5. Assembly programming principles

The previous chapters have covered the ARM instruction set, and using the ARM assembler. Now we are in a position to start programming properly. Since we are assuming you can program in BASIC, most of this chapter can be viewed as a conversion course. It illustrates with examples how the programming techniques that you use when writing in a high-level language translate into assembler.

5.1 Control structures

Some theory

A program is made up of instructions which implement the solution to a problem. Any such solution, or algorithm, may be expressed in terms of a few fundamental concepts. Two of the most important are program decomposition and flow of control.

The composition of a program relates to how it is split into smaller units which solve a particular part of the problem. When combined, these units, or sub-programs, form a solution to the problem as a whole. In high-level languages such as BASIC and Pascal, the procedure mechanism allows the practical decomposition of programs into smaller, more manageable units. Down at the assembly language level, subroutines perform the same function.

Flow of control in a program is the order in which the instructions are executed. The three important types of control structure that have been identified are: the sequence, iteration, and decision.

An instruction sequence is simply the act of executing instructions one after another in the order in which they appear in the program. On the ARM, this action is a consequence of the PC being incremented after each instruction, The second type of control flow is decision: the ability to execute a sequence of instructions only if a certain condition holds (e.g. **IF…THEN…**). Extensions of this are the ability to take two separate, mutually exclusive paths (**IF**...**THEN**...**ELSE**...), and a multi-way decision based on some value (**ON**...**PROC**...). All of these structures are available to the assembly language programmer, but he has to be more explicit about his intentions.

Iteration means looping. Executing the same set of instructions over and over again is one of the computer's fortes. High-level languages provide constructs such as **REPEAT**..**UNTIL** and **FOR**...**NEXT** to implement iteration. Again, in assembler you have to spell out the desired action a little more explicitly, using backward (perhaps conditional) branches.

Some practice

Having talked about program structures in a fairly abstract way, we now look at some concrete examples. Because we are assuming you have some knowledge of BASIC, or similar high-level language, the structures found therein will be used as a starting point. We will present faithful copies of **IF**...**THEN**...**ELSE**, **FOR**...**NEXT** etc. using ARM assembler. However, one of the advantages of using assembly language is its versatility; you shouldn't restrict yourself to slavishly mimicking the techniques you use in BASIC, if some other more appropriate method suggests itself.

Position-independence

Some of the examples below (for example the **ON**...**PROC** implementation using a branch table) may seem slightly more complex than necessary. In particular, addressing of data and routines is performed not by loading addresses into registers, but by performing a calculation (usually 'hidden' in an **ADR** directive) to obtain the same address. This seemingly needless complexity is due to a desire to make the programs position-independent.

Position-independent code has the property that it will execute correctly no matter where in memory it is loaded. In order to possess this property, the code must contain no references to absolute objects. That is, any internal data or routines accessed must be referenced with respect to some fixed point in the program. As the offset from the required location to the fixed point remains constant, the address of the object may be calculated regardless of where the program was loaded. Usually, addresses are calculated with respect to the current instruction. You would often see instructions of the form:

```
.here  ADD ptr,  pc,  #object-(here+8)
```

to obtain the address of **`object`** in the register **`ptr`**. The +8 part occurs because the PC is always two instructions (8 bytes) further on than the instruction which is executing, due to pipelining.

It is because of the frequency with which this calculation crops up that the **`ADR`** directive is provided. As we explained in Chapter Four, the line above could be written:

```
    ADR ptr,  object
```

There is no need for a label: BASIC performs the calculation using the current value of P%.

Instead of using PC offsets, a program can also access its data using base-relative addressing. In this scheme, a register is chosen to store the base address of the program's data. It is initialised in some position-independent way at the start of the program, then all data accesses are relative to this. The ARM's register-offset address mode in **`LDR`** and **`STR`** make this quite a straightforward way of accessing data.

Why strive for position-independence? In a typical ARM system, the programs you write will be loaded into RAM, and may have to share that RAM with other programs. The operating system will find a suitable location for the program and load it there. As 'there' might be anywhere in the available memory range, your program can make no assumptions about the location of its internal routines and data. Thus all references must be relative to the PC. It is for this reason that branches use offsets instead of absolute addresses, and that the assembler provides the

```
    LDR     <dest>,<expression>
```

form of **`LDR`** and **`STR`** to automatically form PC-relative addresses.

Many microprocessors (especially the older, eight-bit ones) make it impossible to write position-independent code because of unsuitable instructions and architectures. The ARM makes it relatively easy, and you should take advantage of this.

Of course, there are bound to be some absolute references in the program. You may have to call external subroutines in the operating system. The usual way of doing this is to use a **`SWI`**, which implicitly calls absolute address &0000008. Pointers handed to the program by memory-allocation routines will be absolute, but as they are external to the program, this doesn't matter. The thing to avoid is absolute references to internal objects.

Sequences

These barely warrant a mention. As we have already implied, ARM instructions execute sequentially unless the processor is instructed to do otherwise. Sequence of high-level assignments:

```
LET a = b+c
LET d = b-c
```

would be implemented by a similar sequence of ARM instructions:

```
ADD     ra, rb, rc
SUB     rd, rb, rc
```

IF-type conditions

Consider the BASIC statement:

```
IF a=b THEN count=count+1
```

This maps quite well into the following ARM sequence:

```
CMP     ra, rb
ADDEQ   count, count, #1
```

In this and other examples, we will assume operands are in registers to avoid lots of **LDR**s and **STR**s. In practice, you may find a certain amount of processor-to-memory transfer has to be made.

The ARM's ability to execute any instruction conditionally enables us to make a straightforward conversion from BASIC. Similarly, a simple **IF**..**THEN**...**ELSE** such as this one

```
IF val<0 THEN sign=-1 ELSE sign=1
```

leads to the ARM equivalent:

```
TEQ     val, #0
MVNMI   sign, #0
MOVPL   sign, #1
```

The opposite conditions (**MI** and **PL**) on the two instructions make them mutually exclusive (i.e. one and only one of them will be executed after the **TEQ**), corresponding to the same property in the **THEN** and **ELSE** parts of the BASIC statement.

There is usually a practical limit to how many instructions may be executed conditionally in one sequence. For example, one of the conditional instructions may itself affect the flags, so the original condition no longer couldn't be performed using conditional execution. The solution (and the *only* method that most processors can use) is to conditionally branch over unwanted instructions.

Below is an example of a two-word add which executes only if R0=R1:

```
        CMP     R0, R1
        BNE     noAdd
        ADDS    lo1, lo1, lo2
        ADC     hi1, hi1, hi2
  .noAdd    ....
```

Notice that the condition used in the branch is the opposite to that under which the **ADD** is to be performed. Here is the general translation of the BASIC statements:

```
    IF cond THEN sequence1 ELSE sequence2  statement

    ;'ARM' version
    ;Obtain <cond>
    B<NOT cond>  seq2     ;If <cond> fails then jump to ELSE
    sequence1             ;Otherwise do the THEN part
            ...
    BAL    endSeq2        ;Skip over the ELSE part
.seq2
    sequence2             ;This gets executed if <cond> fails
            ...
.endSeq2
    statement             ;The paths re-join here
```

At the end of the **THEN** sequence is an unconditional branch to skip the **ELSE** part. The two paths rejoin at **endSeq2**.

It is informative to consider the relative timings of skipped instructions and conditionally executed ones. Suppose the conditional sequence consists of X group one instructions. The table below gives the timings in cycles for the cases when they are executed and not executed, using each method:

	Branch	**Conditional**
Executed	s + Xs	Xs
Not executed	2n+s	Xs

In the case where the instructions are executed, the branch method has to execute the un-executed branch, giving an extra cycle. This gives us the rather predictable result that if the conditional sequence is only one instruction, the conditional execution method should always be used.

When the sequence is skipped because the condition is false, the branch method takes 2n+s, or the equivalent to 5s cycles. The conditional branch method takes one s cycles for each un-executed instruction. So, if there are four or fewer instructions, at least one cycle is saved using conditional instructions. Of course, whether this makes the program execute any faster depends on the ratio between failures and successes of the condition.

Before we leave the **IF**-type constructions, we present a nice way of implementing conditions such as:

```
IF a=1 OR a=5 OR a=12...
```

It uses conditional execution:

```
TEQ     a,#1
TEQNE   a,#5
TEQNE   a,#12
BNE     failed
```

If the first **TEQ** gives an **EQ** result (i.e. **a**=1), the next two instructions are skipped and the sequence ends with the desired flag state. If **a**<>1, the next **TEQ** is executed, and again if this gives an **EQ** result, the last instruction is skipped. If neither of those two succeed, the result of the whole sequence comes from the final **TEQ**.

Another useful property of **TEQ** is that it can be used to test the sign and zero-ness of a register in one instruction. So a three-way decision could be made according to whether an operand was less than zero, equal to zero, or greater than zero:

```
TEQ     R0,#0
BMI     neg
BEQ     zero
BPL     plus
```

In this example, one of three labels is jumped to according to the sign of R0. Note that the last instruction could be an unconditional branch, as **PL** must be true if we've got that far.

The sequence below performs the BASIC assignment **a=ABS(a)** using conditional instructions:

```
TEQ     a, #0
RSBMI   a, #0  ;if a<0 then a=0-a
```

As you have probably realised, conditional instructions allow the elegant expression of many simple types of **IF**... construct.

Multi-way branches

Often, a program needs to take one of several possible actions, depending on a value or a condition. There are two main ways of implementing such a branch, depending on the tests made.

If the action to be taken depends on one of a few specific conditions, it is best implemented using explicit comparisons and branches. For example, suppose we wanted to take one of three actions depending on whether the character in the lowest byte of R0 was a letter, a digit or some other character. Assuming that the character set being used is ASCII, then this can be achieved thus:

```
    CMP     R0,#ASC"0" ;Less than the lowest digit?
    BCC     doOther    ;Yes, so must be 'other'
    CMP     R0,#ASC"9" ;Is it a digit?
    BLS     doDigit    ;Yes
    CMP     R0,#ASC"A" ;Between digits and upper case?
    BCC     doOther    ;Yes, so 'other'
    CMP     R0,#ASC"Z" ;Is it upper case?
    BLS     doLetter   ;Yes
    CMP     R0,#ASC"a" ;Between upper and lower case?
    BLT     doOther    ;Yes, so 'other'
    CMP     R0,#ASC"z" ;Lower case?
    BHI     doOther    ;No, so 'other'
.doLetter
    ....
    B       nextChar   ;Process next character
.doDigit
    ....
    B       nextChar   ;Process next character
.doOther
    ....
.nextChar
    ....
```

Note that by the time the character has been sorted out, the flow of control has been divided into three possible routes. To make the program easier to follow, the three destination labels should be close to each other. It is very possible that after each routine has done its job, the three paths will converge again into a single thread. To make this clear, each routine is terminated by a commented branch to the meeting point.

A common requirement is to branch to a given routine according to a range of values. This is typified by BASIC's **ON...PROC** and **CASE** statements. For example:

```
ON x PROCadd,PROCdelete,PROCamend,PROClist ELSE PROCerror
```

According to whether **x** has the value 1, 2, 3 or 4, one of the four procedures listed is executed. The **ELSE**... part allows for **x** containing a value outside of the expected range.

One way of implementing an **ON**... type structure in assembly language is using repeated comparisons:

```
    CMP choice,  #1      ;Check against lower limit
    BCC error            ;Lower, so error
    BEQ add              ;choice = 1 so add
    CMP choice,  #3      ;Check for 2 or 3
    BLT delete           ;choice = 2 so delete
    BEQ amend            ;choice = 3 so amend
    CMP choice,  #4      ;Check against upper limit
    BEQ list             ;If choice = 4 list else error
.error
    .....
```

Although this technique is fine for small ranges, it becomes large and slow for wide ranges of **choice**. A better technique in this case it to use a branch table. A list of branches to the routines is stored near the program, and this is used to branch to the appropriate routine. Below is an implementation of the previous example using this technique.

```
 DIM org 200
 choice = 0
 t = 1
 sp = 13
 link = 14

 REM Range of legal values
 min = 1
 max = 4
 WriteS = 1
 NewLine = 3
 FOR pass=0 TO 2 STEP 2
 P%=org
 [ opt pass
 ;Multiway branch in ARM assembler
 ;choice contains code, min..max of routine to call
 ;If out of range, error is called
 ;
    STMFD  (sp)!,{t,link}
    SUBS   choice, choice, #min ;Choice <min?
    BCC    error                ;Yes, so error
    CMP    choice, #max-min     ;Choice >max?
    BHI    error                ;Yes, so error
    ADR    link, return         ;Set-up return address
    ADR    t,table              ;Get address of table base
    ADD    PC, t, choice, LSL #2;Jump to table+choice*4
 ;
```

```
   SWI    WriteS
   EQUS   "Range  error"
   EQUB   0
   ALIGN
;
.return
   SWI    NewLine
   LDMFD  (sp)!,{t,PC}
;
;
;Table  of  branches  to  routines
.table
   B      add
   B      delete
   B      amend
   B      list
;
.add
   SWI    WriteS
   EQUS   "Add  command"
   EQUB   0
   ALIGN
   MOV    PC,link
;
.delete
   SWI    WriteS
   EQUS       "Delete  command"
   EQUB   0
   ALIGN
   MOV    PC,link
;
.amend
   SWI    WriteS
   EQUS   "Amend  command"
   EQUB   0
   ALIGN
   MOV    PC,link
;
.list
   SWI    WriteS
   EQUS   "List  command"
   EQUB   0
   ALIGN
   MOV    PC,link
]
NEXT
REPEAT
   INPUT "Choice ",A%
   CALLorg
UNTIL  FALSE
```

The first four lines check the range of the value in **choice**, and call **error** if it is outside of the range **min** to **max**. It is important to do this, otherwise uses **SUBS** instead of **CMP**, so choice is adjusted to the range 0 to **max-min** instead of **min** to **max**.

Next, the return address is placed in R14. The routines **add**, **delete** etc. return as if they had been called using **BL**, i.e. use a return address in R14. To do this, we use **ADR** to place the address of the label **return** into R14, this being where we want to resume execution.

The next **ADR** obtains the base address of the jump table in the register **t**. Finally, the **ADD** multiplies **choice** by 4 (using two left shifts) and adds this offset to the table's base address. The result of the addition is placed in the program counter. This causes execution to jump to the branch instruction in the table that was denoted by **choice**. From there, the appropriate routine is called, with the return address still in R14.

As we mentioned in the position-independent code section, this may seem a little bit involved just to jump to one of four locations. Remember though that the technique will work for an arbitrary number of entries in the table, and will work at whatever address the program is loaded.

Loops

Looping is vital to any non-trivial program. Many problems have solutions that are expressed in an iterative fashion. There are two important classes of looping construct. The first is looping while, or until, a given condition is met (e.g. **REPEAT** and **WHILE** loops in BASIC). The second is looping for a given number of iterations (e.g. **FOR** loops). In fact, the second class is really a special case of the general conditional loop, the condition being that the loop has iterated the correct number of times.

An important characteristic of any looping construct is where the test of the looping condition is made. In BASIC **REPEAT** loops, for example, the test is made at the corresponding **UNTIL**. This means that the instructions in the loop are always executed at least once. Consider this example:

```
REPEAT
   IF a>b THEN a=a-b ELSE b=b-a
UNTIL a=b
```

This is a simple way to find the greatest common divisor (GCD) of **a** and **b**. If **a**=**b** (and **a**<>0) when the loop is entered, the result is an infinite loop as on the first iteration **b=b-a** will be executed, setting **b** to 0. From then on, **a=a-0** will be executed, which will never make **a**=**b**.

The **WHILE** loop tests the condition at the 'top', before its statements have

```
WHILE a<>b
    IF a>b THEN a=a-b ELSE b=b-a
ENDWHILE
```

This time, if **a**=**b**, the condition at the top will fail, so the loop will never be executed, leaving **a**=**b**=GCD(**a**,**b**).

Below are the two ARM equivalents of the **REPEAT** and **WHILE** loop versions of the GCD routine:

```
;Find the GCD of ra,rb.
;Fallible version using 'repeat' loop
.repeat
    CMP    ra,rb      ;REPEAT IF a>b
    SUBGT  ra,ra,rb   ;   THEN a=a-b
    SUBLE  rb,rb,ra   ;   ELSE b=b-a
    CMP    ra,rb      ;UNTIL
    BNE    repeat     ;a=b
;
;Find GCD of ra,rb, using 'while' loop
.while
    CMP    ra,rb      ;WHILE a<>b
    BNE    endwhile
    SUBGT  ra,ra,rb   ;   IF a>b THEN a=a-b
    SUBLE  rb,rb,ra   ;           ELSE b=b-a
    B      while      ;ENDWHILE
.endwhile
```

Notice that the difference between the two is that the **WHILE** requires a forward branch to skip the instructions in the body of the loop. This is not a problem for an assembler, which has to cope with forward references to be of any use at all. In an interpreted language like BASIC, though, the need to scan through a program looking for a matching **ENDWHILE** is something of a burden, which is why some BASIC's don't have such structures.

Because both of the code sequences above are direct translations of high-level versions, they are indicative of what we might expect a good compiler to produce. However, we are better than any compiler, and can optimise both sequences slightly by a bit of observation. In the first loop, we branch back to an instruction which we have just executed, wasting a little time. In the second case, we can use the conditional instructions to eliminate the first branch entirely. Here are the hand-coded versions:

```
;Fallible version using 'repeat'
    CMP    ra,rb      ;REPEAT IF a>b
.repeat
    SUBGT  ra,ra,rb   ;   THEN a=a-b
    SUBLE  rb,rb,ra   ;   ELSE b=b-a
    CMP    ra,rb      ;UNTIL
```

```
;Find GCD of ra,rb, using 'while' loop
.while
    CMP    ra,rb      ;REPEAT
    SUBGT  ra,ra,rb   ;   IF a>b THEN a=a-b
    SUBLT  rb,rb,ra   ;      ELSE IF a<b b=b-a
    BNE    while      ;UNTIL a=b    endwhile
```

By optimising, we have converted the **WHILE** loop into a **REPEAT** loop with a slightly different body.

In general, a **REPEAT**-type structure is used when the processing in the 'body' of the loop will be needed at least once, whereas **WHILE**-type loops have to be used in situations where the 'null' case is a distinct possibility. For example, string handling routines in the BASIC interpreter have to deal with zero-length strings, which often means a **WHILE** looping structure is used. (See the string-handling examples later.)

A common special case of the **REPEAT** loop is the infinite loop, expressed as:

```
REPEAT
    REM do something
UNTIL FALSE
```

or in ARM assembler:

```
.loop
;   do something
    BAL loop
```

Programs which exhibit this behaviour are often interactive ones which take an arbitrary amount of input from the user. Again the BASIC interpreter is a good example. The exit from such programs is usually through some 'back door' method (e.g. calling another program) rather than some well-defined condition.

Since **FOR** loops are a special case of general loops, they can be expressed in terms of them. The **FOR** loop in BBC BASIC exhibits a **REPEAT**-like behaviour, in that the test for termination is performed at the end, and it executes at least once. Below is a typical **FOR** loop and its **REPEAT** equivalent:

```
REM A typical for loop
FOR ch=32 TO 126
    VDU ch
NEXT ch

REM REPEAT loop equivalent
ch=32
REPEAT
```

```
    ch=ch+1
UNTIL ch>126
```

The initial assignment is placed just before the **REPEAT**. The body of the **REPEAT** is the same as that for the **FOR**, with the addition of the incrementing of **ch** just before the condition. The condition is that **ch** is greater than the limit given in the **FOR** statement.

We can code the **FOR** loop in ARM assembler by working from the **REPEAT** loop version:

```
;Print characters 32..126 using a FOR loop-type construct
;R0 holds the character
    MOV     R0, #32         ;Init the character
.loop
    SWI     WriteC      ;Print it
    ADD     R0, R0, #1 ;Increment it
    CMP     R0, #126    ;Check the limit
    BLE     loop        ;Loop if not finished
;
```

Very often, we want to do something a fixed number of times, which could be expressed as a loop beginning **FOR i=1 TO n**... in BASIC. When such loops are encountered in assembler, we can use the fact that zero results of group one instructions can be made to set the Z flag. In such cases, the updating of the looping variable and the test for termination can be combined into one instruction.

For example, to print ten stars on the screen:

```
FOR i=1 TO 10
    PRINT "*";
NEXT i
```

could be re-coded in the form:

```
;Print ten stars on the screen
;R0 holds the star character, R1 the count
    MOV     R0,#ASC"*" ;Init char to print
    MOV     R1,#10      ;Init count
.loop
    SWI     WriteC      ;Print a star
    SUBS    R1,R1,#1    ;Next
    BNE     loop
;
```

The **SUBS** will set the Z flag after the tenth time around the loop (i.e. when R1 reaches 0), so we do not have to make an explicit test.

Of course, if the looping variable's current value was used in the body of the loop, this method could not be used (unless the loop was of the form **`FOR i=n TO 1 STEP -1`**...) as we are counting down from the limit, instead of up from 1.

Some high-level languages provide means of repeating a loop before the end or exiting from the current loop prematurely. These two looping 'extras' are typified by the **`continue`** and **`break`** statements in the C language. **`Continue`** causes a jump to be made to just after the last statement inside the current **`FOR`**, **`WHILE`** or **`REPEAT`**-type loop, and **`break`** does a jump to the first statement after the current loop.

Because **`continue`** and **`break`** cause the flow of control to diverge from the expected action of a loop, they can make the program harder to follow and understand. They are usually only used to 'escape' from some infrequent or error condition. Both constructs may be implemented in ARM using conditional or unconditional branches.

5.2 Subroutines and procedures

We have now covered the main control flow structures. Programs written using just these constructs would be very large and hard to read. The sequence, decision and loop constructs help to produce an ordered solution to a given problem. However, they do not contribute to the division of the problem into smaller, more manageable units. This is where subroutines come in.

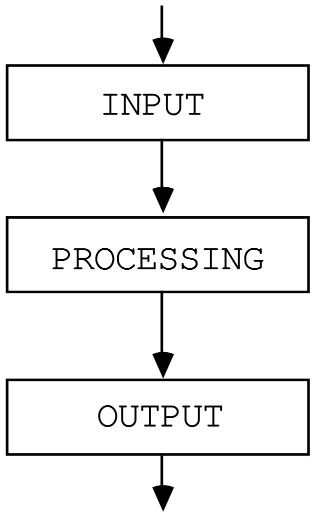
Even the most straightforward of problems that one is likely to use computer to solve can be decomposed into a set of simpler, shorter sub-programs. The motivations for performing this decomposition are several. Humans can only take in so much information at once. In terms of programming, a page of listing is a useful limit to how much a programmer can reasonably be expected to digest in one go. Also, by implementing the solution to a small part of a problem, you may be writing the same part of a later program. It is surprising how much may be accomplished using existing 'library' routines, without having to re-invent the wheel every time.

The topics of program decomposition and top-down, structured programming are worthy of books in their own right, and it is recommended that you consult these if you wish to write good programs in any language. The discipline of structured programming is even more important in assembler than in, say, Pascal, because it is easier to write treacherously unreadable code in assembler.

A minimal decomposition of most programs is shown in the block diagram overleaf. Data is taken in, processed in some way, then results output. If you think about it, most programs would be rather boring if they depended on absolutely no external stimulus for their results.

Once the input, processing and output stages have been identified, work can begin on solving these individual parts. Almost invariably this will involve further decomposition, until eventually a set of routines will be obtained which can be written directly in a suitably small number of basic instructions.

The way in which these routines are linked together, and how they communicate with each other, are the subjects of the next sections.



A minimal useful program

Branch and link

The ARM **BL** instruction is a subroutine-calling primitive. Primitive in this context means an operation which is implemented at the lowest level, with no more hidden detail.

Recall from Chapter Three that **BL** causes a branch to a given address, and stores the return address in R14. We will illustrate the use of **BL** to call the three routines which solve a very simple problem. This may be expressed as follows: repeatedly read a single character from the keyboard and if it is not the NUL character (ASCII code 0), print the number of 1 bits in the code.

For comparison, the BASIC program below solves the problem using exactly the same structure as the following ARM version:

```
REPEAT ch = FNreadChar
   IF ch<>0  PROCoutput(FNprocess(ch))
UNTIL ch=0
END
REM ****************************************
DEF FNreadChar=GET
REM ****************************************
DEF FNprocess(ch)
   LOCAL count
   count=0
   REPEAT
      count=count + ch MOD 2
      ch=ch DIV 2
   UNTIL ch=0
=count
REM ****************************************
DEF PROCoutput(num)
   PRINT num
ENDPROC
```

There are four entities, separated by the lines of asterisks. At the top is the 'main program'. This is at the highest level and is autonomous: no other routine calls the program. The next three sections are the routines which the main program uses to solve the problem. As this is a fairly trivial example, none of the subroutines calls any other; they are all made from primitive instructions. Usually (and especially in assembly language where primitives are just that), these 'second level' routines would call even simpler ones, and so on.

Below is the listing of the ARM assembler version of the program:

```
DIM org 200
sp = 13
link = 14
REM SWI numbers
WriteC = 0
NewLine = 3
ReadC = 4
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;Read characters and print the number of 1 bits in the
;ASCII code, as long as the code isn't zero.
  STMFD   (sp)!,{link}   ; Save return address
.repeat
  BL      readChar       ;Get a character in R0
  CMP     R0,#0          ;Is it zero?
  LDMEQFD(sp)!,{PC}      ;Yes, so return to caller
  BL      process        ;Get the count in R1
  BL      output         ;Print R1 as a digit
  B       repeat         ;Do it again
```

```
 ;readChar - This returns a character in R0
 ;All other registers preserved
 ;
 .readChar
   SWI    ReadC          ;Call the OS for the read
   MOV    PC, link       ;Return using R14
 ;
 ;process - This counts the number of 1s in R0 bits 0..7
 ;It returns the result in R1
 ;On exit, R1=count, R0=0, all others preserved
 ;
 .process
   AND    R0, R0, #&FF   ;Zero bits 8..31 of R0
   MOV    R1, #0         ;Init the bit count
 .procLoop
   MOVS   R0, R0, LSR #1;DIV 2 and get MOD 2 in carry
   ADC    R1, R1, #0     ;Add carry to count
   BNE    procLoop       ;More to do
   MOV    PC, link       ;Return with R1=count
 ;
 ;output - print R1 as a single digit
 ;On exit, R0=R1 + "0", all others preserved
 ;
 .output
   ADD    R0, R1,#ASC"0";Convert R1 to ASCII in R0
   SWI    WriteC         ;Print the digit
   SWI    NewLine        ;And a newline
   MOV    PC, link       ;Return
]
 NEXT
 CALL org
```

Because of the way in which the program closely follows the BASIC version, you should not have much difficulty following it. Here are some points to note. In the BASIC version, two of the subroutines, **`process`** and **`readChar`**, are functions and **`print`** is a procedure. In the ARM version, there is no such obvious distinction in the way the routines are called. However, the fact that **`process`** and **`readChar`** return values to their caller makes them equivalent to function, whereas process, which returns no value of use to the caller, is a procedure equivalent.

At the start of each routine is a short description of what it does and how it affects the registers. Such documentation is the bare minimum that you should provide when writing a routine, so that problems such as registers being changed unexpectedly are easier to track down. In order to do this when the operating system routines are used (e.g. the **`SWI WriteC`** call), you have to know how those routines affect the registers. This information should be provided in the system documentation. For now, we assume that no registers are altered except those in which results are returned, e.g. R0 in **`SWI ReadC`**

In the routine **process** we use the ability to (a) set the C flag from the result of shifting an **<rhs>** operand, and (b) preserve the state of the Z flag over the **ADC** by not specifying the **S** option. This enables us to write an efficient three-instruction version of the BASIC loop.

The routine **output** assumes that the codes of the digit symbols run contiguously from 0, 1, ...9. Using this assumption it is a simple matter to convert the binary number 1..8 (remember &00 will never have its 1 bits counted) into the equivalent printable code. As the ASCII code exhibits the desired contiguous property, and is almost universally used for character representation, the assumption is a safe one.

As none of the routines change the link register, R14, they all return using a simple move from the link register to the PC. We do not bother to use **MOVS** to restore the flags too, as they are not expected by the main program to be preserved.

If a subroutine calls another one using **BL**, then the link register will be overwritten with the return address for this later call. In order for the earlier routine to return, it must preserve R14 before calling the second routine. As subroutines very often call other routines (i.e. are 'nested'), to an arbitrary depth, some way is needed of saving any number of return addresses. The most common way of doing this is to save the addresses on the stack.

The program fragment below shows how the link register may be saved at the entry to a routine, and restored directly into the PC at the exit. Using this technique, any other registers which have to be preserved by the routine can be saved and restored in the same instructions:

```
;
 ;subEg.  This is an example of using the stack to save
;the return address of a subroutine.  In addition, R0,R1
;and R2 are preserved.
;
.subEg
    STMFD  (sp)!,{R0-R2,link};Save link and R0-R2
    ....                     ;Do some processing
    ....
    LDMFD  (sp)!,{R0-R2,pc}^ ;Load PC, flags and R0-R2
;
```

The standard forms of **LDM** and **STM** are used, meaning that the stack is a 'full, descending' one. Write-back is enabled on the stack pointer, since it almost always will be for stacking operations, and when the PC is loaded from the stack the flags are restored too, due to the **^** in the instruction.

Note that if the only 'routines' called are **`SWI`** ones, then there is no need to save the link register, R14, on the stack. Although **`SWI`** saves the PC and flags in R14, it is the supervisor mode's version of this register which is used, and the user's one remains intact.

Parameter passing

When values are passed to a routine, they are called the parameters, or arguments, of the routine. A routine performs some general task. When supplied with a particular set of arguments, it performs a more specific action (it has been parameterized, if you like), and the job it performs is usually the same for a particular set of arguments. When a routine returns one or more values to its caller, these values are known as the results of the routine.

The term 'subroutine' is usually applied to a primitive operation such as branch and link, which enables a section of code to be called then returned from. When a well-defined method of passing parameters is combined with the basic subroutine mechanism, we usually call this a procedure. For example, **`output`** in the example above is a procedure which takes a number between 0 and 9 in R1 and prints the digit corresponding to this. When a procedure is called in order to obtain the results it returns, it is called a function.

You may have heard the terms procedure and function in relation to high-level languages. The concept is equally valid in assembler, and when the procedures and functions of a high-level language are compiled (i.e. converted to machine code or assembler) they use just the primitive subroutine plus parameter passing mechanisms that we describe in this section.

In the example program of the previous section, the BASIC version used global variables as parameters and results, and the assembler version used registers. Usually, high-level languages provide a way of passing parameters more safely than using global variables. The use of globals is not desirable because (a) the caller and callee have to know the name of the variable being used and (b) global variables are prone to corruption by routines which do not 'realise' they are being used elsewhere in the program.

Using registers is just one of the ways in which arguments and results can be passed between caller and callee. Other methods include using fixed memory areas and the stack. Each method has its own advantages and drawbacks. These are described in the next few sections.

Register parameters

On a machine like the ARM, using the registers for the communication of arguments and results is the obvious choice. Registers are fairly plentiful (13 left after the PC, link and stack pointer have been reserved), and access to them is rapid. Remember that before the ARM can perform any data processing instructions, the operands must be loaded into registers. It makes sense then to ensure that they are already in place when the routine is called.

The operating system routines that we use in the examples use the registers for parameter passing. In general, registers which are not used to pass results back are preserved during the routine, i.e. their values are unaltered when control passes back to the caller. This is a policy you should consider using when writing your own routines. If the procedure itself preserves and restores the registers, there is no need for the caller to do so every time it uses the routine.

The main drawback of register parameters is that they can only conveniently be used to hold objects up to the size of a word - 32-bits or four bytes. This is fine when the data consists of single characters (such as the result of **`SWI ReadC`**) and integers. However, larger objects such as strings of characters or arrays of numbers cannot use registers directly.

Reference parameters

To overcome the problem of passing large objects, we resort to a slightly different form of parameter passing. Up until now, we have assumed that the contents of a register contain the value of the character or integer to be passed or returned. For example, when we use the routine called **`process`** in the earlier example, R0 held the value of the character to be processed, and on exit R1 contained the value of the count of the number one bits. Not surprisingly, this method is called call-by-value.

If instead of storing the object itself in a register, we store the object's address, the size limitations of using registers to pass values disappear. For example, suppose a routine requires the name of a file to process. It is obviously impractical to pass an arbitrarily long string using the registers, so we pass the address of where the string is stored in memory instead.

The example below shows how a routine called **`wrchS`** might be written and called. **`WrchS`** takes the address of a string in R1, and the length of the string in R2. It prints the string using **`SWI WriteC`**.

Note that the test program obtains the address in a position-independent way, using **`ADR`**. The first action of **`wrchS`** is to save R0 and the link register data was mentioned in ChapterThree, and we shall have more to say about them later. We save R0 because the specification in the comments states that all registers except R1 and R2 are preserved. Since we need to use R0 for calling **SWI WriteC**, its contents must be saved.

The main loop of the routine is of the **WHILE** variety, with the test at the top. This enables it to cope with lengths of less than or equal to zero. The **SUBS** has the dual effect of decreasing the length count by one and setting the flags for the termination condition. An **LDRB** is used to obtain the character from memory, and post-indexing is used to automatically update the address in R1.

```
DIM org 200
sp = 13
link = 14
cr = 13 : lf = 10
WriteC = 0
FOR pass=0 TO 2 STEP 2
P%=org
[ opt pass
;
;Example showing the use of wrchS
;
.testWrchS
  STMFD   (sp)!,{link}       ;Save return address
  ADR     R1, string         ;Get address of string
  MOV     R2,#strEnd-string  ;Load string length
  BL      wrchS              ;Print it
  LDMFD   (sp)!,{PC}         ;Return
;
.string
  EQUS    "Test string"      ;The string to be printed
  EQUB    cr
  EQUB    lf
.strEnd
;
;
;Subroutine to print a string addressed by R1
;R2 contains the number of bytes in the string
;On exit, R1 points the to byte after the string
;          R2 contains -1
;All other registers preserved
.wrchS
  STMFD   (sp)!, {R0,link}   ;Save R0 and return address
.wrchsLp
  SUBS    R2, R2, #1         ;End of string?
  LDMMIFD(sp)!, {R0,PC}      ;Yes, so exit
  LDRB    R0, [R1], #1       ;Get a char and inc R1
  SWI     WriteC             ;Print this character
  B       wrchsLp            ;Next char
]
```

```
NEXT
CALL  testWrchS
```

When the **`LDMMI`** is executed we restore R0 and return to the caller, using a single instruction. If we had not stored the link on the stack (as we did in the first instruction), an extra **`MOV pc,link`** would have been required to return.

Call-by-reference, or call-by-address is the term used when parameters are passed using their addresses instead of their values. When high-level languages use call-by-reference (e.g. **`var`** parameters in Pascal), there is usually a motive beyond the fact that registers cannot be used to store the value. Reference parameters are used to enable the called routine to alter the object whose address is passed. In effect, a reference parameter can be used to pass a result back, and the address of the result is passed to the routine in a register.

To illustrate the use of reference results, we present below a routine called **`readS`**. This is passed the address of an area of memory in R1. A string of characters is read from the keyboard using **`SWI ReadC`**, and stored at the given address. The length of the read string is returned in R0.

```
 DIM org 100, buffer 256
 WriteC = 0
 ReadC = 4
 NewLine = 3
 cr = &0D
 sp = 13
 link = 14
 FOR pass=0 TO 2 STEP 2
 P%=org
 [ opt pass
 ;
 ;readS. Reads a string from keyboard to memory
 ;addressed by R1. The string is terminated by the
character
 ;&0D (carriage return) On exit R0 contains the length of
 ;the string, including the CR
 ;All other registers are preserved
 ;
 .readS
   STMFD  (sp)!, {link}     ;Save return address
   MOV    R2, #0            ;Init the length
.readS1p
   SWI    ReadC             ;Get char in R0
   TEQ    R0, #cr           ;Was it carriage return?
   SWINE  WriteC            ;Echo the character if not
   STRB   R0, [R1, R2]      ;Store the char
   ADD    R2, R2, #1        ;Increment the count
   BNE    readS1p           ;If not CR, loop
```

```
   MOV    R0, R2            ;Return count in R0 for USR
   LDMFD  (sp)!, {PC}       ;Return
]
NEXT
B%=buffer
PRINT"String: ";
len%=USR readS
PRINT"Length was ";len%
PRINT"String was "$buffer
```

This time, a **REPEAT**-type loop is used because the string will always contain at least one character, the carriage return. Of course, a routine such as this would not be very practical to use: there is no checking for a maximum string length; no action on special keys such as **DELETE** or **ESCAPE** is taken. It does, however, show how a reference parameter might be used to pass the address of a variable which is to be updated by the routine.

Parameter blocks

A parameter block, or control block, is closely related to reference parameters. When we pass a parameter block to a routine, we give it the address of an area of memory in which it may find one or more parameters. For example, suppose we wrote a routine to save an area of memory as a named file on the disk drive. Several parameters would be required:

Name of the file on the disk
Start address of data
End address (or length) of data
Load address of data
Execution address (in case it is a program)
Attributes (read, write etc.)

Now, all of these items may be passed in registers. If we assume the name is passed by address and has some delimiting character on the end, six registers would be required. Alternatively, the information could be passed in a parameter block, the start address of which is passed in a single register. The file save routine could access the component parts of the block using, for example

```
   LDR    [base,#offset]
```

where **base** is the register used to pass the start address, and **offset** is the address of the desired word relative to **base**.

As the address of the parameter block is passed to the routine,the parameters may be altered as well as read. Thus parameter blocks are addition to passing it. For example, the parameter block set up for a disk load operation could have its entries updated from the data stored for the file in the disk catalog (load address, length etc.)

Parameter blocks are perhaps less useful on machines with generous register sets like the ARM than on processors which are less well-endowed, e.g. 8-bit micros such as the 6502. However, you should remember the advantage of only one register being needed to pass several parameters, and be ready to use the technique if appropriate.

Stack parameters

The final parameter passing technique which we will describe uses the stack to store arguments and results. In chapter three we described the **LDM** and **STM** instructions, for which the main use is dealing with a stack-type structure. Information is pushed on to a stack using **STM** and pulled from it using **LDM**. We have already seen how these instructions are used to preserve the return address and other registers.

To pass parameters on the stack, the caller must push them just before calling the routine. It must also make room for any results which it expects to be returned on the stack. The example below calls a routine which expects to find two arguments on the stack, and returns a single result. All items are assumed to occupy a single word.

```
;
;StackEg.  This shows how the stack might be used
;to pass arguments and receive results from a stack.
;Before entry, two arguments are pushed, and on exit a
;single result replaces them.  ;
.stackEg
    STMFD  (sp)!,{R0,R1} ;Save the arguments
    BL     stackSub      ;Call the routine
    LDMFD  (sp)!,{R0}    ;Get the result
    ADD    sp,sp,#8      ;'Lose' the arguments
    ....
    ....
.stackSub
    LDMFD  (sp)!,{R4,R5} ;Get the arguments
    ....                 ;Do some processing
    ....
    STMFD  (sp)!,{R2}    ;Save the result
    MOV    pc,link       ;Return
```

Looking at this code, you may think to yourself 'what a waste of time.' As soon as one routine pushes a value, the other pulls it again. It would seem much more sensible to simply pass the values in registers in the first place. Notice, though, that when **stackSub** is called, the registers used to set-up is one of the advantages of stacked parameters: all the caller and callee need to know is the size, number and order of the parameters, not (explicitly) where they are stored.

In practice, it is rare to find the stack being used for parameter passing by pure assembly language programs, as it is straightforward to allocate particular registers. Where the stack scheme finds more use is in compiled high-level language procedures. Some languages, such as C, allow the programmer to assume that the arguments to a procedure can be accessed in contiguous memory locations. Moreover, many high-level languages allow recursive procedures, i.e. procedures which call themselves. Since a copy of the parameters is required for each invocation of a procedure, the stack is an obvious place to store them. See the Acorn ARM Calling Standard for an explanation of how high-level languages use the stack.

Although the stack is not often used to pass parameters in assembly language programs, subroutines frequently save registers in order to preserve their values across calls to the routine. We have already seen how the link register (and possibly others) may be saved using **STM** at the start of a procedure, and restored by **LDM** at the exit. To further illustrate this technique, the program below shows how a recursive procedure might use the stack to store parameters across invocations.

The technique illustrated is very similar to the way parameters (and local variables) work in BBC BASIC. All variables are actually global. When a procedure with the first line

```
DEF  PROCeg(int%)
```

is called using the statement **PROCeg(42)**, the following happens. The value of **int%** is saved on the stack. Then **int%** is assigned the value 42, and this is the value it has throughout the procedure. When the procedure returns using **ENDPROC**, the previous value of **int%** is pulled from the stack, restoring its old value.

The assembly language equivalent of this method is to pass parameters in registers. Just before a subroutine is called, registers which have to be preserved across the call are pushed, and then the parameter registers are set-up. When the routine exits, the saved registers are pulled from the stack.

There are several routines which are commonly used to illustrate recursion. The one used here is suitable because of its simplicity; the problem to be solved does not get in the way of showing how recursion is used. The Fibonacci sequence is a series of numbers thus:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each number is the sum of its two predecessors. It can be expressed mathematically in terms of some functions:

$f(0) = 0$
$f(1) = 1$
$f(n) = f(n-2) + f(n-1)$

where f(n) means the nth number in the sequence starting from zero. It can easily be translated into a BASIC function:

```
DEF FNfib(n) IF n<=1 THEN =n ELSE =FNfib(n-2)+FNfib(n-1)
```

To convert this into ARM assembler, we will assume that the number **n** is passed in R1 and the result fib(**n**) returned in R0.

```
DIM org 200
link=14
sp=13
FOR pass=0 TO 2 STEP 2
P%=org

[ opt pass
;Fibonacci routine to return fib(n)
;On entry, R1 contains n
;On exit, R0 contains fib(n), R1 preserved, R2 corrupt
;
.fib
  CMP    R1,#1          ;See if it's an easy case
  MOVLE  R0,R1          ;Yes, so return it in R0
  MOVLE  PC,link        ;And return
  STMFD  (sp)!,{link}   ;Save return address
  SUB    R1,R1,#2       ;Get fib(n-2) in R0
  BL     fib
  STMFD  (sp)!,{R0}     ;Save it on the stack
  ADD    R1,R1,#1       ;Get fib(n-1) in R0
  BL     fib
  LDMFD  (sp)!,{R2}     ;Pull fib(n-2)
  ADD    R0,R0,R2       ;Add fib(n-2) and fib(n-1) in R0
  ADD    R1,R1,#1       ;Restore R1 to entry value
  LDMFD  (sp)!,{PC}     ;Return
]
NEXT
FOR B%=0 TO 25

   PRINT "Fib(";B%") is ";USR fib
NEXT B%
```

The routine does not use the stack in exactly the same way as BBC BASIC, but the saving of intermediate results on the stack enables **fib** to be called preserved, i.e. contains **n**, as specified in the comments. This is because whenever **fib** is called recursively the caller expects R1 to be left intact so that it can calculate the next value correctly. In the cases when R1=0 or 1 on entry it is clearly preserved; in the other cases, by observation R1 is changed by -2, +1 and +1, i.e. there is no net change in its value.

You should note that, like a lot of routines that are expressed elegantly using recursion, this Fibonacci program becomes very inefficient of time and stack space for quite small values of **n**. This is due to the number of recursive calls made. (For an exercise you could draw a 'tree' of the calls for some start value, say 6.) A better solution is a counting loop. This is expressed in BASIC and ARM assembler below.

```
DEF  FNfib(n)
    IF  n  <=  1  THEN  =n
    LOCAL  f1,f2
    f2=0  :  f1  =  1
    FOR  i=0  TO  n-2
        f1  =  f1+f2
        f2  =  f1-f2
    NEXT  i
=  f1

 DIM  org  200
 i  =  2        :  REM  Work  registers
 f1  =  3
 f2  =  4
 sp  =  13

 link  =  14
 FOR  pass=0  TO  2  STEP  2
 P%=org
 [  opt  pass

 ;fib  -  using  iteration  instead  of  recursion
 ;On  entry,  R1  =  n
 ;On  exit,  R0  =  fib(n)
 ;
.fib
    CMP     R1,#1               ;Trivial  test  first
    MOVLE   R0,R1
    MOVLE   PC,link
    STMFD   (sp)!,{i,f1,f2,link}  ;Save  work  registers  and
link
    MOV     f1,#1               ;Initialise  fib(n-1)
    MOV     f2,#0               ;and  fib(n-2)
    SUB     i,R1,#2             ;Set-up  loop  count
.fibLp
    ADD     f1,f1,f2            ;Do  calculation
    SUB     f2,f1,f2
    SUBS    i,i,#1
```

```
   MOV     R0,f1              ;Return result in R0
   LDMFD   (sp)!,{i,f1,f2,PC};Restore and return
]
NEXT pass
FOR B%=0 TO 25

    PRINT"Fib(";B%;") is ";USR fib
NEXT B%
```

Summary

The main thrust of this chapter has been to show how some of the familiar concepts of high-level languages can be applied to assembler. Most control structures are easily implemented in terms of branches, though more complex ones (such as multi-way branching) can require slightly more work. This is especially true if the code is to exhibit the desirable property of position-indepence.

We also saw how parameters may be passed between routines - in registers, on the stack, or in parameter blocks. Using the stack has the advantage of allowing recursion, but is less efficient than passing information in registers.