APPENDIX

B

ARM Instruction Set

This appendix contains a summary of version v3 of the ARM instruction set architecture (ISA), which was described in Part I of Chapter 3. A brief discussion of enhancements introduced in later versions of the ISA is also included. The ARM register structure is shown in Figure 3.1. Figure 3.2 shows the general format of an instruction. Here, we give the details for the different types of instructions. All instructions are encoded into a 32-bit word. The memory is byte addressable and addresses are 32 bits long. There are two operand sizes: word (32 bits) and byte (8 bits). A byte operand occupies the lower 8 bits of a processor register. When a byte operand is loaded into a register, the high-order three bytes are cleared to zero.

B.1 INSTRUCTION ENCODING

The encodings for five types of instructions are shown in Figure B.1. Instruction types are distinguished by the bit patterns starting at bit position b_{27}. The multiply instructions in Figure B.1*b* are detected to be different from the group containing the other arithmetic and logic instructions, shown in Figure B.1*a*, as follows. When I = 0 in the latter group, either bit b_7 or bit b_4 is 0, whereas both of these bits are 1 in the multiply instructions. Note that the R*n* and R*d* fields are reversed in the multiply instructions.

The sections that follow give the encoding details, with examples, for each of the five types of instructions. The full ARM architecture has additional instructions associated with coprocessor operations. We provide a brief discussion of them.

Conditional Execution of Instructions

The conditions for conditional execution of instructions are listed in Table B.1. The mnemonic for a desired condition is added to an instruction OP-code mnemonic as a suffix. The AL condition specifies that the instruction is executed irrespective of the state of the condition code flags. This is the default condition if the suffix is omitted in assembly language programs. For example, ADD (Add) and B (Branch) are always executed, but ADDEQ and BEQ are executed only if Z = 1. Conditional execution of an instruction often follows a Compare instruction. The Name column in Table B.1 is written with this in mind.

B.1.1 ARITHMETIC AND LOGIC INSTRUCTIONS

Arithmetic and logic operations, as well as compare, test, and move operations, are performed by instructions with the format shown in Figure B.2. The first operand is contained in register R*n*. The second operand is contained in register R*m* or is an unsigned 8-bit immediate operand, as indicated by the I bit. The result of the operation specified by the 4-bit OP code is placed in register R*d*. If the S bit is equal to 1, condition code flags are affected by the result; otherwise (S = 0), they are not.

The general assembly language form for these instructions is

OP{Cond}{S} R*d*,R*n*,Operand 2

31–28	27–26	25	24–21	20	19–16	15–12	11–0
Condition	0 0	I	OP code	S	Rn	Rd	Operand 2

(a) Arithmetic, logic, compare, test, and move

31–28	27–22	21	20	19–16	15–12	11–8	7–4	3–0
Condition	0 0 0 0 0 0	A	S	Rd	Rn	Rs	1 0 0 1	Rm

(b) Multiply and Multiply Accumulate

31–28	27–26	25	24	23	22	21	20	19–16	15–12	11–0
Condition	0 1	I	P	U	B	W	L	Rn	Rd	Offset

(c) Single word or byte transfer from/to memory

31–28	27–25	24	23	22	21	20	19–16	15–0
Condition	1 0 0	P	U	-	W	L	Rn	Register list

(d) Multiple word transfer from/to memory

31–28	27–25	24	23–0
Condition	1 0 1	K	Offset

(e) Branch and Branch with Link

I	Immediate	P	Pre/Post-index	W	Writeback
S	Set	U	Up/Down	L	Load/Store
A	Accumulate	B	Byte/Word	K	Link

Figure B.1 ARM instruction encoding formats.

Table B.1 Condition field encoding in ARM instructions

Condition field $b_{31} \ldots b_{28}$	Condition suffix	Name	Condition code test
0 0 0 0	EQ	Equal (zero)	$Z = 1$
0 0 0 1	NE	Not equal (nonzero)	$Z = 0$
0 0 1 0	CS/HS	Carry set/Unsigned higher or same	$C = 1$
0 0 1 1	CC/LO	Carry clear/Unsigned lower	$C = 0$
0 1 0 0	MI	Minus (negative)	$N = 1$
0 1 0 1	PL	Plus (positive or zero)	$N = 0$
0 1 1 0	VS	Overflow	$V = 1$
0 1 1 1	VC	No overflow	$V = 0$
1 0 0 0	HI	Unsigned higher	$\overline{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
1 0 1 0	GE	Signed greater than or equal	$N \oplus V = 0$
1 0 1 1	LT	Signed less than	$N \oplus V = 1$
1 1 0 0	GT	Signed greater than	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	Always	
1 1 1 1		Not used	

For example, if the second operand is contained in a register ($I = 0$), the instruction

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

is executed unconditionally and performs the operation

$$R0 \leftarrow [R1] + [R2]$$

without affecting the condition code flags. If the OP code is changed to ADDS, the flags are affected by the result of the operation. If the latter instruction is to be executed conditionally, on the equal condition (EQ), the OP code is written as ADDEQS.

If the second operand is an immediate value ($I = 1$), it is given by the expression #constant. For example,

```
ADD    R0,R1,#17
```

performs the operation

$$R0 \leftarrow [R1] + 17$$

The immediate value is zero-extended to 32 bits before being used in the operation.

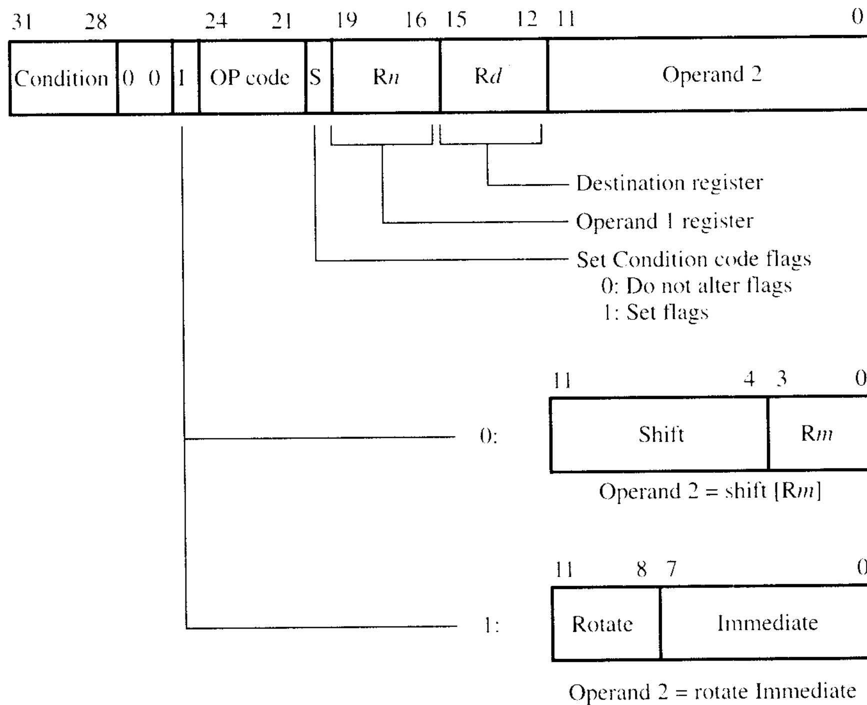


Figure B.2 ARM arithmetic, logic, compare, test, and move instructions.

Shifting of Operand 2

If Operand 2 is contained in a register (I = 0), it can be shifted before being used, as shown in Figure B.3. The shift is specified in bits b_{11-4}. If $b_4 = 0$, a shift amount in the range 0 through 31 is given by the 5-bit unsigned number in bits b_{11-7}. The type of shift is specified in bits b_6 and b_5. The condition code flag C is involved in the four shifts as shown in Figures 2.30 and 2.32. The rotate right operation (ROR) is done without the C bit when the shift amount is nonzero. However, when the shift amount is zero, the meaning is: rotate right one bit position including the C bit, as shown in Figure 2.32*d*. This operation can be indicated in the assembly language by using the mnemonic RRX (rotate right extended) with no shift amount specified. An example of the instruction syntax when the shift amount is specified directly in the instruction, as just described, is

```
ADD    R0,R1,R2, LSL #4
```

which shifts the operand in R2 left 4 bit positions (thus multiplying it by 16) before adding it to the contents of R1. If $b_4 = 1$, the shift amount is specified in the low-order 5 bits of register R*s*, as shown in Figure B.3. For example, the instruction

```
ADD    R0,R1,R2, LSR R3
```

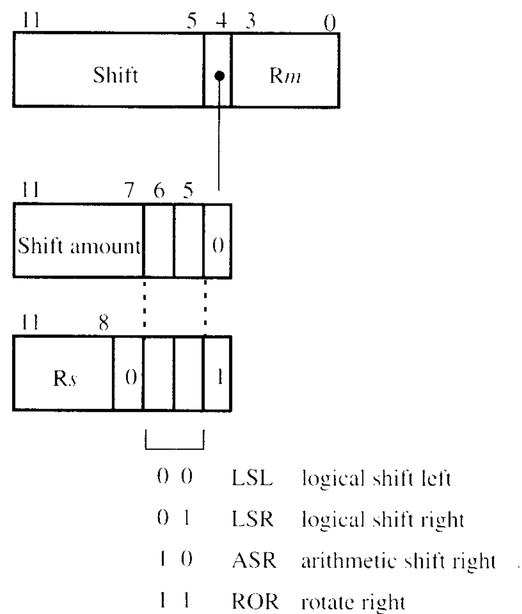


Figure B.3 ARM shift operations on operand 2 (Figure B.2) or address offset (Figure B.5) contained in register R*m*.

shifts the contents of the operand in R2 to the right a number of positions specified by the contents of R3.

- If Operand 2 is an immediate operand (I = 1), it can be rotated to the right as indicated in Figure B.2. This option permits a large number of 32-bit values to be generated by rotating the unsigned 8-bit immediate value. The number of rotation positions is $2n$, where n is the 4-bit number contained in bits b_{11-8}. Therefore, the range of rotations is an even number of bits from 0 through 30. This feature of the ARM instruction set partly compensates for the lack of 32-bit immediate operands in instructions. It is clear, however, that not all 32-bit values can be generated this way. A short sequence of instructions, including rotations and OR operations, can be used to synthesize 32-bit values that cannot be specified by rotation of a single 8-bit value.

The full set of 16 arithmetic and logic instructions is shown in Tables B.2 and B.3, along with the two multiply instructions discussed in the next subsection. There are six Add and Subtract instructions. The Add-with-carry and Subtract-with-carry instructions are needed to provide the capability to operate with multiple-word operands. Since only Operand 2 can be shifted, the two Reverse Subtract instructions are provided to allow the shifted operand to be the first operand of the subtract operation.

Table B.2 ARM arithmetic instructions

Mnemonic (Name)	OP code $b_{24} \ldots b_{21}$	Operation performed	CC flags affected if S = 1 N	Z	V	C
ADD (Add)	0 1 0 0	$Rd \leftarrow [Rn] + \text{Oper2}$	x	x	x	x
ADC (Add with carry)	0 1 0 1	$Rd \leftarrow [Rn] + \text{Oper2} + [C]$	x	x	x	x
SUB (Subtract)	0 0 1 0	$Rd \leftarrow [Rn] - \text{Oper2}$	x	x	x	x
SBC (Subtract with carry)	0 1 1 0	$Rd \leftarrow [Rn] - \text{Oper2} + [C] - 1$	x	x	x	x
RSB (Reverse subtract)	0 0 1 1	$Rd \leftarrow \text{Oper2} - [Rn]$	x	x	x	x
RSC (Reverse subtract with carry)	0 1 1 1	$Rd \leftarrow \text{Oper2} - [Rn] + [C] - 1$	x	x	x	x
MUL (Multiply)	(See Figure B.4)	$Rd \leftarrow [Rm] \times [Rs]$	x	x		
MLA (Multiply accumulate)	(See Figure B.4)	$Rd \leftarrow [Rm] \times [Rs] + [Rn]$	x	x		

Table B.3 ARM logic, compare, test, and move instructions

Mnemonic (Name)	OP code	Operation performed	CC flags affected if S = 1			
	$b_{24} \ldots b_{21}$		N	Z	V	C
AND (Logical AND)	0 0 0 0	$Rd \leftarrow [Rn] \wedge \text{Oper2}$	x	x		x
ORR (Logical OR)	1 1 0 0	$Rd \leftarrow [Rn] \vee \text{Oper2}$	x	x		x
EOR (Exclusive-OR)	0 0 0 1	$Rd \leftarrow [Rn] \oplus \text{Oper2}$	x	x		x
BIC (Bit clear)	1 1 1 0	$Rd \leftarrow [Rn] \wedge \neg\text{Oper2}$	x	x		x
CMP (Compare)	1 0 1 0	$[Rn] - \text{Oper2}$	x	x	x	x
CMN (Compare negative)	1 0 1 1	$[Rn] + \text{Oper2}$	x	x	x	x
TST (Bit test)	1 0 0 0	$[Rn] \wedge \text{Oper2}$	x	x		x
TEQ (Test equal)	1 0 0 1	$[Rn] \oplus \text{Oper2}$	x	x		x
MOV (Move)	1 1 0 1	$Rd \leftarrow \text{Oper2}$	x	x		x
MVN (Move complement)	1 1 1 1	$Rd \leftarrow \neg\text{Oper2}$	x	x		x

When an immediate value is used as Operand 2 in an Add or Subtract instruction, it can only be a positive value. But the assembly language allows the instructions

ADD R0,R1,#−5

and

SUB R0,R1,#−7

to be used, assembling them as

SUB R0,R1,#5

and

ADD R0,R1,#7

respectively.

In addition to the four logic instructions, AND, ORR, EOR, and BIC, the Move complement (MVN) instruction performs the logical NOT operation. The compare and test instructions always affect the condition code flags.

The two Move instructions are used to transfer Operand 2 or its bit complement into the destination register. Operand 2 can be contained in a register or it can be an immediate operand. Thus, in addition to performing register transfers, these two instructions are used to load constants into registers. The MVN instruction can be used to load negative numbers in the 2's-complement representation as follows. The instruction

MOV R0,#−10

is assembled as

MVN R0,#9

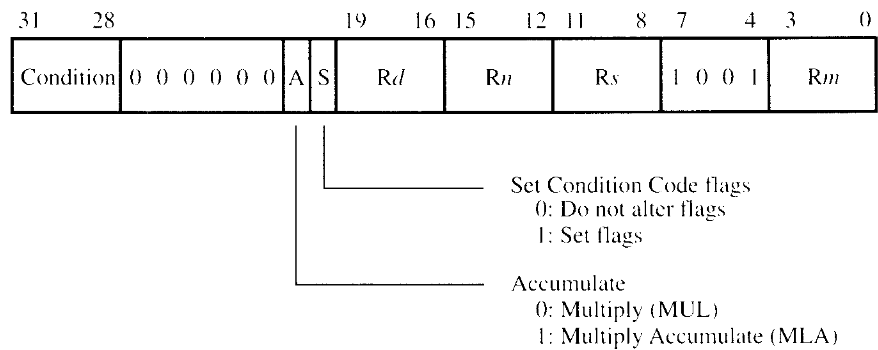
The bit-complement of $9 = 0\ldots01001$ is $1\ldots10110$, which is the 2's-complement representation for -10.

Multiply Instructions

The format and operation for the two multiply instructions are shown in Figure B.4 and Table B.2. None of the operands can be shifted. The product generated is a single-precision 32-bit value.

B.1.2 MEMORY LOAD AND STORE INSTRUCTIONS

The format for the two instructions used to access memory is shown in Figure B.5, and their operation is shown in Table B.4. The L bit, b_{20}, is 1 for a Load (LDR) instruction and 0 for a Store (STR) instruction. The B bit, b_{22}, is 1 for a byte operand and is 0 for a 32-bit word operand. A byte operand is located in the low-order byte position of R*d*. The effective address of the memory operand is determined by adding (U = 1) or subtracting (U = 0) the offset specified by the Offset field with the contents of register R*n*. The P and W bits determine the pre- or post-indexing and writeback operations as



MUL: Rd ← [Rm] x [Rs]

MLA: Rd ← [Rm] x [Rs] + [Rn]

Figure B.4 ARM Multiply and Multiply Accumulate instructions.

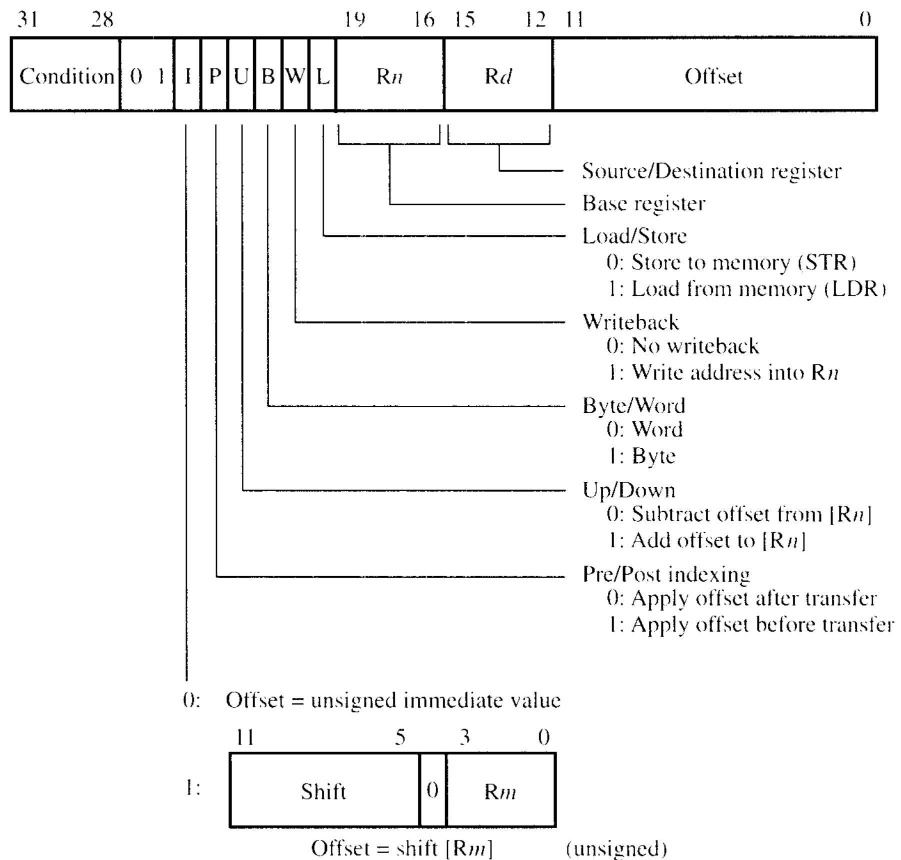


Figure B.5 ARM Load and Store instructions.

Table B.4 ARM instructions for single word or byte transfer from/to memory

Mnemonic (Name)	Instruction bits B L	Operation performed
LDR (Load word)	0 1	R*d* ← [EA]
LDRB (Load byte)	1 1	R*d* ← [EA]
STR (Store word)	0 0	EA ← [R*d*]
STRB (Store byte)	1 0	EA ← [R*d*]

indicated in Figure B.5 and described in Table 3.1. The I bit determines how the Offset field is interpreted, as specified in Figures B.5 and B.3. Note that only one of the two shifting methods can be used on the contents of register R*m*.

Examples of memory access operations are as follows. For the instruction

LDR R0,[R1,#100]

the operation performed is

$$R0 \leftarrow [[R1] + 100]$$

and the bit settings are I = 0, P = 1, U = 1, B = 0, W = 0, and L = 1. The range of offsets is ±4095. For the instruction

LDR R0,[R1,R2]

the operation performed is

$$R0 \leftarrow [[R1] + [R2]]$$

with the I bit changed to 1 and all the other settings left the same.

When the offset is contained in a register, it can be shifted before being added to or subtracted from the base register R*n*. The shift can only be specified by the 5-bit immediate method shown in Figure B.3. For example, the instruction

LDR R0,[R1,−R2, LSL #4]!

performs the operation

$$R0 \leftarrow [[R1] - 16 \times [R2]]$$

and the effective address is written back into R1. The bit settings for this instruction are I = 1, P = 1, U = 0, B = 0, W = 1, and L = 1.

If the program counter, R15, is specified as the base register, the Relative addressing mode is implemented as indicated in Table 3.1. In this case, pre-indexing without writeback, along with an immediate offset, is used to generate the effective address of the operand. The assembly language allows the absolute address of the operand to be named. The assembler computes the offset value relative to the updated contents of the program counter at the time the instruction is executed. For example, if the instruction

```
LDR    R0,PARAMETER
```

is to be placed at address 1000 and the label PARAMETER represents the address 1100, the assembler will generate the instruction

```
LDR    R0,[R15,#92]
```

At the time the offset is added to the contents of the program counter, the counter has been updated to contain 1008, so the offset must be 92 to generate the correct effective address of 1100 = 1008 + 92.

B.1.3 BLOCK LOAD AND STORE INSTRUCTIONS

Figure B.6 shows the encoding for the instructions used to transfer data between a block of consecutive memory words and a specified subset of the 16 processor registers. The

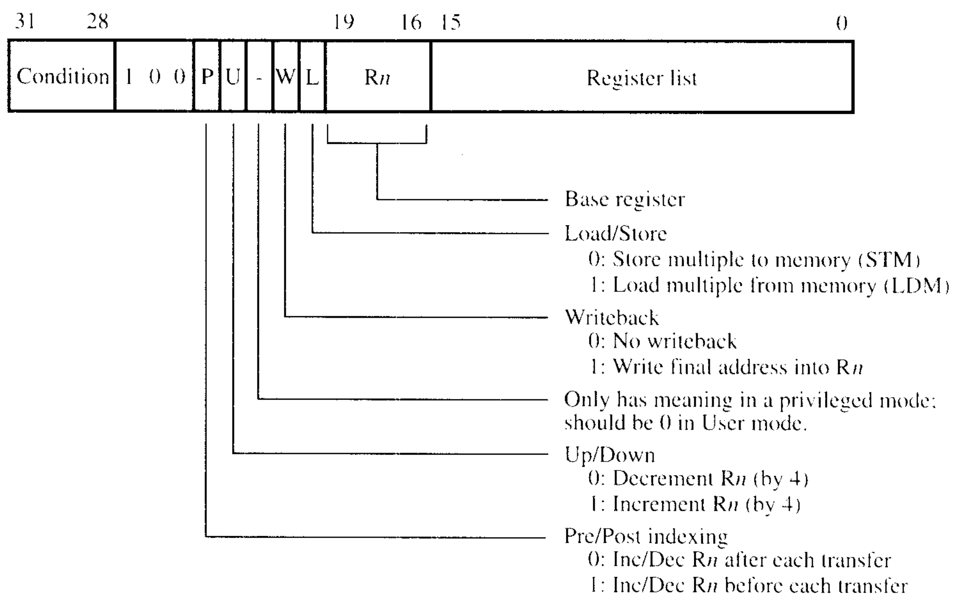


Figure B.6 ARM block transfer instructions.

OP code LDM (Load multiple) is used for loading memory operands into the registers, and STM (Store multiple) is used for the store operation. The L bit is 1 for the load operation and 0 for the store operation. The registers involved are specified by the locations of 1s in the 16-bit Register-list field in bits b_{15-0}. The location of the beginning of the block of words in memory is specified by the contents of the base register R*n*. The block runs toward higher addresses if the U bit is 1 and toward lower addresses if the U bit is 0. Pre- or post-indexing of R*n* is specified by the P bit. The index value is always 4 because the operands are consecutive 4-byte words. The final address generated in performing the block transfer is written back into R*n* if the W bit is 1; otherwise (W = 0), R*n* is left containing the initial address. Irrespective of whether or not the block runs toward higher or lower addresses, the lowest number register is always associated with the lowest address value in the block. Table B.5 shows the OP-code mnemonics for all possible settings of the P, U, and L bits. Suffixes on the LDM and STM OP codes indicate the settings of the P and U bits. For example, the P = 0 and U = 1 settings in the first entry in Table B.5 are indicated by the suffix IA, standing for "increment after," meaning that the contents of the base register R*n* are incremented by 4 after each transfer is performed, that is, R*n* is post-indexed. The alternate mnemonics and names shown in Table B.5 are explained next.

The main use for block transfers is in saving and restoring registers on a stack on entry to and return from subroutines. If we assume that R13 is used as the stack pointer and R14 (the link register) holds the return address, then the instruction

```
STMDB   R13!,{R0–R3,R14}
```

which is the last entry in Table B.5, pushes the contents of registers R0 through R3 and R14 onto the stack. The stack grows toward lower memory addresses and the contents of R0 are transferred last into the lowest address. The corresponding instruction

```
LDMIA   R13!,{R0–R3,R15}
```

which is the first entry in Table B.5, pops the saved contents of R0 through R3 back into those registers, and pops the saved value from R14 (the return address) into R15, the program counter, implementing the return operation. The contents of the highest address are transferred last into R15. The suffixes DB and IA on these two OP codes stand for "decrement before" (DB) and "increment after" (IA), describing how the base register contents are manipulated. The alternate OP-code mnemonics (shown in Table B.5) that can be used for the same instructions are STMFD and LDMFD. The suffix FD stands for "full descending." This is meant to describe the fact that the stack grows toward lower memory addresses (descending) and the initial contents of the base register, R13, are the address of the current top element of the stack (full). For a stack that grows toward higher addresses and uses a stack pointer that points to the empty location beyond the current top element, the descriptor name is "empty ascending," and the suffix is EA. The use of LDM and STM instructions in entering and returning from interrupt service routines is discussed in Chapter 4.

Table B.5 ARM instructions for multiple word transfers from/to memory

Mnemonic (Name)	Instruction bits P U L	Operation performed
LDMIA/LDMFD (Increment after/ Full descending)	0 1 1	$R_{low}, \ldots, R_{high} \leftarrow [[Rn]], [[Rn]+4], \ldots$
LDMIB/LDMED (Increment before/ Empty descending)	1 1 1	$R_{low}, \ldots, R_{high} \leftarrow [[Rn]+4], [[Rn]+8], \ldots$
LDMDA/LDMFA (Decrement after/ Full ascending)	0 0 1	$R_{high}, \ldots, R_{low} \leftarrow [[Rn]], [[Rn]-4], \ldots$
LDMDB/LDMEA (Decrement before/ Empty ascending)	1 0 1	$R_{high}, \ldots, R_{low} \leftarrow [[Rn]-4], [[Rn]-8], \ldots$
STMIA/STMEA (Increment after/ Empty ascending)	0 1 0	$[Rn], [Rn]+4, \ldots \leftarrow [R_{low}], \ldots, [R_{high}]$
STMIB/STMFA (Increment before/ Full ascending)	1 1 0	$[Rn]+4, [Rn]+8, \ldots \leftarrow [R_{low}], \ldots, [R_{high}]$
STMDA/STMED (Decrement after/ Empty descending)	0 0 0	$[Rn], [Rn]-4, \ldots \leftarrow [R_{high}], \ldots, [R_{low}]$
STMDB/STMFD (Decrement before/ Full descending)	1 0 0	$[Rn]-4, [Rn]-8, \ldots \leftarrow [R_{high}], \ldots, [R_{low}]$

31 28		23	0
Condition	1 0 1	K	Offset

K=0 :Branch (B)
K=1 :Branch with Link (BL): store return address in register R14

Figure B.7 ARM Branch and Branch with Link instructions.

B.1.4 BRANCH AND BRANCH WITH LINK INSTRUCTIONS

Figure B.7 shows the encoding of the Branch (B) and Branch with Link (BL) instructions. Offset is a signed 24-bit number. It is shifted left two bit positions (all branch targets are aligned word addresses), sign extended to 32 bits, and added to the updated PC to generate the branch target address. The updated PC points to the instruction that is two words (8 bytes) forward from the branch instruction.

The assembly language permits the absolute address of the branch target to be used. For example, the instruction

BEQ ROUTINE

is a conditional branch on the condition $Z = 1$ to the location ROUTINE. If the Branch instruction is at address 2000 and ROUTINE is the address 3000, the assembler will compute the offset value to be inserted into the instruction as 248. The actual distance to the target address from the updated contents, 2008, of the program counter is 992. This value is 248 shifted left 2 bit positions (multiplied by 4). Thus, the branch target address is computed as $2008 + 992 = 3000$.

The Branch with Link (BL) instruction is used to call a subroutine. Before branching to the subroutine, the address of the instruction that immediately follows the BL instruction (the return address) is stored in register R14, which is used as a link register. The return from a subroutine is handled as described in Section 3.6.

B.1.5 MACHINE CONTROL INSTRUCTIONS

Software Interrupt

When execution of a user program is completed, control is transferred to a supervisor program (part of the operating system) by a software interrupt instruction (SWI). We did not show the SWI instruction in the example programs in Part I of Chapter 3. An example of where it needs to be placed is immediately after the STR instruction in the program in Figure 3.8. An SWI instruction is also used in transferring control to operating system routines that run in supervisor mode while executing input/output operations for a user program, as described in Chapter 4.

The assembly language instruction format for the SWI instruction is shown in Table B.6, along with the operations performed. The OP code SWI is encoded by the

Table B.6 ARM instructions for status register transfers, software interrupt, and data swap

Mnemonic (Name)	Instruction formats	Operation performed
MRS (Copy status register)	User mode:	
	MRS R*d*,CPSR	R*d* ← [CPSR]
	Privileged mode:	
	MRS R*d*,CPSR	R*d* ← [CPSR]
	MRS R*d*,SPSR	R*d* ← [SPSR_*mode*]
MSR (Write to status register)	User mode:	
	MSR CPSR,R*m*	$CPSR_{31-28} \leftarrow [Rm]_{31-28}$
	MSR CPSR,imm32	$CPSR_{31-28} \leftarrow imm32_{31-28}$
	Privileged mode:	
	MSR CPSR,R*m*	CPSR ← [R*m*]
	MSR CPSR_flg,R*m*	$CPSR_{31-28} \leftarrow [Rm]_{31-28}$
	MSR CPSR_flg,imm32	$CPSR_{31-28} \leftarrow imm32_{31-28}$
	MSR SPSR,R*m*	SPSR_*mode* ← [R*m*]
	MSR SPSR_flg,R*m*	$SPSR_mode_{31-28} \leftarrow [Rm]_{31-28}$
	MSR SPSR_flg,imm32	$SPSR_mode_{31-28} \leftarrow imm32_{31-28}$
SWI (Software interrupt)	SWI imm24	R14_svc ← updated[PC]; SPSR_svc ← [CPSR]; PC ← 0x08
SWP (Swap)	SWP R*d*,R*m*,[R*n*]	R*d* ← [[R*n*]]; [R*n*] ← [R*m*]

pattern 1111 in the instruction bit field $b_{27\ldots24}$. As with all other ARM instructions, it can be executed conditionally. The low-order 24 bits of the instruction contain an immediate operand that is ignored during execution of the instruction. The user program can use this field to pass a parameter to the operating system to declare the service being requested, such as an I/O operation.

Processor Status Register Transfers

Instructions that handle the current processor status register, CPSR, and the saved processor status registers, SPSR_*mode* (see Figures 3.1 and 4.12) are provided, mainly for use by privileged mode programs. Limited use of these instructions in user programs is also permitted. When an external interrupt suspends execution of a user program, the current CPSR contents are automatically saved in an SPSR_*mode* register while a privileged mode routine handles the interrupt. Such routines have to manipulate status register contents. These issues are discussed in Chapter 4. The MRS and MSR instructions, shown in Table B.6, are used for reading and writing the contents of the CPSR and SPSR_*mode* registers. Both user mode and privileged mode programs can read the status registers. User mode programs can write only to the N, Z, V, and C condition code flags in bit field $b_{31\ldots28}$ of the CPSR register. Privileged mode programs can write to all 32 bits of the CPSR and SPSR_*mode* registers, and can also write selectively to just the condition code flag field. The source operand for write operations is either the contents of a general-purpose register or a 32-bit immediate value denoted imm32 in Table B.6. Only the high-order 4 bits of an immediate operand are used, so the operand can always be generated by a rotation of the short 8-bit immediate field value in an instruction.

The MRS and MSR instructions are encoded in the machine instruction format normally used for arithmetic and logic instructions (see Figure B.1). The CMP, CMN, TST, and TEQ instructions in this group, shown in Table B.3, always set the condition code flags. Therefore, the S bit is always set to 1 in these instructions. When the S bit is set to 0, the four OP codes for these instructions represent the MRS and MSR instructions operating on either the CPSR or the SPSR_*mode* registers. Other instruction bit positions are used to distinguish between full or partial writes and register or immediate source operands for the MSR instruction.

Register/Memory Swap

An instruction is provided that reads the contents of a memory location into one register and writes the contents of another register into the same memory location in an uninterruptible pair of operations. This Swap instruction (with the mnemonic SWP), shown in Table B.6, can be executed in either user or privileged modes. Its main use is in implementing operations on lock variables to coordinate correct operations on memory data that are shared between programs in multiprocessor configurations (see Chapter 12). Being "uninterruptible" means that no access to memory by a different processor is permitted between the read and write operations performed by the Swap instruction. The registers R*m* and R*d* can be the same, effecting an exchange operation between the register and the memory operand.

B.2 OTHER ARM INSTRUCTIONS

In this section, we briefly describe coprocessor instructions and instructions introduced in versions v4 and v5 of the architecture.

B.2.1 COPROCESSOR INSTRUCTIONS

Hardware units for executing operations not performed by the defined ARM instruction set are called coprocessors. An example is a hardware unit for executing operations on floating-point numbers. Other examples include application-specific processing on digital signals or video data that may be required in an embedded system. If an ARM processor design is provided in a software-synthesizable form, as described in Chapters 9 and 11, a software module that specifies a coprocessor can be integrated with the software description of the processor and used to generate a single-chip implementation. Programming of the combined units is facilitated by including instruction templates in the ARM instruction set for directing the coprocessor to perform its operations, transferring data between coprocessor registers and memory, and transferring data between coprocessor registers and ARM registers.

B.2.2 VERSIONS V4 AND V5 INSTRUCTIONS

An extended set of memory access and multiply instructions have been included in the two versions of the instruction set architecture that follow version v3. Versions v4 and v5 have Load and Store instructions that transfer signed bytes and signed/unsigned 16-bit half words between memory and the processor registers. Internally, ARM processors perform operations only on 32-bit operands. When signed bytes or signed half words are loaded into processor registers by v4 and v5 instructions, they are sign-extended to the 32-bit length.

Versions v4 and v5 also provide additional forms of the MUL and MULA instructions found in version v3 (see Figure B.4). Signed and unsigned versions of these two instructions that produce 64-bit products are provided.

B.3 PROGRAMMING EXPERIMENTS

The ARM web site URL: www.arm.com/hr.ns4/html/SDT202u contains software development tools that can be used to enter, edit, assemble, and run (simulated) ARM assembly language programs. For the program in Figure 3.8, the AREA directives should be changed to

```
AREA    addloop,    CODE
AREA    addloopdata,    DATA
```

to enable assembly. Also, as described in Section B.1.5, a software interrupt instruction in the form

```
SWI    0x123456
```

is needed after the SRT instruction.