ARM System Developer's Guide

Designing and Optimizing System Software

Andrew N. Sloss

Dominic Symes

Chris Wright

With a contribution by John Rayfield

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Contents

Chapter 4

Chapter 5

Chapter 6

Chapter 3

Introduction to the ARM Instruction Set

This introduction to the ARM instruction set is a fundamental chapter since the information presented here is used throughout the rest of the book. Consequently, it is placed here before we start going into any depth on optimization and efficient algorithms. This chapter introduces the most common and useful ARM instructions and builds on the ARM processor fundamentals covered in the last chapter. Chapter 4 introduces the Thumb instruction set, and Appendix A gives a complete description of all ARM instructions.

Different ARM architecture revisions support different instructions. However, new revisions usually add instructions and remain backwardly compatible. Code you write for architecture ARMv4T should execute on an ARMv5TE processor. Table 3.1 provides a complete list of ARM instructions available in the ARMv5E instruction set architecture (ISA). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set. The "ARM ISA" column lists the ISA revision in which the instruction was introduced. Some instructions have extended functionality in later architectures; for example, the CDP instruction has an ARMv5 variant called CDP2. Similarly, instructions such as LDR have ARMv5 additions but do not require a new or extended mnemonic.

We illustrate the processor operations using examples with pre- and post-conditions, describing registers and memory before and after the instruction or instructions are

Table 3.1 ARM instruction set.

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL*xy*	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW*y*	v5E	signed multiply accumulate instruction $(((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

continued

Table 3.1 ARM instruction set. (*Continued*)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULW*y*	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

executed. We will represent hexadecimal numbers with the prefix 0x and binary numbers with the prefix 0b. The examples follow this format:

```
PRE    <pre-conditions>
       <instruction/s>
POST   <post-conditions>
```

In the pre- and post-conditions, memory is denoted as

```
mem<data_size>[address]
```

This refers to *data_size* bits of memory starting at the given byte *address*. For example, *mem32*[1024] is the 32-bit value starting at address 1 KB.

ARM instructions process data held in registers and only access memory with load and store instructions. ARM instructions commonly take two or three operands. For instance the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

In the following sections we examine the function and syntax of the ARM instructions by instruction class—data processing instructions, branch instructions, load-store instructions, software interrupt instruction, and program status register instructions.

3.1 Data Processing Instructions

The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing instructions can process one of their operands using the barrel shifter.

If you use the `S` suffix on a data processing instruction, then it updates the flags in the `cpsr`. Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*. The carry flag is set from the result of the barrel shift as the last bit shifted out. The *N* flag is set to bit 31 of the result. The *Z* flag is set if the result is zero.

3.1.1 Move Instructions

Move is the simplest ARM instruction. It copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

`MOV`	Move a 32-bit value into a register	$Rd = N$
`MVN`	move the NOT of the 32-bit value into a register	$Rd = \sim N$

Table 3.3, to be presented in Section 3.1.2, gives a full description of the values allowed for the second operand *N* for all data processing instructions. Usually it is a register *Rm* or a constant preceded by #.

Example 3.1

This example shows a simple move instruction. The `MOV` instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value 5, and overwriting the value 8 in register *r7*.

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5     ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

3.1.2 Barrel Shifter

In Example 3.1 we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

Data processing instructions are processed within the arithmetic logic unit (ALU). A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

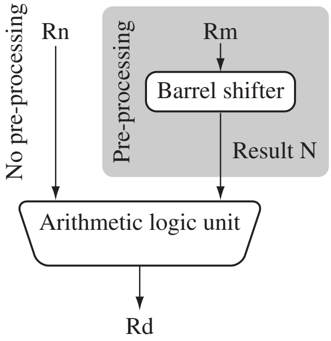


Figure 3.1 Barrel shifter and ALU.

To illustrate the barrel shifter we will take the example in Figure 3.1 and add a shift operation to the move instruction example. Register *Rn* enters the ALU without any pre-processing of registers. Figure 3.1 shows the data flow between the ALU and the barrel shifter.

Example 3.2 We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard C language shift operator ≪ to the register. The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in Table 3.2.

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5, LSL #2   ; let r7 = r5*4 = (r5<<2)

POST    r5 = 5
        r7 = 20
```

The example multiplies register *r5* by four and then places the result into register *r7*. ■

The five different shift operations that you can use within the barrel shifter are summarized in Table 3.2.

Figure 3.2 illustrates a logical shift left by one. For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared. The *C* flag is updated with the last bit shifted out of the register. This is bit $(32 - y)$ of the original value, where y is the shift amount. When y is greater than one, then a shift by y positions is the same as a shift by one position executed y times.

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(c flag $\ll$ 31) | ((unsigned)$x \gg 1$)	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

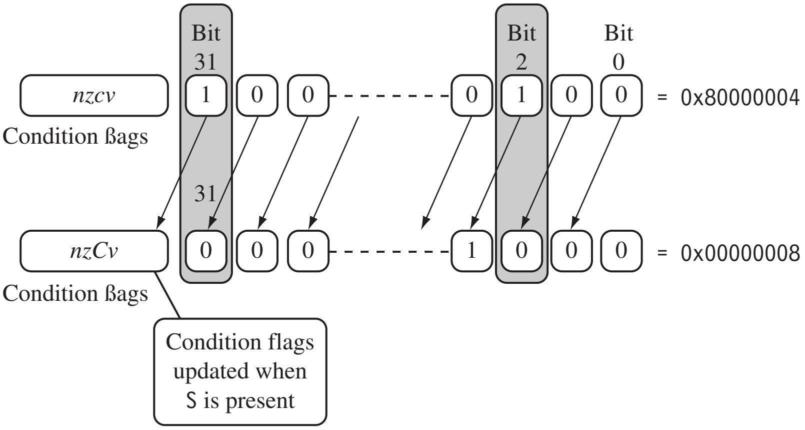


Figure 3.2 Logical shift left by one.

Table 3.3 Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

EXAMPLE 3.3

This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the *C* flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

         MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

Table 3.3 lists the syntax for the different barrel shift operations available on data processing instructions. The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

3.1.3 Arithmetic Instructions

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ `carry`
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$ `!(carry flag)`
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$ `!(carry flag)`
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

Example 3.4

This simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

Example 3.5

This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

Example 3.6

The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the *ZC* flags being set.

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

        SUBS r1, r1, #1
```

```
POST   cpsr = nZCvqiFt_USER
       r1 = 0x00000000
```

■

3.1.4 Using the Barrel Shifter with Arithmetic Instructions

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register *r1* by three.

Example 3.7

Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE    r0 = 0x00000000
       r1 = 0x00000005

       ADD    r0, r1, r1, LSL #1

POST   r0 = 0x0000000f
       r1 = 0x00000005
```

■

3.1.5 Logical Instructions

Logical instructions perform bitwise logical operations on the two source registers.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example 3.8

This example shows a logical OR operation between registers *r1* and *r2*. *r0* holds the result.

```
PRE    r0 = 0x00000000
       r1 = 0x02040608
       r2 = 0x10305070
```

```
        ORR   r0, r1, r2

POST    r0 = 0x12345678
```

EXAMPLE 3.9 This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
        Rd = Rn AND NOT(N)
```

In this example, register *r2* contains a binary pattern where every binary 1 in *r2* clears a corresponding bit location in register *r1*. This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

3.1.6 Comparison Instructions

The comparison instructions are used to compare or test a register with a 32-bit value. They update the *cpsr* flag bits according to the result, but do not affect other registers. After the bits have been set, the information can then be used to change program flow by using conditional execution. For more information on conditional execution take a look at Section 3.8. You do not need to apply the S suffix for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

EXAMPLE 3.10

This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*. This change indicates *equality*.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation. For each, the results are discarded but the condition bits are updated in the *cpsr*. It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

3.1.7 Multiply Instructions

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register. The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*. For more details on cycle timings, see Appendix D.

EXAMPLE 3.11 This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2   ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *RdLo* and *RdHi*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result. Example 3.12 shows an example of a long unsigned multiply instruction.

EXAMPLE 3.12 The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

3.2 Branch Instructions

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, *if-then-else* structures, and loops.

The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction. *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example 3.13

This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B     forward
        ADD   r1, r2, #4
        ADD   r0, r6, #2
        ADD   r3, r7, #4
forward
        SUB   r1, r2, #4
```

```
backward
        ADD   r1, r2, #4
        SUB   r1, r2, #4
        ADD   r4, r6, r7
        B     backward
```

Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels. In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset. ■

Example 3.14 The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call. This example shows a simple fragment of code that branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction. The BX instruction uses an absolute address stored in register *Rm*. It is primarily used to branch to and from Thumb code, as shown in Chapter 4. The *T* bit in the *cpsr* is updated by the least significant bit of the branch register. Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address. ■

3.3 Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

3.3.1 Single-Register Transfer

These instructions are used for moving a single data item in and out of a register. The datatypes supported are signed and unsigned words (32-bit), halfwords (16-bit), and bytes. Here are the various load-store single-register transfer instructions.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Tables 3.5 and 3.7, to be presented is Section 3.3.2, describe the *addressing*^{1} and *addressing*^{2} syntax.

Example 3.15

LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]          ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]          ; = STR r0, [r1, #0]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

3.3.2 Single-Register Load-Store Addressing Modes

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex (see Table 3.4).

Table 3.4 Index methods.

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

EXAMPLE 3.16 Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only updates the address base register after the address is used. The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

```
PRE      r0 = 0x00000000
         r1 = 0x00090000
         mem32[0x00009000] = 0x01010101
         mem32[0x00009004] = 0x02020202

         LDR    r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)  r0 = 0x02020202
         r1 = 0x00009004

         LDR    r0, [r1, #4]
```

Preindexing:

```
POST(2)  r0 = 0x02020202
         r1 = 0x00009000

         LDR    r0, [r1], #4
```

Postindexing:

```
POST(3)  r0 = 0x01010101
         r1 = 0x00009004
```

Table 3.5 Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

Example 3.15 used a preindex method. This example shows how each indexing method effects the address held in register *r1*, as well as the data loaded into register *r0*. Each instruction shows the result of the index method with the same pre-condition. ■

The addressing modes available with a particular load or store instruction depend on the instruction class. Table 3.5 shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

A signed offset or register is denoted by "+/−", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.

Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction. *Register* means the address is calculated using the base address register and a specific register's contents. *Scaled* means the address is calculated using the base address register and a barrel shift operation.

Table 3.6 provides an example of the different variations of the LDR instruction. Table 3.7 shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data.

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes. Table 3.8 shows the variations for STRH instructions.

3.3.3 Multiple-Register Transfer

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn* pointing into memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.

Table 3.6 Examples of LDR instructions using different addressing modes.

	Instruction	*r0* =	*r1* + =
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

Table 3.7 Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

Addressing^2 mode and index method	Addressing^2 syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

Table 3.8 Variations of STRH instructions.

	Instruction	Result	*r1* + =
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Load-store multiple instructions can increase interrupt latency. ARM implementations do not usually interrupt instructions while they are executing. For example, on an ARM7 a load multiple instruction takes $2 + Nt$ cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory. If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

Compilers, such as armcc, provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

`LDM`	load multiple registers	$\{Rd\}^{*N}$ <- mem32[start address + 4*N] optional Rn updated
`STM`	save multiple registers	$\{Rd\}^{*N}$ -> mem32[start address + 4*N] optional Rn updated

Table 3.9 shows the different addressing modes for the load-store multiple instructions. Here N is the number of registers in the list of registers.

Any subset of the current bank of registers can be transferred to memory or fetched from memory. The base register *Rn* determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similiar to the single-register load-store using preindex with writeback.

Table 3.9 Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
`IA`	increment after	Rn	$Rn + 4*N - 4$	$Rn + 4*N$
`IB`	increment before	$Rn + 4$	$Rn + 4*N$	$Rn + 4*N$
`DA`	decrement after	$Rn - 4*N + 4$	Rn	$Rn - 4*N$
`DB`	decrement before	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Example 3.17

In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

        LDMIA   r0!, {r1-r3}

POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```

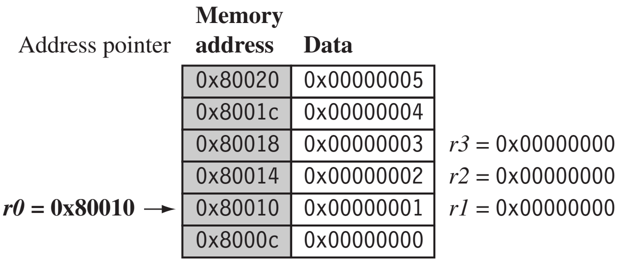
Figure 3.3 shows a graphical representation.

The base register *r0* points to memory address 0x80010 in the PRE condition. Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively. After the load multiple instruction executes registers *r1*, *r2*, and *r3* contain these values as shown in Figure 3.4. The base register *r0* now points to memory address 0x8001c after the last loaded word.

Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions. The first word pointed to by register *r0* is ignored and register *r1* is loaded from the next memory location as shown in Figure 3.5.

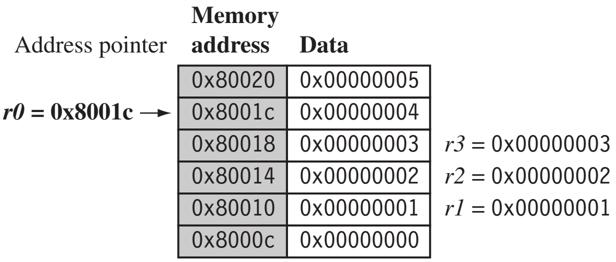
After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location. ■

The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations. This is equivalent to descending memory but accessing the register list in reverse order. With the increment and decrement load multiples, you can access arrays forwards or backwards. They also allow for stack push and pull operations, illustrated later in this section.



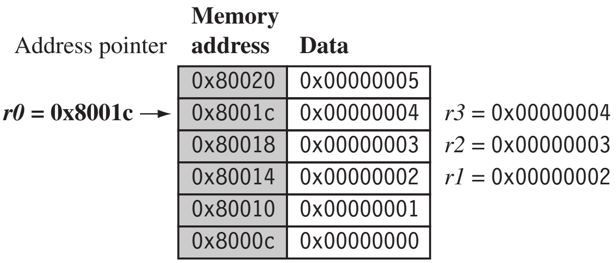
Address pointer	**Memory address**	**Data**	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

Figure 3.3 Pre-condition for LDMIA instruction.



Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Figure 3.4 Post-condition for LDMIA instruction.



Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	*r3* = 0x00000004
	0x80018	0x00000003	*r2* = 0x00000003
	0x80014	0x00000002	*r1* = 0x00000002
	0x80010	0x00000001	
	0x8000c	0x00000000	

Figure 3.5 Post-condition for LDMIB instruction.

Table 3.10 Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Table 3.10 shows a list of load-store multiple instruction pairs. If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is useful when you need to temporarily save a group of registers and restore them later.

Example 3.18

This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

```
PRE      r0 = 0x00009000
         r1 = 0x00000009
         r2 = 0x00000008
         r3 = 0x00000007

         STMIB   r0!, {r1-r3}

         MOV    r1, #1
         MOV    r2, #2
         MOV    r3, #3

PRE(2)   r0 = 0x0000900c
         r1 = 0x00000001
         r2 = 0x00000002
         r3 = 0x00000003

         LDMDA r0!, {r1-r3}

POST     r0 = 0x00009000
         r1 = 0x00000009
         r2 = 0x00000008
         r3 = 0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example 3.19

We illustrate the use of the load-store multiple instructions with a *block memory copy* example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-store multiple instructions, which use the same *increment after* addressing mode.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source

loop
        ; load 32 bytes from source and update r9 pointer
        LDMIA   r9!, {r0-r7}
```

```
; store 32 bytes to destination and update r10 pointer
STMIA   r10!, {r0-r7} ; and store them

; have we reached the end
CMP    r9, r11
BNE    loop
```

This routine relies on registers *r9*, *r10*, and *r11* being set up before the code is executed. Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data. LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9* to point to the next block of data to be copied. STMIA copies the contents of registers *r0* to *r7* to the destination memory address pointed to by register *r10*. It also updates *r10* to point to the next destination location. CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached. If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.

The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to *not equal*, the branch instruction is executed.

Figure 3.6 shows the memory map of the block memory copy and how the routine moves through memory. Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory. ■

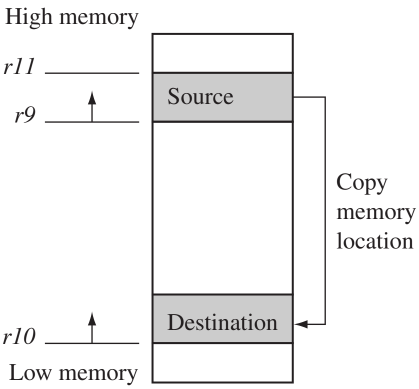


Figure 3.6 Block memory copy in the memory map.

3.3.3.1 Stack Operations

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The *pop* operation (removing data from a stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction.

When using a stack you have to decide whether the stack will grow up or down in memory. A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.

When you use a *full stack* (F), the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack). In contrast, if you use an *empty stack* (E) the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

There are a number of load-store multiple addressing mode aliases available to support stack operations (see Table 3.11). Next to the *pop* column is the actual load multiple instruction equivalent. For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.

ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the pop and push functions, respectively.

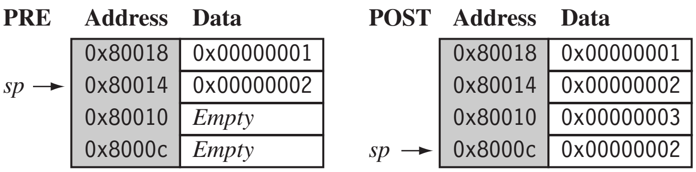
EXAMPLE 3.20 The STMFD instruction pushes registers onto the stack, updating the *sp*. Figure 3.7 shows a push onto a full descending stack. You can see that when the stack grows the stack pointer points to the last full entry in the stack.

```
PRE    r1 = 0x00000002
       r4 = 0x00000003
       sp = 0x00080014

       STMFD   sp!, {r1,r4}
```

Table 3.11 Addressing methods for stack operations.

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA



PRE	Address	Data	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
sp →	0x80014	0x00000002		0x80014	0x00000002
	0x80010	*Empty*		0x80010	0x00000003
	0x8000c	*Empty*	*sp* →	0x8000c	0x00000002

Figure 3.7 STMFD instruction—full stack push operation.

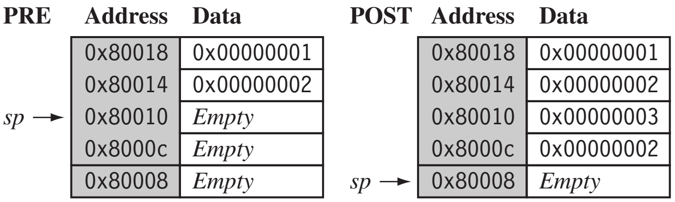
```
POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x0008000c
```

EXAMPLE 3.21 In contrast, Figure 3.8 shows a push operation on an empty stack using the STMED instruction. The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1,r4}

POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080008
```



PRE	Address	Data	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
sp →	0x80010	*Empty*		0x80010	0x00000003
	0x8000c	*Empty*		0x8000c	0x00000002
	0x80008	*Empty*	*sp* →	0x80008	*Empty*

Figure 3.8 STMED instruction—empty stack push operation.

When handling a checked stack there are three attributes that need to be preserved: the *stack base*, the *stack pointer*, and the *stack limit*. The stack base is the starting address of the stack in memory. The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a *stack overflow error* has occurred. Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow

SUB sp, sp, #size
CMP sp, r10
BLLO _stack_overflow ; condition
```

ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled. The BLLO instruction is a branch with link instruction plus the condition mnemonic LO. If *sp* is less than register *r10* after the new items are pushed onto the stack, then stack overflow error has occurred. If the stack pointer goes back past the stack base, then a *stack underflow error* has occurred.

3.3.4 Swap Instruction

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Swap cannot be interrupted by any other instruction or any other bus access. We say the system “holds the bus” until the transaction is complete.

Example 3.22 The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

```
PRE     mem32[0x9000] = 0x12345678
         r0 = 0x00000000
         r1 = 0x11112222
         r2 = 0x00009000

         SWP    r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
         r0 = 0x12345678
         r1 = 0x11112222
         r2 = 0x00009000
```

This instruction is particularly useful when implementing semaphores and mutual exclusion in an operating system. You can see from the syntax that this instruction can also have a byte size qualifier B, so this instruction allows for both a word and a byte swap.

EXAMPLE 3.23

This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
spin
        MOV     r1, =semaphore
        MOV     r2, #1
        SWP     r3, r2, [r1] ; hold the bus until complete
        CMP     r3, #1
        BEQ     spin
```

The address pointed to by the semaphore either contains the value 0 or 1. When the semaphore equals 1, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value 0.

3.4 Software Interrupt Instruction

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	$lr\_svc =$ address of instruction following the SWI $spsr\_svc = cpsr$ $pc =$ vectors + 0x8 $cpsr$ mode = SVC $cpsr\ I = 1$ (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table. The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

EXAMPLE 3.24 Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff; lr = r14
        r0 = 0x12

     0x00008000   SWI     0x123456

POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter 0x12. The return values are also passed back via registers.

Code called the *SWI handler* is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

```
SWI_Number = <SWI instruction> AND NOT(0xff000000)
```

Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

EXAMPLE 3.25 This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*. You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
        ;
        ; Store registers r0-r12 and the link register
```

```
;
STMFD   sp!, {r0-r12, lr}

; Read the SWI instruction
LDR       r10, [lr, #-4]

; Mask off top 8 bits
BIC       r10, r10, #0xff000000

; r10 - contains the SWI number
BL        service_routine

; return from SWI handler
LDMFD   sp!, {r0-r12, pc}^
```

The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

3.5 Program Status Register Instructions

The ARM instruction set provides two instructions to directly control a program status register (*psr*). The MRS instruction transfers the contents of either the *cpsr* or *spsr* into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the *cpsr* or *spsr*. Together these instructions are used to read and write the *cpsr* and *spsr*.

In the syntax you can see a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*). These fields relate to particular byte regions in a *psr*, as shown in Figure 3.9.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

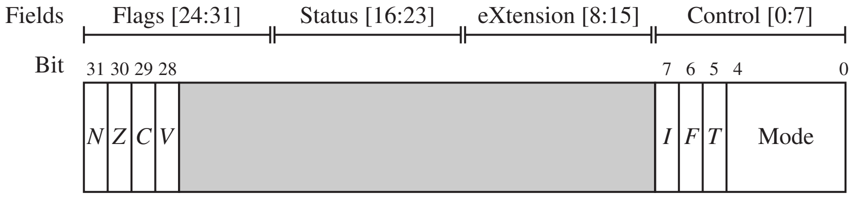


Figure 3.9 *psr* byte fields.

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

The *c* field controls the interrupt masks, Thumb state, and processor mode. Example 3.26 shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

EXAMPLE 3.26 The MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit 7 of *r1*. Register *r1* is then copied back into the *cpsr*, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

```
PRE    cpsr = nzcvqIFt_SVC

        MRS    r1, cpsr
        BIC    r1, r1, #0x80 ; 0b01000000
        MSR    cpsr_c, r1

POST   cpsr = nzcvqiFt_SVC
```

This example is in *SVC* mode. In *user* mode you can read all *cpsr* bits, but you can only update the condition flag field *f*. ■

3.5.1 Coprocessor Instructions

Coprocessor instructions are used to extend the instruction set. A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management. The coprocessor instructions include data processing, register transfer, and memory transfer instructions. We will provide only a short overview since these instructions are coprocessor specific. Note that these instructions are only used by cores with a coprocessor.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

In the syntax of the coprocessor instructions, the *cp* field represents the coprocessor number between *p0* and *p15*. The *opcode* fields describe the operation to take place on the coprocessor. The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor. The coprocessor operations and registers depend on the specific coprocessor you are using. Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

EXAMPLE 3.27

This example shows a CP15 register being copied into a general-purpose register.

```
; transferring the contents of CP15 register c0 to register r10
        MRC p15, 0, r10, c0, c0, 0
```

Here CP15 *register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*. ■

3.5.2 Coprocessor 15 Instruction Syntax

CP15 configures the processor core and has a set of dedicated registers to store configuration information, as shown in Example 3.27. A value written into a register sets a configuration attribute—for example, switching on the cache.

CP15 is called the *system control coprocessor*. Both MRC and MCR instructions are used to read and write to CP15, where register *Rd* is the core destination register, *Cn* is the primary register, *Cm* is the secondary register, and *opcode2* is a secondary register modifier. You may occasionally hear secondary registers called "extended registers."

As an example, here is the instruction to move the contents of CP15 control register *c1* into register *r1* of the processor core:

```
        MRC p15, 0, r1, c1, c0, 0
```

We use a shorthand notation for CP15 reference that makes referring to configuration registers easier to follow. The reference notation uses the following format:

CP15:cX:cY:Z

The first term, *CP15*, defines it as coprocessor 15. The second term, after the separating colon, is the primary register. The primary register *X* can have a value between 0 and 15. The third term is the secondary or extended register. The secondary register *Y* can have a value between 0 and 15. The last term, *opcode2*, is an instruction modifier and can have a value between 0 and 7. Some operations may also use a nonzero value *w* of *opcode1*. We write these as *CP15:w:cX:cY:Z*.

3.6 Loading Constants

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.

To aid programming there are two pseudoinstructions to move a 32-bit value into a register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

The first pseudoinstruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.

The second pseudoinstruction writes a relative address into a register, which will be encoded using a *pc*-relative expression.

Example 3.28

This example shows an LDR instruction loading a 32-bit constant 0xff00ffff into register *r0*.

```
        LDR     r0, [pc, #constant_number-8-{PC}]
        :
constant_number
        DCD     0xff00ffff
```

This example involves a memory access to load the constant, which can be expensive for time-critical routines. ■

Example 3.29 shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Table 3.12 LDR pseudoinstruction conversion.

Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

Example 3.29

Loading the constant 0xff00ffff using an MVN.

```
PRE     none...

        MVN     r0, #0x00ff0000

POST    r0 = 0xff00ffff
```

As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load. Compilers and assemblers use clever techniques to avoid loading a constant from memory. These tools have algorithms to find the optimal number of instructions required to generate a constant in a register and make extensive use of the barrel shifter. If the tools cannot generate the constant by these methods, then it is loaded from memory. The LDR pseudoinstruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a *literal pool*—a data area embedded within the code.

Table 3.12 shows two pseudocode conversions. The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load. We recommended that you use this pseudoinstruction to load a constant. To see how the assembler has handled a particular load constant, you can pass the output through a disassembler, which will list the instruction chosen by the tool to load the constant.

Another useful pseudoinstruction is the ADR instruction, or *address relative*. This instruction places the address of the given *label* into register *Rd*, using a *pc*-relative add or subtract.

3.7 ARMv5E Extensions

The ARMv5E extensions provide many new instructions (see Table 3.13). One of the most important additions is the signed multiply accumulate instructions that operate on 16-bit data. These operations are single cycle on many ARMv5E implementations.

ARMv5E provides greater flexibility and efficiency when manipulating 16-bit values, which is important for applications such as 16-bit digital audio processing.

Table 3.13 New instructions provided by the ARMv5E extensions.

Instruction	Description
`CLZ {<cond>} Rd, Rm`	count leading zeros
`QADD {<cond>} Rd, Rm, Rn`	signed saturated 32-bit add
`QDADD{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit add
`QDSUB{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit subtract
`QSUB{<cond>} Rd, Rm, Rn`	signed saturated 32-bit subtract
`SMLAxy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (1)
`SMLALxy{<cond>} RdLo, RdHi, Rm, Rs`	signed multiply accumulate 64-bit
`SMLAWy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (2)
`SMULxy{<cond>} Rd, Rm, Rs`	signed multiply (1)
`SMULWy{<cond>} Rd, Rm, Rs`	signed multiply (2)

3.7.1 Count Leading Zeros Instruction

The count leading zeros instruction counts the number of zeros between the most significant bit and the first bit set to 1. Example 3.30 shows an example of a CLZ instruction.

Example 3.30

You can see from this example that the first bit set to 1 has 27 zeros preceding it. CLZ is useful in routines that have to normalize numbers.

```
PRE     r1 = 0b00000000000000000000000000010000

        CLZ r0, r1

POST    r0 = 27
```

3.7.2 Saturated Arithmetic

Normal ARM arithmetic instructions wrap around when you overflow an integer value. For example, `0x7fffffff + 1 = -0x80000000`. Thus, when you design an algorithm, you have to be careful not to exceed the maximum representable value in a 32-bit integer.

Example 3.31

This example shows what happens when the maximum value is exceeded.

```
PRE     cpsr = nzcvqiFt_SVC
         r0 = 0x00000000
         r1 = 0x70000000 (positive)
         r2 = 0x7fffffff (positive)
```

```
        ADDS    r0, r1, r2

POST    cpsr = NzcVqiFt_SVC
        r0 = 0xefffffff (negative)
```

In the example, registers *r1* and *r2* contain positive numbers. Register *r2* is equal to 0x7fffffff, which is the maximum positive value you can store in 32 bits. In a perfect world adding these numbers together would result in a large positive number. Instead the value becomes negative and the overflow flag, *V*, is set. ■

In contrast, using the ARMv5E instructions you can *saturate* the result—once the highest number is exceeded the results remain at the maximum value of 0x7fffffff. This avoids the requirement for any additional code to check for possible overflows. Table 3.14 lists all the ARMv5E saturation instructions.

Table 3.14 Saturation instructions.

Instruction	Saturated calculation
QADD	$Rd = Rn + Rm$
QDADD	$Rd = Rn + (Rm*2)$
QSUB	$Rd = Rn - Rm$
QDSUB	$Rd = Rn - (Rm*2)$

Example 3.32 This example shows the same data being passed into the QADD instruction.

```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        QADD    r0, r1, r2

POST    cpsr = nzcvQiFt_SVC
        r0 = 0x7fffffff
```

You will notice that the saturated number is returned in register *r0*. Also the *Q* bit (bit 27 of the *cpsr*) has been set, indicating saturation has occurred. The *Q* flag is sticky and will remain set until explicitly cleared. ■

3.7.3 ARMv5E Multiply Instructions

Table 3.15 shows a complete list of the ARMv5E multiply instructions. In the table, x and y select which 16 bits of a 32-bit register are used for the first and second

Table 3.15 Signed multiply and multiply accumulate instructions.

Instruction	Signed Multiply [Accumulate]	Signed result	Q flag updated	Calculation
SMLAxy	(16-bit *16-bit)+ 32-bit	32-bit	yes	$Rd = (Rm.x * Rs.y) + Rn$
SMLALxy	(16-bit *16-bit)+ 64-bit	64-bit	—	$[RdHi, RdLo] += Rm.x * Rs.y$
SMLAWy	((32-bit *16-bit) ≫ 16)+ 32-bit	32-bit	yes	$Rd = ((Rm * Rs.y) \gg 16) + Rn$
SMULxy	(16-bit *16-bit)	32-bit	—	$Rd = Rm.x * Rs.y$
SMULWy	((32-bit *16-bit)≫ 16)	32-bit	—	$Rd = (Rm * Rs.y) \gg 16$

operands, respectively. These fields are set to a letter T for the top 16-bits, or the letter B for the bottom 16 bits. For multiply accumulate operations with a 32-bit result, the *Q* flag indicates if the accumulate overflowed a signed 32-bit value.

EXAMPLE 3.33 This example shows how you use these operations. The example uses a signed multiply accumulate instruction, SMLATB.

```
PRE     r1 = 0x20000001
        r2 = 0x20000001
        r3 = 0x00000004

        SMLATB r4, r1, r2, r3

POST    r4 = 0x00002004
```

The instruction multiplies the top 16 bits of register *r1* by the bottom 16 bits of register *r2*. It adds the result to register *r3* and writes it to destination register *r4*. ■

3.8 Conditional Execution

Most ARM instructions are *conditionally executed*—you can specify that the instruction only executes if the condition code flags pass a given condition or test. By using conditional execution instructions you can increase performance and code density.

The condition field is a two-letter mnemonic appended to the instruction mnemonic. The default mnemonic is AL, or *always execute*.

Conditional execution reduces the number of branches, which also reduces the number of pipeline flushes and thus improves the performance of the executed code. Conditional execution depends upon two components: the condition field and condition flags. The condition field is located in the instruction, and the condition flags are located in the *cpsr*.

EXAMPLE 3.34 This example shows an ADD instruction with the EQ condition appended. This instruction will only be executed when the zero flag in the *cpsr* is set to 1.

```
        ; r0 = r1 + r2 if zero flag is set
        ADDEQ r0, r1, r2
```

Only comparison instructions and data processing instructions with the S suffix appended to the mnemonic update the condition flags in the *cpsr*. ■

EXAMPLE 3.35 To help illustrate the advantage of conditional execution, we will take the simple C code fragment shown in this example and compare the assembler output using nonconditional and conditional instructions.

```
while (a!=b)
{
  if (a>b) a -= b; else b -= a;
}
```

Let register *r1* represent *a* and register *r2* represent *b*. The following code fragment shows the same algorithm written in ARM assembler. This example only uses conditional execution on the branch instructions:

```
        ; Greatest Common Divisor Algorithm
gcd
        CMP    r1, r2
        BEQ    complete
        BLT    lessthan
        SUB    r1, r1, r2
        B      gcd

lessthan
        SUB    r2, r2, r1
        B      gcd

complete
...
```

Now compare the same code with full conditional execution. As you can see, this dramatically reduces the number of instructions:

```
gcd
        CMP     r1, r2
```

```
SUBGT   r1, r1, r2
SUBLT   r2, r2, r1
BNE     gcd
```

3.9 Summary

In this chapter we covered the ARM instruction set. All ARM instructions are 32 bits in length. The arithmetic, logical, comparisons, and move instructions can all use the inline barrel shifter, which pre-processes the second register *Rm* before it enters into the ALU.

The ARM instruction set has three types of load-store instructions: single-register load-store, multiple-register load-store, and swap. The multiple load-store instructions provide the push-pop operations on the stack. The ARM-Thumb Procedure Call Standard (ATPCS) defines the stack as being a full descending stack.

The software interrupt instruction causes a software interrupt that forces the processor into *SVC* mode; this instruction invokes privileged operating system routines. The program status register instructions write and read to the *cpsr* and *spsr*. There are also special pseudoinstructions that optimize the loading of 32-bit constants.

The ARMv5E extensions include count leading zeros, saturation, and improved multiply instructions. The count leading zeros instruction counts the number of binary zeros before the first binary one. Saturation handles arithmetic calculations that overflow a 32-bit integer value. The improved multiply instructions provide better flexibility in multiplying 16-bit values.

Most ARM instructions can be conditionally executed, which can dramatically reduce the number of instructions required to perform a specific algorithm.

This Page Intentionally Left Blank

Chapter 6

Writing and Optimizing ARM Assembly Code

Embedded software projects often contain a few key subroutines that dominate system performance. By optimizing these routines you can reduce the system power consumption and reduce the clock speed needed for real-time operation. Optimization can turn an infeasible system into a feasible one, or an uncompetitive system into a competitive one.

If you write your C code carefully using the rules given in Chapter 5, you will have a relatively efficient implementation. For maximum performance, you can optimize critical routines using hand-written assembly. Writing assembly by hand gives you direct control of three optimization tools that you cannot explicitly use by writing C source:

- *Instruction scheduling:* Reordering the instructions in a code sequence to avoid processor stalls. Since ARM implementations are pipelined, the timing of an instruction can be affected by neighboring instructions. We will look at this in Section 6.3.
- *Register allocation:* Deciding how variables should be allocated to ARM registers or stack locations for maximum performance. Our goal is to minimize the number of memory accesses. See Section 6.4.
- *Conditional execution:* Accessing the full range of ARM condition codes and conditional instructions. See Section 6.5.

It takes additional effort to optimize assembly routines so don't bother to optimize noncritical ones. When you take the time to optimize a routine, it has the side benefit of giving you a better understanding of the algorithm, its bottlenecks, and dataflow.

Section 6.1 starts with an introduction to assembly programming on the ARM. It shows you how to replace a C function by an assembly function that you can then optimize for performance.

We describe common optimization techniques, specific to writing ARM assembly. Thumb assembly is not covered specifically since ARM assembly will always give better performance when a 32-bit bus is available. Thumb is most useful for reducing the compiled size of C code that is not critical to performance and for efficient execution on a 16-bit data bus. Many of the principles covered here apply equally well to Thumb and ARM.

The best optimization of a routine can vary according to the ARM core used in your target hardware, especially for signal processing (covered in detail in Chapter 8). However, you can often code a routine that is reasonably efficient for all ARM implementations. To be consistent this chapter uses ARM9TDMI optimizations and cycle counts in the examples. However, the examples will run efficiently on all ARM cores from ARM7TDMI to ARM10E.

6.1 Writing Assembly Code

This section gives examples showing how to write basic assembly code. We assume you are familiar with the ARM instructions covered in Chapter 3; a complete instruction reference is available in Appendix A. We also assume that you are familiar with the ARM and Thumb procedure call standard covered in Section 5.4.

As with the rest of the book, this chapter uses the ARM macro assembler *armasm* for examples (see Section A.4 in Appendix A for *armasm* syntax and reference). You can also use the GNU assembler *gas* (see Section A.5 for details of the GNU assembler syntax).

Example 6.1 This example shows how to convert a C function to an assembly function—usually the first stage of assembly optimization. Consider the simple C program `main.c` following that prints the squares of the integers from 0 to 9:

```
#include <stdio.h>

int square(int i);

int main(void)
{
  int i;

  for (i=0; i<10; i++)
  {
    printf("Square of %d is %d\n", i, square(i));
  }
}

int square(int i)
```

```
{
  return i*i;
}
```

Let's see how to replace `square` by an assembly function that performs the same action. Remove the C definition of `square`, but not the declaration (the second line) to produce a new C file `main1.c`. Next add an *armasm* assembler file `square.s` with the following contents:

```
        AREA     |.text|, CODE, READONLY

        EXPORT   square

        ; int square(int i)
square
        MUL   r1, r0, r0   ; r1 = r0 * r0
        MOV   r0, r1       ; r0 = r1
        MOV   pc, lr       ; return r0
        END
```

The `AREA` directive names the area or code section that the code lives in. If you use nonalphanumeric characters in a symbol or area name, then enclose the name in vertical bars. Many nonalphanumeric characters have special meanings otherwise. In the previous code we define a read-only code area called `.text`.

The `EXPORT` directive makes the symbol `square` available for external linking. At line six we define the symbol `square` as a code label. Note that *armasm* treats nonindented text as a label definition.

When `square` is called, the parameter passing is defined by the ATPCS (see Section 5.4). The input argument is passed in register *r0*, and the return value is returned in register *r0*. The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into *r1* and move this to *r0*.

The `END` directive marks the end of the assembly file. Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

Example 6.1 only works if you are compiling your C as ARM code. If you compile your C as Thumb code, then the assembly routine must return using a `BX` instruction.

EXAMPLE 6.2 When calling ARM code from C compiled as Thumb, the only change required to the assembly in Example 6.1 is to change the return instruction to a `BX`. `BX` will return to ARM or Thumb state according to bit 0 of *lr*. Therefore this routine can be called from ARM or Thumb. Use `BX lr` instead of `MOV pc,lr` whenever your processor supports `BX` (ARMv4T and above). Create a new assembly file `square2.s` as follows:

```
        AREA     |.text|, CODE, READONLY

        EXPORT   square

        ; int square(int i)
square
        MUL   r1, r0, r0   ; r1 = r0 * r0
        MOV   r0, r1       ; r0 = r1
        BX    lr           ; return r0

        END
```

With this example we build the C file using the Thumb C compiler *tcc*. We assemble the assembly file with the interworking flag enabled so that the linker will allow the Thumb C code to call the ARM assembly code. You can use the following commands to build this example:

```
tcc -c main1.c
armasm -apcs /interwork square2.s
armlink -o main2.axf main1.o square2.o
```

■

Example 6.3 This example shows how to call a subroutine from an assembly routine. We will take Example 6.1 and convert the whole program (including `main`) into assembly. We will call the C library routine `printf` as a subroutine. Create a new assembly file `main3.s` with the following contents:

```
        AREA     |.text|, CODE, READONLY

        EXPORT   main

        IMPORT   |Lib$$Request$$armlib|, WEAK
        IMPORT   __main    ; C library entry
        IMPORT   printf    ; prints to stdout

i       RN 4

        ; int main(void)

main
        STMFD   sp!, {i, lr}
        MOV     i, #0
```

```
loop
        ADR     r0, print_string
        MOV     r1, i
        MUL     r2, i, i
        BL      printf
        ADD     i, i, #1
        CMP     i, #10
        BLT     loop
        LDMFD   sp!, {i, pc}

print_string
        DCB      "Square of %d is %d\n", 0

        END
```

We have used a new directive, IMPORT, to declare symbols that are defined in other files. The imported symbol Lib$$Request$$armlib makes a request that the linker links with the standard ARM C library. The WEAK specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero. The second imported symbol __main is the start of the C library initialization code. You only need to import these symbols if you are defining your own main; a main defined in C code will import these automatically for you. Importing printf allows us to call that C library function.

The RN directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*. Using register names makes the code more readable. It is also easier to change the allocation of variables to registers at a later date.

Recall that ATPCS states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i*(*r4*), and calling printf will corrupt *lr*. Therefore we stack these two registers at the start of the function using an STMFD instruction. The LDMFD instruction pulls these registers from the stack and returns by writing the return address to *pc*.

The DCB directive defines byte data described as a string or a comma-separated list of bytes.

To build this example you can use the following command line script:

```
armasm main3.s
armlink -o main3.axf main3.o
```

Note that Example 6.3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 6.2 then we must be capable of returning to Thumb code. For architectures before ARMv5 we must use a BX to return. Change the last instruction to the two instructions:

```
        LDMFD   sp!, {i, lr}
        BX      lr
```

Finally, let's look at an example where we pass more than four parameters. Recall that ATPCS places the first four arguments in registers *r0* to *r3*. Subsequent arguments are placed on the stack.

EXAMPLE 6.4

This example defines a function `sumof` that can sum any number of integers. The arguments are the number of integers to sum followed by a list of the integers. The `sumof` function is written in assembly and can accept any number of arguments. Put the C part of the example in a file `main4.c`:

```
#include <stdio.h>

/* N is the number of values to sum in list ... */
int sumof(int N, ...);

int main(void)
{
   printf("Empty sum=%d\n", sumof(0));
   printf("1=%d\n", sumof(1,1));
   printf("1+2=%d\n", sumof(2,1,2));
   printf("1+2+3=%d\n", sumof(3,1,2,3));
   printf("1+2+3+4=%d\n", sumof(4,1,2,3,4));
   printf("1+2+3+4+5=%d\n", sumof(5,1,2,3,4,5));
   printf("1+2+3+4+5+6=%d\n", sumof(6,1,2,3,4,5,6));
}
```

Next define the `sumof` function in an assembly file `sumof.s`:

```
        AREA    |.text|, CODE, READONLY

        EXPORT  sumof

N       RN 0    ; number of elements to sum
sum     RN 1    ; current sum

        ; int sumof(int N, ...)
sumof
        SUBS    N, N, #1        ; do we have one element
        MOVLT   sum, #0         ; no elements to sum!
        SUBS    N, N, #1        ; do we have two elements
        ADDGE   sum, sum, r2
        SUBS    N, N, #1        ; do we have three elements
        ADDGE   sum, sum, r3
        MOV     r2, sp          ; top of stack
loop
        SUBS    N, N, #1        ; do we have another element
        LDMGEFD r2!, {r3}       ; load from the stack
        ADDGE   sum, sum, r3
        BGE     loop
        MOV     r0, sum
        MOV     pc, lr          ; return r0

        END
```

The code keeps count of the number of remaining values to sum, N. The first three values are in registers *r1*, *r2*, *r3*. The remaining values are on the stack. You can build this example using the commands

```
armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o
```

■

6.2 Profiling and Cycle Counting

The first stage of any optimization process is to identify the critical routines and measure their current performance. A *profiler* is a tool that measures the proportion of time or processing cycles spent in each subroutine. You use a profiler to identify the most critical routines. A *cycle counter* measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.

The ARM simulator used by the ADS1.1 debugger is called the ARMulator and provides profiling and cycle counting features. The ARMulator profiler works by sampling the program counter *pc* at regular intervals. The profiler identifies the function the *pc* points to and updates a hit counter for each function it encounters. Another approach is to use the trace output of a simulator as a source for analysis.

Be sure that you know how the profiler you are using works and the limits of its accuracy. A *pc*-sampled profiler can produce meaningless results if it records too few samples. You can even implement your own *pc*-sampled profiler in a hardware system using timer interrupts to collect the *pc* data points. Note that the timing interrupts will slow down the system you are trying to measure!

ARM implementations do not normally contain cycle-counting hardware, so to easily measure cycle counts you should use an ARM debugger with ARM simulator. You can configure the ARMulator to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.

6.3 Instruction Scheduling

The time taken to execute instructions depends on the implementation pipeline. For this chapter, we assume ARM9TDMI pipeline timings. You can find these in Section D.3 of

Appendix D. The following rules summarize the cycle timings for common instruction classes on the ARM9TDMI.

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ALU operations such as addition, subtraction, and logical operations take one cycle. This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.
- Load instructions that load *N* 32-bit words of memory such as `LDR` and `LDM` take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system. An `LDM` of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
- Load instructions that load 16-bit or 8-bit data such as `LDRB, LDRSB, LDRH, and LDRSH` take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system, or a cache hit for a cached system.
- Branch instructions take three cycles.
- Store instructions that store *N* values take *N* cycles. This assumes zero-wait-state memory for an uncached system, or a cache hit or a write buffer with *N* free entries for a cached system. An `STM` of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product (see Table D.6 in Section D.3).

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The ARM9TDMI processor performs five operations in parallel:

- *Fetch:* Fetch from memory the instruction at address *pc*. The instruction is loaded into the core and then processes down the core pipeline.
- *Decode:* Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- *ALU:* Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address $pc - 8$ (ARM state) or $pc - 4$ (Thumb state). Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation. Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.

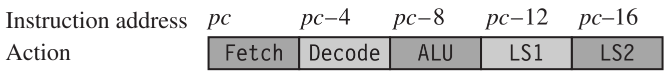


Figure 6.1 ARM9TDMI pipeline executing in ARM state.

- *LS1:* Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- *LS2:* Extract and zero- or sign-extend the data loaded by a byte or halfword load instruction. If the instruction is not a load of an 8-bit byte or 16-bit halfword item, then this stage has no effect.

Figure 6.1 shows a simplified functional view of the five-stage ARM9TDMI pipeline. Note that multiply and register shift operations are not shown in the figure.

After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that *pc* points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address *pc* − 8 in parallel with fetching the instruction at address *pc*.

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline. To work out how many cycles a block of code will take, use the tables in Appendix D that summarize the cycle timings and interlock cycles for a range of ARM cores.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a pipeline *hazard* or pipeline *interlock*.

Example 6.5

This example shows the case where there is no interlock.

```
ADD   r0, r0, r1
ADD   r0, r0, r2
```

This instruction pair takes two cycles. The ALU calculates *r0* + *r1* in one cycle. Therefore this result is available for the ALU to calculate *r0* + *r2* in the second cycle.

Example 6.6

This example shows a one-cycle interlock caused by load use.

```
LDR   r1, [r2, #4]
ADD   r0, r0, r1
```

This instruction pair takes three cycles. The ALU calculates the address *r2* + 4 in the first cycle while decoding the ADD instruction in parallel. However, the ADD cannot proceed on

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

Figure 6.2 One-cycle interlock caused by load use.

the second cycle because the load instruction has not yet loaded the value of *r1*. Therefore the pipeline stalls for one cycle while the load instruction completes the LS1 stage. Now that *r1* is ready, the processor executes the ADD in the ALU on the third cycle.

Figure 6.2 illustrates how this interlock affects the pipeline. The processor stalls the ADD instruction for one cycle in the ALU stage of the pipeline while the load instruction completes the LS1 stage. We've denoted this stall by an italic ADD. Since the LDR instruction proceeds down the pipeline, but the ADD instruction is stalled, a gap opens up between them. This gap is sometimes called a pipeline *bubble*. We've marked the bubble with a dash. ■

Example 6.7 This example shows a one-cycle interlock caused by delayed load use.

```
LDRB   r1, [r2, #1]
ADD    r0, r0, r2
EOR    r0, r0, r1
```

This instruction triplet takes four cycles. Although the ADD proceeds on the cycle following the load byte, the EOR instruction cannot start on the third cycle. The *r1* value is not ready until the load instruction completes the LS2 stage of the pipeline. The processor stalls the EOR instruction for one cycle.

Note that the ADD instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not! Figure 6.3 shows how this sequence progresses through the processor pipeline. The ADD doesn't cause any stalls since the ADD does not use *r1*, the result of the load. ■

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB	...	
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	*EOR*	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Figure 6.3 One-cycle interlock caused by delayed load use.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Figure 6.4 Pipeline flush caused by a branch.

Example 6.8 This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.

```
        MOV   r1, #1
        B     case1
        AND   r0, r0, r1
        EOR   r2, r2, r3
        ...
case1
        SUB   r0, r0, r1
```

The three executed instructions take a total of five cycles. The MOV instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles. Finally, the SUB instruction executes normally. Figure 6.4 illustrates the pipeline state on each cycle. The pipeline drops the two instructions following the branch when the branch takes place.

6.3.1 Scheduling of Load Instructions

Load instructions occur frequently in compiled code, accounting for approximately one-third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problems of C that we looked at in Section 5.6 limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Let's consider an example of a memory-intensive task. The following function, str_tolower, copies a zero-terminated string of characters from in to out. It converts the string to lowercase in the process.

```
void str_tolower(char *out, char *in)
{
  unsigned int c;

  do
  {
    c = *(in++);
    if (c>='A' && c<='Z')
    {
      c = c + ('a' -'A');
    }
    *(out++) = (char)c;
  } while (c);
}
```

The ADS1.1 compiler generates the following compiled output. Notice that the compiler optimizes the condition `(c>='A' && c<='Z')` to the check that `0<=c-'A'<='Z'-'A'`. The compiler can perform this check using a single unsigned comparison.

```
str_tolower
        LDRB    r2,[r1],#1    ; c = *(in++)
        SUB     r3,r2,#0x41   ; r3 = c -'A'
        CMP     r3,#0x19      ; if (c <='Z'-'A')
        ADDLS   r2,r2,#0x20   ;     c +='a'-'A'
        STRB    r2,[r0],#1    ; *(out++) = (char)c
        CMP     r2,#0         ; if (c!=0)
        BNE     str_tolower   ;     goto str_tolower
        MOV     pc,r14        ; return
```

Unfortunately, the `SUB` instruction uses the value of `c` directly after the `LDRB` instruction that loads `c`. Consequently, the ARM9TDMI pipeline will stall for two cycles. The compiler can't do any better since everything following the load of `c` depends on its value. However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by *preloading* and *unrolling*.

6.3.1.1 Load Scheduling by Preloading

In this method of load scheduling, we load the data required for the loop at the end of the previous loop, rather than at the beginning of the current loop. To get performance improvement with little increase in code size, we don't unroll the loop.

EXAMPLE 6.9

This assembly applies the preload method to the `str_tolower` function.

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string
```

```
c       RN 2    ; character loaded
t       RN 3    ; scratch register
        ; void str_tolower_preload(char *out, char *in)
        str_tolower_preload
        LDRB    c, [in], #1       ; c = *(in++)
loop
        SUB     t, c, #'A'        ; t = c-'A'
        CMP     t, #'Z'-'A'       ; if (t <= 'Z'-'A')
        ADDLS   c, c, #'a'-'A'    ;   c += 'a'-'A';
        STRB    c, [out], #1      ; *(out++) = (char)c;
        TEQ     c, #0             ; test if c==0
        LDRNEB  c, [in], #1       ; if (c!=0) { c=*in++;
        BNE     loop              ;             goto loop; }
        MOV     pc, lr            ; return
```

The scheduled version is one instruction longer than the C version, but we save two cycles for each inner loop iteration. This reduces the loop from 11 cycles per character to 9 cycles per character on an ARM9TDMI, giving a 1.22 times speed improvement. ■

The ARM architecture is particularly well suited to this type of preloading because instructions can be executed conditionally. Since loop *i* is loading the data for loop *i* + 1 there is always a problem with the first and last loops. For the first loop, we can preload data by inserting extra load instructions before the loop starts. For the last loop it is essential that the loop does not read any data, or it will read beyond the end of the array. This could cause a data abort! With ARM, we can easily solve this problem by making the load instruction conditional. In Example 6.9, the preload of the next character only takes place if the loop will iterate once more. No byte load occurs on the last loop.

6.3.1.2 Load Scheduling by Unrolling

This method of load scheduling works by unrolling and then interleaving the body of the loop. For example, we can perform loop iterations *i*, *i* + 1, *i* + 2 interleaved. When the result of an operation from loop *i* is not ready, we can perform an operation from loop *i* + 1 that avoids waiting for the loop *i* result.

EXAMPLE 6.10 The assembly applies load scheduling by unrolling to the `str_tolower` function.

```
out     RN 0    ; pointer to output string
in      RN 1    ; pointer to input string
ca0     RN 2    ; character 0
t       RN 3    ; scratch register
```

```
ca1     RN 12   ; character 1
ca2     RN 14   ; character 2
        ; void str_tolower_unrolled(char *out, char *in)
        str_tolower_unrolled
        STMFD   sp!, {lr}         ; function entry
loop_next3
        LDRB    ca0, [in], #1     ; ca0 = *in++;
        LDRB    ca1, [in], #1     ; ca1 = *in++;
        LDRB    ca2, [in], #1     ; ca2 = *in++;
        SUB     t, ca0, #'A'      ; convert ca0 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca0, ca0, #'a'-'A'
        SUB     t, ca1, #'A'      ; convert ca1 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca1, ca1, #'a'-'A'
        SUB     t, ca2, #'A'      ; convert ca2 to lower case
        CMP     t, #'Z'-'A'
        ADDLS   ca2, ca2, #'a'-'A'
        STRB    ca0, [out], #1    ; *out++ = ca0;
        TEQ     ca0, #0           ; if (ca0!=0)
        STRNEB  ca1, [out], #1    ;   *out++ = ca1;
        TEQNE   ca1, #0           ; if (ca0!=0 && ca1!=0)
        STRNEB  ca2, [out], #1    ;   *out++ = ca2;
        TEQNE   ca2, #0           ; if (ca0!=0 && ca1!=0 && ca2!=0)
        BNE     loop_next3        ;   goto loop_next3;
        LDMFD   sp!, {pc}         ; return;
```

This loop is the most efficient implementation we've looked at so far. The implementation requires seven cycles per character on ARM9TDMI. This gives a 1.57 times speed increase over the original `str_tolower`. Again it is the conditional nature of the ARM instructions that makes this possible. We use conditional instructions to avoid storing characters that are past the end of the string.

However, the improvement in Example 6.10 does have some costs. The routine is more than double the code size of the original implementation. We have assumed that you can read up to two characters beyond the end of the input string, which may not be true if the string is right at the end of available RAM, where reading off the end will cause a data abort. Also, performance can be slower for very short strings because (1) stacking *lr* causes additional function call overhead and (2) the routine may process up to two characters pointlessly, before discovering that they lie beyond the end of the string.

You should use this form of scheduling by unrolling for time-critical parts of an application where you know the data size is large. If you also know the size of the data at compile time, you can remove the problem of reading beyond the end of the array.

Summary **Instruction Scheduling**

- ARM cores have a pipeline architecture. The pipeline may delay the results of certain instructions for several cycles. If you use these results as source operands in a following instruction, the processor will insert stall cycles until the value is ready.
- Load and multiply instructions have delayed results in many implementations. See Appendix D for the cycle timings and delay for your specific ARM processor core.
- You have two software methods available to remove interlocks following load instructions: You can preload so that loop *i* loads the data for loop *i* + 1, or you can unroll the loop and interleave the code for loops *i* and *i* + 1.

6.4 Register Allocation

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer *r13* and the program counter *r15*. For a function to be ATPCS compliant it must preserve the callee values of registers *r4* to *r11*. ATPCS also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:

```
routine_name
        STMFD sp!,      {r4-r12, lr}      ; stack saved registers
          ; body of routine
          ; the fourteen registers r0-r12 and lr are available
        LDMFD sp!,      {r4-r12, pc}      ; restore registers and return
```

Our only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other ATPCS routines. For ARMv5 and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an ARMv4T processor, then modify the template as follows:

```
routine_name
       STMFD sp!,      {r4-r12, lr}      ; stack saved registers
         ; body of routine
         ; registers r0-r12 and lr available
       LDMFD sp!,      {r4-r12, lr}      ; restore registers
       BX         lr                     ; return, with mode switch
```

In this section we look at how best to allocate variables to register numbers for register-intensive tasks, how to use more than 14 local variables, and how to make the best use of the 14 available registers.

6.4.1 Allocating Variables to Register Numbers

When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are orthogonal with respect to register number. In other words, specific register numbers do not have specific roles. If you swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change. However, there are several cases where the physical number of the register is important:

- *Argument registers.* The ATPCS convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack. The return value must be placed in *r0*.
- *Registers used in a load or store multiple.* Load and store multiple instructions `LDM` and `STM` operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- *Load and store double word.* The `LDRD` and `STRD` instructions introduced in ARMv5E operate on a pair of registers with sequential register numbers, *Rd* and *Rd* + 1. Furthermore, *Rd* must be an even register number.

For an example of how to allocate registers when writing assembly, suppose we want to shift an array of N bits upwards in memory by k bits. For simplicity assume that N is large and a multiple of 256. Also assume that $0 \leq k < 32$ and that the input and output pointers are word aligned. This type of operation is common in dealing with the arithmetic of multiple precision numbers where we want to multiply by 2^k. It is also useful to block copy from one bit or byte alignment to a different bit or byte alignment. For example, the C library function `memcpy` can use the routine to copy an array of bytes using only word accesses.

The C routine `shift_bits` implements the simple k-bit shift of N bits of data. It returns the k bits remaining following the shift.

```
unsigned int shift_bits(unsigned int *out, unsigned int *in,
                        unsigned int N, unsigned int k)
{
  unsigned int carry=0, x;

  do
  {
    x = *in++;
    *out++ = (x<<k) | carry;
    carry = x>>(32-k);
    N -= 32;
  } while (N);

  return carry;
}
```

The obvious way to improve efficiency is to unroll the loop to process eight words of 256 bits at a time so that we can use load and store multiple operations to load and store eight words at a time for maximum efficiency. Before thinking about register numbers, we write the following assembly code:

```
shift_bits
        STMFD   sp!, {r4-r11, lr}       ; save registers
        RSB     kr, k, #32              ; kr = 32-k;
        MOV     carry, #0
loop
        LDMIA   in!, {x_0-x_7}          ; load 8 words
        ORR     y_0, carry, x_0, LSL k  ; shift the 8 words
        MOV     carry, x_0, LSR kr
        ORR     y_1, carry, x_1, LSL k
        MOV     carry, x_1, LSR kr
        ORR     y_2, carry, x_2, LSL k
        MOV     carry, x_2, LSR kr
        ORR     y_3, carry, x_3, LSL k
        MOV     carry, x_3, LSR kr
        ORR     y_4, carry, x_4, LSL k
        MOV     carry, x_4, LSR kr
        ORR     y_5, carry, x_5, LSL k
        MOV     carry, x_5, LSR kr
        ORR     y_6, carry, x_6, LSL k
        MOV     carry, x_6, LSR kr
        ORR     y_7, carry, x_7, LSL k
        MOV     carry, x_7, LSR kr
        STMIA   out!, {y_0-y_7}         ; store 8 words
        SUBS    N, N, #256              ; N -= (8 words * 32 bits)
        BNE     loop                    ; if (N!=0) goto loop;
        MOV     r0, carry               ; return carry;
        LDMFD   sp!, {r4-r11, pc}
```

Now to the register allocation. So that the input arguments do not have to move registers, we can immediately assign

```
out    RN 0
in     RN 1
```

```
N      RN 2
k      RN 3
```

For the load multiple to work correctly, we must assign x_0 through x_7 to sequentially increasing register numbers, and similarly for y_0 through y_7. Notice that we finish with x_0 before starting with y_1. In general, we can assign x_n to the same register as y_{n+1}. Therefore, assign

```
x_0    RN 5
x_1    RN 6
x_2    RN 7
x_3    RN 8
x_4    RN 9
x_5    RN 10
x_6    RN 11
x_7    RN 12
y_0    RN 4
y_1    RN x_0
y_2    RN x_1
y_3    RN x_2
y_4    RN x_3
y_5    RN x_4
y_6    RN x_5
y_7    RN x_6
```

We are nearly finished, but there is a problem. There are two remaining variables *carry* and *kr*, but only one remaining free register *lr*. There are several possible ways we can proceed when we run out of registers:

- Reduce the number of registers we require by performing fewer operations in each loop. In this case we could load four words in each load multiple rather than eight.
- Use the stack to store the least-used values to free up more registers. In this case we could store the loop counter *N* on the stack. (See Section 6.4.2 for more details on swapping registers to the stack.)
- Alter the code implementation to free up more registers. This is the solution we consider in the following text. (For more examples, see Section 6.4.3.)

We often iterate the process of implementation followed by register allocation several times until the algorithm fits into the 14 available registers. In this case we notice that the carry value need not stay in the same register at all! We can start off with the carry value in y_0 and then move it to y_1 when x_0 is no longer required, and so on. We complete the routine by allocating *kr* to *lr* and recoding so that carry is not required.

EXAMPLE 6.11

This assembly shows our final shift_bits routine. It uses all 14 available ARM registers.

```
kr   RN lr

shift_bits
             STMFD   sp!, {r4-r11, lr}      ; save registers
             RSB     kr, k, #32             ; kr = 32-k;
             MOV     y_0, #0                ; initial carry
loop
             LDMIA   in!, {x_0-x_7}         ; load 8 words
             ORR     y_0, y_0, x_0, LSL k   ; shift the 8 words
             MOV     y_1, x_0, LSR kr       ; recall x_0 = y_1
             ORR     y_1, y_1, x_1, LSL k
             MOV     y_2, x_1, LSR kr
             ORR     y_2, y_2, x_2, LSL k
             MOV     y_3, x_2, LSR kr
             ORR     y_3, y_3, x_3, LSL k
             MOV     y_4, x_3, LSR kr
             ORR     y_4, y_4, x_4, LSL k
             MOV     y_5, x_4, LSR kr
             ORR     y_5, y_5, x_5, LSL k
             MOV     y_6, x_5, LSR kr
             ORR     y_6, y_6, x_6, LSL k
             MOV     y_7, x_6, LSR kr
             ORR     y_7, y_7, x_7, LSL k
             STMIA   out!, {y_0-y_7}        ; store 8 words
             MOV     y_0, x_7, LSR kr
             SUBS    N, N, #256             ; N -= (8 words * 32 bits)
             BNE     loop                   ; if (N!=0) goto loop;
             MOV     r0, y_0                ; return carry;
             LDMFD   sp!, {r4-r11, pc}
```

■

6.4.2 Using More than 14 Local Variables

If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack. The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

EXAMPLE 6.12

This example shows three nested loops, each loop requiring state information inherited from the loop surrounding it. (See Section 6.6 for further ideas and examples of looping constructs.)

```
nested_loops
        STMFD   sp!, {r4-r11, lr}
        ; set up loop 1
loop1
        STMFD   sp!, {loop1 registers}
        ; set up loop 2
loop2
        STMFD   sp!, {loop2 registers}
        ; set up loop 3
loop3
        ; body of loop 3
        B{cond} loop3
        LDMFD   sp!, {loop2 registers}
        ; body of loop 2
        B{cond} loop2
        LDMFD   sp!, {loop1 registers}
        ; body of loop 1
        B{cond} loop1
        LDMFD   sp!, {r4-r11, pc}
```

You may find that there are insufficient registers for the innermost loop even using the construction in Example 6.12. Then you need to swap inner loop variables out to the stack. Since assembly code is very hard to maintain and debug if you use numbers as stack address offsets, the assembler provides an automated procedure for allocating variables to the stack.

EXAMPLE 6.13

This example shows how you can use the ARM assembler directives MAP (alias ^) and FIELD (alias #) to define and allocate space for variables and arrays on the processor stack. The directives perform a similar function to the struct operator in C.

```
         MAP     0    ; map symbols to offsets starting at offset 0
a        FIELD   4    ; a is 4 byte integer (at offset 0)
b        FIELD   2    ; b is 2 byte integer (at offset 4)
c        FIELD   2    ; c is 2 byte integer (at offset 6)
d        FIELD   64   ; d is an array of 64 characters (at offset 8)
length   FIELD   0    ; length records the current offset reached

example
         STMFD   sp!, {r4-r11, lr}   ; save callee registers
         SUB     sp, sp, #length     ; create stack frame
         ; ...
         STR     r0, [sp, #a]        ; a = r0;
         LDRSH   r1, [sp, #b]        ; r1 = b;
```

```
ADD     r2, sp, #d          ; r2 = &d[0]
; ...
ADD     sp, sp, #length     ; restore the stack pointer
LDMFD   sp!, {r4-r11, pc}   ; return
```

6.4.3 Making the Most of Available Registers

On a load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance. This section presents three examples showing how you can pack multiple variables into a single ARM register.

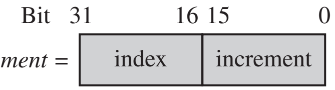
Example 6.14

Suppose we want to step through an array by a programmable increment. A common example is to step through a sound sample at various rates to produce different pitched notes. We can express this in C code as

```
sample = table[index];
index += increment;
```

Commonly index and increment are small enough to be held as 16-bit values. We can pack these two variables into a single 32-bit variable indinc:

indinc = *(index*<<16) + *increment* =



The C code translates into assembly code using a single register to hold indinc:

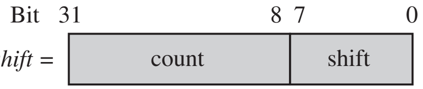
```
LDRB    sample, [table, indinc, LSR#16]  ; table[index]
ADD     indinc, indinc, indinc, LSL#16   ; index+=increment
```

Note that if index and increment are 16-bit values, then putting index in the top 16 bits of indinc correctly implements 16-bit-wrap-around. In other words, index = (short)(index + increment). This can be useful if you are using a buffer where you want to wrap from the end back to the beginning (often known as a *circular buffer*).

Example 6.15

When you shift by a register amount, the ARM uses bits 0 to 7 as the shift amount. The ARM ignores bits 8 to 31 of the register. Therefore you can use bits 8 to 31 to hold a second variable distinct from the shift amount.

This example shows how to combine a register-specified shift `shift` and loop counter `count` to shift an array of 40 entries right by `shift` bits. We define a new variable `cntshf` that combines `count` and `shift`:

cntshf = (*count*<<8) + *shift* = 

```
out      RN 0   ; address of the output array
in       RN 1   ; address of the input array
cntshf   RN 2   ; count and shift right amount
x        RN 3   ; scratch variable
         ; void shift_right(int *out, int *in, unsigned shift);
shift_right
         ADD    cntshf, cntshf, #39<<8 ; count = 39
shift_loop
         LDR    x, [in], #4
         SUBS   cntshf, cntshf, #1<<8  ; decrement count
         MOV    x, x, ASR cntshf       ; shift by shift
         STR    x, [out], #4
         BGE    shift_loop             ; continue if count>=0
         MOV    pc, lr
```

EXAMPLE 6.16 If you are dealing with arrays of 8-bit or 16-bit values, it is sometimes possible to manipulate multiple values at a time by packing several values into a single 32-bit register. This is called *single issue multiple data* (SIMD) processing.

ARM architecture versions up to ARMv5 do not support SIMD operations explicitly. However, there are still areas where you can achieve SIMD type compactness. Section 6.6 shows how you can store multiple loop values in a single register. Here we look at a graphics example of how to process multiple 8-bit pixels in an image using normal `ADD` and `MUL` instructions to achieve some SIMD operations.

Suppose we want to merge two images X and Y to produce a new image Z. Let x_n, y_n, and z_n denote the nth 8-bit pixel in these images, respectively. Let $0 \leq a \leq 256$ be a scaling factor. To merge the images, we set

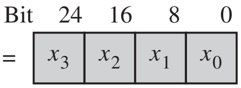
$$z_n = (ax_n + (256 - a)y_n)/256 \qquad (6.1)$$

In other words image Z is image X scaled in intensity by $a/256$ added to image Y scaled by $1 - (a/256)$. Note that

$$z_n = w_n/256, \quad \text{where } w_n = a(x_n - y_n) + 256y_n \qquad (6.2)$$

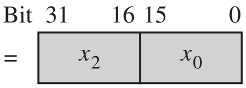
Therefore each pixel requires a subtract, a multiply, a shifted add, and a right shift. To process multiple pixels at a time, we load four pixels at once using a word load. We use a bracketed notation to denote several values packed into the same word:

[*x3, x2, x1, x0*] = $x_3 2^{24} + x_2 2^{16} + x_1 2^8 + x_0$ =



We then unpack the 8-bit data and promote it to 16-bit data using an AND with a mask register. We use the notation

[*x2, x0*] = $x_2 2^{16} + x_0$ =



Note that even for signed values $[a, b] + [c, d] = [a + b, c + d]$ if we interpret $[a, b]$ using the mathematical equation $a2^{16} + b$. Therefore we can perform SIMD operations on these values using normal arithmetic instructions.

The following code shows how you can process four pixels at a time using only two multiplies. The code assumes a 176 × 144 sized quarter CIF image.

```
IMAGE_WIDTH       EQU 176      ; QCIF width
IMAGE_HEIGHT      EQU 144      ; QCIF height

pz      RN 0    ; pointer to destination image (word aligned)
px      RN 1    ; pointer to first source image (word aligned)
py      RN 2    ; pointer to second source image (word aligned)
a       RN 3    ; 8-bit scaling factor (0-256)

xx      RN 4    ; holds four x pixels [x3, x2, x1, x0]
yy      RN 5    ; holds four y pixels [y3, y2, y1, y0]
x       RN 6    ; holds two expanded x pixels [x2, x0]
y       RN 7    ; holds two expanded y pixels [y2, y0]
z       RN 8    ; holds four z pixels [z3, z2, z1, z0]
count   RN 12   ; number of pixels remaining
mask    RN 14   ; constant mask with value 0x00ff00ff

        ; void merge_images(char *pz, char *px, char *py, int a)
merge_images
        STMFD   sp!, {r4-r8, lr}
        MOV     count, #IMAGE_WIDTH*IMAGE_HEIGHT
        LDR     mask, =0x00FF00FF   ; [   0, 0xFF,    0, 0xFF ]
 merge_loop
        LDR     xx, [px], #4        ; [  x3,   x2,   x1,   x0 ]
        LDR     yy, [py], #4        ; [  y3,   y2,   y1,   y0 ]
        AND     x, mask, xx         ; [   0,   x2,    0,   x0 ]
        AND     y, mask, yy         ; [   0,   y2,    0,   y0 ]
        SUB     x, x, y             ; [   (x2-y2),    (x0-y0) ]
```

```
MUL     x, a, x             ; [ a*(x2-y2),   a*(x0-y0) ]
ADD     x, x, y, LSL#8      ; [        w2,          w0 ]
AND     z, mask, x, LSR#8   ; [   0,   z2,     0,   z0 ]
AND     x, mask, xx, LSR#8  ; [   0,   x3,     0,   x1 ]
AND     y, mask, yy, LSR#8  ; [   0,   y3,     0,   y1 ]
SUB     x, x, y             ; [   (x3-y3),     (x1-y1) ]
MUL     x, a, x             ; [ a*(x3-y3),   a*(x1-y1) ]
ADD     x, x, y, LSL#8      ; [        w3,          w1 ]
AND     x, mask, x, LSR#8   ; [   0,   z3,     0,   z1 ]
ORR     z, z, x, LSL#8      ; [  z3,   z2,    z1,   z0 ]
STR     z, [pz], #4         ; store four z pixels
SUBS    count, count, #4
BGT     merge_loop
LDMFD   sp!, {r4-r8, pc}
```

The code works since

$$0 \le w_n \le 255a + 255(256 - a) = 256 \times 255 = \texttt{0xFF00} \qquad (6.3)$$

Therefore it is easy to separate the value $[w_2, w_0]$ into w_2 and w_0 by taking the most significant and least significant 16-bit portions, respectively. We have succeeded in processing four 8-bit pixels using 32-bit load, stores, and data operations to perform operations in parallel.

SUMMARY **Register Allocation**

- ARM has 14 available registers for general-purpose use: *r0* to *r12* and *r14*. The stack pointer *r13* and program counter *r15* cannot be used for general-purpose data. Operating system interrupts often assume that the *user* mode *r13* points to a valid stack, so don't be tempted to reuse *r13*.
- If you need more than 14 local variables, swap the variables out to the stack, working outwards from the innermost loop.
- Use register names rather than physical register numbers when writing assembly routines. This makes it easier to reallocate registers and to maintain the code.
- To ease register pressure you can sometimes store multiple values in the same register. For example, you can store a loop counter and a shift in one register. You can also store multiple pixels in one register.

6.5 Conditional Execution

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to the execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register. See Table A.2 in Appendix A for the list of possible ARM conditions. Also see Sections 2.2.6 and 3.8 for an introduction to conditional execution.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an S suffix to the instruction mnemonic. Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the S suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example 6.17

The following C code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character c:

```
if (i<10)
{
  c = i + '0';
}
else
{
  c = i + 'A'-10;
}
```

We can write this in assembly using conditional execution rather than conditional branches:

```
    CMP      i, #10
    ADDLO    c, i, #'0'
    ADDHS    c, i, #'A'-10
```

The sequence works since the first ADD does not change the condition codes. The second ADD is still conditional on the result of the compare. Section 6.3.1 shows a similar use of conditional execution to convert to lowercase.

Conditional execution is even more powerful for cascading conditions.

Example 6.18

The following C code identifies if c is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
   vowel++;
}
```

In assembly you can write this using conditional comparisons:

```
TEQ     c, #'a'
TEQNE   c, #'e'
TEQNE   c, #'i'
TEQNE   c, #'o'
TEQNE   c, #'u'
ADDEQ   vowel, vowel, #1
```

As soon as one of the TEQ comparisons detects a match, the *Z* flag is set in the *cpsr*. The following TEQNE instructions have no effect as they are conditional on $Z = 0$.

The next instruction to have effect is the ADDEQ that increments vowel. You can use this method whenever all the comparisons in the if statement are of the same type. ■

EXAMPLE 6.19

Consider the following code that detects if c is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
   letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq$ c $\leq$ *limit*. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

For more complicated decisions involving switches, see Section 6.8. ■

Note that the logical operations AND and OR are related by the standard logical relations as shown in Table 6.1. You can invert logical expressions involving OR to get an expression involving AND, which can often be useful in simplifying or rearranging logical expressions.

SUMMARY

Conditional Execution

- You can implement most if statements with conditional execution. This is more efficient than using a conditional branch.

Table 6.1 Inverted logical relations

Inverted expression	Equivalent
`!(a && b)`	`(!a) || (!b)`
`!(a || b)`	`(!a) && (!b)`

- You can implement `if` statements with the logical AND or OR of several similar conditions using compare instructions that are themselves conditional.

6.6 Looping Constructs

Most routines critical to performance will contain a loop. We saw in Section 5.3 that on the ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

6.6.1 Decremented Counted Loops

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```
        MOV i, N
loop
        ; loop body goes here and i=N,N-1,...,1
        SUBS i, i, #1
        BGT  loop
```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On ARM7 and ARM9 this overhead costs four cycles per loop. If i is an array index, then you may want to count down from $N - 1$ to 0 inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```
        SUBS i, N, #1
loop
        ; loop body goes here and i=N-1,N-2,...,0
        SUBS i, i, #1
        BGE  loop
```

In this arrangement the Z flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the EQ and NE conditions. For example, if you preload data for the next loop (as discussed in Section 6.3.1.1), then you want to avoid the preload on the last loop. You can make all preload operations conditional on NE as in Section 6.3.1.1.

There is no reason why we must decrement by one on each loop. Suppose we require $N/3$ loops. Rather than attempting to divide N by three, it is far more efficient to subtract three from the loop counter on each iteration:

```
        MOV i, N
loop
        ; loop body goes here and iterates (round up)(N/3) times
        SUBS i, i, #3
        BGT  loop
```

6.6.2 Unrolled Counted Loops

This brings us to the subject of loop unrolling. Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome. What if the loop count is not a multiple of the unroll amount? What if the loop count is smaller than the unroll amount? We looked at these questions for C code in Section 5.3. In this section we look at how you can handle these issues in assembly.

We'll take the C library function memset as a case study. This function sets N bytes of memory at address s to the byte value c. The function needs to be efficient, so we will look at how to unroll the loop without placing extra restrictions on the input operands. Our version of memset will have the following C prototype:

```
void my_memset(char *s, int c, unsigned int N);
```

To be efficient for large N, we need to write multiple bytes at a time using STR or STM instructions. Therefore our first task is to align the array pointer s. However, it is only worth us doing this if N is sufficiently large. We aren't sure yet what "sufficiently large" means, but let's assume we can choose a threshold value T_1 and only bother to align the array when $N \geq T_1$. Clearly $T_1 \geq 3$ as there is no point in aligning if we don't have four bytes to write!

Now suppose we have aligned the array s. We can use store multiples to set memory efficiently. For example, we can use a loop of four store multiples of eight words each to set 128 bytes on each loop. However, it will only be worth doing this if $N \geq T_2 \geq 128$, where T_2 is another threshold to be determined later on.

Finally, we are left with $N < T_2$ bytes to set. We can write bytes in blocks of four using STR until $N < 4$. Then we can finish by writing bytes singly with STRB to the end of the array.

EXAMPLE 6.20

This example shows the unrolled memset routine. We've separated the three sections corresponding to the preceding paragraphs with rows of dashes. The routine isn't finished until we've decided the best values for T_1 and T_2.

```
s      RN 0    ; current string pointer
c      RN 1    ; the character to fill with
N      RN 2    ; the number of bytes to fill
c_1    RN 3    ; copies of c
c_2    RN 4
c_3    RN 5
c_4    RN 6
c_5    RN 7
c_6    RN 8
c_7    RN 12

      ; void my_memset(char *s, unsigned int c, unsigned int N)
my_memset
        ;-----------------------------------------------
        ; First section aligns the array
        CMP     N, #T_1          ; We know that T_1>=3
        BCC     memset_1ByteBlk ; if (N<T_1) goto memset_1ByteBlk
        ANDS    c_1, s, #3       ; find the byte alignment of s
        BEQ     aligned          ; branch if already aligned
        RSB     c_1, c_1, #4     ; number of bytes until alignment
        SUB     N, N, c_1        ; number of bytes after alignment
        CMP     c_1, #2
        STRB    c, [s], #1
        STRGEB  c, [s], #1       ; if (c_1>=2) then output byte
        STRGTB  c, [s], #1       ; if (c_1>=3) then output byte
aligned                          ;the s array is now aligned
        ORR     c, c, c, LSL#8   ; duplicate the character
        ORR     c, c, c, LSL#16  ; to fill all four bytes of c
        ;-----------------------------------------------
        ; Second section writes blocks of 128 bytes
        CMP     N, #T_2          ; We know that T_2 >= 128
        BCC     memset_4ByteBlk ; if (N<T_2) goto memset_4ByteBlk
        STMFD   sp!, {c_2-c_6}    ; stack scratch registers
        MOV     c_1, c
        MOV     c_2, c
        MOV     c_3, c
        MOV     c_4, c
        MOV     c_5, c
        MOV     c_6, c
        MOV     c_7, c
        SUB     N, N, #128      ; bytes left after next block
loop128 ; write 32 words = 128 bytes
        STMIA   s!, {c, c_1-c_6, c_7} ; write 8 words
        STMIA   s!, {c, c_1-c_6, c_7} ; write 8 words
        STMIA   s!, {c, c_1-c_6, c_7} ; write 8 words
        STMIA   s!, {c, c_1-c_6, c_7} ; write 8 words
        SUBS    N, N, #128      ; bytes left after next block
        BGE     loop128
        ADD     N, N, #128      ; number of bytes left
        LDMFD   sp!, {c_2-c_6}  ; restore corrupted registers
        ;--------------------------------------------
        ; Third section deals with left over bytes
memset_4ByteBlk
        SUBS    N, N, #4        ; try doing 4 bytes
loop4   ; write 4 bytes
        STRGE   c, [s], #4
        SUBGES  N, N, #4
        BGE     loop4
        ADD     N, N, #4        ; number of bytes left
memset_1ByteBlk
        SUBS    N, N, #1
loop1   ; write 1 byte
        STRGEB  c, [s], #1
        SUBGES  N, N, #1
        BGE     loop1
        MOV     pc, lr          ; finished so return
```

It remains to find the best values for the thresholds T_1 and T_2. To determine these we need to analyze the cycle counts for different ranges of N. Since the algorithm operates on blocks of size 128 bytes, 4 bytes, and 1 byte, respectively, we start by decomposing N with respect to these block sizes:

$$N = 128N_h + 4N_m + N_l, \quad \text{where } 0 \le N_m < 32 \quad \text{and} \quad 0 \le N_l < 4$$

We now partition into three cases. To follow the details of these cycle counts, you will need to refer to the instruction cycle timings in Appendix D.

- Case $0 \le N < T_1$: The routine takes $5N + 6$ cycles on an ARM9TDMI including the return.
- Case $T_1 \le N < T_2$: The first algorithm block takes 6 cycles if the *s* array is word aligned and 10 cycles otherwise. Assuming each alignment is equally likely, this averages to $(6 + 10 + 10 + 10)/4 = 9$ cycles. The second algorithm block takes 6 cycles. The final

Table 6.2 Cycles taken for each range of N values.

N range	Cycles taken
$0 \leq N < T_1$	$640N_h + 20N_m + 5N_l + 6$
$T_1 \leq N < T_2$	$160N_h + 5N_m + 5N_l + 17 + 5Z_l$
$T_2 \leq N$	$36N_h + 5N_m + 5N_l + 32 + 5Z_l + 5Z_m$

block takes $5(32N_h + N_m) + 5(N_l + Z_l) + 2$ cycles, where Z_l is 1 if $N_l = 0$, and 0 otherwise. The total cycles for this case is $5(32N_h + N_m + N_l + Z_l) + 17$.

- Case $N \geq T_2$: As in the previous case, the first algorithm block averages 9 cycles. The second algorithm block takes $36N_h + 21$ cycles. The final algorithm block takes $5(N_m + Z_m + N_l + Z_l) + 2$ cycles, where Z_m is 1 if N_m is 0, and 0 otherwise. The total cycles for this case is $36N_h + 5(N_m + Z_m + N_l + Z_l) + 32$.

Table 6.2 summarizes these results. Comparing the three table rows it is clear that the second row wins over the first row as soon as $N_m \geq 1$, unless $N_m = 1$ and $N_l = 0$. We set $T_1 = 5$ to choose the best cycle counts from rows one and two. The third row wins over the second row as soon as $N_h \geq 1$. Therefore take $T_2 = 128$.

This detailed example shows you how to unroll any important loop using threshold values and provide good performance over a range of possible input values. ■

6.6.3 Multiple Nested Loops

How many loop counters does it take to maintain multiple nested loops? Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32. We can combine the loop counts within a single register, placing the innermost loop count at the highest bit positions. This section gives an example showing how to do this. We will ensure the loops count down from $max - 1$ to 0 inclusive so that the loop terminates by producing a negative result.

Example 6.21 This example shows how to merge three loop counts into a single loop count. Suppose we wish to multiply matrix B by matrix C to produce matrix A, where A, B, C have the following constant dimensions. We assume that R, S, T are relatively large but less than 256.

Matrix A:	R rows × T columns
Matrix B:	R rows × S columns
Matrix C:	S rows × T columns

We represent each matrix by a lowercase pointer of the same name, pointing to an array of words organized by row. For example, the element at row *i*, column *j*, A[*i*, *j*], is at the byte address

```
&A[i,j] = a + 4*(i*T+j)
```

A simple C implementation of the matrix multiply uses three nested loops *i*, *j*, and *k*:

```
#define R 40
#define S 40
#define T 40

void ref_matrix_mul(int *a, int *b, int *c)
{
unsigned int i,j,k;
int sum;

for (i=0; i<R; i++)
{
  for (j=0; j<T; j++)
  {
    /* calculate a[i,j] */
    sum = 0;
    for (k=0; k<S; k++)
    {
     /* add b[i,k]*c[k,j] */
     sum += b[i*S+k]*c[k*T+j];
    }
    a[i*T+j] = sum;
  }
 }
}
```

There are many ways to improve the efficiency here, starting by removing the address indexing calculations, but we will concentrate on the looping structure. We allocate a register counter *count* containing all three loop counters *i*, *j*, *k*:

```
   Bit 31     24 23     16 15      8 7        0
count = |    0    |  S-1-k  |  T-1-j  |  R-1-i  |
```

Note that $S - 1 - k$ counts from $S - 1$ down to 0 rather than counting from 0 to $S - 1$ as *k* does. The following assembly implements the matrix multiply using this single counter in register *count*:

```
R        EQU 40
S        EQU 40
```

```
T        EQU 40

a        RN 0    ; points to an R rows × T columns matrix
b        RN 1    ; points to an R rows × S columns matrix
c        RN 2    ; points to an S rows × T columns matrix
sum      RN 3
bval     RN 4
cval     RN 12
count    RN 14

        ; void matrix_mul(int *a, int *b, int *c)
matrix_mul
         STMFD  sp!, {r4, lr}
         MOV    count, #(R-1)                ; i=0
loop_i
         ADD    count, count, #(T-1)<<8      ; j=0
loop_j
         ADD    count, count, #(S-1)<<16     ; k=0
         MOV    sum, #0
loop_k
         LDR    bval, [b], #4                ; bval = B[i,k], b=&B[i,k+1]
         LDR    cval, [c], #4*T              ; cval = C[k,j], c=&C[k+1,j]
         SUBS   count, count, #1<<16         ; k++
         MLA    sum, bval, cval, sum         ; sum += bval*cval
         BPL    loop_k                       ; branch if k<=S-1
         STR    sum, [a], #4                 ; A[i,j] = sum, a=&A[i,j+1]
         SUB    c, c, #4*S*T                 ; c = &C[0,j]
         ADD    c, c, #4                     ; c = &C[0,j+1]
         ADDS   count, count, #(1<<16)-(1<<8)  ; zero (S-1-k), j++
         SUBPL  b, b, #4*S                   ; b = &B[i,0]
         BPL    loop_j                       ; branch if j<=T-1
         SUB    c, c, #4*T                   ; c = &C[0,0]
         ADDS   count, count, #(1>>8)-1      ; zero (T-1-j), i++
         BPL    loop_i                       ; branch if i<=R-1
         LDMFD  sp!, {r4, pc}
```

The preceding structure saves two registers over a naive implementation. First, we decrement the *count* at bits 16 to 23 until the result is negative. This implements the *k* loop, counting down from $S - 1$ to 0 inclusive. Once the result is negative, the code adds 2^{16} to clear bits 16 to 31. Then we subtract 2^8 to decrement the count stored at bits 8 to 15, implementing the *j* loop. We can encode the constant $2^{16} - 2^8 =$ 0xFF00 efficiently using a single ARM instruction. Bits 8 to 15 now count down from $T - 1$ to 0. When the result of the combined add and subtract is negative, then we have finished the *j* loop. We repeat the same process for the *i* loop. ARM's ability to handle a wide range of rotated constants in addition and subtraction instructions makes this scheme very efficient.

6.6.4 Other Counted Loops

You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from N to 1 or $N - 1$ to 0. For example, you may want to select bits out of a data register one at a time; in this case you may want a power-of-two mask that doubles on each iteration.

The following subsections show useful looping structures that count in different patterns. They use only a single instruction combined with a branch to implement the loop.

6.6.4.1 Negative Indexing

This loop structure counts from $-N$ to 0 (inclusive or exclusive) in steps of size *STEP*.

```
        RSB     i, N, #0        ; i=-N
loop
        ; loop body goes here and i=-N,-N+STEP,...,
        ADDS    i, i, #STEP
        BLT     loop            ; use BLT or BLE to exclude 0 or not
```

6.6.4.2 Logarithmic Indexing

This loop structure counts down from 2^N to 1 in powers of two. For example, if $N = 4$, then it counts 16, 8, 4, 2, 1.

```
        MOV     i, #1
        MOV     i, i, LSL N
loop
        ; loop body
        MOVS    i, i, LSR#1
        BNE     loop
```

The following loop structure counts down from an N-bit mask to a one-bit mask. For example, if $N = 4$, then it counts 15, 7, 3, 1.

```
        MOV     i, #1
        RSB     i, i, i, LSL N  ; i=(1<<N)-1
```

```
loop
        ; loop body
        MOVS    i, i, LSR#1
        BNE     loop
```

SUMMARY **Looping Constructs**

- ARM requires two instructions to implement a counted loop: a subtract that sets flags and a conditional branch.
- Unroll loops to improve loop performance. Do not overunroll because this will hurt cache performance. Unrolled loops may be inefficient for a small number of iterations. You can test for this case and only call the unrolled loop if the number of iterations is large.
- Nested loops only require a single loop counter register, which can improve efficiency by freeing up registers for other uses.
- ARM can implement negative and logarithmic indexed loops efficiently.

6.7 Bit Manipulation

Compressed file formats pack items at a bit granularity to maximize the data density. The items may be of a fixed width, such as a length field or version field, or they may be of a variable width, such as a Huffman coded symbol. Huffman codes are used in compression to associate with each symbol a code of bits. The code is shorter for common symbols and longer for rarer symbols.

In this section we look at methods to handle a bitstream efficiently. First we look at fixed-width codes, then variable width codes. See Section 7.6 for common bit manipulation routines such as endianness and bit reversal.

6.7.1 Fixed-Width Bit-Field Packing and Unpacking

You can extract an unsigned bit-field from an arbitrary position in an ARM register in one cycle provided that you set up a mask in advance; otherwise you require two cycles. A signed bit-field always requires two cycles to unpack unless the bit-field lies at the top of a word (most significant bit of the bit-field is the most significant bit of the register). On the ARM we use logical operations and the barrel shifter to pack and unpack codes, as in the following examples.

EXAMPLE 6.22 The assembly code shows how to unpack bits 4 to 15 of register *r0*, placing the result in *r1*.

```
        ; unsigned unpack with mask set up in advance
        ; mask=0x00000FFF
```

```
AND     r1, mask, r0, LSR#4

; unsigned unpack with no mask
MOV     r1, r0, LSL#16  ; discard bits 16-31
MOV     r1, r1, LSR#20  ; discard bits 0-3 and zero extend

; signed unpack
MOV     r1, r0, LSL#16  ; discard bits 16-31
MOV     r1, r1, ASR#20  ; discard bits 0-3 and sign extend
```

■

Example 6.23 Packing the value *r1* into the bit-packed register *r0* requires one cycle if *r1* is already restricted to the correct range and the corresponding field of *r0* is clear. In this example, *r1* is a 12-bit number to be inserted at bit 4 of *r0*.

```
; pack r1 into r0
ORR     r0, r0, r1, LSL #4
```

Otherwise you need a mask register set up:

```
; pack r1 into r0
; mask=0x00000FFF set up in advance
AND     r1, r1, mask         ; restrict the r1 range
BIC     r0, r0, mask, LSL#4  ; clear the destination bits
ORR     r0, r0, r1, LSL#4    ; pack in the new data
```

■

6.7.2 Variable-Width Bitstream Packing

Our task here is to pack a series of variable-length codes to create a bitstream. Typically we are compressing a datastream and the variable-length codes represent Huffman or arithmetic coding symbols. However, we don't need to make any assumptions about what the codes represent to pack them efficiently.

We do need to be careful about the packing endianness. Many compressed file formats use a big-endian bit-packing order where the first code is placed at the most significant bits of the first byte. For this reason we will use a big-endian bit-packing order for our examples. This is sometimes known as *network order*. Figure 6.5 shows how we form a bytestream out of variable-length bitcodes using a big-endian packing order. *High* and *low* represent the most and least significant bit ends of the byte.

To implement packing efficiently on the ARM we use a 32-bit register as a buffer to hold four bytes, in big-endian order. In other words we place byte 0 of the bytestream in the most significant 8 bits of the register. Then we can insert codes into the register one at a time, starting from the most significant bit and working down to the least significant bit.

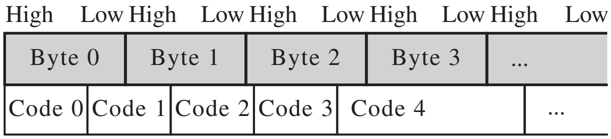


Figure 6.5 Big-endian bitcodes packed into a bytestream.

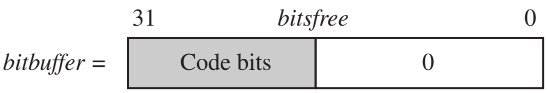


Figure 6.6 Format of *bitbuffer*.

Once the register is full we can store 32 bits to memory. For a big-endian memory system we can store the word without modification. For a little-endian memory system we need to reverse the byte order in the word before storing.

We call the 32-bit register we insert codes into *bitbuffer*. We need a second register *bitsfree* to record the number of bits that we haven't used in *bitbuffer*. In other words, *bitbuffer* contains 32 − *bitsfree* code bits, and *bitsfree* zero bits, as in Figure 6.6. To insert a code of *k* bits into *bitbuffer*, we subtract *k* from *bitsfree* and then insert the code with a left shift of *bitsfree*.

We also need to be careful about alignment. A bytestream need not be word aligned, and so we can't use word accesses to write it. To allow word accesses we will start by backing up to the last word-aligned address. Then we fill the 32-bit register *bitbuffer* with the backed-up data. From then on we can use word (32-bit) read and writes.

Example 6.24

This example provides three functions `bitstream_write_start`, `bitstream_write_code`, and `bitstream_write_flush`. These are not ATPCS-compliant functions because they assume registers such as *bitbuffer* are preserved between calls. In practice you will inline this code for efficiency, and so this is not a problem.

The `bitstream_write_start` function aligns the bitstream pointer `bitstream` and initializes the 32-bit buffer `bitbuffer`. Each call to `bitstream_write_code` inserts a value `code` of bit-length `codebits`. Finally, the `bitstream_write_flush` function writes any remaining bytes to the bitstream to terminate the stream.

```
bitstream     RN 0  ; current byte address in the output bitstream
code          RN 4  ; current code
```

```
codebits      RN 5  ; length in bits of current code
bitbuffer     RN 6  ; 32-bit output big-endian bitbuffer
bitsfree      RN 7  ; number of bits free in the bitbuffer
tmp           RN 8  ; scratch register
mask          RN 12 ; endian reversal mask 0xFFFF00FF

bitstream_write_start
        MOV     bitbuffer, #0
        MOV     bitsfree, #32
align_loop
        TST     bitstream, #3
        LDRNEB  code, [bitstream, #-1]!
        SUBNE   bitsfree, bitsfree, #8
        ORRNE   bitbuffer, code, bitbuffer, ROR #8
        BNE     align_loop
        MOV     bitbuffer, bitbuffer, ROR #8
        MOV     pc, lr

bitstream_write_code
        SUBS    bitsfree, bitsfree, codebits
        BLE     full_buffer
        ORR     bitbuffer, bitbuffer, code, LSL bitsfree
        MOV     pc, lr
full_buffer
        RSB     bitsfree, bitsfree, #0
        ORR     bitbuffer, bitbuffer, code, LSR bitsfree
        IF {ENDIAN}="little"
          ; byte reverse the bit buffer prior to storing
          EOR   tmp, bitbuffer, bitbuffer, ROR #16
          AND   tmp, mask, tmp, LSR #8
          EOR   bitbuffer, tmp, bitbuffer, ROR #8
        ENDIF
        STR     bitbuffer, [bitstream], #4
        RSB     bitsfree, bitsfree, #32
        MOV     bitbuffer, code, LSL bitsfree
        MOV     pc, lr

bitstream_write_flush
        RSBS    bitsfree, bitsfree, #32
flush_loop
        MOVGT   bitbuffer, bitbuffer, ROR #24
        STRGTB  bitbuffer, [bitstream], #1
        SUBGTS  bitsfree, bitsfree, #8
        BGT     flush_loop
        MOV     pc, lr
```

6.7.3 Variable-Width Bitstream Unpacking

It is much harder to unpack a bitstream of variable-width codes than to pack it. The problem is that we usually don't know the width of the codes we are unpacking! For Huffman-encoded bitstreams you must derive the length of each code by looking at the next sequence of bits and working out which code it can be.

Here we will use a lookup table to speed up the unpacking process. The idea is to take the next N bits of the bitstream and perform a lookup in two tables, *look_codebits*[] and *look_code*[], each of size 2^N entries. If the next N bits are sufficient to determine the code, then the tables tell us the code length and the code value, respectively. If the next N bits are insufficient to determine the code, then the `look_codebits` table will return an *escape* value of `0xFF`. An escape value is just a flag to indicate that this case is exceptional.

In a sequence of Huffman codes, common codes are short and rare codes are long. So, we expect to decode most common codes quickly, using the lookup tables. In the following example we assume that $N = 8$ and use 256-entry lookup tables.

Example 6.25

This example provides three functions to unpack a big-endian bitstream stored in a bytestream. As with Example 6.24, these functions are not ATPCS compliant and will normally be inlined. The function `bitstream_read_start` initializes the process, starting to decode a bitstream at byte address `bitstream`. Each call to `bitstream_read_code` returns the next code in register `code`. The function only handles short codes that can be read from the lookup table. Long codes are trapped at the label `long_code`, but the implementation of this function depends on the codes you are decoding.

The code uses a register `bitbuffer` that contains $N + bitsleft$ code bits starting at the most significant bit (see Figure 6.7).

```
bitstream     RN 0   ; current byte address in the input bitstream
look_code     RN 2   ; lookup table to convert next N bits to a code
look_codebits RN 3   ; lookup table to convert next N bits to a code length
code          RN 4   ; code read
codebits      RN 5   ; length of code read
bitbuffer     RN 6   ; 32-bit input buffer (big endian)
bitsleft      RN 7   ; number of valid bits in the buffer - N
```

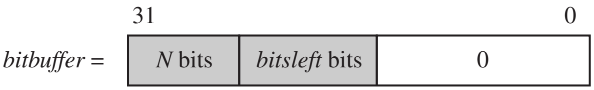


Figure 6.7 Format of `bitbuffer`.

```
tmp             RN 8   ; scratch
tmp2            RN 9   ; scratch
mask            RN 12 ; N-bit extraction mask (1<<N)-1

N               EQU 8  ; use a lookup table on 8 bits (N must be <= 9)

bitstream_read_start
        MOV     bitsleft, #32
read_fill_loop
        LDRB    tmp, [bitstream], #1
        ORR     bitbuffer, tmp, bitbuffer, LSL#8
        SUBS    bitsleft, bitsleft, #8
        BGT     read_fill_loop
        MOV     bitsleft, #(32-N)
        MOV     mask, #(1<<N)-1
        MOV     pc, lr

bitstream_read_code
        LDRB    codebits, [look_codebits, bitbuffer, LSR# (32-N)]
        AND     code, mask, bitbuffer, LSR#(32-N)
        LDR     code, [look_code, code, LSL#2]
        SUBS    bitsleft, bitsleft, codebits
        BMI     empty_buffer_or_long_code
        MOV     bitbuffer, bitbuffer, LSL codebits
        MOV     pc, lr
empty_buffer_or_long_code
        TEQ     codebits, #0xFF
        BEQ     long_code
        ; empty buffer - fill up with 3 bytes
        ; as N <= 9, we can fill 3 bytes without overflow
        LDRB    tmp, [bitstream], #1
        LDRB    tmp2, [bitstream], #1
        MOV     bitbuffer, bitbuffer, LSL codebits
        LDRB    codebits, [bitstream], #1
        ORR     tmp, tmp2, tmp, LSL#8
        RSB     bitsleft, bitsleft, #(8-N)
        ORR     tmp, codebits, tmp, LSL#8
        ORR     bitbuffer, bitbuffer, tmp, LSL bitsleft
        RSB     bitsleft, bitsleft, #(32-N)
        MOV     pc, lr

long_code
        ; handle the long code case depending on the application
        ; here we just return a code of -1
        MOV     code, #-1
        MOV     pc, lr
```

The counter bitsleft actually counts the number of bits remaining in the buffer bitbuffer less the N bits required for the next lookup. Therefore we can perform the next table lookup as long as bitsleft ≥ 0. As soon as bitsleft < 0 there are two possibilities. One possibility is that we found a valid code but then have insufficient bits to look up the next code. Alternatively, codebits contains the escape value 0xFF to indicate that the code was longer than N bits. We can trap both these cases at once using a call to empty_buffer_or_long_code. If the buffer is empty, then we fill it with 24 bits. If we have detected a long code, then we branch to the long_code trap.

The example has a best-case performance of seven cycles per code unpack on an ARM9TDMI. You can obtain faster results if you know the sizes of the packed bitfields in advance. ■

SUMMARY **Bit Manipulation**

- The ARM can pack and unpack bits efficiently using logical operations and the barrel shifter.
- To access bitstreams efficiently use a 32-bit register as a bitbuffer. Use a second register to keep track of the number of valid bits in the bitbuffer.
- To decode bitstreams efficiently, use a lookup table to scan the next N bits of the bitstream. The lookup table can return codes of length at most N bits directly, or return an escape character for longer codes.

6.8 Efficient Switches

A *switch* or *multiway branch* selects between a number of different actions. In this section we assume the action depends on a variable x. For different values of x we need to perform different actions. This section looks at assembly to implement a switch efficiently for different types of x.

6.8.1 Switches on the Range $0 \leq x < N$

The example C function ref_switch performs different actions according to the value of x. We are only interested in x values in the range $0 \leq x < 8$.

```
int ref_switch(int x)
{
  switch (x)
  {
    case 0: return method_0();
    case 1: return method_1();
    case 2: return method_2();
    case 3: return method_3();
    case 4: return method_4();
    case 5: return method_5();
    case 6: return method_6();
    case 7: return method_7();
    default: return method_d();
  }
}
```

There are two ways to implement this structure efficiently in ARM assembly. The first method uses a table of function addresses. We load *pc* from the table indexed by *x*.

Example 6.26

The `switch_absolute` code performs a switch using an inlined table of function pointers:

```
x   RN 0

    ; int switch_absolute(int x)
switch_absolute
    CMP     x, #8
    LDRLT   pc, [pc, x, LSL#2]
    B       method_d
    DCD     method_0
    DCD     method_1
    DCD     method_2
    DCD     method_3
    DCD     method_4
    DCD     method_5
    DCD     method_6
    DCD     method_7
```

The code works because the *pc* register is pipelined. The *pc* points to the `method_0` word when the ARM executes the `LDR` instruction.

The method above is very fast, but has one drawback: The code is not position independent since it stores absolute addresses to the method functions in memory. Position-independent code is often used in modules that are installed into a system at run time. The next example shows how to solve this problem.

Example 6.27

The code `switch_relative` is slightly slower compared to `switch_absolute`, but it is position independent:

```
    ; int switch_relative(int x)
switch_relative
```

```
        CMP     x, #8
        ADDLT   pc, pc, x, LSL#2
        B       method_d
        B       method_0
        B       method_1
        B       method_2
        B       method_3
        B       method_4
        B       method_5
        B       method_6
        B       method_7
```

There is one final optimization you can make. If the method functions are short, then you can inline the instructions in place of the branch instructions.

Example 6.28 Suppose each nondefault method has a four-instruction implementation. Then you can use code of the form

```
        CMP     x, #8
        ADDLT   pc, pc, x, LSL#4 ; each method is 16 bytes long
        B       method_d
method_0
        ; the four instructions for method_0 go here
method_1
        ; the four instructions for method_1 go here
        ; ... continue in this way ...
```

6.8.2 Switches on a General Value x

Now suppose that x does not lie in some convenient range $0 \leq x < N$ for N small enough to apply the methods of Section 6.8.1. How do we perform the switch efficiently, without having to test x against each possible value in turn?

A very useful technique in these situations is to use a *hashing function*. A hashing function is any function $y = f(x)$ that maps the values we are interested in into a continuous range of the form $0 \leq y < N$. Instead of a switch on x, we can use a switch on $y = f(x)$. There is a problem if we have a collision, that is, if two x values map to the same y value. In this case we need further code to test all the possible x values that could have led to the y value. For our purposes a good hashing function is easy to compute and does not suffer from many collisions.

To perform the switch we apply the hashing function and then use the optimized switch code of Section 6.8.1 on the hash value y. Where two x values can map to the same hash, we need to perform an explicit test, but this should be rare for a good hash function.

EXAMPLE 6.29

Suppose we want to call method_k when $x = 2^k$ for eight possible methods. In other words we want to switch on the values 1, 2, 4, 8, 16, 32, 64, 128. For all other values of x we need to call the default method method_d. We look for a hash function formed out of multiplying by powers of two minus one (this is an efficient operation on the ARM). By trying different multipliers we find that $15 \times 31 \times x$ has a different value in bits 9 to 11 for each of the eight switch values. This means we can use bits 9 to 11 of this product as our hash function.

The following switch_hash assembly uses this hash function to perform the switch. Note that other values that are not powers of two will have the same hashes as the values we want to detect. The switch has narrowed the case down to a single power of two that we can test for explicitly. If x is not a power of two, then we fall through to the default case of calling method_d.

```
x       RN 0
hash    RN 1

        ; int switch_hash(int x)
switch_hash
        RSB     hash, x, x, LSL#4       ; hash=x*15
        RSB     hash, hash, hash, LSL#5 ; hash=x*15*31
        AND     hash, hash, #7<<9       ; mask out the hash value
        ADD     pc, pc, hash, LSR#6
        NOP
        TEQ     x, #0x01
        BEQ     method_0
        TEQ     x, #0x02
        BEQ     method_1
        TEQ     x, #0x40
        BEQ     method_6
        TEQ     x, #0x04
        BEQ     method_2
        TEQ     x, #0x80
        BEQ     method_7
        TEQ     x, #0x20
        BEQ     method_5
        TEQ     x, #0x10
        BEQ     method_4
        TEQ     x, #0x08
        BEQ     method_3
        B       method_d
```

SUMMARY

Efficient Switches

- Make sure the switch value is in the range $0 \leq x < N$ for some small N. To do this you may have to use a hashing function.

- Use the switch value to index a table of function pointers or to branch to short sections of code at regular intervals. The second technique is position independent; the first isn't.

6.9 Handling Unaligned Data

Recall that a load or store is *unaligned* if it uses an address that is not a multiple of the data transfer width. For code to be portable across ARM architectures and implementations, you must avoid unaligned access. Section 5.9 introduced unaligned accesses and ways of handling them in C. In this section we look at how to handle unaligned accesses in assembly code.

The simplest method is to use byte loads and stores to access one byte at a time. This is the recommended method for any accesses that are not speed critical. The following example shows how to access word values in this way.

Example 6.30

This example shows how to read or write a 32-bit word using the unaligned address p. We use three scratch registers t0, t1, t2 to avoid interlocks. All unaligned word operations take seven cycles on an ARM9TDMI. Note that we need separate functions for 32-bit words stored in big- or little-endian format.

```
p       RN 0
x       RN 1
t0      RN 2
t1      RN 3
t2      RN 12

        ; int load_32_little(char *p)
load_32_little
        LDRB    x,  [p]
        LDRB    t0, [p, #1]
        LDRB    t1, [p, #2]
        LDRB    t2, [p, #3]
        ORR     x, x, t0, LSL#8
        ORR     x, x, t1, LSL#16
        ORR     r0, x, t2, LSL#24
        MOV     pc, lr

        ; int load_32_big(char *p)
load_32_big
        LDRB    x,  [p]
        LDRB    t0, [p, #1]
        LDRB    t1, [p, #2]
        LDRB    t2, [p, #3]
        ORR     x, t0, x, LSL#8
        ORR     x, t1, x, LSL#8
        ORR     r0, t2, x, LSL#8
        MOV     pc, lr

        ; void store_32_little(char *p, int x)
store_32_little
        STRB    x,  [p]
        MOV     t0, x, LSR#8
        STRB    t0, [p, #1]
        MOV     t0, x, LSR#16
        STRB    t0, [p, #2]
        MOV     t0, x, LSR#24
        STRB    t0, [p, #3]
        MOV     pc, lr

        ; void store_32_big(char *p, int x)
store_32_big
        MOV     t0, x, LSR#24
        STRB    t0, [p]
        MOV     t0, x, LSR#16
        STRB    t0, [p, #1]
        MOV     t0, x, LSR#8
        STRB    t0, [p, #2]
        STRB    x, [p, #3]
        MOV     pc, lr
```

■

If you require better performance than seven cycles per access, then you can write several variants of the routine, with each variant handling a different address alignment. This reduces the cost of the unaligned access to three cycles: the word load and the two arithmetic instructions required to join values together.

EXAMPLE 6.31 This example shows how to generate a checksum of N words starting at a possibly unaligned address data. The code is written for a little-endian memory system. Notice how we can use the assembler MACRO directive to generate the four routines checksum_0, checksum_1, checksum_2, and checksum_3. Routine checksum_a handles the case where data is an address of the form $4q + a$.

Using a macro saves programming effort. We need only write a single macro and instantiate it four times to implement our four checksum routines.

```
sum     RN 0    ; current checksum
N       RN 1    ; number of words left to sum
```

```
data    RN 2    ; word aligned input data pointer
w       RN 3    ; data word

        ; int checksum_32_little(char *data, unsigned int N)
checksum_32_little
        BIC     data, r0, #3       ; aligned data pointer
        AND     w, r0, #3          ; byte alignment offset
        MOV     sum, #0            ; initial checksum
        LDR     pc, [pc, w, LSL#2] ; switch on alignment
        NOP                        ; padding
        DCD     checksum_0
        DCD     checksum_1
        DCD     checksum_2
        DCD     checksum_3

        MACRO
        CHECKSUM $alignment
checksum_$alignment
        LDR     w, [data], #4      ; preload first value
10      ; loop
        IF $alignment<>0
          ADD   sum, sum, w, LSR#8*$alignment
          LDR   w, [data], #4
          SUBS  N, N, #1
          ADD   sum, sum, w, LSL#32-8*$alignment
        ELSE
          ADD   sum, sum, w
          LDR   w, [data], #4
          SUBS  N, N, #1
        ENDIF
        BGT     %BT10
        MOV     pc, lr
        MEND

        ; generate four checksum routines
        ; one for each possible byte alignment
        CHECKSUM 0
        CHECKSUM 1
        CHECKSUM 2
        CHECKSUM 3
```

You can now unroll and optimize the routines as in Section 6.6.2 to achieve the fastest speed. Due to the additional code size, only use the preceding technique for time-critical routines.

SUMMARY **Handling Unaligned Data**

- If performance is not an issue, access unaligned data using multiple byte loads and stores. This approach accesses data of a given endianness regardless of the pointer alignment and the configured endianness of the memory system.
- If performance is an issue, then use multiple routines, with a different routine optimized for each possible array alignment. You can use the assembler `MACRO` directive to generate these routines automatically.

6.10 Summary

For the best performance in an application you will need to write optimized assembly routines. It is only worth optimizing the key routines that the performance depends on. You can find these using a profiling or cycle counting tool, such as the ARMulator simulator from ARM.

This chapter covered examples and useful techniques for optimizing ARM assembly. Here are the key ideas:

- Schedule code so that you do not incur processor interlocks or stalls. Use Appendix D to see how quickly an instruction result is available. Concentrate particularly on load and multiply instructions, which often take a long time to produce results.
- Hold as much data in the 14 available general-purpose registers as you can. Sometimes it is possible to pack several data items in a single register. Avoid stacking data in the innermost loop.
- For small `if` statements, use conditional data processing operations rather than conditional branches.
- Use unrolled loops that count down to zero for the maximum loop performance.
- For packing and unpacking bit-packed data, use 32-bit register buffers to increase efficiency and reduce memory data bandwidth.
- Use branch tables and hash functions to implement efficient `switch` statements.
- To handle unaligned data efficiently, use multiple routines. Optimize each routine for a particular alignment of the input and output arrays. Select between the routines at run time.