5 The ARM Instruction Set

Summary of chapter contents

In Chapter 3 we looked at user-level ARM assembly language programming and got the general flavour of the ARM instruction set. In this chapter we will look in finer detail at the instruction set to see the full range of instructions that are available in the standard ARM instruction set.

Some ARM cores will also execute a compressed form of the instruction set where a subset of the full ARM instruction set is encoded into 16-bit instructions. These instructions are 'Thumb' instructions, and are discussed in Chapter 7. The only aspects of the Thumb architecture we will see in this chapter are the instructions available in the ARM instruction set which cause the processor to switch to executing Thumb instructions. Likewise, some ARM cores support instruction set extensions to enhance their signal processing capabilities, discussion of which is deferred to Section 8.9 on page 239.

As with any processor's full instruction set, the ARM instruction set has corners which conceal complex behaviour. Often these corners are not at all useful to programmers, in which case ARM Limited does not define the behaviour of the processor in the corner cases and the corresponding instructions should not be used. The fact that a particular implementation of the ARM behaves in a particular way in such a case should not be taken as meaning that future implementations will behave the same way. Programs should only use instructions with defined semantics!

Some ARM instructions are not available on all ARM chips; these will be highlighted as they arise.

5.1 Introduction

The ARM programmers' model was introduced in Figure 2.1 on page 39. In this chapter we will consider the supervisor and exception modes, so now the shaded registers will also come into play.

Data types

ARM processors support six data types:

- 8-bit signed and unsigned bytes.
- 16-bit signed and unsigned half-words; these are aligned on 2-byte boundaries.
- 32-bit signed and unsigned words; these are aligned on 4-byte boundaries.

(Some older ARM processors do not have half-word and signed byte support.)

ARM instructions are all 32-bit words and must be word-aligned. Thumb instructions are half-words and must be aligned on 2-byte boundaries.

Internally all ARM operations are on 32-bit operands; the shorter data types are only supported by data transfer instructions. When a byte is loaded from memory it is zero- or sign-extended to 32 bits and then treated as a 32-bit value for internal processing.

ARM coprocessors may support other data types, and in particular there is a defined set of types to represent floating-point values. There is no explicit support for these types within the ARM core, however, and in the absence of a floating-point coprocessor these types are interpreted by software which uses the standard types listed above.

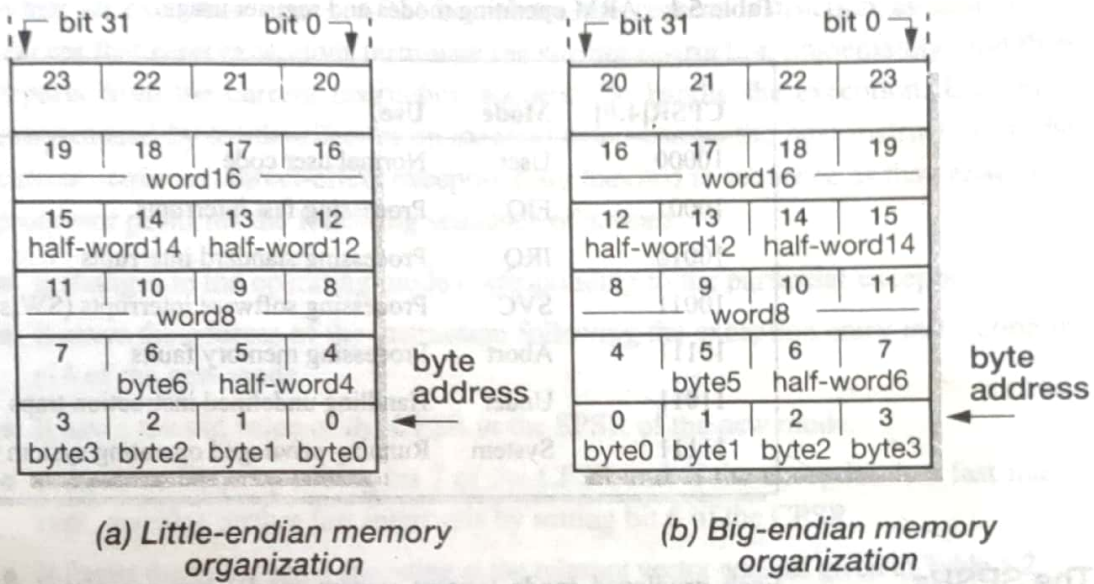
Memory organization

There are two ways to store words in a byte-addressed memory, depending on whether the least significant byte is stored at a lower or higher address than the next most significant byte. Since there is no good reason for choosing one approach over the other the argument as to which is better is more a matter of religion than reason.

The two schemes are illustrated in Figure 5.1 on page 107, which shows how an assortment of data types would be stored under the two schemes. ('half-word12' is found at address 12, and so on.)

The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches is derived from Swift's *Gulliver's Travels*. The inhabitants of Lilliput, who are well known for being rather small are, in addition, constrained by law to break their eggs only at the little end. When this law is imposed, those of their fellow citizens who prefer to break their eggs at the big end take exception to the new rule and civil war breaks out. The big-endians eventually take refuge on a nearby island, which is the kingdom of Blefuscu. The civil war results in many casualties.

The application of the 'big-endian' and 'little-endian' terms to the two ways to organize computer memory comes from 'On Holy Wars and a Plea for Peace' by Danny Cohen in the October 1981 issue of *Computer*.



(a) Little-endian memory organization

(b) Big-endian memory organization

Figure 5.1 Little- and big-endian memory organizations.

To my knowledge, no one has yet been mortally wounded in an argument over byte ordering. However the issue causes significant practical difficulties when datasets are transferred between machines of opposite orderings.

Most ARM chips remain strictly neutral in the dispute and can be configured to work with either memory arrangement, though they default to little-endian. Throughout this book we will assume a little-endian ordering, where bytes of increasing significance are stored at increasing addresses in memory. ARM may be neutral, but I am not!

Privileged modes

Most programs operate in user mode as described in Chapter 3. However, ARM has other **privileged** operating modes which are used to handle exceptions and supervisor calls (which are sometimes called **software interrupts**).

The current operating mode is defined by the bottom five bits of the CPSR (see Figure 2.2 on page 40). The interpretation of these bits is summarized in Table 5.1 on page 108. Where the register set is not the user registers, the relevant shaded registers shown in Figure 2.1 on page 39 replace the corresponding user registers and the current SPSR (**Saved Program Status Register**; see below) also becomes accessible.

Some ARM processors do not support all of the above operating modes, and some also support '26-bit' modes for backwards compatibility with older ARMs; these will be discussed further in Section 5.23 on page 147.

The privileged modes can only be entered through controlled mechanisms; with suitable memory protection they allow a fully protected operating system to be built. This issue will be discussed further in Chapter 11.

Most ARMs are used in embedded systems where such protection is inappropriate, but the privileged modes can still be used to give a weaker level of protection that is useful for trapping errant software.

Table 5.1 ARM operating modes and register usage.

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

The SPSRs

Each privileged mode (except system mode) has associated with it a Saved Program Status Register, or SPSR. This register is used to save the state of the CPSR (Current Program Status Register) when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed. Often the SPSR may be untouched from the time the privileged mode is entered to the time it is used to restore the CPSR, but if the privileged software is to be re-entrant (for example, if supervisor code makes supervisor calls to itself) then the SPSR must be copied into a general register and saved.

5.2 Exceptions

Exceptions are usually used to handle unexpected events which arise during the execution of a program, such as interrupts or memory faults. In the ARM architecture the term is also used to cover software interrupts and undefined instruction traps (which do not really qualify as 'unexpected') and the system reset function which logically arises before rather than during the execution of a program (although the processor may be reset again while running). These events are all grouped under the 'exception' heading because they all use the same basic mechanism within the processor.

ARM exceptions may be considered in three groups:

1. Exceptions generated as the direct effect of executing an instruction.
   Software interrupts, undefined instructions (including coprocessor instructions where the requested coprocessor is absent) and prefetch aborts (instructions that are invalid due to a memory fault occurring during fetch) come under this heading.
2. Exceptions generated as a side-effect of an instruction.
   Data aborts (a memory fault during a load or store data access) are in this class.
3. Exceptions generated externally, unrelated to the instruction flow.
   Reset, IRQ and FIQ fall into this category.

Exception entry

When an exception arises, ARM completes the current instruction as best it can (except that *reset* exceptions terminate the current instruction immediately) and then departs from the current instruction sequence to handle the exception. Exception entry caused by a side-effect or an external event usurps the next instruction in the current sequence; direct-effect exceptions are handled in sequence as they arise. The processor performs the following sequence of actions:

- It changes to the operating mode corresponding to the particular exception.
- It saves the address of the instruction following the exception entry instruction in r14 of the new mode.
- It saves the old value of the CPSR in the SPSR of the new mode.
- It disables IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disables further fast interrupts by setting bit 6 of the CPSR.
- It forces the PC to begin executing at the relevant vector address given in Table 5.2.

Table 5.2 Exception vector addresses.

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Normally the vector address will contain a branch to the relevant routine, though the FIQ code can start immediately since it occupies the highest vector address.

The two banked registers in each of the privileged modes are used to hold the return address and a stack pointer; the stack pointer may be used to save other user registers so that they can be used by the exception handler. FIQ mode has additional private registers to give better performance by avoiding the need to save user registers in most cases where it is used.

Exception return

Once the exception has been handled the user task is normally resumed. This requires the handler code to restore the user state exactly as it was when the exception first arose:

- Any modified user registers must be restored from the handler's stack.
- The CPSR must be restored from the appropriate SPSR.
- The PC must be changed back to the relevant instruction address in the user instruction stream.

Note that the last two of these steps cannot be carried out independently. If the CPSR is restored first, the banked r14 holding the return address is no longer accessible; if the PC is restored first, the exception handler loses control of the instruction stream and cannot cause the restoration of the CPSR to take place. There are also more subtle difficulties to do with ensuring that instructions are always fetched in the correct operating mode to ensure that memory protection schemes are not bypassed. Therefore ARM provides two mechanisms which cause both steps to happen atomically as part of a single instruction. One of these is used when the return address has been kept in the banked r14 and the other when the return address has been saved onto a stack. First we look at the case where the return address is in r14.

- To return from a SWI or undefined instruction trap use:

```
MOVS    pc, r14
```

- To return from an IRQ, FIQ or prefetch abort use:

```
SUBS    pc, r14, #4
```

- To return from a data abort to retry the data access use:

```
SUBS    pc, r14, #8
```

The 'S' modifier after the opcode signifies the special form of the instruction when the destination register is the PC. Note how the return instruction incorporates an adjustment to the return address where necessary:

- IRQ and FIQ must return one instruction early in order to execute the instruction that was 'usurped' for the exception entry.
- Prefetch abort must return one instruction early to execute the instruction that had caused a memory fault when first requested.
- Data abort must return two instructions early to retry the data transfer instruction, which was the instruction *before* the one usurped for exception entry.

If the handler has copied the return address out onto a stack (in order, for example, to allow re-entrant behaviour, though note that in this case the SPSR must be saved as well as the PC) the restoration of the user registers and the return may be implemented with a single multiple register transfer instruction such as:

```
LDMFD   r13!, {r0-r3,pc}^  ; restore and return
```

Here the '^' after the register list (which must include the PC) indicates that this is a special form of the instruction. The CPSR is restored at the same time that the PC is loaded from memory, which will always be the last item transferred from memory since the registers are loaded in increasing order.

The stack pointer (r13) used here is the banked register belonging to the privileged operating mode; each privileged mode can have its own stack pointer which must be initialized during system start-up.

Clearly the stack return mechanism can only be employed if the value in r14 was adjusted, where necessary, before being saved onto the stack.

Exception priorities

Since multiple exceptions can arise at the same time it is necessary to define a priority order to determine the order in which the exceptions are handled. On ARM this is:

1. reset (highest priority);
2. data abort;
3. FIQ;
4. IRQ;
5. prefetch abort;
6. SWI, undefined instruction (including absent coprocessor). These are mutually exclusive instruction encodings and therefore cannot occur simultaneously.

Reset starts the processor from a known state and renders all other pending exceptions irrelevant.

The most complex exception scenario is where an FIQ, an IRQ and a third exception (which is not Reset) happen simultaneously. FIQ has higher priority than IRQ and also masks it out, so the IRQ will be ignored until the FIQ handler explicitly enables IRQ or returns to the user code.

If the third exception is a data abort, the processor will enter the data abort handler and then immediately enter the FIQ handler, since data abort entry does not mask FIQs out. The data abort is 'remembered' in the return path and will be processed when the FIQ handler returns.

If the third exception is not a data abort, the FIQ handler will be entered immediately. When FIQ and IRQ have both completed, the program returns to the instruction which generated the third exception, and in all the remaining cases the exception will recur and be handled accordingly.

Address exceptions

The observant reader will have noticed that Table 5.2 on page 109 shows the use of all of the first eight word locations in memory as exception vector addresses apart from address 0x00000014. This location *was* used on earlier ARM processors which operated within a 26-bit address space to trap load or store addresses which fell outside the address space. These traps were referred to as 'address exceptions'.

Since 32-bit ARMs are unable to generate addresses which fall outside their 32-bit address space, address exceptions have no role in the current architecture and the vector address at 0x00000014 is unused.

5.3 Conditional execution

An unusual feature of the ARM instruction set is that every instruction (with the exception of certain v5T instructions) is conditionally executed. Conditional branches are a standard feature of most instruction sets, but ARM extends the

conditional execution to all of its instructions, including supervisor calls and coprocessor instructions. The condition field occupies the top four bits of the 32-bit instruction field:

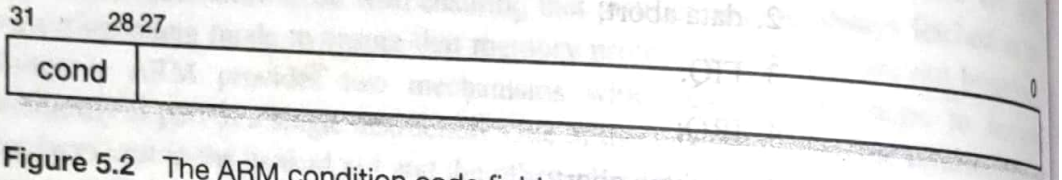


Figure 5.2 The ARM condition code field.

Each of the 16 values of the condition field causes the instruction to be executed or skipped according to the values of the N, Z, C and V flags in the CPSR. The conditions are given in Table 5.3 on page 113. Every ARM instruction mnemonic may be extended by appending the two letters defined in this table, though the 'always' condition (AL) may be omitted since it is the default condition that is assumed if no other condition is specified.

The 'never' condition

The 'never' condition (NV) should not be used – there are plenty of other ways to write no-ops (instructions that have no effect on the processor state) in ARM code. The reason to avoid the 'never' condition is that ARM Limited have indicated that they may use this area of the instruction space for other purposes in the future (and have done so in architecture v5T), so although current ARMs may behave as expected, there is no guarantee that future variants will behave the same way.

Alternative mnemonics

Where alternative mnemonics are shown in the same row in the condition table this indicates that there is more than one way to interpret the condition field. For instance in row 3 the same condition field value can be invoked by the mnemonic extension CS or HS. Both cause the instruction to be executed only if the C bit in the CPSR is set. The alternatives are available because the same test is used in different circumstances. If you have previously added two unsigned integers and want to test whether there was a carry-out from the addition, you should use CS. If you have compared two unsigned integers and want to test whether the first was higher or the same as the second, use HS. The alternative mnemonic removes the need for the programmer to remember that an unsigned comparison sets the carry on higher or the same.

The observant reader will note that the conditions are in pairs where the second condition is the inverse of the first, so for any condition the opposite condition is also available (with the exception of 'always', since 'never' should not be used). Therefore whenever *if...then...* can be implemented with conditional instructions, *...else...* can be added using instructions with the opposite condition.

Table 5.3 ARM condition codes.

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set,and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

5.4 Branch and Branch with Link (B, BL)

Branch and Branch with Link instructions are the standard way to cause a switch in the sequence of instruction execution. The ARM normally executes instructions from sequential word addresses in memory, using conditional execution to skip over individual instructions where required. Whenever the program must deviate from sequential execution a control flow instruction is used to modify the program counter. Although there are several ways to achieve this in special circumstances, Branch and Branch with Link instructions are the standard way.

Binary encoding

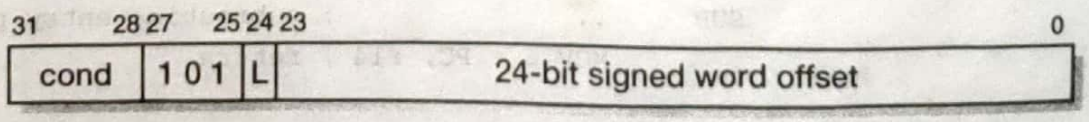


Figure 5.3 Branch and Branch with Link binary encoding.

Description

Branch and Branch with Link instructions cause the processor to begin executing instructions from an address computed by sign extending the 24-bit offset specified in the instruction, shifting it left two places to form a word offset, then adding it to the program counter which contains the address of the branch instruction plus eight bytes. (See 'PC behaviour' on page 78 for an explanation of the PC offset.) The assembler will compute the correct offset under normal circumstances.

The range of the branch instruction is +/– 32 Mbytes.

The Branch with Link variant, which has the L bit (bit 24) set, also moves the address of the instruction following the branch into the link register (r14) of the current processor mode. This is normally used to perform a subroutine call, with the return being caused by copying the link register back into the PC.

Both forms of the instruction may be executed conditionally or unconditionally.

Assembler format

```
B{L}{<cond>}    <target address>
```

'L' specifies the branch and link variant; if 'L' is not included a branch without link is generated.'<cond>' should be one of the mnemonic extensions given in Table 5.3 on page 113 or, if omitted, 'AL' is assumed.'<target address>' is normally a label in the assembler code; the assembler will generate the offset (which will be the difference between the address of the target and the address of the branch instruction plus 8).

Examples

An unconditional jump:

```
        B       LABEL   ; unconditional jump ..
        ..
LABEL   ..              ; .. to here
```

To execute a loop ten times:

```
        MOV     r0, #10 ; initialize loop counter
LOOP    ..
        SUBS    r0, #1  ; decrement counter setting CCs
        BNE     LOOP    ; if counter <> 0 repeat loop..
        ..              ; .. else drop through
```

To call a subroutine:

```
        ..
        BL      SUB     ; branch and link to subroutine SUB
        ..              ; return to here
SUB     ..
        MOV     PC, r14 ; subroutine entry point
                        ; return
```

Conditional subroutine call:

```
...
CMP     r0, #5   ; if r0 < 5
BLLT    SUB1     ; then call SUB1
BLGE    SUB2     ; else call SUB2
...
```

(Note that this example will only work correctly if `SUB1` does not change the condition codes, since if the `BLLT` is taken it will return to the `BLGE`. If the condition codes are changed by `SUB1`, `SUB2` may be executed as well.)

Notes

1. If you are familiar with other RISC processors you might expect ARM to execute the instruction after the branch before moving to `LABEL` in the first example above, following the *delayed branch* model employed by many other RISCs. This expectation will not be fulfilled, however, since ARM does not employ a delayed branch mechanism.
2. Branches which attempt to go past the beginning or the end of the 32-bit address space should be avoided since they may have unpredictable results.

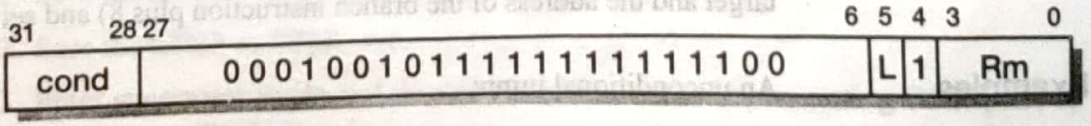
5.5 Branch, Branch with Link and eXchange (BX, BLX)

These instructions are available on ARM chips which support the Thumb (16-bit) instruction set, and are a mechanism for switching the processor to execute Thumb instructions or for returning symmetrically to ARM and Thumb calling routines. A similar Thumb instruction causes the processor to switch back to 32-bit ARM instructions. The Thumb instruction set is described in Chapter 7.

BLX is available only on ARM processors that support architecture v5T.

Binary encoding

(1) `BX | BLX Rm`



(2) `BLX label`

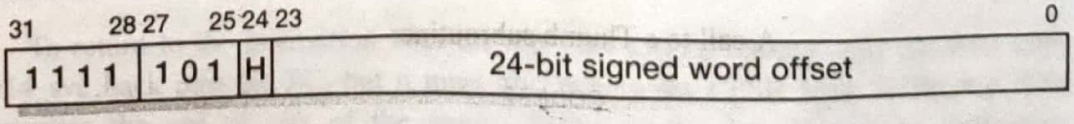


Figure 5.4 Branch (with optional link) and exchange instruction binary encodings.

Description

In the first format the branch target is specified in a register, Rm. Bit[0] of Rm is copied into the T bit in the CPSR and bits[31:1] are moved into the PC:

- If Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit.
- If Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[1].

In the second format the branch target is an address computed by sign extending the 24-bit offset specified in the instruction, shifting it left two places to form a word offset, then adding it to the program counter which contains the address of the BX instruction plus eight bytes. (See 'PC behaviour' on page 78 for an explanation of the PC offset.) The H bit (bit 24) is also added into bit 1 of the resulting address, allowing an odd half-word address to be selected for the target instruction which will always be a Thumb instruction (BL is used to target an ARM instruction). The assembler will compute the correct offset under normal circumstances. The range of the branch instruction is +/– 32 Mbytes.

The Branch with Link variants (BLX, available only on v5T processors) of both formats, which have the L bit (bit 5) set in the first format, also move the address of the instruction following the branch into the link register (r14) of the current processor mode. This is normally used to save the return address when calling a Thumb subroutine. If BX is used as the subroutine return mechanism the instruction set of the calling routine can be saved along with the return address, so the same return mechanism can be used to return symmetrically to either an ARM or a Thumb caller from an ARM or Thumb subroutine.

Format (1) instructions may be executed conditionally or unconditionally, but format (2) instructions are executed unconditionally.

Assembler format

```
1:      B{L}X{<cond>} Rm
2:      BLX <target address>
```

'<target address>' is normally a label in the assembler code; the assembler will generate the offset (which will be the difference between the word address of the target and the address of the branch instruction plus 8) and set the H bit if appropriate.

Examples

An unconditional jump:

```
        BX      r0          ; branch to address in r0,
                            ; enter Thumb state if r0[0] = 1
```

A call to a Thumb subroutine:

```
        CODE32              ; ARM code follows
        ..
        BLX     TSUB        ; call Thumb subroutine
```

```
        ..
        CODE16          ; start of Thumb code
TSUB    ..              ; Thumb subroutine
        BX      r14     ; return to ARM code
```

Notes

1. Some ARM processors which do not support the Thumb instruction set will trap these instructions, allowing software emulation of Thumb.
2. Only processors that implement ARM architecture v5T support either format of the BLX instruction (see Section 5.23 on page 147).

5.6 Software Interrupt (SWI)

The software interrupt instruction is used for calls to the operating system and is often called a 'supervisor call'. It puts the processor into supervisor mode and begins executing instructions from address 0x08.

If this area of memory is suitably protected it is possible to build an operating system on the ARM that is fully protected from a malicious user, though since ARM is rarely used in multi-user applications this level of protection is not often sought.

Binary encoding

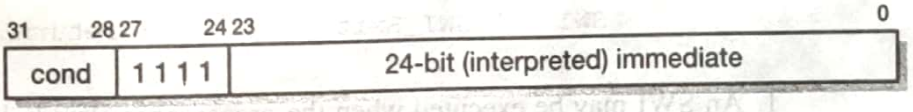


Figure 5.5 Software interrupt binary encoding.

Description

The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code.

If the condition is passed the instruction enters supervisor mode using the standard ARM exception entry sequence. In detail, the processor actions are:

1. Save the address of the instruction after the SWI in r14_svc.
2. Save the CPSR in SPSR_svc.
3. Enter supervisor mode and disable IRQs (but not FIQs) by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1.
4. Set the PC to 08_{16} and begin executing the instructions there.

To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc. This requires the use of one of the special forms of the data processing instruction described in the next section.

Assembler format

```
SWI{<cond>}     <24-bit immediate>
```

Examples

To output the character 'A':

```
MOV     r0, #'A'        ; get 'A' into r0..
SWI     SWI_WriteC      ; .. and print it
```

A subroutine to output a text string following the call:

```
        ..
        BL      STROUT          ; output following message
        =       "Hello World",&0a,&0d,0
        ..                      ; return to here
        ..
STROUT  LDRB    r0, [r14], #1   ; get character
        CMP     r0, #0          ; check for end marker
        SWINE   SWI_WriteC      ; if not end, print..
        BNE     STROUT          ; .. and loop
        ADD     r14, #3         ; align to next word
        BIC     r14, #3
        MOV     pc, r14         ; return
```

To finish executing the user program and return to the monitor program:

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

Notes

1. An SWI may be executed when the processor is already in supervisor mode provided the original return address (in r14_svc) and SPSR_svc have been saved; otherwise these registers will be overwritten when the SWI is executed.
2. The interpretation of the 24-bit immediate is system dependent, but most systems support a standard subset for character I/O and similar basic functions.

   The immediates can be specified as constant expressions, but it is usually better to declare names for the required calls (and set their values) at the start of the program (or import a file which declares their values for the local operating system) and then use these names in the code.

   To see how to declare names and give them values look at the 'Examples and exercises' on page 72.
3. The first instruction executed in supervisor mode, which is at 08_{16}, is normally a branch to the SWI handler which resides somewhere nearby in memory. Writing the SWI handler to start at 08_{16} is not possible because the next memory word, at $0C_{16}$, is the entry point for the prefetch abort handler.

5.7 Data processing instructions

The ARM data processing instructions are used to modify data values in registers. The operations that are supported include arithmetic and bit-wise logical combinations of 32-bit data types. One operand may be shifted or rotated *en route* to the ALU, allowing, for example, shift and add in a single instruction.

Multiply instructions use different formats, so these are considered separately in the next section.

Binary encoding

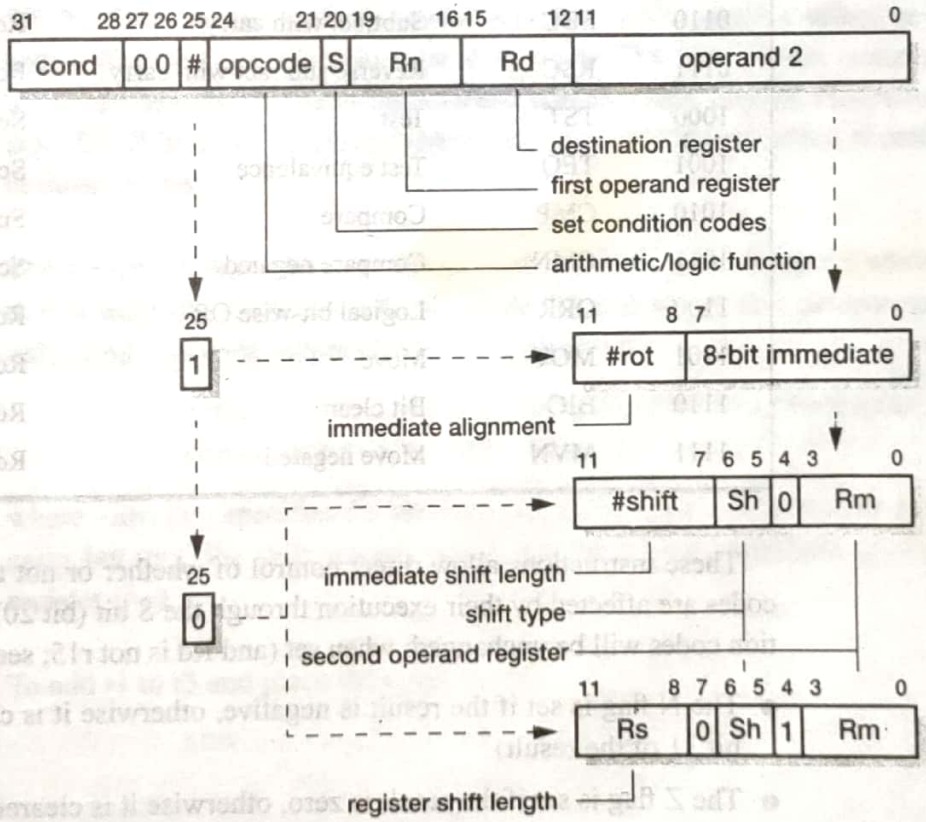


Figure 5.6 Data processing instruction binary encoding.

Description

The ARM data processing instructions employ a 3-address format, which means that the two source operands and the destination register are specified independently. One source operand is always a register; the second may be a register, a shifted register or an immediate value. The shift applied to the second operand, if it is a register, may be a logical or arithmetic shift or a rotate (see Figure 3.1 on page 54), and it may be by an amount specified either as an immediate quantity or by a fourth register.

The operations that may be specified are listed in Table 5.4 on page 120.

When the instruction does not require all the available operands (for instance `MOV` ignores Rn and `CMP` ignores Rd) the unused register field should be set to zero. The assembler will do this automatically.

Table 5.4 ARM data processing instructions.

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn – Op2
0011	RSB	Reverse subtract	Rd := Op2 – Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn – Op2 + C – 1
0111	RSC	Reverse subtract with carry	Rd := Op2 – Rn + C – 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn – Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

These instructions allow direct control of whether or not the processor's condition codes are affected by their execution through the S bit (bit 20). When clear, the condition codes will be unchanged; when set (and Rd is not r15; see below):

- The N flag is set if the result is negative, otherwise it is cleared (that is, N equals bit 31 of the result).
- The Z flag is set if the result is zero, otherwise it is cleared.
- The C flag is set to the carry-out from the ALU when the operation is arithmetic (`ADD`, `ADC`, `SUB`, `SBC`, `RSB`, `RSC`, `CMP`, `CMN`) or to the carry-out from the shifter otherwise. If no shift is required, C is preserved.
- The V flag is preserved in non-arithmetic operations. It is set in an arithmetic operation if there is an overflow from bit 30 to bit 31 and cleared if no overflow occurs. It has significance only when an arithmetic operation has operands that are viewed as 2's complement signed values, and indicates a result that is out of range.

Multiply by a constant

These instructions may be used to multiply a register by a small constant much more efficiently than can be achieved using the multiply instructions described in the next section. Examples are given below.

Use of r15

The PC may be used as a source operand except when a register-specified shift amount is used, in which case none of the three source registers should be r15. When r15 is used as a source operand, the value supplied is the address of the instruction plus eight bytes. (The 8-byte offset exposes the pipelined operation of the processor; see 'PC behaviour' on page 78.)

The PC may also be specified as the destination for the result, in which case the instruction is a form of branch. This is exploited as a standard way to return from a subroutine.

When the PC is specified as the destination register Rd, the S bit still controls the effect of the instruction on the CPSR, but in a rather different way. Instead of updating the flags according to the ALU output as described above, when the S bit is set the SPSR of the current mode is copied into the CPSR, possibly affecting the interrupt enable flags and the processor operating mode. This mechanism restores the PC and the CPSR atomically, and is the standard way to return from an exception. Since there is no SPSR in user and system modes, this form of the instruction should not be used in those modes.

Assembler format

The assembler representation is one of the following, omitting `Rn` when the instruction is monadic (`MOV`, `MVN`) or `Rd` when the instruction is a comparison producing only condition code outputs (`CMP`, `CMN`, `TST`, `TEQ`):

```
<op>{<cond>}{S} Rd, Rn, #<32-bit immediate>
<op>{<cond>}{S} Rd, Rn, Rm, {<shift>}
```

where `<shift>` specifies the shift type (`LSL`, `LSR`, `ASL`, `ASR`, `ROR` or `RRX`) and, in all cases but `RRX`, the shift amount which may be a 5-bit immediate (`#<#shift>`) or a register (`Rs`).

Examples

To add r1 to r3 and place the result in r5:

```
ADD     r5, r1, r3
```

To decrement r2 and check for zero:

```
SUBS    r2, r2, #1          ; dec r2 and set cc
BEQ     LABEL               ; branch if r2 zero
..                          ; .. else fall through
```

To multiply r0 by 5:

```
ADD     r0, r0, r0, LSL #2
```

A subroutine to multiply r0 by 10:

```
        MOV     r0, #3
        BL      TIMES10
        ..
TIMES10 MOV     r0, r0, LSL #1      ; x 2
        ADD     r0, r0, r0, LSL #2  ; x 5
        MOV     pc, r14             ; return
```

To add a 64-bit integer in r0, r1 to one in r2, r3:

```
        ADDS    r2, r2, r0          ; add lower, save carry
        ADC     r3, r3, r1          ; add higher and carry
```

Notes

1. Since the immediate field must be encoded within a subset of a 32-bit instruction, not all 32-bit immediate values can be represented. The binary encoding shown in Figure 5.6 on page 119 shows how the immediate values are encoded. The immediate value is generated by rotating an 8-bit immediate field through an even number of bit positions.

5.8 Multiply instructions

ARM multiply instructions produce the product of two 32-bit binary numbers held in registers. The result of multiplying two 32-bit binary numbers is a 64-bit product. Some forms of the instruction, available only on certain versions of the processor, store the full result into two independently specified registers; other forms store only the least significant 32 bits into a single register.

In all cases there is a multiply-accumulate variant that adds the product to a running total and both signed and unsigned operands may be used. The least significant 32 bits of the result are the same for signed and unsigned operands, so there is no need for separate signed and unsigned versions of the 32-bit result instructions.

Binary encoding

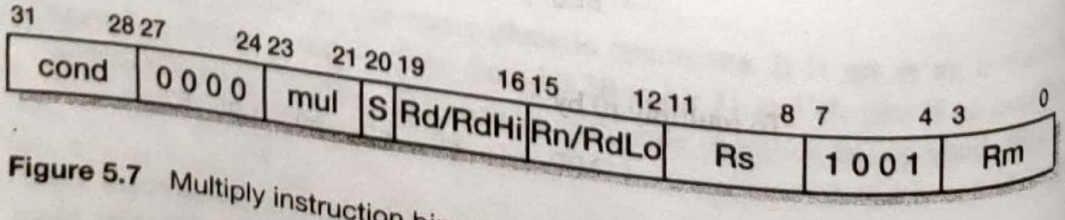


Figure 5.7 Multiply instruction binary encoding.

Table 5.5 Multiply instructions.

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs