3. The instruction set

We now know what the ARM provides by way of memory and registers, and the sort of instructions to manipulate them.This chapter describes those instructions in great detail.

As explained in the previous chapter, all ARM instructions are 32 bits long. Here is a typical one:

```
10101011100101010010100111101011
```

Fortunately, we don't have to write ARM programs using such codes. Instead we use assembly language. We saw at the end of Chapter One a few typical ARM mnemonics. Usually, mnemonics are followed by one or more operands which are used to completely describe the instruction.

An example mnemonic is **ADD**, for 'add two registers'. This alone doesn't tell the assembler which registers to add and where to put the result. If the left and right hand side of the addition are R1 and R2 respectively, and the result is to go in R0, the operand part would be written R0,R1,R2. Thus the complete add instruction, in assembler format, would be:

```
ADD    R0, R1, R2    ;R0 = R1 + R2
```

Most ARM mnemonics consist of three letters, e.g. **SUB**, **MOV**, **STR**, **STM**. Certain 'optional extras' may be added to slightly alter the affect of the instruction, leading to mnemonics such as **ADCNES** and **SWINE**.

The mnemonics and operand formats for all of the ARM's instructions are described in detail in the sections below. At this stage, we don't explain how to create programs, assemble and run them. There are two main ways of assembling ARM programs - using the assembler built-in to BBC BASIC, or using a dedicated assembler. The former method is more convenient for testing short programs, the latter for developing large scale projects. Chapter Four covers the use of the BASIC assembler.

3.1 Condition codes

The property of conditional execution is common to all ARM instructions, so its representation in assembler is described before the syntax of the actual instructions.

As mentioned in chapter two, there are four bits of condition encoded into an instruction word. This allows sixteen possible conditions. If the condition for the current instruction is true, the execution goes ahead. If the condition does not hold, the instruction is ignored and the next one executed.

The result flags are altered mainly by the data manipulation instructions. These instructions only affect the flags if you explicitly tell them to. For example, a **`MOV`** instruction which copies the contents of one register to another. No flags are affected. However, the **`MOVS`** (move with **`S`**et) instruction additionally causes the result flags to be set. The way in which each instruction affects the flags is described below.

To make an instruction conditional, a two-letter suffix is added to the mnemonic. The suffixes, and their meanings, are listed below.

AL Always

An instruction with this suffix is always executed. To save having to type '**`AL`**' after the majority of instructions which are unconditional, the suffix may be omitted in this case. Thus **`ADDAL`** and **`ADD`** mean the same thing: add unconditionally.

NV Never

All ARM conditions also have their inverse, so this is the inverse of always. Any instruction with this condition will be ignored. Such instructions might be used for 'padding' or perhaps to use up a (very) small amount of time in a program.

EQ Equal

This condition is true if the result flag Z (zero) is set. This might arise after a compare instruction where the operands were equal, or in any data instruction which received a zero result into the destination.

NE Not equal

This is clearly the opposite of **`EQ`**, and is true if the Z flag is cleared. If Z is set, and instruction with the **`NE`** condition will not be executed.

VS Overflow set

This condition is true if the result flag V (overflow) is set. Add, subtract and compare instructions affect the V flag.

VC Overflow clear

The opposite to **`VS`**.

MI Minus

Instructions with this condition only execute if the N (negative) flag is set. Such a condition would occur when the last data operation gave a result which was negative. That is, the N flag reflects the state of bit 31 of the result. (All data operations work on 32-bit numbers.)

PL Plus

This is the opposite to the **`MI`** condition and instructions with the **`PL`** condition will only execute if the N flag is cleared.

The next four conditions are often used after comparisons of two unsigned numbers. If the numbers being compared are n1 and n2, the conditions are n1>=n2, n1<n2, n1>n2 and n1<=n2, in the order presented.

CS Carry set

This condition is true if the result flag C (carry) is set. The carry flag is affected by arithmetic instructions such as **`ADD`**, **`SUB`** and **`CMP`**. It is also altered by operations involving the shifting or rotation of operands (data manipulation instructions).

When used after a compare instruction, **`CS`** may be interpreted as 'higher or same', where the operands are treated as unsigned 32-bit numbers. For example, if the left hand operand of **`CMP`** was 5 and the right hand operand was 2, the carry would be set. You can use **`HS`** instead of **`CS`** for this condition.

CC Carry clear

This is the inverse condition to **`CS`**. After a compare, the **`CC`** condition may be interpreted as meaning 'lower than', where the operands are again treated as unsigned numbers. An synonym for **`CC`** is **`LO`**.

HI Higher

This condition is true if the C flag is set and the Z flag is false. After a operand being greater than the right hand one, where the operands are treated as unsigned.

LS Lower or same

This condition is true if the C flag is cleared or the Z flag is set. After a compare or subtract, this combination may be interpreted as the left hand operand being less than or equal to the right hand one, where the operands are treated as unsigned.

The next four conditions have similar interpretations to the previous four, but are used when signed numbers have been compared. The difference is that they take into account the state of the V (overflow) flag, whereas the unsigned ones don't.

Again, the relationships between the two numbers which would cause the condition to be true are n1>=n2, n1<n2, n1>n2, n1<=n2.

GE Greater than or equal

This is true if N is cleared and V is cleared, or N is set and V is set.

LT Less than

This is the opposite to **`GE`** and instructions with this condition are executed if N is set and V is cleared, or N is cleared and V is set.

GT Greater than

This is the same as **`GE`**, with the addition that the Z flag must be cleared too.

LE Less than or equal

This is the same as **`LT`**, and is also true whenever the Z flag is set.

Note that although the conditions refer to signed and unsigned numbers, the operations on the numbers are identical regardless of the type. The only things that change are the flags used to determine whether instructions are to be obeyed or not.

The flags may be set and cleared explicitly by performing operations directly on R15, where they are stored.

3.2 Group one - data manipulation

This group contains the instructions which do most of the manipulation of data in ARM programs. The other groups are concerned with moving data between the processor and memory, or changing the flow of control.

The group comprises sixteen distinct instructions. All have a very similar format with respect to the operands they take and the 'optional extras'. We shall describe them generically using **ADD**, then give the detailed operation of each type.

Assembler format

ADD has the following format:

```
ADD{cond}{S}  <dest>,  <lhs>,  <rhs>
```

The parts in curly brackets are optional. **Cond** is one of the two-letter condition codes listed above. If it is omitted, the 'always' condition **AL** is assumed. The **S**, if present, causes the instruction to affect the result flags. If there is no **S**, none of the flags will be changed. For example, if an instruction **ADDS** … yields a result which is negative, then the N flag will be set. However, just **ADD** … will not alter N (or any other flag) regardless of the result.

After the mnemonic are the three operands. **<dest>** is the destination, and is the register number where the result of the **ADD** is to be stored. Although the assembler is happy with actual numbers here, e.g. 0 for R0, it recognises R0, R1, R2 etc. to stand for the register numbers. In addition, you can define a name for a register and use that instead. For example, in BBC BASIC you could say:-

```
iac = 0
```

where **iac** stands for, say, integer accumulator. Then this can be used in an instruction:-

```
ADD    iac,  iac,  #1
```

The second operand is the left hand side of the operation. In general, the group one instructions act on two values to provide the result. These are referred to as the left and right hand sides, implying that the operation determined by the mnemonic would be written between them in mathematics. For example, the instruction:

```
ADD R0,  R1,  R2
```

has R1 and R2 as its left and right hand sides, and R0 as the result. This is analogous to an assignment such as **R0=R1+R2** in BASIC, so the operands are sometimes said to be in 'assignment order'.

The **<lhs>** operand is always a register number, like the destination. The **<rhs>** may either be a register, or an immediate operand, or a shifted or rotated register. It is the versatile form that the right hand side may take which gives much of the power to these instructions.

If the **<rhs>** is a simple register number, we obtain instructions such as the first **ADD** example above. In this case, the contents of R1 and R2 are added (as signed, 32-bit numbers) and the result stored in R0. As there is no condition after the instruction, the **ADD** instruction will always be executed. Also, because there was no **S**, the result flags would not be affected.

The three examples below all perform the same **ADD** operation (if the condition is true):

```
ADDNE  R0, R0, R2
ADDS   R0, R0, R2
ADDNES R0, R0, R2
```

They all add R2 to R0. The first has a **NE** condition, so the instruction will only be executed if the Z flag is cleared. If Z is set when the instruction is encountered, it is ignored. The second one is unconditional, but has the **S** option. Thus the N, Z, V and C flags will be altered to reflect the result. The last example has the condition and the **S**, so if Z is cleared, the **ADD** will occur and the flags set accordingly. If Z is set, the **ADD** will be skipped and the flags remain unaltered.

Immediate operands

Immediate operands are written as a **#** followed by a number. For example, to increment R0, we would use:

```
ADD    R0, R0, #1
```

Now, as we know, an ARM instruction has 32 bits in which to encode the instruction type, condition, operands etc. In group one instructions there are twelve bits available to encode immediate operands. Twelve bits of binary can represent numbers in the range 0..4095, or -2048..+2047 if we treat them as signed.

The designers of the ARM decided not to use the 12 bits available to them for immediate operands in the obvious way just mentioned. Remember that some of the status bits are stored in bits 26..31 of R15. If we wanted to store an immediate value there using a group one instruction, there's no way we could using the straightforward twelve-bit number approach.

To get around this and related problems, the immediate operand is split into two fields, called the position (the top four bits) and the value (stored in the lower eight bits). The value is an eight bit number representing 256 possible combinations. The position is a four bit field which determines where in the 32-bit word the value lies. Below is a diagram showing how the sixteen values of the position determine where the value goes. The bits of the value part are shown as 0, 1, 2 etc.

The way of describing this succinctly is to say that the value is rotated by 2*position bits to the right within the 32-bit word. As you can see from the diagram, when position=&03, all of the status bits in R15 can be reached.

```
|Bit 31                    Bit 0|      Position
|                               |
........................76543210       &00
10........................765432       &01
3210........................7654       &02
543210........................76       &03
76543210........................       &04
..76543210......................       &05
....76543210....................       &06
......76543210..................       &07
........76543210................       &08
..........76543210..............       &09
............76543210............       &0A
..............76543210..........       &0B
................76543210........       &0C
..................76543210......       &0D
....................76543210....       &0E
......................76543210..       &0F
```

The sixteen immediate shift positions

When using immediate operands, you don't have to specify the number in terms of position and value. You just give the number you want, and the assembler tries to generate the appropriate twelve-bit field. If you specify a value which can't be generated, such as &101 (which would require a nine-bit value), an error is generated. The **ADD** instruction below adds 65536 (&1000) to R0:

```
ADD    R0, R0, #&1000
```

To get this number, the assembler might use a position value of 8 and value of 1, though other combinations could also be used.

Shifted operands

If the **`<rhs>`** operand is a register, it may be manipulated in various ways before it is used in the instruction. The contents of the register aren't altered, just the value given to the ALU, as applied to this operation (unless the same register is also used as the result, of course).

The particular operations that may be performed on the **`<rhs>`** are various types of shifting and rotation. The number of bits by which the register is shifted or rotated may be given as an immediate number, or specified in yet another register.

Shifts and rotates are specified as left or right, logical or arithmetic. A left shift is one where the bits, as written on the page, are moved by one or more bits to the left, i.e. towards the more significant end. Zero-valued bits are shifted in at the right and the bits at the left are lost, except for the final bit to be shifted out, which is stored in the carry flag.

Left shifts by n bits effectively multiply the number by 2^n, assuming that no significant bits are 'lost' at the top end.

A right shift is in the opposite direction, the bits moving from the more significant end to the lower end, or from left to right on the page. Again the bits shifted out are lost, except for the last one which is put into the carry. If the right shift is logical then zeros are shifted into the left end. In arithmetic shifts, a copy of bit 31 (i.e. the sign bit) is shifted in.

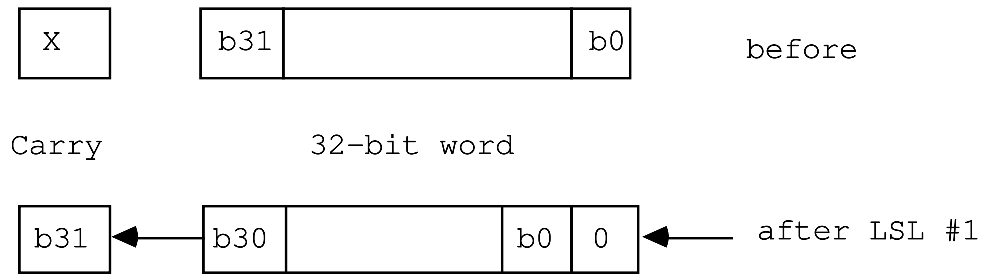
Right arithmetic shifts by n bits effectively divide the number by 2^n, rounding towards minus infinity (like the BASIC **`INT`** function).

A rotate is like a shift except that the bits shifted in to the left (right) end are those which are coming out of the right (left) end.

Here are the types of shifts and rotates which may be used:

LSL #n Logical shift left immediate

`n` is the number of bit positions by which the value is shifted. It has the value 0..31. An **`LSL`** by one bit may be pictured as below:



After n shifts, n zero bits have been shifted in on the right and the carry is set to bit 32-n of the original word.

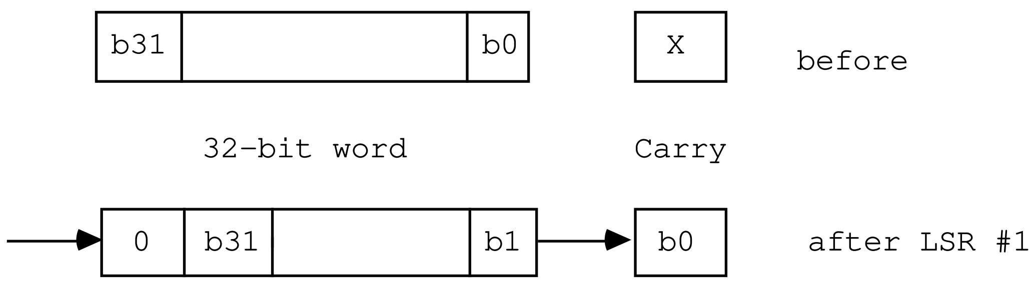
Note that if there is no shift specified after the **<rhs>** register value, **LSL #0** is used, which has no effect at all.

ASL #n Arithmetic shift left immediate

This is a synonym for **LSL #n** and has an identical effect.

LSR #n Logical shift right immediate

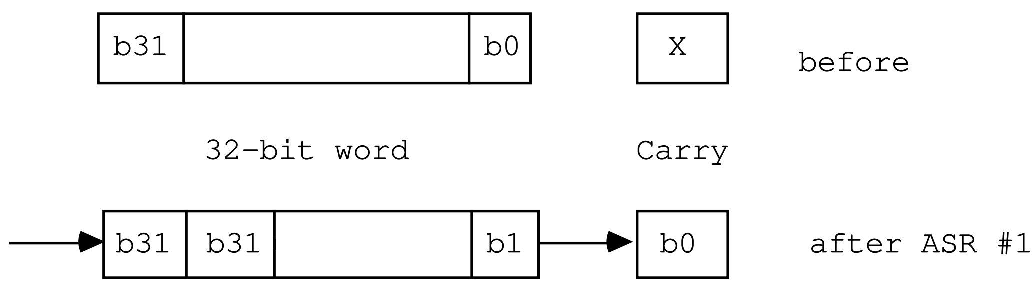
n is the number of bit positions by which the value is shifted. It has the value 1..32. An **LSR** by one bit is shown below:



After n of these, n zero bits have been shifted in on the left, and the carry flag is set to bit n-1 of the original word.

ASR #n Arithmetic shift right immediate

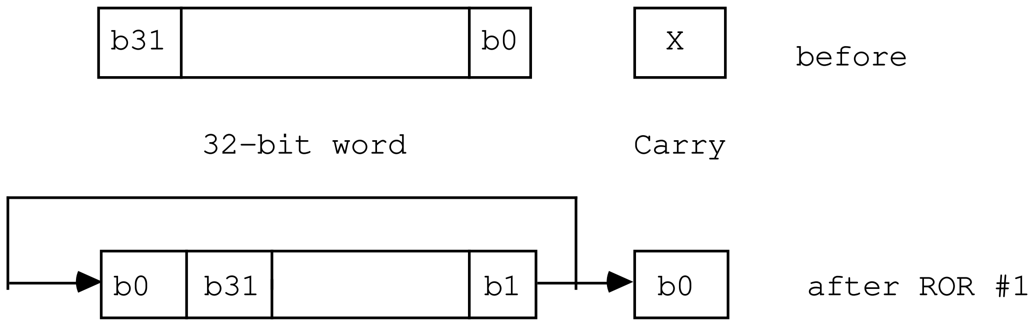
n is the number of bit positions by which the value is shifted. It has the value 1..32. An **ASR** by one bit is shown below:



If ' sign' is the original value of bit 31 then after n shifts, n 'sign' bits have been shifted in on the left, and the carry flag is set to bit n-1 of the original word.

ROR #n Rotate right immediate

n is the number of bit positions to rotate in the range 1..31. A rotate right



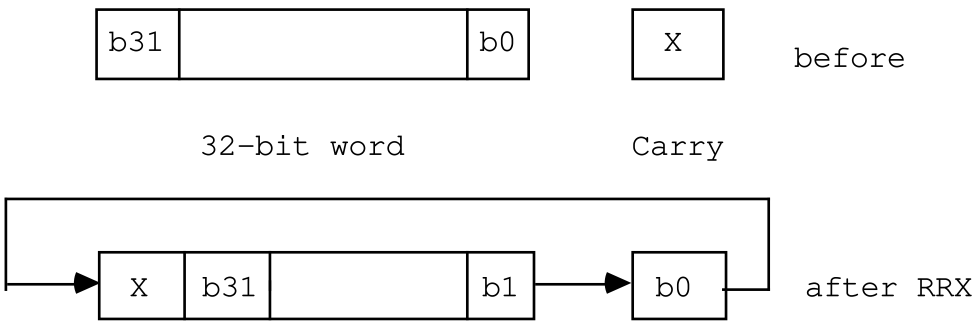
by one bit is shown below:

After n of these rotates, the old bit n is in the bit 0 position; the old bit (n-1) is in bit 31 and in the carry flag.

Note that a rotate left by n positions is the same as a rotate right by (32-n). Also note that there is no rotate right by 32 bits. The instruction code which would do this has been reserved for rotate right with extend (see below).

RRX Rotate right one bit with extend

This special case of rotate right has a slightly different effect from the usual



rotates. There is no count; it always rotates by one bit only. The pictorial representation of **RRX** is:
The old bit 0 is shifted into the carry. The old content of the carry is shifted into bit 31.

Note that there is no equivalent **RLX** rotate. However, the same effect may be obtained using the instruction:

After this, the carry will contain the old bit 31 of R0, and bit 0 of R0 will contain the old carry flag.

LSL rn Logical shift left by a register.
ASL rn Arithmetic shift left by a register.
LSR rn Logical shift right by a register.
ASR rn Arithmetic shift right by a register.
ROR rn Rotate right by a register

This group acts like the immediate shift group, but takes the count from the contents of a specified register instead of an immediate value. Only the least significant byte of the register is used, so the shift count is in the range 0..255. Of course, only counts in the range 0..32 are useful anyway.

We now have the complete syntax of group one instructions:

```
ADD{cond}{S}  <dest>,<lhs>,<rhs>
```

where **<dest>** and **<lhs>** are registers, and **<rhs>** is:

```
#<value>   or
<reg>  {,<shift>}
```

where **<value>** is a shifted immediate number as explained above, **<reg>** is a register and **<shift>** is:

```
<shift type> #<shift count> or
<shift type> <reg> or
RRX
```

where **<shift type>** is **LSL**, **ASL**, **LSR**, **ASR**, **ROR** and **<shift count>** is in the range 0..32, depending on the shift type.

Here is an example of the **ADD** instruction using shifted operands:

```
ADD    R0, R0, R0, LSL #1   ;R0 = R0+2*R0 = 3*R0
```

Instruction descriptions

The foregoing applies to all group one instructions. We now list and explain the individual operations in the group. They may be divided into two sub-groups: logical and arithmetic. These differ in the way in which the result flags are affected (assuming the **S** is present in the mnemonic). Arithmetic instructions alter N, Z, V and C according to the result of the addition, subtraction etc.

Logical instructions affect N and Z according to the result of the operation, and C according to any shifts or rotates that occurred in the **<rhs>**. For example, the instruction:

```
ANDS   R0,R1,R2, LSR #1
```

will set C from bit 0 of R2. Immediate operands will generally leave the carry unaffected if the position value is 0 (i.e. an operand in the range 0..255). For other immediate values, the state of C is hard to predict after the instruction, and it is probably best not to assume anything about the state of the carry after a logical operation which uses the **S** option and has an immediate operand.

To summarise the state of the result flags after any logical operation, if the **S** option was not given, then there is no change to the flags. Otherwise:

If result is negative (bit 31=1), N is set, otherwise it is cleared.
If result is zero, Z is set, otherwise it is cleared.
If **<rhs>** involved a shift or rotate, C is set from this, otherwise it is unaffected by the operation.

V is unaffected

AND Logical AND

The **AND** instruction produces the bit-wise AND of its operands. The AND of two bits is 1 only if both of the bits are 1, as summarised below:

<lhs>	<rhs>	<lhs> AND <rhs>
0	0	0
0	1	0
1	0	0
1	1	1

In the ARM **AND** instruction, this operation is applied to all 32 bits in the operands. That is, bit 0 of the **<lhs>** is **AND**ed with bit 0 of the **<rhs>** and stored in bit 0 of the **<dest>**, and so on.

Examples:

```
ANDS   R0, R0, R5    ;Mask wanted bits using R5
AND    R0, R0, #&DF  ;Convert character to upper case
```

BIC Clear specified bits

The **BIC** instruction produces the bit-wise **AND** of **<lhs>** and **NOT <rhs>**. This has the effect of clearing the **<lhs>** bit if the **<rhs>** bit is set, and leaving the **<lhs>** bit unaltered otherwise.

```
<lhs>  <rhs>    NOT <rhs>   <lhs> AND NOT <rhs>
  0      0         1              0
  0      1         0              0
  1      0         1              1
  1      1         0              0
```

In the ARM **BIC** instruction, this operation is applied to all bits in the operands. That is, bit 0 of the **<lhs>** is **AND**ed with **NOT** bit 0 of the **<rhs>** and stored in bit 0 of the **<dest>**, and so on.

Examples:

```
BICS    R0,R0,R5   ;Zero unwanted bits using R5
BIC     R0,R0,#&20 ;Convert to caps by clearing bit 5
```

TST Test bits

The **TST** instruction performs the AND operation on its **<lhs>** and **<rhs>** operands. The result is not stored anywhere, but the result flags are set according to the result. As there are only two operands, the format of **TST** is:

```
TST     <lhs>,<rhs>
```

Also, as the only purpose of executing a **TST** is to set the flags according to the result, the assembler assumes the **S** is present whether you specify it or not, i.e. **TST** always affects the flags.

See the section 'Using R15 in group one instructions' below.

Examples:

```
TST     R0,R5      ;Test bits using r5, setting flags
TST     R0,#&20    ;Test case of character in R0
```

ORR Logical OR

The **ORR** instruction produces the bit-wise OR of its operands. The OR of two bits is 1 if either or both of the bits is 1, as summarised below:

```
<lhs> <rhs>  <lhs> OR <rhs>
  0     0          0
  0     1          1
  1     0          1
  1     1          1
```

In the ARM **ORR** instruction, this operation is applied to all bits in the operands. That is, bit 0 of the **<lhs>** is ORed with bit 0 of the **<rhs>** and stored in bit 0 of the **<dest>**, and so on. This instruction is often used to set specific bits in a register without affecting the others. It can be regarded as a

Examples:

```
ORRS   R0,R0,R5   ;Set desired bits using R5
ORR    R0,R0,&80000000 ;Set top bit of R0
```

EOR Logical exclusive-OR

The **EOR** instruction produces the bit-wise exclusive-OR of its operands. The EOR of two bits is 1 only if they are different, as summarised below:

```
<lhs> <rhs> <lhs> EOR <rhs>
  0     0         0
  0     1         1
  1     0         1
  1     1         0
```

In the ARM **EOR** instruction, this operation is applied to all bits in the operands. That is, bit 0 of the **<lhs>** is EORed with bit 0 of the **<rhs>** and stored in bit 0 of the **<dest>**, and so on. The **EOR** instruction is often used to invert the state of certain bits of a register without affecting the rest.

Examples:

```
EORS   R0,R0,R5   ;Invert bits using R5, setting flags
EOR    R0,R0,#1   ;'Toggle' state of bit 0
```

TEQ Test equivalence

The **TEQ** instruction performs the EOR operation on its **<lhs>** and **<rhs>** operands. The result is not stored anywhere, but the result flags are set according to the result. As there are only two operands, the format of **TEQ** is:

```
TEQ    <lhs>,<rhs>
```

Also, as the only purpose of executing a **TEQ** is to set the flags according to the result, the assembler assumes the **S** is present whether you specify or not, i.e. **TEQ** always affects the flags.

One use of **TEQ** is to test if two operands are equal without affecting the state of the C flag, as the **CMP** instruction does (see below). After such an operation, Z=1 if the operands are equal, or 0 if not. The second example below illustrates this.

See the section 'Using R15 in group one instructions' below.

Examples:

```
TEQ R0,R5      ;Test bits using R5, setting flags
TEQ R0,#0      ;See if R0 = 0
```

MOV Move value

The **MOV** instruction transfers its **<rhs>** operand to the register specified by **<dest>**. There is no **<lhs>** specified in the instruction, so the assembler format is:

```
MOV     <dest>,<rhs>
```

Examples:

```
MOV     R0, R0,LSL #2 ;Multiply R0 by four.
MOVS    R0, R1        ;Put R1 in R0, setting flags
MOV     R1, #&80      ;Set R1 to 128
```

MVN Move value

The **MOV** instruction transfers the logical **NOT** of its **<rhs>** operand to the register specified by **<dest>**. Inverting the **<rhs>** before transfer enables negative immediate operands to be loaded. There is no **<lhs>** specified in the instruction, so the assembler format is:

```
MVN     <dest>,<rhs>
```

In general, -n = NOT (n-1). This means that to load a negative number, you subtract one from its positive value and use that in the **MVN**. For example, to load the number -128 you would do a **MVN** of 127.

Examples:

```
MVNS    R0, R0      ;Invert all bits of R0, setting flags
MVN     R0, #0      ;Set R0 to &FFFFFFFF, i.e. -1
```

Descriptions of the arithmetic instructions follow. Here is a summary of how the result flags are affected by these instructions. If the **S** option is not specified, no change takes place. Otherwise:

If result is negative (bit 31=1), N is set, otherwise it is cleared.
If result is zero, Z is set, otherwise it is cleared.
If the operation generated a carry, C is set, otherwise it is cleared.
If the operation generated an overflow, V is set, otherwise it is cleared.

Note that if C=1 after an **ADDS**, the implication is that a carry was generated, i.e. the addition overflowed into bit 32. This does not mean that the result is wrong. For example, adding -1 to 10 will produce the following:–

```
    00000000000000000000000000001010
+   11111111111111111111111111111111
```

A carry is generated, but the result we obtain is the desired one, i.e. 9. The overflow flag (see below) is used for detecting errors in the arithmetic. In this case, V=0 after the operation, implying no overflow.

The state of the carry after a **SUBS** is the opposite to that after an **ADDS**. If C=1, no borrow was generated during the subtract. If C=0, a borrow was required. Note that by this definition, borrow = **NOT** carry, so the summary of the flags' states above is correct.

The precise definition of the overflow state is the exclusive-**OR** of the carries from bits 30 and 31 during the add or subtract operation. What this means in real terms is that if the V flag is set, the result is too large to fit in a single 32-bit word. In this case, the sign of the result will be wrong, which is why the signed condition codes take the state of the V flag into account.

ADD Addition

This instruction adds the **<lhs>** operand to the **<rhs>** operand, storing the result in **<dest>**. The addition is a thirty-two bit signed operation, though if the operands are treated as unsigned numbers, then the result can be too.

Examples:

```
    ADD     R0,R0,#1          ;Increment R0
    ADD     R0,R0,R0,LSL#2    ;Multiple R0 by 5
    ADDS    R2,R2,R7          ;Add result; check for overflow
```

ADC Addition with carry

The add with carry operation is very similar to **ADD**, with the difference that the carry flag is added too. Whereas the function of **ADD** can be written:

```
    <dest> = <lhs> + <rhs>
```

we must add an extra part for **ADC**:

```
    <dest> = <lhs> + <rhs> + <carry>
```

where **<carry>** is 0 if C is cleared and 1 if it is set. The purpose of **ADC** is to allow multi-word addition to take place, as the example illustrates.

Example:

```
;Add the 64-bit number in R2,R3 to that in R0,R1
    ADDS    R0,R0,R2   ;Add the lower words, getting carry
    ADC     R1,R1,R3   ;Add upper words, using carry
```

SUB Subtract

This instruction subtracts the **<rhs>** operand from the **<lhs>** operand, storing the result in **<dest>**. The subtraction is a thirty-two bit signed operation.

Examples:

```
    SUB     R0,R0,#1        ;Decrement R0
    SUB     R0,R0,R0,ASR#2 ;Multiply R0 by 3/4 (R0=R0-R0/4)
```

SBC Subtract with carry

This has the same relationship to **SUB** as **ADC** has to **ADD**. The operation may be written:

```
    <dest> = <lhs> - <rhs> - NOT <carry>
```

Notice that the carry is inverted because the C flag is cleared by a subtract that needed a borrow and set by one that didn't. As long as multi-word subtracts are performed using **SUBS** for the lowest word and **SBCS** for subsequent ones, the way in which the carry is set shouldn't concern the programmer.

Example:

```
;Subtract the 64-bit number in R2,R3 from that in R0,R1
    SUBS    R0,R0,R2   ;Sub the lower words, getting borrow
    SBC     R1,R1,R3   ;Sub upper words, using borrow
```

RSB Reverse subtract

This instruction performs a subtract without carry, but reverses the order in which its operands are subtracted. The instruction:

```
    RSB     <dest>,<lhs>,<rhs>
```

performs the operation:

```
    <dest> = <rhs> - <lhs>
```

The instruction is provided so that the full power of the **<rhs>** operand (register, immediate, shifted register) may be used on either side of a subtraction. For example, in order to obtain the result 1-R1 in the register R0, you would have to use:

```
    MVN     R0,R1      ;get NOT (R1) = -R1-1
    ADD     R0,R0,#2   ;get -R1-1+2 = 1-R1
```

However, using **RSB** this could be written:

```
    RSB    R0, R1, #1 ;R0 = 1 - R0
```

In more complex examples, extra registers might be needed to hold temporary results if subtraction could only operate in one direction.

Example:

```
;Multiply R0 by 7
    RSB    R0, R0, R0, ASL #3   ;Get 8*R0-R0 = 7*R0
```

RSC   Reverse subtract with carry

This is the 'with carry' version of **RSB**. Its operation is:

```
    <dest> = <rhs> - <lhs> - NOT <carry>
```

It is used to perform multiple-word reversed subtracts.

Example

```
    ;Obtain &100000000-R0,R1 in R0,R1
    RSBS   R0,R0,#0   ;Sub the lower words, getting borrow
    RSC    R1,R1,#1   ;Sub the upper words
```

CMP Compare

The **CMP** instruction is used to compare two numbers. It does this by subtracting one from the other, and setting the status flags according to the result. Like **TEQ** and **TST**, **CMP** doesn't actually store the result anywhere. Its format is:

```
    CMP    <lhs>, <rhs>
```

Also like **TST** and **TEQ** it doesn't require the **S** option to be set, as **CMP** without setting the flags wouldn't do anything at all.

After a **CMP**, the various conditions can be used to execute instructions according to the relationship between the integers. Note that the two operands being compared may be regarded as either signed (two's complement) or unsigned quantities, depending on which of the conditions is used.

See the section 'Using R15 in group one instructions' below.

Examples:

```
    CMP    R0,#&100       ;Check R0 takes a single byte

    CMP    R0,R1          ;Get greater of R1, R0 in R0
    MOVLT  R0,R1
```

CMN Compare negative

The **CMN** instruction compares two numbers, but negates the right hand side before performing the comparison. The 'subtraction' performed is therefore **<lhs>--<rhs>**, or simply **<lhs>+<rhs>**. The main use of **CMN** is in comparing a register with a small negative immediate number, which would not otherwise be possible without loading the number into a register (using **MVN**). For example, a comparison with -1 would require

```
MVN     R1,#0       ;Get -1 in R1
CMP     R0,R1       ;Do the comparison
```

which uses two instructions and an auxiliary register, compared to this:

```
CMN     R0,#1       ;Compare R0 with -1
```

Note that whereas **MVN** is 'move *NOT*', **CMN** is 'compare *negative*', so there is a slight difference in the way **<rhs>** is altered before being used in the operation.

See the section 'Using R15 in group one instructions' below.

Example

```
CMN     R0,#256         ;Make sure R0 >= -256
MVNLT   R0,#255
```

Using R15 in group one instructions

As we know, R15 is a general-purpose register, and as such may be cited in any situation where a register is required as an operand. However, it is also used for storing both the program counter and the status register. Because of this, there are some special rules which have to be obeyed when you use R15. These are described in this section.

The first rule concerns how much of R15 is 'visible' when it is used as one of the source operands, i.e. in an **<rhs>** or **<lhs >** position. Simply stated, it is:

if **<lhs>** is R15 then only bits 2..25 (the PC) are visible
if **<rhs>** is R15 then all bits 0..31 are visible

So, in the instruction:

```
ADD     R0,R15,#128
```

the result of adding 128 to the PC is stored in R0. The eight status bits of R15 are seen as zeros by the ALU, and so they don't figure in the addition.

the addition is eight greater than the address of the **ADD** itself, because of pipelining (this is described in more detail below).

In the instruction

```
MOV     R0,R15,ROR  #26
```

all 32 bits of R15 are used, as this time the register is being used in an **<rhs>** position. The effect of this instruction is to obtain the eight status bits in the least significant byte of R0.

The second rule concerns using R15 as the destination operand. In the instruction descriptions above we stated that (if **S** is present) the status bits N, Z, C and C are determined by the outcome of the instruction. For example, a result of zero would cause the Z bit to be set.

In the cases when R15 itself is the destination register, this behaviour changes. If **S** is not given in the instruction, only the PC bits of R15 are affected, as usual. So the instruction

```
ADD     R15,R15,R0
```

adds some displacement which is stored in R0 to the program counter, leaving the status unaffected.

If **S** is present in the instruction, and the instruction isn't one of **TST**, **TEQ**, **CMP**, or **CMN** (which are explained below), the status bits which are allowed to be altered in the current processor mode are overwritten directly by the result. (As opposed to the status of the result.) An example should make this clear. To explicitly set the carry flag (bit 29 of R15) we might use:

```
ORRS    R15,R15,#&20000000
```

Now, as the second R15 is in a **<lhs>** position, the status bits are presented as zeros to the ALU (because of the first rule described above). Thus the value written into the status register is (in binary) 001000...00. In fact, in user modes, only the top four bits may be affected (i.e. the interrupt masks and processor mode bits can't be altered in user mode).

The example above has the (usually) unfortunate side effect of skipping the two instructions which follow the **ORR**. This is, as usual, due to pipelining. The R15 value which is transferred into the ALU holds the address of the third instruction after the current one, thus the intervening two are never executed. (They are pre-fetched into pipeline, but whenever the PC is altered, the ARM has to disregard the current pre-fetched instructions.)

To overcome this there is a special way of writing to the status bits, and only the status bits, of R15. It involves using the four instructions which don't usually have a destination register: **TST**, **TEQ**,, **CMP**, and **CMN**. As we know, these usually affect the flags from the result of the operation because they have an implicit **S** option built-in. Also, usually the assembler makes the **<dest>** part of the instruction code R0 (it still has this field in the instruction, even if it isn't given in the assembly language).

Now, if the **<dest>** field of one of these instructions is made R15 instead, a useful thing happens. The status bits are updated from the result of the operation (the **AND**, **EOR**, **SUB** or**ADD** as appropriate), but the PC part remains unaltered.

The problem is how to make the assembler use R15 in the **<dest>** field instead of R0. This is done by adding a **P** to the instruction. To give a concrete example, we will show how the carry flag can be set *without* skipping over the next two instructions:

```
TEQP   R15,#&20000000
```

This works as follows. The R15 is a **<lhs>** so the status bits appear as zeros to the ALU. Thus **EOR**ing the mask for the carry flag with this results in that bit being set when the result is transferred to R15. The rest of the status register (or at least the bits which are alterable) will be cleared.

Setting and clearing only selected status bits while leaving the rest alone takes a bit more effort. First the current value of the bits must be obtained. Then the appropriate masking is performed and the result stored in R15. For example, to clear the overflow flag (bit 28) while preserving the rest, something like this is needed:

```
MOV    tmp,R15        ;Get the status
BIC    tmp,tmp,#1<<28;Clear bit 28
TEQP   R15,tmp        ;Store the new status
```

Finally, we have to say something about performing arithmetic on R15 in order to alter the execution path of the program. As we will see later in this chapter, there is a special **B** (for Branch) instruction, which causes the PC to take on a new value, This causes a jump to another part of the program, similar to the BASIC **GOTO** statement. However, by changing the value of R15 using group one instructions such as **ADD**, we can achieve more versatile control, for example emulating the BASIC **ON..GOTO** statement.

The important thing to bear in mind when dealing with the PC has already been mentioned once or twice: the effect of pipelining. The value obtained when R15 is used as an operand is 8 bytes, or 2 words, greater than the address of the current instruction. Thus if the instruction

```
MOV     R0,R15
```

was located at address &8000, then the value loaded into R15 would be &8008. Chapters Five and Six contain several examples of the use of R15 where pipelining is taken into account.

Group One A

There is a small class of instructions which is similar in form to the group one instructions, but doesn't belong in that group. These are the multiply instructions, whose form bears a similarity to the simplest form of group one instructions.

Two distinct operations make up this group, multiply and multiply with accumulate. The formats are:

```
MUL{cond}{S}  <dest,<lhs>,<rhs>
MLA{cond}{S}  <dest>,<lhs>,<rhs>,<add>
```

All operands are simple registers; there are no immediate operands or shifted registers. **MUL** multiplies **<lhs>** by **<rhs>** and stores the result in **<dest>**. MLA does the same, but adds register **<add>** before storing the result.

You must obey certain restrictions on the operands when using these instructions. The registers used for **<rhs>** and **<dest>** must be different. Additionally, you should not use R15 as a destination, as this would produce a meaningless result. In fact, R15 is protected from modification by the multiply instructions. There are no other restrictions.

If the **S** option is specified, the flags are affected as follows. The N and Z flags are set to reflect the status of the result register, the same as the rest of the group one instructions. The overflow flag is unaltered, and the carry flag is undefined.

You can regard the operands of the multiply instructions as unsigned or as two's complement signed numbers. In both cases, the correct results will be obtained.

Example:

```
MUL     R0,R1,R2
```

Summary of group one

The group one instructions have the following form:

```
<op1>{cond}{S}{P}     <dest>,<lhs>,<rhs>
<op2>{cond}{S}{P}     <dest>,<rhs>
<op3>{cond}{S}{P}     <lhs>,<rhs>
```

where **<op1>** is one of **ADD**, **ADC**, **AND**, **BIC**, **EOR**, **ORR**, **RSB**, **RSC**, **SBC**, **SUB**, **<op2>** is one of **MOV**, **MVN**, and **<op3>** is one of **TEQ**, **TST**,**CMN**, **CMP**.

The following **<op3>**s have no **<dest>** field: **CMN**, **CMP**, **TEQ**, **TST**. They allow the use of the **{P}** option to set the **<dest>** field in the machine code to R15 instead of the default R0.

The following **<op2>**s have no **<lhs>** field: **MOV**, **MVN**.

<dest> and **<lhs>** are registers.

<rhs> is **#<expression>** where **<expression>** is a 12-bit shifted immediate operand
or:
<register>
or:
<register>, <shift type> <count>

where **<shift type>** is **LSL**, **ASL**, **ASR**, **LSR**, **ROR** and where **<count>** is **#<expression>** or **<register>** where **<expression>** is five-bit unsigned value,
or:
<register>, RRX

3.3 Group two - load and store

We turn now to the first set of instructions concerned with getting data in and out of the processor. There are only two basic instructions in this category. **LDR** loads a register from a specified location and **STR** saves a register.

As with group one, there are several options which make the instructions very versatile. As the main difference between **LDR** and **STR** is the direction in which the data is transferred (i.e. to or from the processor), we will explain only one of the instructions in detail - **STR**. Notes about **LDR** follow this description.

STR Store a word or byte

Addressing modes

When storing data into memory, you have to be able to specify the desired location. There are two main ways of giving the address, called addressing modes. These are known as pre-indexed and post-indexed addressing.

Pre-indexed addressing

Pre-indexed addressing is specified as below:

```
STR{cond}  <srce>,[<base>{,<offset>}]
```

`<srce>` is the register from which the data is to be transferred to memory. **`<base>`** is a register containing the base address of memory location required. **`<offset>`** is an optional number which is added to the address before the data is stored. So the address actually used to store the **`<srce>`** data is **`<base>+<offset>`**

Offset formats

The **`<offset>`** is specified in one of two ways. It may be an immediate number in the range 0 to 4095, or a (possibly) shifted register. For example:

```
STR     R0,[R1,#20]
```

will store R0 at byte address R1+20. The offset may be specified as negative, in which case it is subtracted from the base address. An example is:

```
STR     R0,[R1,#-200]
```

which stores R0 at R1-200.

(Note for alert readers. The immediate offset is stored as a 12-bit magnitude plus one bit 'direction', not as a 13-bit two's complement number. Therefore the offset range really is -4095 to +4095, not -4096 to +4095, as you might expect.)

If the offset is specified as a register, it has the same syntax as the **`<rhs>`** of group one instructions. That is, it could be a simple register contents, or the contents of a register shifted or rotated by an immediate number.

Note: the offset register can only have an immediate shift applied to it. In this respect, the offset differs from the **`<rhs>`** of group one instructions. The latter can also have a shift which is stored in a register.

This example stores R0 in the address given by R1+R2*4:

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

Again, the offset may be negative as this example illustrates:

```
STR     R0,[R1,-R2,LSL#3]
```

Write-back

Quite frequently, once the address has been calculated, it is convenient to update the base register from it. This enables it to be used in the next instruction. This is useful when stepping through memory at a fixed offset. By adding a **!** to the end of the instruction, you can force the base register to be updated from the **<base>+<offset>** calculation. An example is:

```
STR     R0,[R1,#-16]!
```

This will store R0 at address R1-16, and then perform an automatic:

```
SUB     R1,R1,#16
```

which, because of the way in which the ARM works, does not require any extra time.

Byte and word operations

All of the examples of **STR** we have seen so far have assumed that the final address is on a word boundary, i.e. is a multiple of 4. This is a constraint that you must obey when using the **STR** instruction. However, it is possible to store single bytes which may be located at any address. The byte form of **STR** is obtained by adding **B** at the end. For example:

```
STRB    R0,[R1,#1]!
```

will store the least significant byte of R0 at address R1+1 and store this address in R1 (as **!** is used).

Post-indexed addressing

The second addressing mode that **STR** and **LDR** use is called post-indexed addressing. In this mode, the **<offset>** isn't added to the **<base>** until after the instruction has executed. The general format of a post-indexed **STR** is:

```
STR{cond}  <srce>,[<base>],<offset>
```

The **<base>** and **<offset>** operands have the same format as pre-indexed addressing. Note though that the **<offset>** is always present, and write-back always occurs (so no **!** is needed). Thus the instruction:

will save the least significant byte of R0 at address R1. Then R1 is set to R1+R2. The **<offset>** is used only to update the **<base>** register at the end of the instruction. An example with an immediate **<offset>** is:

```
STR    R2,[R4],#-16
```

which stores R2 at the address held in R4, then decrements R4 by 4 words.

LDR Load a word or byte

The **LDR** instruction is similar in most respects to **STR**. The main difference is, of course, that the register operand is loaded from the given address, instead of saved to it.

The addressing modes are identical, and the **B** option to load a single byte (padded with zeros) into the least significant byte of the register is provided.

When an attempt is made to **LDR** a word from a non-word boundary, special corrective action is taken. If the load address is **addr**, then the word at **addr AND &3FFFFFFC** is loaded. That is, the two least significant bits of the address are set to zero, and the contents of that word address are accessed. Then, the register is rotated right by **(addr MOD 4)*8** bits. To see why this is done, consider the following example. Suppose the contents of address &1000 is &76543210. The table below shows the contents of R0 after a word load from various addresses:

Address	**R0**
&1000	&76543210
&1001	&10765432
&1002	&32107654
&1003	&54321076

After the rotation, at least the least significant byte of R0 contains the correct value. When **addr** is &1000, it is a word-boundary load and the complete word is as expected. When **addr** is &1001, the first three bytes are as expected, but the last byte is the contents of &1000, rather than the desired &1004. This (at first sight) odd behaviour facilitates writing code to perform word loads on arbitrary byte boundaries. It also makes the implementation of extra hardware to perform correct non-word-aligned loads easier.

Note that **LDR** and **STR** will never affect the status bits, even if **<dest>** or **<base>** is R15. Also, if R15 is used as the base register, pipelining means that the value used will be eight bytes higher than the address of the instruction. This is taken into account automatically when PC-relative addressing is used.

PC relative addressing

The assembler will accept a special form of pre-indexed address in the **LDR** instruction, which is simply:

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

where **<expression>** yields an address. In this case, the instruction generated will use R15 (i.e. the program counter) as the base register, and calculate the immediate offset automatically. If the address given is not in the correct range (-4095 to +4095) from the instruction, an error is given.

An example of this form of instruction is:

```
LDR     R0,default
```

(We assume that **default** is a label in the program. Labels are described more fully in the next chapter, but for now suffice is to say that they are set to the address of the point in the program where they are defined.)

As the assembler knows the value of the PC when the program is executed, it can calculate the immediate offset required to access the location **default**. This must be within the range -4095 to +4095 of course. This form of addressing is used frequently to access constants embedded in the program.

Summary of LDR and STR

Below is a summary of the syntax for the register load and save instructions.

```
<op>{cond}{B}  <dest>,[<base>{,#<imm>}]{!}

<op>{cond}{B}  <dest>,[<base>,{+|-}<off>{,<shift>}]{!}

<op>{cond}{B}  <dest>,<expression>

<op>{cond}{B}  <dest>,[<base>],#<imm>

<op>{cond}{B}  <dest>,[<base>],{+|-}<off>{,<shift>}
```

Although we haven't used any explicit examples, it is implicit given the regularity of the ARM instruction set that any **LDR** or **STR** instruction may be made conditional by adding the two letter code. Any **B** option follows this.

<op> means **LDR** or **STR**. **<imm>** means an immediate value between -4095 and +4095. **{+|-}** means an optional + or - sign may be present. **<off>** is the offset register number. **<base>** is a base register, and **<shift>** refers to the standard immediate (but *not* register shift) described in the section

Note that the case of:

```
STR     R0,label
```

is covered. Although the assembler will accept this and generate the appropriate PC-relative instruction, its use implies that the program is writing over or near its code. Generally this is not advisable because (a) it may lead inadvertently to over-writing the program with dire consequences, and (b) if the program is to be placed in ROM, it will cease to function, as ROMs are generally read-only devices, on the whole.

3.4 Group three - multiple load and store

The previous instruction group was eminently suitable for transferring single data items in and out of the ARM processor. However, circumstances often arise where several registers need to be saved (or loaded) at once. For example, a program might need to save the contents of R1 to R12 while these registers are used for some calculation, then load them back when the result has been obtained. The sequence:

```
STR R1,[R0],#4
STR R2,[R0],#4
STR R3,[R0],#4
STR R4,[R0],#4
STR R5,[R0],#4
STR R6,[R0],#4
STR R7,[R0],#4
STR R8,[R0],#4
STR R9,[R0],#4
STR R10,[R0],#4
STR R11,[R0],#4
STR R12,[R0],#4
```

to save them is inefficient in terms of both space and time. The **LDM** (load multiple) and **STM** (store multiple) instructions are designed to perform tasks such as the one above in an efficient manner.

As with **LDR** and **STR** we will describe one variant in detail, followed by notes on the differences in the other. First though, a word about stacks.

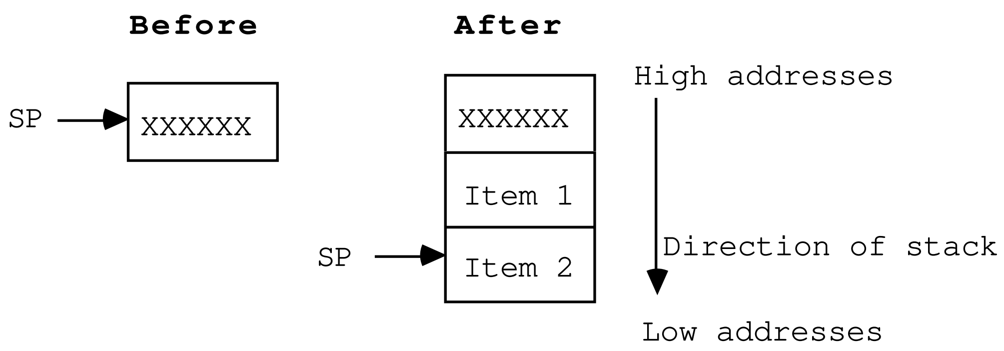
About stacks

LDM and **STM** are frequently used to store registers on, and retrieve them from, a stack. A stack is an area of memory which 'grows' in one direction as items are added to it, and 'shrinks' as they are taken off. Items are always removed from the stack in the reverse order to which they are added.

The term 'stack' comes from an analogy with stacks of plates that you see in self-service restaurants. Plates are added to the top, then removed in the opposite order. Another name for a stack is a last-in, first-out structure or 'LIFO'.

Computing stacks have a value associated with them called the stack pointer, or SP. The SP holds the address of the next item to be added to (pushed onto) or removed (pulled) from the stack. In an ARM program, the SP will almost invariably be stored in one of the general-purpose registers. Usually, a high-numbered register, e.g. R12 or R13 is used. The Acorn ARM Calling Standard, for example, specifies R12, whereas BASIC uses R13.

Here is a pictorial representation of two items being pushed onto a stack.



Before the items are pushed, SP points to (holds the address of) the previous item that was pushed. After two new words have been pushed, the stack pointer points to the second of these, and the first word pushed lies 'underneath' it.

Stacks on the ARM have two attributes which must be decided on before any **STM**/**LDM** instructions are used. The attributes must be used consistently in any further operation on the stack.

The first attribute is whether the stack is 'full' or 'empty'. A full stack is one in which the SP points to the last item pushed (like the example above). An empty stack is where the stack pointer holds the address of the empty slot where the *next* item will be pushed.

Secondly, a stack is said to be ascending or descending. An ascending stack is one where the SP is incremented when items are pushed and decremented when items are pulled. A descending stack grows and shrinks in the opposite direction. Given the direction of growth in the example above,the stack can be seen to be descending.

The above-mentioned Acorn Calling Standard (which defines the way in which separate programs may communicate) specifies a full, descending stack, and BASIC also uses one.

STM Store multiple registers

In the context of what has just been said, **STM** is the 'push items onto stack' instruction. Although it has other uses, most **STM**s are stack oriented.

The general form of the STM instruction is:

```
STM<type>  <base>{!},  <registers>
```

We have omitted the **{cond}** for clarity, but as usual a condition code may be included immediately after the mnemonic. The **<type>** consists of two letters which determine the **F**ull/**E**mpty, **A**scending/**D**escending mode of the stack. The first letter is **F** or **E**; the second is **A** or **D**.

The **<base>** register is the stack pointer. As with **LDR**/**STR**, the **!** option will cause write-back if present. **<registers>** is a list of the registers which we want to push. It is specified as a list of register numbers separated by commas, enclosed in braces (curly brackets). You can also specify a list of registers using a **–** sign, e.g. **R0–R4** is shorthand for **R0,R1,R2,R3,R4**.

Our first example of an **STM** instruction is:

```
STMED  R13!,{R1,R2,R5}
```

This will save the three registers R1, R2 and R5 using R13 as the SP. As write-back is specified, R13 will be updated by subtracting 3*4 (12) from its original value (subtracting as we are using a descending stack). Because we have specified the **E** option, the stack is empty, i.e. after the **STM**, R13 will hold the address of the next word to be pushed. The address of the last word pushed is R13+4.

When using **STM** and **LDM** for stack operations, you will almost invariably use the **!** option, as the stack pointer has to be updated. Exceptions are when you want to obtain a copy of an item without pulling it, and when you need to store a result in a 'slot' which has been created for it on the stack. Both these techniques are used in the 'find substring' example of Chapter Six.

More on FD etc.

The ascending/descending mode of a stack is analogous to positive and negative offsets of the previous section. During an **STM**, if the stack is descending, a negative offset of 4 is used. If the stack is ascending, a positive

Similarly, the difference between what the ARM does for full and empty stacks during an **STM** is analogous to the pre- and post-indexed addressing modes of the previous section. Consider a descending stack. An empty stack's pointer is post-decremented when an item is pushed. That is, the register to be pushed is stored at the current location, then the stack pointer is decremented (by a fixed offset of 4), ready for the next one. A full stack is pre-decremented on a push: first SP is decremented by 4, then the register is stored there. This is repeated for each register in the list.

Direction of storage

Below is an **STM** which stores all of the registers at the location pointed to by R13, but without affecting R13 (no write-back).

```
STMFD  R13,{R0-R15}
```

Although the stack is an **FD** one, because there is no write-back, R13 is not actually decremented. Thus after the operation, R13 points to some previous item, and below that in memory are the sixteen pushed words.

Now, as R0 appears first in the list, you might expect that to be the first register to be pushed, and to be stored at address R13-4. Well, you would be wrong. Firstly, the order in which the register list appears has no bearing at all on the order in which things are done. All the assembler is looking for is a mention of a register's name: it can appear anywhere within the braces. Secondly, the registers are always stored in the same order in memory, whether the stack is ascending or descending, full or empty.

When you push one or more registers, the ARM stores the lowest-numbered one at the lowest address, the next highest-numbered one at the next address and so on. This always occurs. For ascending stacks, the location pointer is updated, the first register stored (or vice versa for empty stacks) and so on for each register. Finally, if write-back is enabled, the stack pointer is updated by the location pointer.

For descending stacks, the final value of the stack pointer is calculated first. Then the registers are stored in increasing memory locations, and finally the stack pointer is updated if necessary.

We go into this detail because if you need to access pushed registers directly from the stack, it is important to know where they are!

Saving the stack pointer and PC

In the previous example, the stack pointer (R13) was one of the registers stored. However, in cases where write-back is enabled, the SP changes value at some time during the instruction, so the 'before' or 'after' version might be saved.

The rule which determines which version of the base register is saved is quite simple. If the stack pointer is the lowest-numbered one in the list, then the value stored is the original, unaltered one. Otherwise, the written-back value of the base register is that stored on the stack. Since we have standardised on R13 for the stack pointer, it is almost always the new value which is stored.

When R15 is included in the register list, all 32-bits are stored, i.e. both the PC and status register parts.

LDM Load multiple registers

LDM perform the stack 'pull' (or pop as it is also known) operation. It is very similar in form to the **STM** instruction:

```
LDM<type>  <base>{!},<registers>{^}
```

As before, the **<type>** gives the direction and type of the stack. When pulling registers, you must use the same type of stack as when they were pushed, otherwise the wrong values will be loaded. **<base>**, **!** and **<registers>** are all as **STM**.

The only extra 'twiddle' provided by **LDM** is the **^** which may appear at the end. If present, this indicates that if R15 is in the register list, then all 32-bits are to be updated from the value pulled from the stack. If there is no **^** in the instruction, then if R15 is pulled, only the PC portion (bits 2-25) are affected. The status bits of R15 will remain unaltered by the operation.

This enables you to decide whether subroutines (which are explained in detail in the next section) preserve the status flags or use them to return results. If you want the routine to preserve the flags, then use **^** to restore them from the stack. Otherwise omit it, and the flags will retain their previous contents.

Loading the stack pointer and PC

If the register being used as the stack pointer is in the **LDM** register list, the register's value after the instruction is always that which was loaded, irrespective of where it appeared in the list, and whether write-back was specified.

If R15 appears in the list, then the **^** option is used to determine whether the

An alternative notation

Not all of the uses for **LDM** and **STM** involve stacks. In these situations, the full/empty, ascending/descending view of the operations may not be the most suitable. The assembler recognises an alternative set of option letters, which in fact mirrors more closely what the ARM instructions really do.

The alternative letter pairs are **I**/**D** for increment/decrement, and **B**/**A** for before/after. Thus the instruction:

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

stores R1 and R2 at the address in R0. For each store, a post-increment (**I**ncrement **A**fter) is performed, and the new value (R0+8) is written back. Similarly:

```
LDMDA  R0!,{R1,R3,R4}
```

loads the three registers specified, decrementing the address after each one. The write-back option means that R0 will be decremented by 12. Remember that registers are always stored lowest at lowest address, so in terms of the original R0, the registers in the list will be loaded from addresses R0-8, R0-4 and R0-0 respectively.

The table below shows the equivalents for two types of notation:

Full/Empty	Decrement/Increment equivalent
STMFD	STMDB
LDMFD	LDMIA
STMFA	STMIB
LDMFA	LDMDA
STMED	STMDA
LDMED	LDMIB
STMEA	STMIA
LDMEA	LDMDB

The increment/decrement type notation shows how push and pull operations for a given type of stack are exact opposites.

Summary of LDM/STM

```
LDM{cond}<type1>  <base>{!},<registers>{^}
LDM{cond}<type2>  <base>{!},<registers>{^}
STM{cond}<type1>  <base>{!},<registers>{^}
STM{cond}<type2>  <base>{!},<registers>{^}
```

where:

cond is defined at the start of this chapter.
<type1> is **F|E A|D**
<type2> is**I|D B|A**
<base> is a register
<registers> is open-brace comma-separated-list close-brace

Notice that **^** may be used in **STM** as well as **LDM**. Its use in **STM** is only relevant to non-user modes, and as such is described in Chapter Seven.

3.5 Group four - branch

In theory, the group one data instructions could be used to make any change to the PC required by a program. For example, an **ADD** to R15 could be used to move the PC on by some amount, causing a jump forward in the program. However, there are limits to what you can achieve conveniently using these general-purpose instructions. The branch group provides the ability to locate any point in the program with a single operation.

The simple branch

Group four consists of just two variants of the branch instruction. There are (refreshingly perhaps) no alternative operand formats or optional extras. The basic instruction is a simple branch, whose mnemonic is just **B**. The format of the instruction is:

```
B{cond}    <expression>
```

The optional condition, when present, makes the mnemonic a more normal-looking three-letter one, e.g. **BNE**, **BCC**. If you prefer three letters, you could always express the unconditional branch as **BAL**, though the assembler would be happy with just **B**.

<expression> is the address within the program to which you wish to transfer control. Usually, it is just a label which is defined elsewhere in the program. For example, a simple counting loop would be implemented thus:

```
      MOV    R0,#0          ;Init the count to zero
.LOOP MOVS   R1,R1,LSR#1    ;Get next 1 bit in Carry
      ADC    R0,#0          ;Add it to count
      BNE    LOOP           ;Loop if more to do
```

The 'program' counts the number of 1 bits in R1 by shifting them a bit at a time into the carry flag and adding the carry to R0. If the **MOV** left a non-zero result in R1, the branch causes the operation to repeat (note that the **ADC** doesn't affect the status as it has no **S** option).

Offsets and pipelining

When the assembler converts a branch instruction into the appropriate binary code, it calculates the offset from the current instruction to the destination address. The offset is encoded in the instruction word as a 24-bit word address, like the PC in R15. This is treated as a signed number, and when a branch is executed the offset is added to the current PC to reach the destination.

Calculation of the offset is a little more involved than you might at first suspect - once again due to pipelining. Suppose the location of a branch instruction is &1230, and the destination label is at address &1288. At first sight it appears the byte offset is &1288-&1230 = &58 bytes or &16 words. This is indeed the difference between the addresses. However, by the time the ARM gets round to executing an instruction, the PC has already moved two instructions further on.

Given the presence of pipelining, you can see that by the time the ARM starts to execute the branch at &1230, the PC contains &1238, i.e. the address of two instructions along. It is this address from which the offset to the destination must be calculated. To complete the example, the assembler adds &8 to &1230 to obtain &1238. It then subtracts this from the destination address of &1288 to obtain an offset of &50 bytes or &14 words, which is the number actually encoded in the instruction. Luckily you don't have to think about this when using the **B** instruction.

Branch with link

The single variant on the branch theme is the option to perform a link operation before the branch is executed. This simply means storing the current value of R15 in R14 before the branch is taken, so that the program has some way of returning there. Branch with link, **BL**, is used to implement subroutines, and replaces the more usual **BSR** (branch to subroutine) and **JSR** (jump to subroutine) instructions found on some computers.

Most processors implement **BSR** by saving the return address on the stack. Since ARM has no dedicated stack, it can't do this. Instead it copies R15 into R14. Then, if the called routine needs to use R14 for something, it can save it on the stack explicitly. The ARM method has the advantage that subroutines which don't need to save R14 can be called and return very quickly. The disadvantage is that all other routines have the overhead of explicitly saving R14.

The address that ARM saves in R14 is that of the instruction immediately following the **BL**. (Given pipelining, it should be the one after that, but the processor automatically adjusts the value saved.) So, to return from a subroutine, the program simply has to move R14 back into R15:

```
MOVS   R15,R14
```

The version shown will restore the original flags too, automatically making the subroutine preserve them. If some result was to be passed back in the status register, a **MOV** without **S** could be used. This would only transfer the PC portion of R14 back to R15, enabling the subroutine to pass status information back in the flags.

BL has the expected format:

```
BL{cond}   <expression>
```

3.6 Group five - software interrupt

The final group is the most simple, and the most complex. It is very simple because it contains just one instruction, **SWI**, whose assembler format has absolutely no variants or options.

The general form of **SWI** is:

```
SWI{cond}  <expression>
```

It is complex because depending on **<expression>**, **SWI** will perform tasks as disparate as displaying characters on the screen, setting the auto-repeat speed of the keyboard and loading a file from the disc.

SWI is the user's access to the operating system of the computer. When a **SWI** is executed, the CPU enters supervisor mode, saves the return address in R14_SVC, and jumps to location 8. From here, the operating system takes over. The way in which the **SWI** is used depends on **<expression>**. This is encoded as a 24-bit field in the instruction. The operating system can examine the instruction using, for example:.

```
STMFD  R13!,{R0-R12}        ;Save user's registers
BIC    R14,R14,#&FC000003   ;Mask status bits
LDR    R13,[R14,#-4]        ;Load SWI instruction
```

to find out what **<expression>** is.

Since the interpretation of **<expression>** depends entirely on the system in which the program is executing, we cannot say much more about **SWI** here. However, as practical programs need to use operating system screen, and the latter reads a character from the keyboard and returns it in the bottom byte of R0.

Note: The code in the example above will be executed in **SVC** mode, so the accesses to R13 and R14 are actually to R13_SVC and R14_SVC. Thus the user's versions of these registers do not have to be saved.

3.7 Instruction timings

It is informative to know how quickly you can expect instructions to execute. This section gives the timing of all of the ARM instructions. The times are expressed in terms of 'cycles'. A cycle is one tick of the crystal oscillator clock which drives the ARM. In fact there are three types of cycles, called sequential, non-sequential and internal.

Sequential (s) cycles are those used to access memory locations in sequential order. For example, when the ARM is executing a series of group one instructions with no interruption from branches and load/store operations, sequential cycles will be used.

Non-sequential (n) cycles are those used to access external memory when non-consecutive locations are required. For example, the first instruction to be loaded after a branch instruction will use an n-cycle.

Internal (i) cycles are those used when the ARM is not accessing memory at all, but performing some internal operation.

On a typical ARM, the clock speed is 8MHz (eight million cycles a second). S cycles last 125 nanoseconds for RAM and 250ns for ROM. All n-cycles are 250ns. All i-cycles are 125ns in duration.

Instructions which are skipped due to the condition failing always execute in 1s cycle.

Group one

MOV, **ADD** etc. 1 s-cycle. If **<rhs>** contains a shift count in a register (i.e. not an immediate shift), add 1 s-cycle. If **<dest>** is R15, add 1 s + 1 n-cycle.

Group one A

MUL, **MLA**. 1 s + 16 i- cycles worst-case.

Group two

LDR. 1 s + 1 n + 1 i-cycle. If **<dest>** is R15, add 1 s + 1n-cycle.

STR. 2 n-cycles.

Group three

LDM. (regs-1) s + 1 n + 1 i-cycle. Regs is the number of registers loaded. Add 1 s + 1 n-cycles if R15 is loaded.

STM. 2 n + (regs-1) s-cycles.

Group four

B, **BL**. 2 s + 1 n-cycles.

Group five

SWI. 2 s + 1 n-cycles.

From these timings you can see that when the ARM executes several group one instructions in sequence, it does so at a rate of eight million a second. The overhead of calling and returning from a subroutine is 3 s+1 n-cycles, or 0.525 microseconds if the return address is stored, or 2 s+5 n-cycles, or 1.5µs if the return address is stacked.

A multiply by zero or one takes 125ns. A worst-case multiply takes 2.125µs. Saving eight registers on the stack takes 7 s + 2 n-cycles, or 1.375µs. Loading them back, if one of them is the PC, takes 1.75 µ6s.