3 ARM Assembly Language Programming

Summary of chapter contents

The ARM processor is very easy to program at the assembly level, though for most applications it is more appropriate to program in a high-level language such as C or C++.

Assembly language programming requires the programmer to think at the level of the individual machine instruction. An ARM instruction is 32 bits long, so there are around 4 billion different binary machine instructions. Fortunately there is considerable structure within the instruction space, so the programmer does not have to be familiar with each of the 4 billion binary encodings on an individual basis. Even so, there is a considerable amount of detail to be got right in each instruction. The assembler is a computer program which handles most of this detail for the programmer.

In this chapter we will look at ARM assembly language programming at the user level and see how to write simple programs which will run on an ARM development board or an ARM emulator (for example, the ARMulator which comes as part of the ARM development toolkit). Once the basic instruction set is familiar we will move on, in Chapter 5, to look at system-level programming and at some of the finer details of the ARM instruction set, including the binary-level instruction encoding.

Some ARM processors support a form of the instruction set that has been compressed into 16-bit 'Thumb' instructions. These are discussed in Chapter 7.

3.1 Data processing instructions

ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers. All other instructions just move data around and control the sequence of program execution, so the data processing instructions are the only instructions which modify data values. These instructions typically require two operands and produce a single result, though there are exceptions to both of these rules. A characteristic operation is to add two values together to produce a single result which is the sum.

Here are some rules which apply to ARM data processing instructions:

- All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself.
- The result, if there is one, is 32 bits wide and is placed in a register.

  (There is an exception here: long multiply instructions produce a 64-bit result; they are discussed in Section 5.8 on page 122.)
- Each of the operand registers and the result register are independently specified in the instruction. That is, the ARM uses a '3-address' format for these instructions.

Simple register operands

A typical ARM data processing instruction is written in assembly language as shown below:

```
ADD     r0, r1, r2          ; r0 := r1 + r2
```

The semicolon in this line indicates that everything to the right of it is a comment and should be ignored by the assembler. Comments are put into the assembly source code to make reading and understanding it easier.

This example simply takes the values in two registers (r1 and r2), adds them together, and places the result in a third register (r0). The values in the source registers are 32 bits wide and may be considered to be either unsigned integers or signed 2's-complement integers. The addition may produce a carry-out or, in the case of signed 2's-complement values, an internal overflow into the sign bit, but in either case this is ignored.

Note that in writing the assembly language source code, care must be taken to write the operands in the correct order, which is result register first, then the first operand and lastly the second operand (though for **commutative** operations the order of the first and second operands is not significant when they are both registers). When this instruction is executed the only change to the system state is the value of the destination register r0 (and, optionally, the N, Z, C and V flags in the CPSR, as we shall see later).

The different instructions available in this form are listed below in their classes:

- Arithmetic operations.

  These instructions perform binary arithmetic (addition, subtraction and reverse subtraction, which is subtraction with the operand order reversed) on two 32-bit operands. The operands may be unsigned or 2's-complement signed integers; the carry-in, when used, is the current value of the C bit in the CPSR.

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

  'ADD' is simple addition, 'ADC' is add with carry, 'SUB' is subtract, 'SBC' is subtract with carry, 'RSB' is reverse subtraction and 'RSC' reverse subtract with carry.

- Bit-wise logical operations.

  These instructions perform the specified Boolean logic operation on each bit pair of the input operands, so in the first case *r0[i]*:= *r1[i]* AND *r2[i]* for each value of *i* from 0 to 31 inclusive, where *r0[i]* is the *i*th bit of r0.

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

  We have met AND, OR and XOR (here called EOR) logical operations at the hardware gate level in Section 1.2 on page 3; the final mnemonic, BIC, stands for 'bit clear' where every '1' in the second operand clears the corresponding bit in the first. (The 'not' operation in the assembly language comment inverts each bit of the following operand.)

- Register movement operations.

  These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand (possibly bit-wise inverted) to the destination.

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

  The 'MVN' mnemonic stands for 'move negated'; it leaves the result register set to the value obtained by inverting every bit in the source operand.

- Comparison operations.

  These instructions do not produce a result (which is therefore omitted from the assembly language format) but just set the condition code bits (N, Z, C and V) in the CPSR according to the selected operation.

```
CMP     r1, r2          ; set cc on r1 - r2
CMN     r1, r2          ; set cc on r1 + r2
TST     r1, r2          ; set cc on r1 and r2
TEQ     r1, r2          ; set cc on r1 xor r2
```

  The mnemonics stand for 'compare' (CMP), 'compare negated' (CMN), '(bit) test' (TST) and 'test equal' (TEQ).

Immediate operands

If, instead of adding two registers, we simply wish to add a constant to a register we can replace the second source operand with an immediate value, which is a literal constant, preceded by '#':

```
ADD     r3, r3, #1          ; r3 := r3 + 1
AND     r8, r7, #&ff        ; r8 := r7[7:0]
```

The first example also illustrates that although the 3-address format allows source and destination operands to be specified separately, they are not required to be distinct registers. The second example shows that the immediate value may be specified in hexadecimal (base 16) notation by putting '&' after the '#'.

Since the immediate value is coded within the 32 bits of the instruction, it is not possible to enter every possible 32-bit value as an immediate. The values which can be entered correspond to any 32-bit binary number where all the binary ones fall within a group of eight adjacent bit positions on a 2-bit boundary. Most valid immediate values are given by:

$$immediate = (0 \rightarrow 255) \times 2^{2n}$$

Equation 10

where $0 \leq n \leq 12$. The assembler will also replace MOV with MVN, ADD with SUB, and so on, where this can bring the immediate within range.

This may appear a complex constraint on the immediate values, but it does, in practice, cover all the most common cases such as a byte value at any of the four byte positions within a 32-bit word, any power of 2, and so on. In any case the assembler will report any value which is requested that it cannot encode.

(The reason for the constraint on immediate values is the way they are specified at the binary instruction level. This is described in the Chapter 5, and the reader who wishes to understand this issue fully should look there for the complete explanation.)

Shifted register operands

A third way to specify a data operation is similar to the first, but allows the second register operand to be subject to a shift operation before it is combined with the first operand. For example:

```
ADD     r3, r2, r1, LSL #3 ; r3 := r2 + 8 x r1
```

Note that this is still a single ARM instruction, executed in a single clock cycle. Most processors offer shift operations as separate instructions, but the ARM combines them with a general ALU operation in a single instruction.

Here 'LSL' indicates 'logical shift left by the specified number of bits', which in this example is 3. Any number from 0 to 31 may be specified, though using 0 is equivalent to omitting the shift altogether. As before, '#' indicates an immediate quantity. The available shift operations are:

- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.
- ASL: arithmetic shift left; this is a synonym for LSL.
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positive, or with ones if the source operand was negative.
- ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.
- RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.

These shift operations are illustrated in Figure 3.1 on page 54.

It is also possible to use a register value to specify the number of bits the second operand should be shifted by:

```
ADD     r5, r5, r3, LSL r2 ; r5 := r5 + r3 x 2^r2
```

This is a 4-address instruction. Only the bottom eight bits of r2 are significant, but since shifts by more than 32 bits are not very useful this limitation is not important for most purposes.

Setting the condition codes

Any data processing instruction can set the condition codes (N, Z, C and V) if the programmer wishes it to. The comparison operations only set the condition codes, so there is no option with them, but for all other data processing instructions a

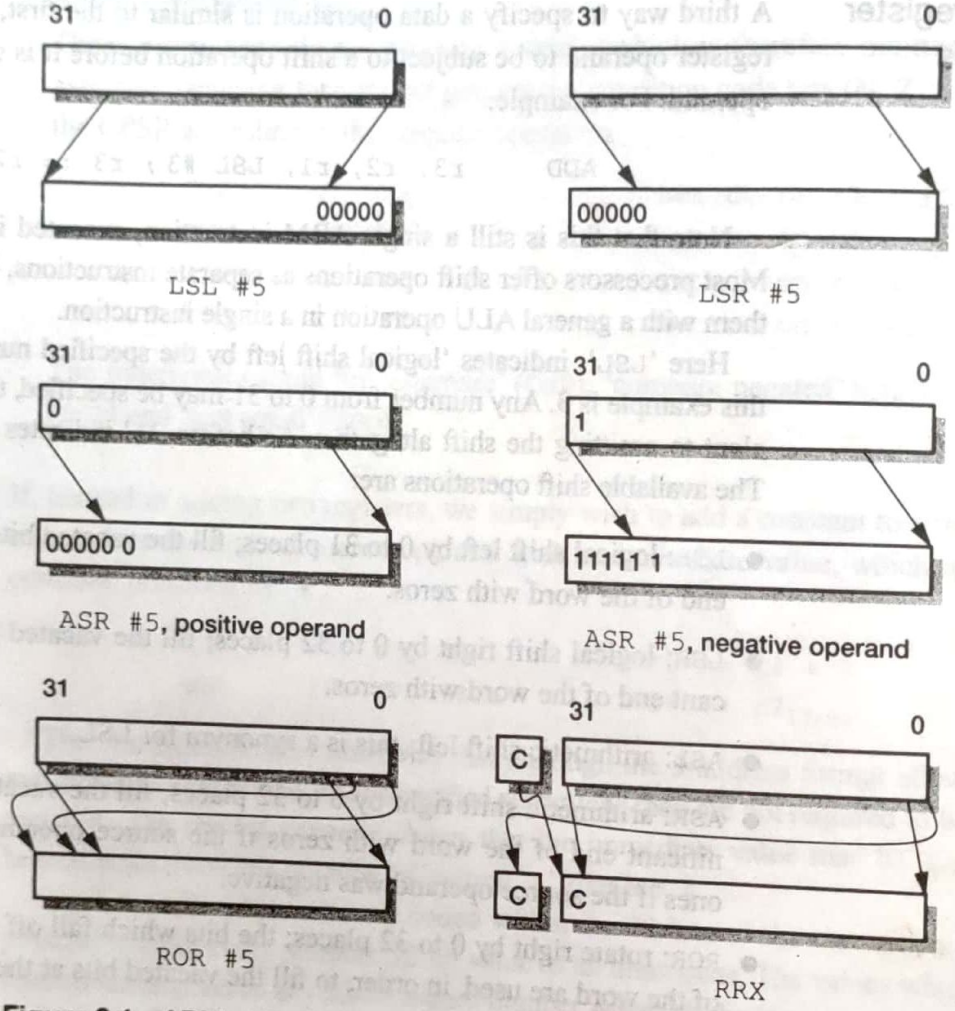

Figure 3.1 ARM shift operations

specific request must be made. At the assembly language level this request is indicated by adding an 'S' to the opcode, standing for 'Set condition codes'. As an example, the following code performs a 64-bit addition of two numbers held in r0–r1 and r2–r3, using the C condition code flag to store the intermediate carry:

```
ADDS    r2, r2, r0 ; 32-bit carry out -> C..
ADC     r3, r3, r1 ; .. and added into high word
```

Since the S opcode extension gives the programmer control over whether or not an instruction modifies the condition codes, the codes can be preserved over long instruction sequences when it is appropriate to do so.

An arithmetic operation (which here includes CMP and CMN) sets all the flags according to the arithmetic result. A logical or move operation does not produce a meaningful value for C or V, so these operations set N and Z according to the result but preserve V, and either preserve C when there is no shift operation, or set C to the value of the last bit to fall off the end of the shift. This detail is not often significant.

Use of the condition codes

We have already seen the C flag used as an input to an arithmetic data processing instruction. However we have not yet seen the most important use of the condition codes, which is to control the program flow through the conditional branch instructions. These will be described in Section 3.3 on page 63.

Multiplies

A special form of the data processing instruction supports multiplication:

```
MUL     r4, r3, r2          ; r4 := (r3 x r2)[31:0]
```

There are some important differences from the other arithmetic instructions:

- Immediate second operands are not supported.
- The result register must not be the same as the first source register.
- If the 'S' bit is set the V flag is preserved (as for a logical instruction) and the C flag is rendered meaningless.

Multiplying two 32-bit integers gives a 64-bit result, the least significant 32 bits of which are placed in the result register and the rest are ignored. This can be viewed as multiplication in modulo 2^{32} arithmetic and gives the correct result whether the operands are viewed as signed or unsigned integers. (ARMs also support long multiply instructions which place the most significant 32 bits into a second result register; these are described in Section 5.8 on page 122.)

An alternative form, subject to the same restrictions, adds the product to a running total. This is the multiply-accumulate instruction:

```
MLA     r4, r3, r2, r1      ; r4 := (r3 x r2 + r1)[31:0]
```

Multiplication by a constant can be implemented by loading the constant into a register and then using one of these instructions, but it is usually more efficient to use a short series of data processing instructions using shifts and adds or subtracts. For example, to multiply r0 by 35:

```
ADD     r0, r0, r0, LSL #2; r0' := 5 x r0
RSB     r0, r0, r0, LSL #3; r0" := 7 x r0' (= 35 x r0)
```

3.2 Data transfer instructions

Data transfer instructions move data between ARM registers and memory. There are three basic forms of data transfer instruction in the ARM instruction set:

- Single register load and store instructions.

  These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 32-bit word, or a 16-bit half-word. (Older ARM chips may not support half-words.)

- Multiple register load and store instructions.

  These instructions are less flexible than single register transfer instructions, but enable large quantities of data to be transferred more efficiently. They are used for procedure entry and exit, to save and restore workspace registers, and to copy blocks of data around memory.

- Single register swap instructions.

  These instructions allow a value in a register to be exchanged with a value in memory, effectively doing both a load and a store operation in one instruction. They are little used in user-level programs, so they will not be discussed further in this section. Their principal use is to implement semaphores to ensure mutual exclusion on accesses to shared data structures in multi-processor systems, but don't worry if this explanation has little meaning for you at the moment.

It is quite possible to write any program for the ARM using only the single register load and store instructions, but there are situations where the multiple register transfers are much more efficient, so the programmer should be familiar with them.

Register-indirect addressing

Towards the end of Section 1.4 on page 14 there was a discussion of memory addressing mechanisms that are available to the processor instruction set designer. The ARM data transfer instructions are all based around register-indirect addressing, with modes that include base-plus-offset and base-plus-index addressing.

Register-indirect addressing uses a value in one register (the **base** register) as a memory address and either **loads** the value from that address into another register or **stores** the value from another register into that memory address.

These instructions are written in assembly language as follows:

```
LDR     r0, [r1]          ; r0 := mem32[r1]
STR     r0, [r1]          ; mem32[r1] := r0
```

Other forms of addressing all build on this form, adding immediate or register offsets to the base address. In all cases it is necessary to have an ARM register loaded with an address which is near to the desired transfer address, so we will begin by looking at ways of getting memory addresses into a register.

Initializing an address pointer

To load or store from or to a particular memory location, an ARM register must be initialized to contain the address of that location, or, in the case of single register transfer instructions, an address within 4 Kbytes of that location (the 4 Kbyte range will be explained later).

If the location is close to the code being executed it is often possible to exploit the fact that the program counter, r15, is close to the desired address. A data processing instruction can be employed to add a small offset to r15, but calculating the appropriate offset may not be that straightforward. However, this is the sort of tricky calculation that assemblers are good at, and ARM assemblers have an inbuilt 'pseudo instruction', ADR, which makes this easy. A pseudo instruction looks like a normal

instruction in the assembly source code but does not correspond directly to a particular ARM instruction. Instead, the assembler has a set of rules which enable it to select the most appropriate ARM instruction or short instruction sequence for the situation in which the pseudo instruction is used. (In fact, ADR is always assembled into a single ADD or SUB instruction.)

As an example, consider a program which must copy data from TABLE1 to TABLE2, both of which are near to the code:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

Here we have introduced **labels** (COPY, TABLE1 and TABLE2) which are simply names given to particular points in the assembly code. The first ADR pseudo instruction causes r1 to contain the address of the data that follows TABLE1; the second ADR likewise causes r2 to hold the address of the memory starting at TABLE2.

Of course any ARM instruction can be used to compute the address of a data item in memory, but for the purposes of small programs the ADR pseudo instruction will do what we require.

Single register load and store instructions

These instructions compute an address for the transfer using a base register, which should contain an address near to the target address, and an offset which may be another register or an immediate value.

We have just seen the simplest form of these instructions, which does not use an offset:

```
LDR     r0, [r1]            ; r0 := mem32[r1]
STR     r0, [r1]            ; mem32[r1] := r0
```

The notation used here indicates that the data quantity is the 32-bit memory word addressed by r1. The word address in r1 should be aligned on a 4-byte boundary, so the two least significant bits of r1 should be zero. We can now copy the first word from one table to the other:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        LDR     r0, [r1]            ; load first value...
        STR     r0, [r2]            ; and store it in TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

We could now use data processing instructions to modify both base registers ready for the next transfer:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1]            ; get TABLE1 1st word
        STR     r0, [r2]            ; copy into TABLE2
        ADD     r1, r1, #4          ; step r1 on 1 word
        ADD     r2, r2, #4          ; step r2 on 1 word
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

Note that the base registers are incremented by 4 (bytes), since this is the size of a word. If the base register was word-aligned before the increment, it will be word-aligned afterwards too.

All load and store instructions could use just this simple form of register-indirect addressing. However, the ARM instruction set includes more addressing modes that can make the code more efficient.

Base plus offset addressing

If the base register does not contain exactly the right address, an offset of up to 4 Kbytes may be added (or subtracted) to the base to compute the transfer address:

```
        LDR     r0, [r1,#4]         ; r0 := mem32[r1 + 4]
```

This is a **pre-indexed** addressing mode. It allows one base register to be used to access a number of memory locations which are in the same area of memory.

Sometimes it is useful to modify the base register to point to the transfer address. This can be achieved by using pre-indexed addressing with **auto-indexing**, and allows the program to walk through a table of values:

```
        LDR     r0, [r1,#4]!        ; r0 := mem32[r1 + 4]
                                    ; r1 := r1 + 4
```

The exclamation mark indicates that the instruction should update the base register after initiating the data transfer. On the ARM this auto-indexing costs no extra time since it is performed on the processor's datapath while the data is being fetched from memory. It is exactly equivalent to preceding a simple register-indirect load with a data processing instruction that adds the offset (4 bytes in this example) to the base register, but the time and code space cost of the extra instruction are avoided.

Another useful form of the instruction, called **post-indexed** addressing, allows the base to be used without an offset as the transfer address, after which it is auto-indexed:

```
LDR     r0, [r1], #4        ; r0 := mem32[r1]
                            ; r1 := r1 + 4
```

Here the exclamation mark is not needed, since the only use of the immediate offset is as a base register modifier. Again, this form of the instruction is exactly equivalent to a simple register-indirect load followed by a data processing instruction, but it is faster and occupies less code space.

Using the last of these forms we can now improve on the table copying program example introduced earlier:

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        ???                         ; if more go back to LOOP
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

The load and store instructions are repeated until the required number of values has been copied into TABLE2, then the loop is exited. Control flow instructions are required to determine the loop exit; they will be introduced shortly.

In the above examples the address offset from the base register was always an immediate value. It can equally be another register, optionally subject to a shift operation before being added to the base, but such forms of the instruction are less useful than the immediate offset form. They are described fully in Section 5.10 on page 125.

As a final variation, the size of the data item which is transferred may be a single unsigned 8-bit byte instead of a 32-bit word. This option is selected by adding a letter B onto the opcode:

```
LDRB    r0, [r1]                ; r0 := mem8[r1]
```

In this case the transfer address can have any alignment and is not restricted to a 4-byte boundary, since bytes may be stored at any byte address. The loaded byte is placed in the bottom byte of r0 and the remaining bytes in r0 are filled with zeros.

(All but the oldest ARM processors also support **signed** bytes, where the top bit of the byte indicates whether the value should be treated as positive or negative, and signed and unsigned 16-bit half-words; these variants will be described when we return to look at the instruction set in more detail in Section 5.11 on page 128.)

Multiple register data transfers

Where considerable quantities of data are to be transferred, it is preferable to move several registers at a time. These instructions allow any subset (or all) of the 16 registers to be transferred with a single instruction. The trade-off is that the available addressing modes are more restricted than with a single register transfer instruction.

A simple example of this instruction class is:

```
LDMIA   r1, {r0,r2,r5}      ; r0 := mem32[r1]
                            ; r2 := mem32[r1 + 4]
                            ; r5 := mem32[r1 + 8]
```

Since the transferred data items are always 32-bit words, the base address (r1) should be word-aligned.

The transfer list, within the curly brackets, may contain any or all of r0 to r15. The order of the registers within the list is insignificant and does not affect the order of transfer or the values in the registers after the instruction has executed. It is normal practice, however, to specify the registers in increasing order within the list.

Note that including r15 in the list will cause a change in the control flow, since r15 is the PC. We will return to this case when we discuss control flow instructions and will not consider it further until then.

The above example illustrates a common feature of all forms of these instructions: the lowest register is transferred to or from the lowest address, and then the other registers are transferred in order of register number to or from consecutive word addresses above the first. However there are several variations on how the first address is formed, and auto-indexing is also available (again by adding a '!' after the base register).

Stack addressing

The addressing variations stem from the fact that one use of these instructions is to implement stacks within memory. A stack is a form of last-in-first-out store which supports simple dynamic memory allocation, that is, memory allocation where the address to be used to store a data value is not known at the time the program is compiled or assembled. An example would be a recursive function, where the depth of recursion depends on the value of the argument. A stack is usually implemented as a linear data structure which grows up (an **ascending** stack) or down (a **descending** stack) memory as data is added to it and shrinks back as data is removed. A **stack pointer** holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a **full** stack), or by pointing to the vacant slot where the next data item will be placed (an **empty** stack).

The above description suggests that there are four variations on a stack, representing all the combinations of ascending and descending full and empty stacks. The ARM multiple register transfer instructions support all four forms of stack:

- Full ascending: the stack grows up through increasing memory addresses and the base register points to the highest address containing a valid item.

- Empty ascending: the stack grows up through increasing memory addresses and the base register points to the first empty location above the stack.
- Full descending: the stack grows down through decreasing memory addresses and the base register points to the lowest address containing a valid item.
- Empty descending: the stack grows down through decreasing memory addresses and the base register points to the first empty location below the stack.

Block copy addressing

Although the stack view of multiple register transfer instructions is useful, there are occasions when a different view is easier to understand. For example, when these instructions are used to copy a block of data from one place in memory to another a mechanistic view of the addressing process is more useful. Therefore the ARM assembler supports two different views of the addressing mechanism, both of which map onto the same basic instructions, and which can be used interchangeably. The block copy view is based on whether the data is to be stored above or below the address held in the base register and whether the address incrementing or decrementing begins before or after storing the first value. The mapping between the two views depends on whether the operation is a load or a store, and is detailed in Table 3.1 on page 62.

The block copy views are illustrated in Figure 3.2 on page 62, which shows how each variant stores three registers into memory and how the base register is modified if auto-indexing is enabled. The base register value before the instruction is r9, and after the auto-indexing it is r9'.

To illustrate the use of these instructions, here are two instructions which copy eight words from the location r0 points to to the location r1 points to:

```
LDMIA   r0!, {r2-r9}
STMIA   r1,  {r2-r9}
```

After executing these instructions r0 has increased by 32 since the '!' causes it to auto-index across eight words, whereas r1 is unchanged. If r2 to r9 contained useful values, we could preserve them across this operation by pushing them onto a stack:

```
STMFD   r13!, {r2-r9}      ; save regs onto stack
LDMIA   r0!,  {r2-r9}
STMIA   r1,   {r2-r9}
LDMFD   r13!, {r2-r9}      ; restore from stack
```

Here the 'FD' postfix on the first and last instructions signifies the full descending stack address mode as described earlier. Note that auto-indexing is almost always specified for stack operations in order to ensure that the stack pointer has a consistent behaviour.

The load and store multiple register instructions are an efficient way to save and restore processor state and to move blocks of data around in memory. They save code space and operate up to four times faster than the equivalent sequence of single

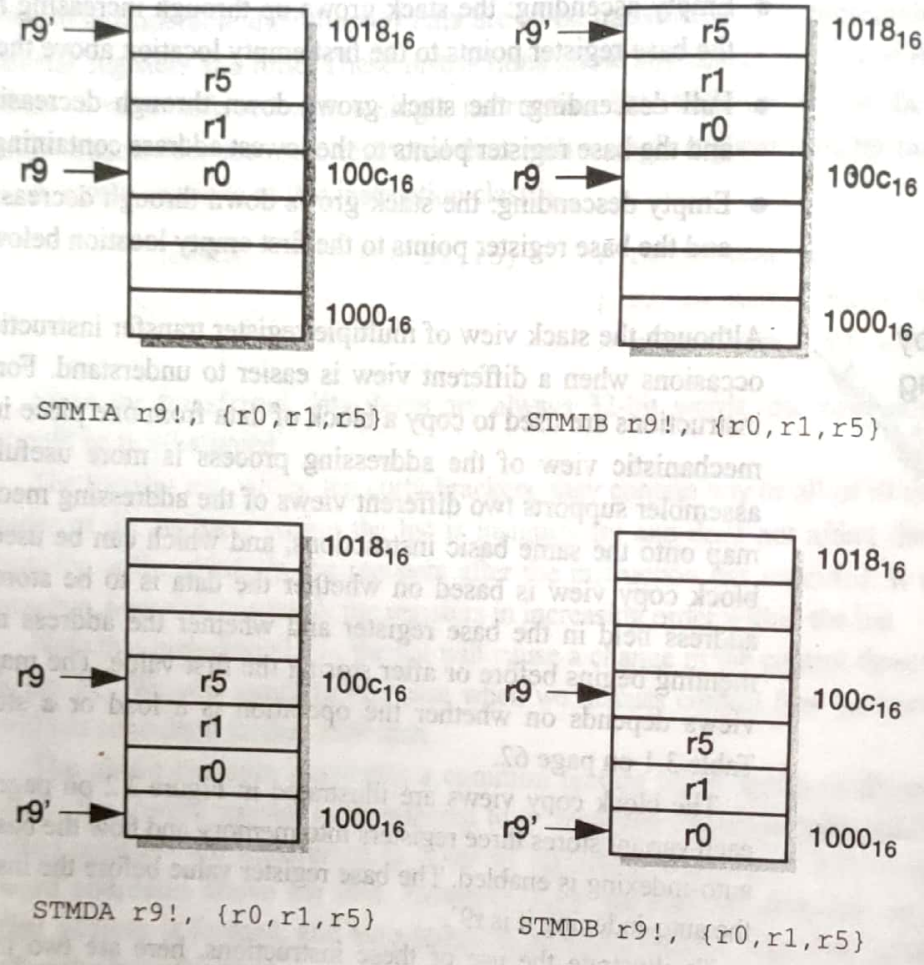

Figure 3.2 Multiple register transfer addressing modes.

Table 3.1 The mapping between the stack and block copy views of the load and store multiple instructions.

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

register load or store instructions (a factor of two due to improved sequential behaviour and another factor of nearly two due to the reduced instruction count). This significant advantage suggests that it is worth thinking carefully about how data is organized in memory in order to maximize the potential for using multiple register data transfer instructions to access it.

These instructions are, perhaps, not pure 'RISC' since they cannot be executed in a single clock cycle even with separate instruction and data caches, but other RISC architectures are beginning to adopt multiple register transfer instructions in order to increase the data bandwidth between the processor's registers and the memory.

On the other side of the equation, load and store multiple instructions are complex to implement, as we shall see later.

The ARM multiple register transfer instructions are uniquely flexible in being able to transfer any subset of the 16 currently visible registers, and this feature is powerfully exploited by the ARM procedure call mechanism which is described in Section 6.8 on page 175.

3.3 Control flow instructions

This third category of instructions neither processes data nor moves it around; it simply determines which instructions get executed next.

Branch instructions

The most common way to switch program execution from one place to another is to use the branch instruction:

```
        B       LABEL
        ..
LABEL   ..
```

The processor normally executes instructions sequentially, but when it reaches the branch instruction it proceeds directly to the instruction at `LABEL` instead of executing the instruction immediately after the branch. In this example `LABEL` comes after the branch instruction in the program, so the instructions in between are skipped. However, `LABEL` could equally well come before the branch, in which case the processor goes back to it and possibly repeats some instructions it has already executed.

Conditional branches

Sometimes you will want the processor to take a decision whether or not to branch. For example, to implement a loop a branch back to the start of the loop is required, but this branch should only be taken until the loop has been executed the required number of times, then the branch should be skipped.

The mechanism used to control loop exit is conditional branching. Here the branch has a condition associated with it and it is only executed if the condition codes have the correct value. A typical loop control sequence might be:

```
        MOV     r0, #0          ; initialize counter
LOOP    ...
        ADD     r0, r0, #1      ; increment loop counter
        CMP     r0, #10         ; compare with limit
        BNE     LOOP            ; repeat if not equal
        ...                     ; else fall through
```

This example shows one sort of conditional branch, BNE, or 'branch if not equal'. There are many forms of the condition. All the forms are listed in Table 3.2, along with their normal interpretations. The pairs of conditions which are listed in the same row of the table (for instance BCC and BLO) are synonyms which result in identical binary code, but both are available because each makes the interpretation of the assembly source code easier in particular circumstances. Where the table refers to signed or unsigned comparisons this does not reflect a choice in the comparison instruction itself but supports alternative interpretations of the operands.

Table 3.2 Branch conditions.

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional execution

An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions. A branch which is used to skip a small number of following instructions may be omitted altogether by giving those instructions the opposite condition. For example, consider the following sequence:

```
        CMP     r0, #5
        BEQ     BYPASS          ; if (r0 != 5) {
        ADD     r1, r1, r0      ;   r1 := r1 + r0 - r2
        SUB     r1, r1, r2      ; }
BYPASS  ..
```

This may be replaced by:

```
        CMP     r0, #5          ; if (r0 != 5) {
        ADDNE   r1, r1, r0      ;   r1 := r1 + r0 - r2
        SUBNE   r1, r1, r2      ; }
        ..
```

The new sequence is both smaller and faster than the old one. Whenever the conditional sequence is three instructions or fewer it is better to exploit conditional execution than to use a branch, provided that the skipped sequence is not doing anything complicated with the condition codes within itself.

(The three instruction guideline is based on the fact that ARM branch instructions typically take three cycles to execute, and it is only a guideline. If the code is to be fully optimized then the decision on whether to use conditional execution or a branch must be based on measurements of the dynamic code behaviour.)

Conditional execution is invoked by adding the 2-letter condition after the 3-letter opcode (and before any other instruction modifier letter such as the 'S' that controls setting the condition codes in a data processing instruction or the 'B' that specifies a byte load or store).

Just to emphasize the scope of this technique, note that every ARM instruction, including supervisor calls and coprocessor instructions, may have a condition appended which causes it to be skipped if the condition is not met.

It is sometimes possible to write very compact code by cunning use of conditionals, for example:

```
; if ((a==b) && (c==d)) e++;

        CMP     r0, r1
        CMPEQ   r2, r3
        ADDEQ   r4, r4, #1
```

Note how if the first comparison finds unequal operands the second is skipped, causing the increment to be skipped also. The logical 'and' in the if clause is implemented by making the second comparison conditional.

Branch and link instructions

A common requirement in a program is to be able to branch to a subroutine in a way which makes it possible to resume the original code sequence when the subroutine has completed. This requires that a record is kept of the value of the program counter just before the branch is taken.

ARM offers this functionality through the branch and link instruction which, as well as performing a branch in exactly the same way as the branch instruction, also saves the address of the instruction following the branch in the link register, r14:

```
        BL      SUBR                ; branch to SUBR
        ..                          ; return to here
SUBR    ..                          ; subroutine entry point
        MOV     pc, r14             ; return
```

Note that since the return address is held in a register, the subroutine should not call a further, nested, subroutine without first saving r14, otherwise the new return address will overwrite the old one and it will not be possible to find the way back to the original caller. The normal mechanism used here is to push r14 onto a stack in memory. Since the subroutine will often also require some work registers, the old values in these registers can be saved at the same time using a store multiple instruction:

```
        BL      SUB1
        ..
SUB1    STMFD   r13!, {r0-r2,r14}  ; save work & link regs
        BL      SUB2
        ..
SUB2    ..
```

A subroutine that does not call another subroutine (a **leaf** subroutine) need not save r14 since it will not be overwritten.

Subroutine return instructions

To get back to the calling routine, the value saved by the branch and link instruction in r14 must be copied back into the program counter. In the simplest case of a leaf subroutine (a subroutine that does not call another subroutine) a MOV instruction suffices, exploiting the visibility of the program counter as r15:

```
SUB2    ..
        MOV     pc, r14 ; copy r14 into r15 to return
```

In fact the availability of the program counter as r15 means that any of the data processing instructions can be used to compute a return address, though the 'MOV' form is by far the most commonly used.

Where the return address has been pushed onto a stack, it can be restored along with any saved work registers using a load multiple instruction:

```
SUB1    STMFD   r13!, {r0-r2,r14}; save work regs & link
        BL      SUB2
        ..
        LDMFD   r13!, {r0-r2,pc} ; restore work regs & return
```

Note here how the return address is restored directly to the program counter, not to the link register. This single restore and return instruction is very powerful. Note also the use of the stack view of the multiple register transfer addressing modes. The same stack model (in this case 'full descending', which is the most common stack type for ARM code) is used for both the store and the load, ensuring that the correct values will be collected. It is important that for any particular stack the same addressing mode is used for every use of the stack, unless you really know what you are doing.

Supervisor calls

Whenever a program requires input or output, for instance to send some text to the display, it is normal to call a supervisor routine. The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly. The limitations on the capabilities of a user-level program vary from system to system, but in many systems the user cannot access hardware facilities directly.

The supervisor provides trusted ways to access system resources which appear to the user-level program rather like special subroutine accesses. The instruction set includes a special instruction, `SWI`, to call these functions. (`SWI` stands for 'SoftWare Interrupt', but is usually pronounced 'Supervisor Call'.)

Although the supervisor calls are implemented in system software, and could therefore be totally different from one ARM system to another, most ARM systems implement a common subset of calls in addition to any specific calls required by the particular application. The most useful of these is a routine which sends the character in the bottom byte of r0 to the user display device:

```
SWI     SWI_WriteC          ; output r0[7:0]
```

Another useful call returns control from a user program back to the monitor program:

```
SWI     SWI_Exit            ; return to monitor
```

The operation of `SWI`s is described in more detail in Section 5.6 on page 117.

Jump tables

Jump tables are not normally used by less experienced programmers, so you can ignore this section if you are relatively new to programming at the assembly level.

The idea of a jump table is that a programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program. It is clearly possible to do this with the instructions we have seen already. Suppose the value is in r0. We can then write:

```
        BL      JUMPTAB
        ..
JUMPTAB CMP     r0, #0
        BEQ     SUB0
        CMP     r0, #1
        BEQ     SUB1
        CMP     r0, #2
        BEQ     SUB2
```

However, this solution becomes very slow when the list of subroutines is long unless there is some reason to think that the later choices will rarely be used. A solution which is more efficient in this case exploits the visibility of the program counter in the general register file:

```
        BL      JUMPTAB
        ..
JUMPTAB ADR     r1, SUBTAB          ; r1 -> SUBTAB
        CMP     r0, #SUBMAX         ; check for overrun..
        LDRLS   pc, [r1,r0,LSL #2]  ;  .. if OK, table jump
        B       ERROR               ;  .. otherwise signal error
SUBTAB  DCD     SUB0                ; table of subroutine
        DCD     SUB1                ;       entry points
        DCD     SUB2
        ..
```

The 'DCD' directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right, which in these cases is just the address of the label.

This approach has a constant performance however many subroutines are in the table and independent of the distribution of frequency of use. Note, however, that the consequences of reading beyond the end of the table are likely to be dire, so checking for overrun is essential! Here, note how the overrun check is implemented by making the load into the PC conditional, so the overrun case skips the load and falls into the branch to the error handler. The only performance cost of checking for overrun is the comparison with the maximum value. More obvious code might have been:

```
        ..
        CMP     r0, #SUBMAX         ; check for overrun..
        BHI     ERROR               ;  .. if overrun call error
        LDR     pc, [r1,r0,LSL #2]  ;  .. else table jump
        ..
```

but note that here the cost of conditionally skipping the branch is borne every time the jump table is used. The original version is more efficient except when overrun is detected, which should be infrequent and, since it represents an error, performance in that case is not of great concern.

An alternative, less obvious way to implement a jump table is discussed in 'switches' on page 171.

3.4 Writing simple assembly language programs

We now have all the basic tools for writing simply assembly language programs. As with any programming task, it is important to have a clear idea of your algorithm before beginning to type instructions into the computer. Large programs are almost certainly better written in C or C++, so we will only look at small examples of assembly language programs.

Even the most experienced programmers begin by checking that they can get a very simple program to run before moving on to whatever their real task is. There are so many complexities to do with learning to use a text editor, working out how to get the assembler to run, how to load the program into the machine, how to get it to start executing and so on. This sort of simple test program is often referred to as a *Hello World* program because all it does is print 'Hello World' on the display before terminating. Here is an ARM assembly language version:

```
        AREA    HelloW,CODE,READONLY ; declare code area
SWI_WriteC      EQU     &0           ; output character in r0
SWI_Exit        EQU     &11          ; finish program
        ENTRY                        ; code entry point
START   ADR     r1, TEXT             ; r1 -> "Hello World"
LOOP    LDRB    r0, [r1], #1         ; get the next byte
        CMP     r0, #0               ; check for text end
        SWINE   SWI_WriteC           ; if not end print ..
        BNE     LOOP                 ;  .. and loop back
        SWI     SWI_Exit             ; end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                          ; end of program source
```

This program illustrates a number of the features of the ARM assembly language and instruction set:

- The declaration of the code 'AREA', with appropriate attributes.
- The definitions of the system calls which will be used in the routine. (In a larger program these would be defined in a file which other code files would reference.)
- The use of the ADR pseudo instruction to get an address into a base register.
- The use of auto-indexed addressing to move through a list of bytes.
- Conditional execution of the SWI instruction to avoid an extra branch.

Note also the use of a zero byte to mark the end of the string (following the line-feed and carriage return special characters). Whenever you use a looping structure, make sure it has a terminating condition.

In order to run this program you will need the following tools, all of which are available within the ARM software development toolkit:

- A text editor to type the program into.
- An assembler to turn the program into ARM binary code.
- An ARM system or emulator to execute the binary on. The ARM system must have some text output capability. (The ARM development board, for example, sends text output back up to the host for output onto the host's display.)

Once you have this program running you are ready to try something more useful. From now on, the only thing that changes is the program text. The use of the editor, the assembler, and the test system or emulator will remain pretty similar to what you have done already, at least up to the point where your program refuses to do what you want and you can't see why it refuses. Then you will need to use a debugger to see what is happening inside your program. This means learning how to use another complex tool, so we will put off that moment for as long as possible.

For the next example, we can now complete the block copy program developed partially earlier in the text. To ensure that we know it has worked properly, we will use a text source string so that we can output it from the destination address, and we will initialize the destination area to something different:

```
        AREA    BlkCpy,CODE,READONLY
SWI_WriteC      EQU     &0
SWI_Exit        EQU     &11             ; output character in r0
        ENTRY                           ; finish program
        ADR     r1, TABLE1              ; code entry point
        ADR     r2, TABLE2              ; r1 -> TABLE1
        ADR     r3, T1END               ; r2 -> TABLE2
LOOP1   LDR     r0, [r1], #4            ; r3 -> T1END
        STR     r0, [r2], #4            ; get TABLE1 1st word
        CMP     r1, r3                  ; copy into TABLE2
        BLT     LOOP1                   ; finished?
        ADR     r1, TABLE2              ; if not, do more
LOOP2   LDRB    r0, [r1], #1            ; r1 -> TABLE2
        CMP     r0, #0                  ; get next byte
        SWINE   SWI_WriteC              ; check for text end
        BNE     LOOP2                   ; if not end, print ..
        SWI     SWI_Exit                ;   .. and loop back
TABLE1  =       "This is the right string!", &0a, &0d, 0
T1END                                   ; finish
        ALIGN                           ; ensure word alignment
TABLE2  =       "This is the wrong string!", 0
        END
```

This program uses word loads and stores to copy the table, which is why the tables must be word-aligned. It then uses byte loads to print out the result using a routine which is the same as that used in the 'Hello World' program.

Note the use of 'BLT' to control the loop termination. If TABLE1 contains a number of bytes which is not a multiple of four, there is a danger that r1 would step past T1END without ever exactly equalling it, so a termination condition based on 'BNE' might fail.

If you have succeeded in getting this program running, you are well on the way to understanding the basic operation of the ARM instruction set. The examples and exercises which follow should be studied to reinforce this understanding. As you attempt more complex programming tasks, questions of detail will arise. These should be answered by the full instruction set description given in Chapter 5.

Program design

With a basic understanding of the instruction set, small programs can be written and debugged without too much trouble by just typing them into an editor and seeing if they work. However, it is dangerous to assume that this simple approach will scale to the successful development of complex programs which may be expected to work for many years, which may be changed by other programmers in the future, and which may end up in the hands of customers who will use use them in unexpected ways.

This book is not a text on program design, but having offered an introduction to programming it would be a serious omission not to point out that there is a lot more to writing a useful program than just sitting down and typing code.

Serious programming should start not with coding, but with careful design. The first step of the development process is to understand the requirements; it is surprising how often programs do not behave as expected because the requirements were not well understood by the programmer! Then the (often informal) requirements should be translated into an unambiguous specification. Now the design can begin, defining a program structure, the data structures that the program works with and the algorithms that are used to perform the required operations on the data. The algorithms may be expressed in **pseudo-code**, a program-like notation which does not follow the syntax of a particular programming language but which makes the meaning clear.

Only when the design is developed should the coding begin. Individual modules should be coded, tested thoroughly (which may require special programs to be designed as 'test-harnesses') and documented, and the program built piece by piece.

Today nearly all programming is based on high-level languages, so it is rare for large programs to be built using assembly programming as described here. Sometimes, however, it may be necessary to develop small software components in assembly language to get the best performance for a critical application, so it is useful to know how to write assembly code for these purposes.

3.5 Examples and exercises

Once you have the basic flavour of an instruction set the easiest way to learn to write programs is to look at some examples, then attempt to write your own program to do something slightly different. To see whether or not your program works you will need an ARM assembler and either an ARM emulator or hardware with an ARM processor in it. The following sections contain example ARM programs and suggestions for modifications to them. You should get the original program working first, then see if you can edit it to perform the modified function suggested in the exercises.

Example 3.1 **Print out r1 in hexadecimal.**

This is a useful little routine which dumps a register to the display in hexadecimal (base 16) notation; it can be used to help debug a program by writing out register values and checking that algorithms are producing the expected results, though in most cases using a debugger is a better way of seeing what is going on inside a program.

```
        AREA    Hex_Out,CODE,READONLY
SWI_WriteC      EQU     &0              ; output character in r0
SWI_Exit        EQU     &11             ; finish program
        ENTRY                           ; code entry point
        LDR     r1, VALUE               ; get value to print
        BL      HexOut                  ; call hexadecimal output
        SWI     SWI_Exit                ; finish
VALUE   DCD     &12345678               ; test value
HexOut  MOV     r2, #8                  ; nibble count = 8
LOOP    MOV     r0, r1, LSR #28         ; get top nibble
        CMP     r0, #9                  ; 0-9 or A-F?
        ADDGT   r0, r0, #"A"-10         ; ASCII alphabetic
        ADDLE   r0, r0, #"0"            ; ASCII numeric
        SWI     SWI_WriteC              ; print character
        MOV     r1, r1, LSL #4          ; shift left one nibble
        SUBS    r2, r2, #1              ; decrement nibble count
        BNE     LOOP                    ; if more do next nibble
        MOV     pc, r14                 ; return
        END
```

Exercise 3.1.1 Modify the above program to output r1 in binary format. For the value loaded into r1 in the example program you should get:

```
00010010001101000101011001111000
```

Exercise 3.1.2 Use HEXOUT as the basis of a program to display the contents of an area of memory.

Example 3.2 **Write a subroutine to output a text string immediately following the call.**

It is often useful to be able to output a text string without having to set up a separate data area for the text (though this is inefficient if the processor has separate data and instruction caches, as does the StrongARM; in this case it is better to set up a separate data area). A call should look like:

```
        BL      TextOut
        =       "Test string",&0a,&0d,0
        ALIGN
        ..                          ; return to here
```

The issue here is that the return from the subroutine must not go directly to the value put in the link register by the call, since this would land the program in the text string. Here is a suitable subroutine and test harness:

```
        AREA    Text_Out,CODE,READONLY
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
        ENTRY                       ; code entry point
        BL      TextOut             ; print following string
        =       "Test string",&0a,&0d,0
        ALIGN
        SWI     SWI_Exit            ; finish
TextOut LDRB    r0, [r14], #1       ; get next character
        CMP     r0, #0              ; test for end mark
        SWINE   SWI_WriteC          ; if not end, print..
        BNE     TextOut             ; .. and loop
        ADD     r14, r14, #3        ; pass next word boundary
        BIC     r14, r14, #3        ; round back to boundary
        MOV     pc, r14             ; return
        END
```

This example shows r14 incrementing along the text string and then being adjusted to the next word boundary prior to the return. If the adjustment (add 3, then clear the bottom two bits) looks like slight of hand, check it; there are only four cases.

Exercise 3.2.1 Using code from this and the previous examples, write a program to dump the ARM registers in hexadecimal with formatting such as:

```
r0 = 12345678
r1 = 9ABCDEF0
```

Exercise 3.2.2 Now try to save the registers you need to work with before they are changed, for instance by saving them near the code using PC-relative addressing.