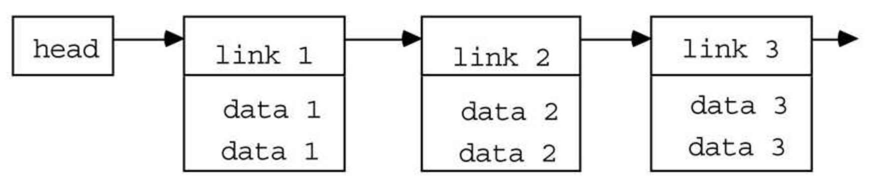
The facilities provided by the operating system for the allocation of memory vary greatly from machine to machine. The UNIX operating system, for example, provides some useful 'library' routines for allocating a given number of bytes, freeing an area so that it can be re-used later, and extending an area already allocated. On the other hand, a simple operating system such as the environment provided by an ARM co-processor connected to a BBC Micro might just hand the program a large chunk of memory and leave the management of it entirely to the program.

In this section, we illustrate a simple way in which memory allocation may be implemented, assuming that the program is just handed a large 'pool' of space by the operating system.

Linked lists

In memory allocation, a structure known as the linked list is often found. A linked list of objects can be regarded as an array where the elements are not contiguous in memory. Instead, each element contains explicit information about the address of the next element. This information is known as the link field of the element. The last item in the list usually has a special link field value, e.g. zero, to mark it as the end, or may contain a link back to the first element (this is called a circular linked list).

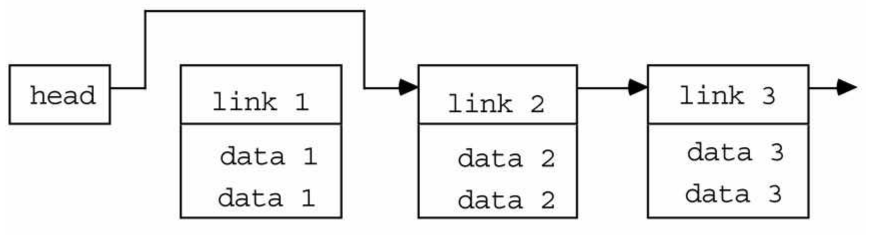
Linked lists can be illustrated pictorially using boxes and arrows. For example, consider a linked list of items which contain two words of information in addition to the link. A list of them might be drawn like this:



The 'head' is a pointer to the first element. This could be stored in a memory location, or in a register in the ARM. The first field in each element is the link pointer, and this is followed by two data words. The pointer for the third element does not point anywhere; it marks the end of the list.

There are many operations that can be performed on linked lists, and large sections of many books on algorithms and data structures are devoted to them. However, we are only interested in a couple of simple operations here: removing an item from the front of the list, and adding a new item to the front of the list.

To remove an item from the front, we just need to replace the head pointer with the link from the first item. When this is done, the list looks like this:



Notice that item number one is now inaccessible through the list, so in order to use it, a pointer to it must be 'remembered' somewhere. Notice also that if the item removed was the last one, the head pointer would be given the end of list value, and would not point at anything. This is known as an empty list.

To insert an item at the front of the list, two actions are required. First, the head pointer is copied into the link field of the item to be inserted. Then, the head pointer is changed to point at this new item.

With this simple level of understanding of linked lists, we can now describe how they are used in memory allocation schemes.

String allocation

The allocation scheme presented is very similar to the one BBC BASIC uses to allocate space for its string variables, and so is suitable for that type of application. Operations which a string allocator must perform are:

Allocate an area. Given a length in bytes, return a pointer to an area where this number of bytes may be stored.

Free an area. Given a length in bytes, and a pointer, free the area of memory so that it can be used by another string when required.

Strings in BBC BASIC may be between 0 and 255 bytes long. The allocator always works in terms of complete words, so strings may occupy between 0 and 64 words. Recall from the discussion of string information blocks earlier that the length is stored along with the address of the characters which make up the string. From this length byte, the number of words required to hold the string can be deduced:

```
words = (len-1) DIV 4 + 1
```

The area of memory BASIC uses for its string storage is called the heap. A word-aligned pointer, which we will call `varTop`, holds the address of the next free byte on the heap. The upper limit on the heap is set by the stack, which grows down from the top of memory.

A very simple memory allocation scheme using the heap would work as follows. To allocate n bytes, calculate how many words are needed, then add this to `varTop`. If this is greater than the stack pointer, SP, give a '`No room`' error. Otherwise, return the old value of `varTop` as the pointer to free space, and update `varTop` by the appropriate number of bytes. To free space, do nothing.

This scheme is clearly doomed to failure in the long run, because memory can never be freed, and so eventually `varTop` could reach the stack and a '`No room`' error be generated. To solve this, BASIC has a way of 'giving back' storage used by strings. There are 64 linked lists, one for each possible number of words that a string can occupy. When a request for n bytes is made, a check is made on the appropriate linked list. If this is not empty, the address of the first item in the list is returned, and this is removed from the list. If the list is empty, the storage is taken from `varTop` as described above. To free n bytes, the area being freed is simply added to the front of the appropriate linked list.

The algorithms for allocate, free and initialise are shown below in BASIC.

```
DEF PROCinit
EMPTY = 0
DIM list(64)
FOR i=1 TO 64
list(i) =EMPTY
NEXT i
varTop = <initial value>
ENDPROC
DEF FNallocate(n)
IF n=0 THEN =0
words = (n-1) DIV 4 + 1
IF list(words) <> EMPTY THEN
addr = list(words)
```

```
list(words) = !list(words)
ELSE
IF varTop + 4*words > SP THEN ERROR 0,"No room"
addr = varTop
varTop += 4*words
ENDIF
= addr
DEF PROCfree(n,addr)
IF n=0 THEN ENDPROC
words = (n-1) DIV 4 + 1
!addr = list(words)
list(words) = addr
ENDPROC
```

The ARM assembler versions of these routines rely on a register called **workSpace**, which always contains the address of the start of a fixed workspace area. In this example, the first word of **workSpace** holds the current value of **varTop**, and the next 64 words are the pointers to the free lists. Another register, **heapLimit**, is assumed to always hold the upper memory limit that the allocater can use. Here are the ARM versions of the three routines.

```
heapSize = 1000
DIM org 600,heap heapSize-1
addr = 0
n = 1
offset = 2
words = 3
tmp = 4
heapLimit = 5
workSpace = 6
sp = 13
link = 14
NULL = 0
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;Init. Intialise the memory allocation system
;It initialises varTop and sets the 64 linked list
;pointers to 'NULL'
.init
STMFD (sp)!,{link}
ADR workSpace,ws ;Init workspace pointer
BL getVarTop ;Get varTop in tmp.
STR tmp,[workSpace] ;Save it
MOV tmp,#NULL ;Init the pointers
MOV offset,#64 ;Word offset into workSpace
.initLp
STR tmp,[workSpace,offset,LSL #2]
SUBS offset,offset,#1
BNE initLp
LDMFD (sp)!,{PC} ;Return
;
;Alloc. Allocate n bytes, returning address of
;memory in addr, or EMPTY if no room
.alloc
SUBS words,n,#1 ;Return immediately for n=0
MOVMI addr,#NULL
MOVMI PC,link
MOV words,words,LSR #2 ;Divide by four
ADD words,words,#1 ;Plus one
;Get pointer for this list
LDR addr,[workSpace,words,LSL#2]
CMP addr,#NULL ;Is it empty?
LDRNE tmp,[addr] ;No, so unlink
STRNE tmp,[workSpace,words,LSL#2]
MOVNE PC,link ;And return
;Empty list so allocate from varTop
LDR addr,[workSpace]
ADD tmp,addr,words,LSL #2 ;Check for no room
CMP tmp,heapLimit
STRLT tmp,[workSpace] ;Update vartop and return
MOVGE addr,#NULL ;Return NULL if no room
MOV PC,link
;
;free. Take a size (in n) and address (in addr)
;of a string, and link it into the appropriate
;free list.
.free
SUBS words,n,#1 ;Return if for n=0
MOVMI PC,link
MOV words,words,LSR #2 ;Divide by four
ADD words,words,#1 ;Plus one
;Get current head pointer for this size
LDR tmp,[workSpace,words,LSL #2]
STR tmp,[addr] ;Store it in new block
;Update head to point to new block
STR addr,[workSpace,words,LSL #2]
MOV PC,link ;Return
;
;Set tmp to point to 'heap' area
;and set up upper limit of heap
.getVarTop
ADR tmp,heap
ADD heapLimit,tmp,#heapSize
MOV PC,link
;
.ws EQUD 0 ;Slot for varTop pointer
]
REM Reserve space for 64 pointers = 256 bytes
P%=P%+256
NEXT pass
```

The way in which `varTop` is initialised depends on the system. In BASIC, for example, `varTop` is initialised to the value of `LOMEM` whenever a `CLEAR`-type operation is performed. `LOMEM` itself is usually set to the top of the BASIC program, but can be altered using an assignment. These three routines show that a relatively small amount of code can perform quite sophisticated memory allocation.

Summary

We have seen that most types of data may be loaded into ARM registers and processed using short sequences of instructions. Simple items may be stored along with the program, but only if the program is executing in RAM. ROM programs may only access fixed tables of data within the program area. Other data must be accessed through pointer registers, using memory allocated by the operating system. Data should be accessed in a position independent manner.