Module 2

Introduction

We illustrate the processor operations using examples with pre- and post-conditions, describing registers and memory before and after the instruction or instructions are executed. We will represent hexadecimal numbers with the prefix 0x and binary numbers with the prefix 0b.

ARM Instruction set

#	Mnemonics	Description
1	ADC	add two 32-bit values and carry
2	ADD	add two 32-bit values
3	AND	logical bitwise AND of two 32-bit values
4	B	branch relative +/− 32 MB
5	BIC	logical bit clear (AND NOT) of two 32-bit values
6	BKPT	breakpoint instructions

7	BL	relative branch with link
8	BLX	branch with link and exchange
9	BX	branch with exchange
10	CDP CDP2	Coprocessor data processing operation
11	CLZ	count leading zeros
12	CMN	compare negative two 32-bit values
13	CMP	compare two 32-bit values
14	EOR	logical exclusive OR of two 32-bit values
15	LDC LDC2	load to coprocessor single or multiple 32-bit values
16	LDM	load multiple 32-bit words from memory to ARM registers
17	LDR	load a single value from a virtual address in memory
18	MCR MCR2 MCRR	move to coprocessor from an ARM register or registers
19	MLA	multiply and accumulate 32-bit values
20	MOV	move a 32-bit value into a register
21	MRC MRC2 MRRC	move to ARM register or registers from a coprocessor
22	MRS	move to ARM register from a status register (cpsr or spsr)
23	MSR	move to a status register (cpsr or spsr) from an ARM register
24	MUL	multiply two 32-bit values
25	MVN	move the logical NOT of 32-bit value into a register
26	ORR	logical bitwise OR of two 32-bit values
27	PLD	preload hint instruction
28	QADD	signed saturated 32-bit add
29	QDADD	signed saturated double and 32-bit add
30	QDSUB	signed saturated double and 32-bit subtract
31	QSUB	signed saturated 32-bit subtract
32	RSB	reverse subtract of two 32-bit values
33	RSC	reverse subtract with carry of two 32-bit integers

34	SBC	subtract with carry of two 32-bit values
35	SMLAxy	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
36	SMLAL	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
37	SMLALxy	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
38	SMLAWy	signed multiply accumulate instruction (((32 × 16) 16) + 32 = 32-bit)
39	SMULL	signed multiply long (32 × 32 = 64-bit)
40	SMULxy	signed multiply instructions (16 × 16 = 32-bit)
41	SMULWy	signed multiply instruction ((32 × 16) 16 = 32-bit)
42	STC STC2	store to memory single or multiple 32-bit values from coprocessor
43	STM	store multiple 32-bit registers to memory
44	STR	store register to a virtual address in memory
45	SUB	subtract two 32-bit values
46	SWI	software interrupt
47	SWP	swap a word/byte in memory with a register, without interruption
48	TEQ	test for equality of two 32-bit values
49	TST	test for bits in a 32-bit value
50	UMLAL	unsigned multiply accumulate long ((32 × 32) + 64 =64-bit)
51	UMULL	unsigned multiply long (32 × 32 = 64-bit)

ARM Inst ructions by Instruction class

1. Data processing instructions,
2. Branch instructions,
3. Load-store instructions,
4. Software interrupt instruction, and
5. Program status registers instructions.

3.1 Data Processing Instructions

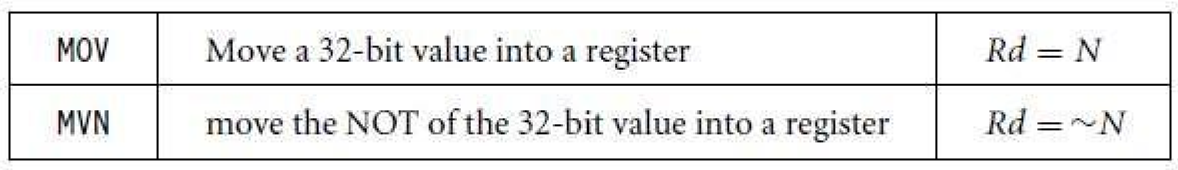
The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing instructions can process one of their operands using the barrel shifter.

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

MVN r7,r5   ;r7=~r5
```

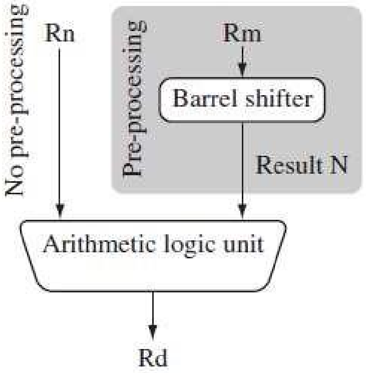
3.1.2 Barrel Shifter

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.



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.

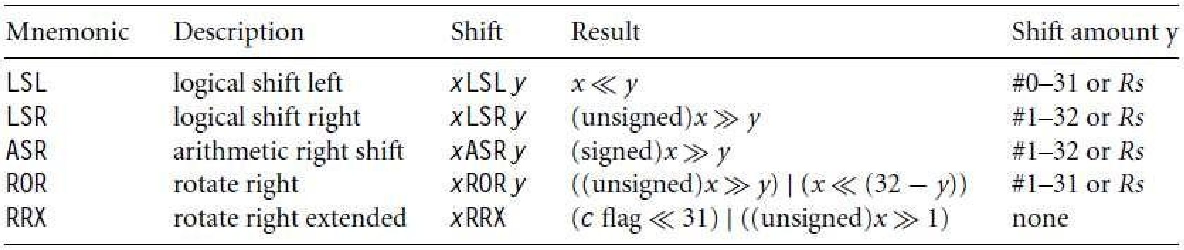
```
PRE     r5 = 5
        r7 = 8
MOV r7, r5, LSL #2 ; let r7 = r5*4 = (r5 << 2)
POST    r5 = 5
        r7 = 20
```

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 as 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 positionsis the same 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.

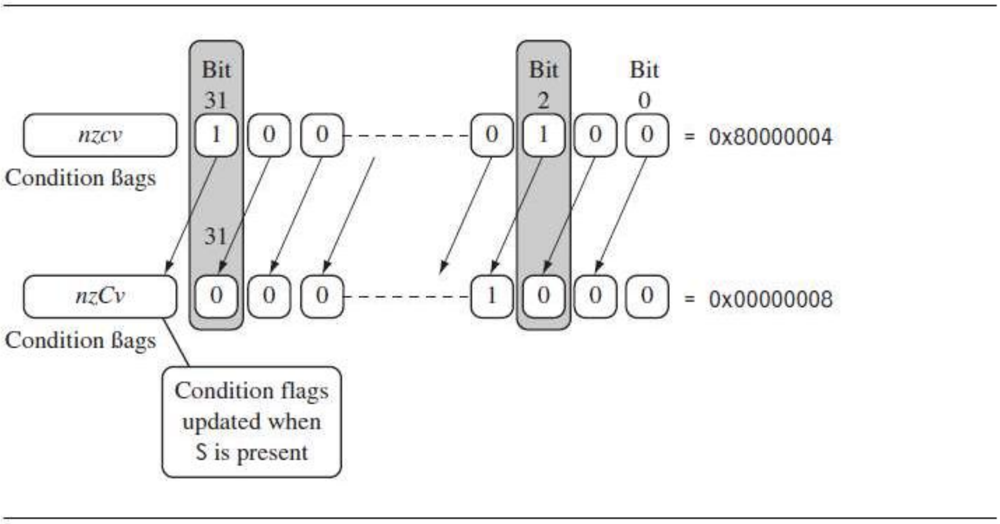
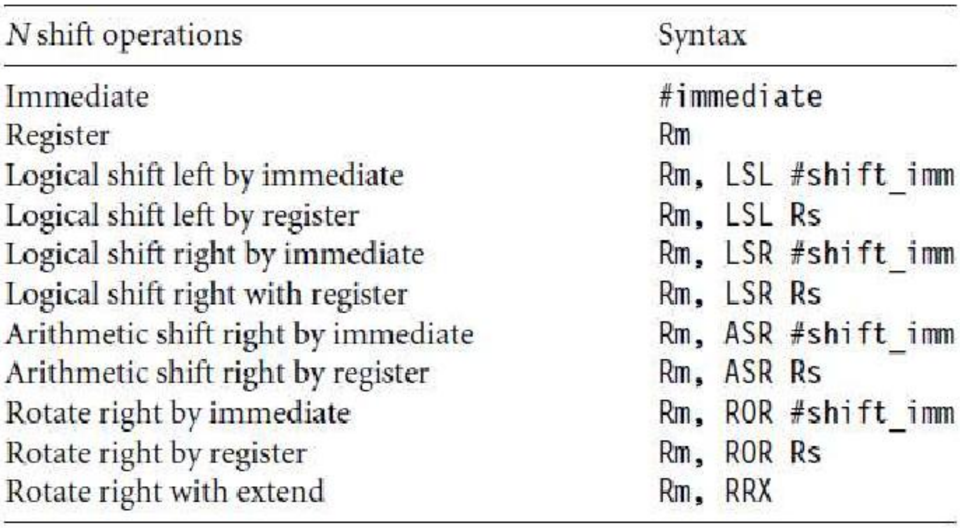


Figure 3.2 Logical shift left by one.

Table 3.3 Barrel shift operation syntax for data processing instructions.



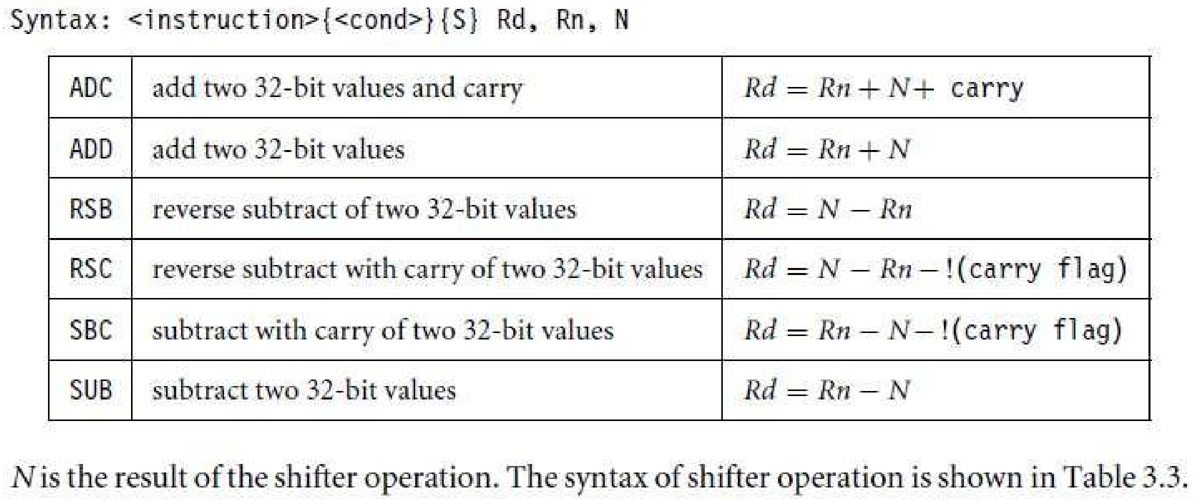
N shift operations	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
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 21. 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 proceeded 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 loopcounters. 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

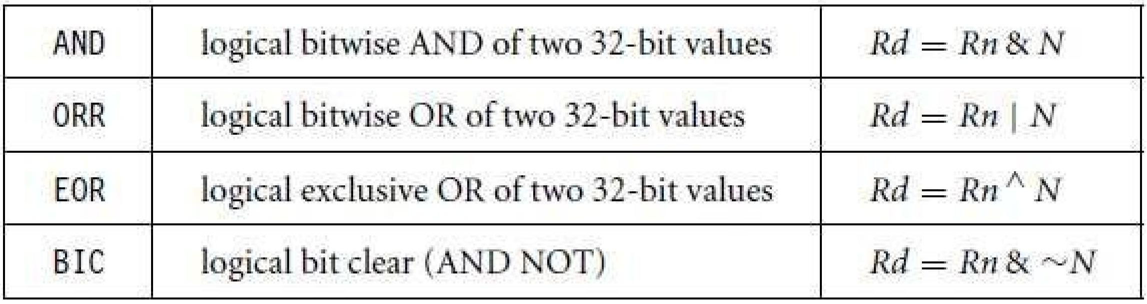
The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 bythree.

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* | *N*
EOR	logical exclusive OR of two 32-bit values	*Rd* = *Rn* ^ *N*
BIC	logical bit clear (AND NOT)	*Rd* = *Rn* & ~*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)

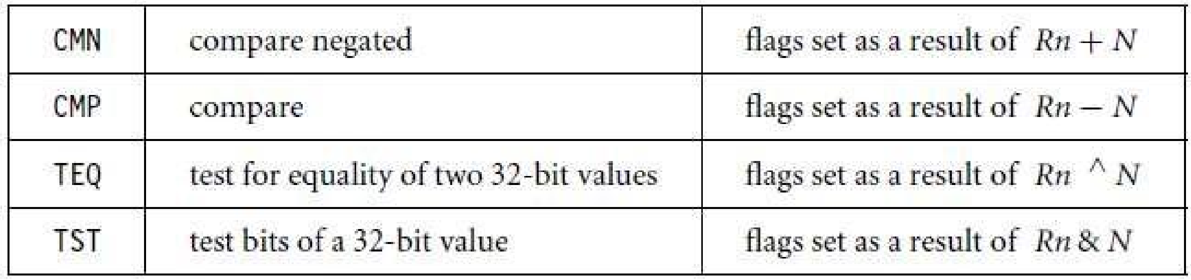
In this example, register r2 contains a binary pattern where every binary 1 in r2 clears a corresponding bit location in register r1. This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the cpsr.

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 ^ 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.

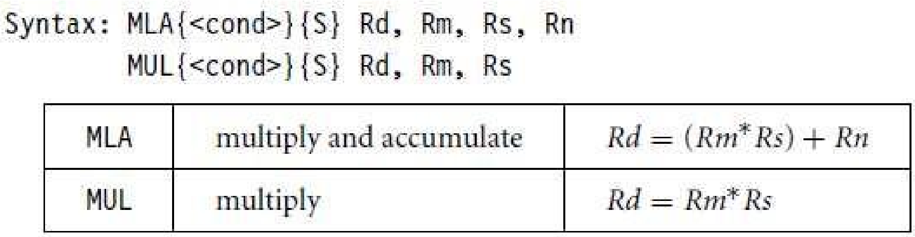
Example 3.10: This example shows a CMP comparison instruction. You can see that both registers, r0 and r9, are equal before executing the instruction. The value of the z flag prior to execution is 0 and is represented by a lowercase z. After execution the z flag changes to 1 or an uppercase Z. This change 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.

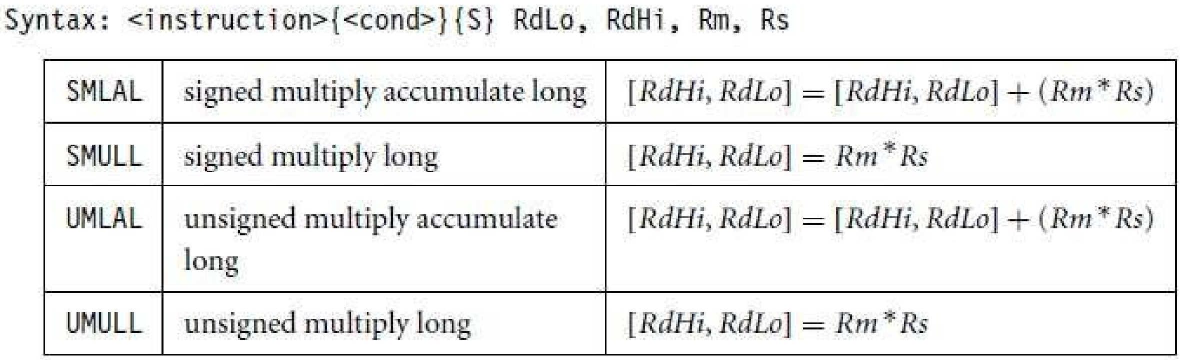
3.1.7 Multiply Instructions

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.

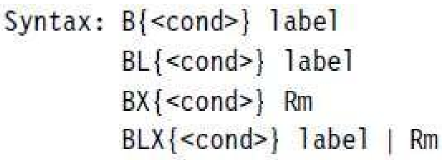
PRE r0 = 0x00000000, r1 = 0x00000000, r2 = 0xf0000002, r3 = 0x00000002

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

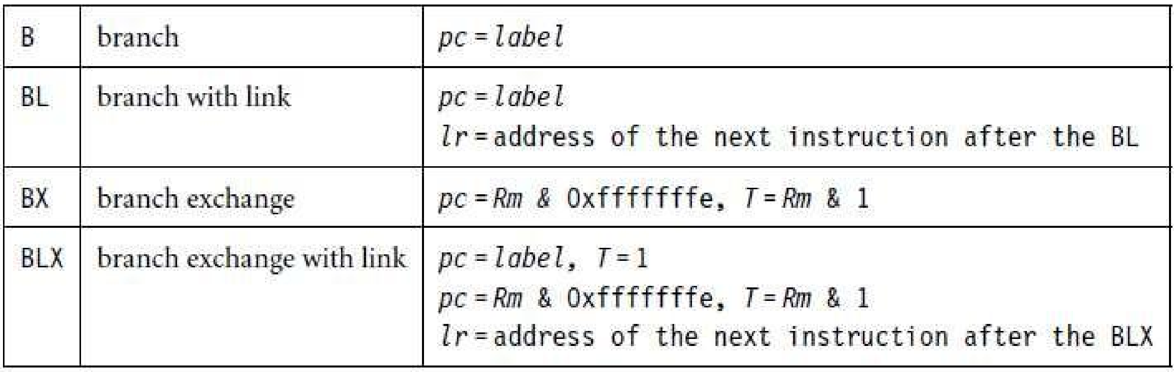
POST r0 = 0xe0000004 ; = RdLo, r1 = 0x00000001 ; = RdHi

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.



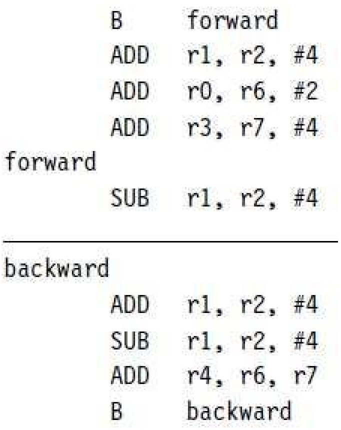
```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```



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.

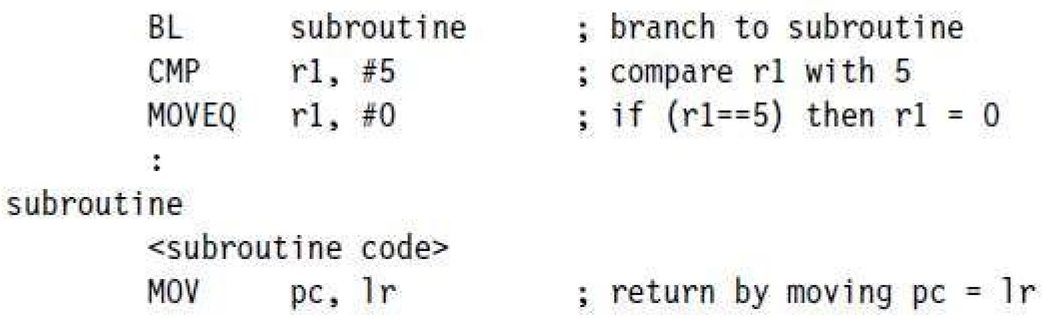
Example 3.13: This example shows a forward and backward branch. Because these loops are addressing specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.



```
        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.

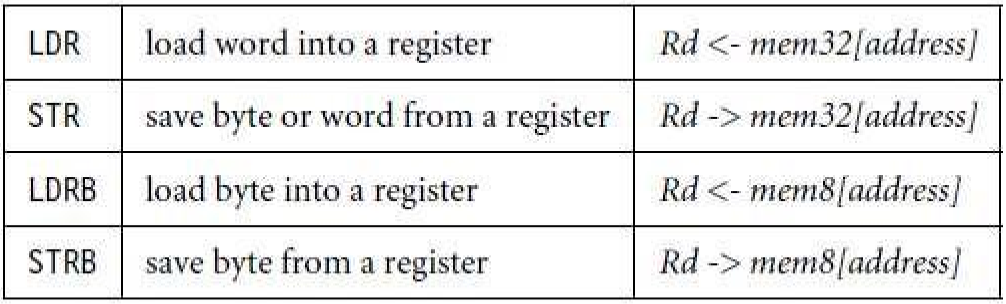
3.3 Load-Store Instructions

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.

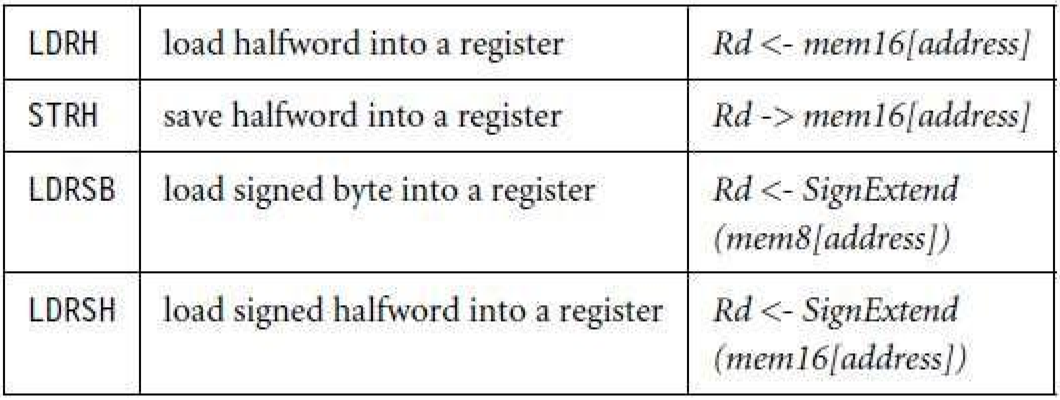
3.3.1 Single-Register Transfer

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,addressing
            LDR{<cond>}SB|H|SH Rd, addressing
            STR{<cond>}H Rd, addressing

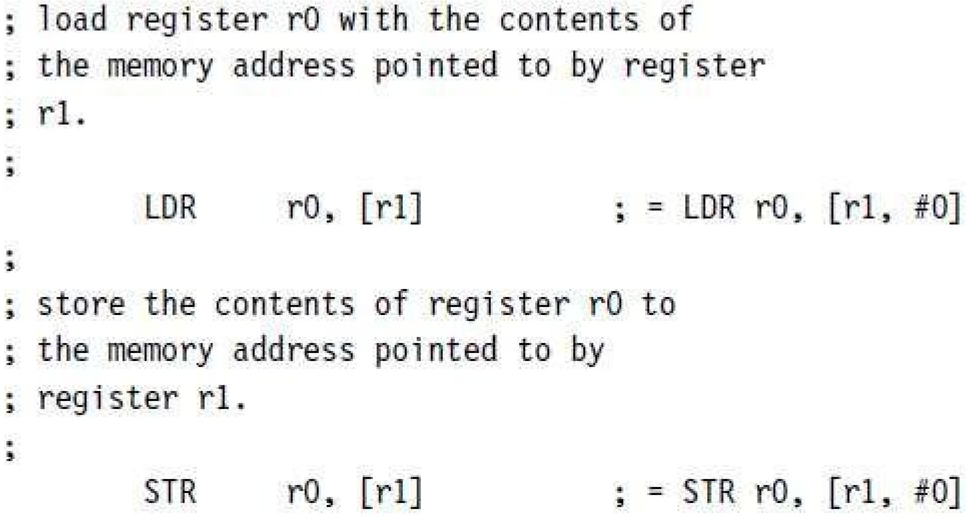


LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



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])*

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.



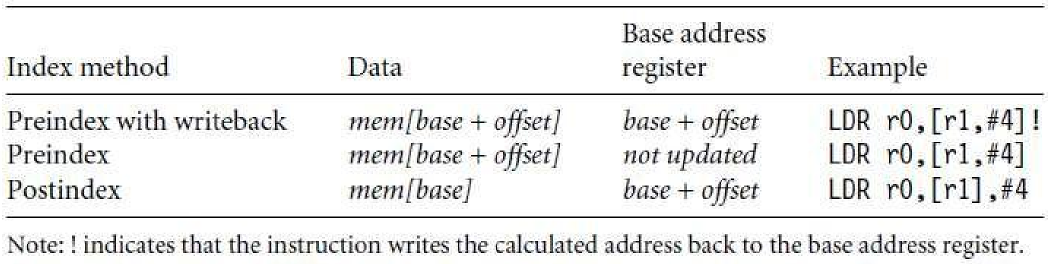
```
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
      LDR    r0, [r1]          ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
      STR    r0, [r1]          ; = STR r0, [r1, #0]
```

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.

Table 3.4 Index methods.

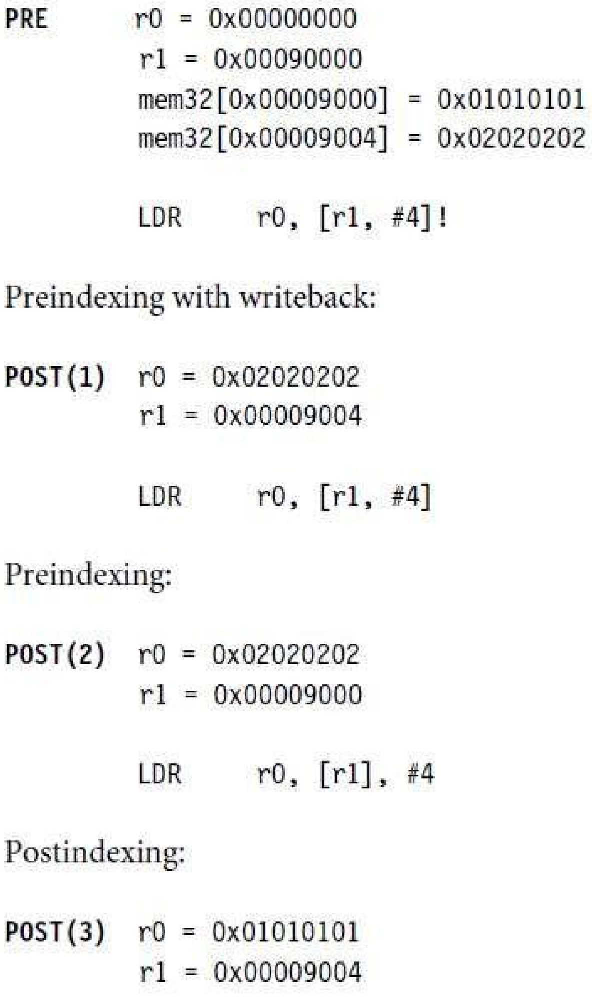


Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

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

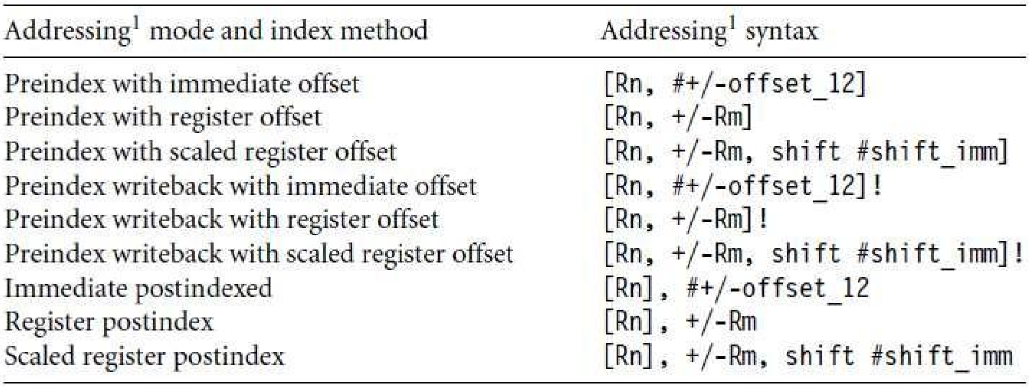
Example 3.16: Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only 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.

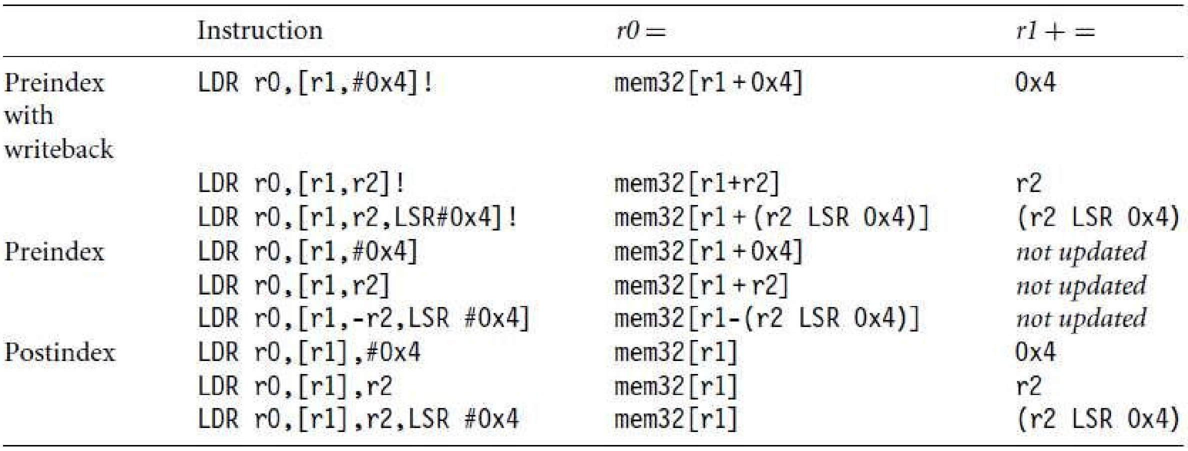
Table 3.6 provides an example of the different variations of the LDR instruction. Table 3.7 shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data.

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

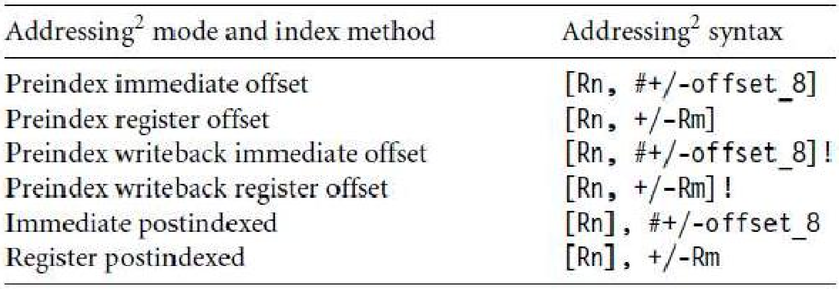
Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register Rn pointing into memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks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

Table 3.6 Examples of LDR instructions using different addressing modes.



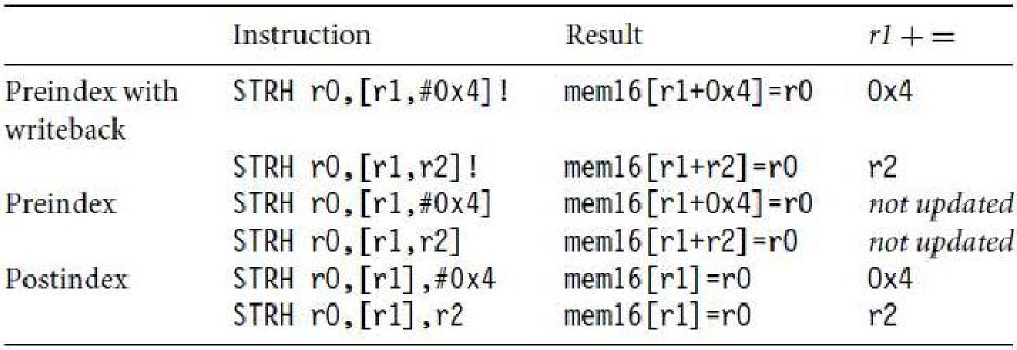
	Instruction	r0 =	r1 +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	mem32[r1+0x4]	0x4
	`LDR r0,[r1,r2]!`	mem32[r1+r2]	r2
	`LDR r0,[r1,r2,LSR#0x4]!`	mem32[r1+(r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	`LDR r0,[r1,#0x4]`	mem32[r1+0x4]	*not updated*
	`LDR r0,[r1,r2]`	mem32[r1+r2]	*not updated*
	`LDR r0,[r1,-r2,LSR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

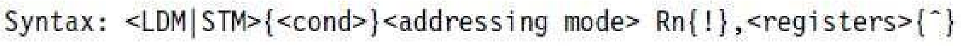
Table 3.8 Variations of STRH instructions.



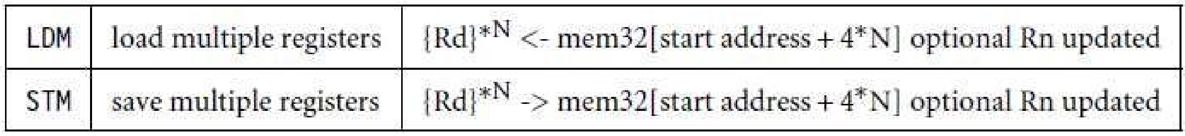
	Instruction	Result	r1 +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	mem16[r1+0x4]=r0	0x4
	`STRH r0,[r1,r2]!`	mem16[r1+r2]=r0	r2
Preindex	`STRH r0,[r1,#0x4]`	mem16[r1+0x4]=r0	*not updated*
	`STRH r0,[r1,r2]`	mem16[r1+r2]=r0	*not updated*
Postindex	`STRH r0,[r1],#0x4`	mem16[r1]=r0	0x4
	`STRH r0,[r1],r2`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

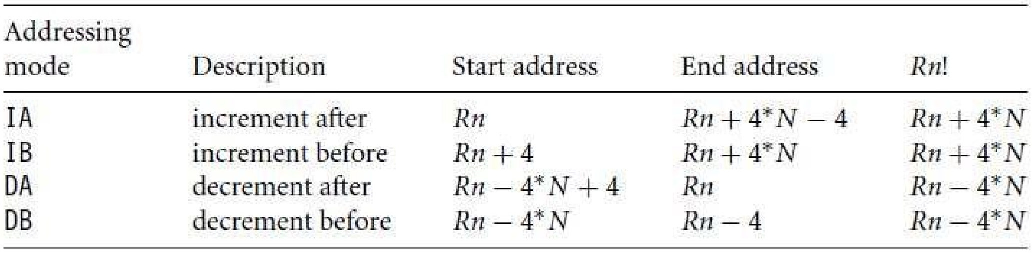


LDM	load multiple registers	{Rd}^{*N} <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}^{*N} -> mem32[start address + 4*N] optional Rn updated

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 loadstore 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

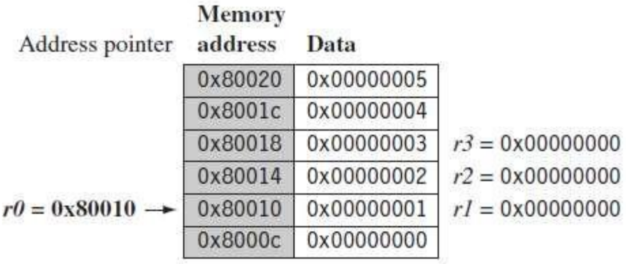
Figure 3.3 shows a graphical representation.

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.

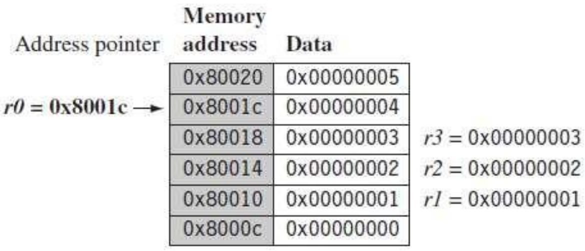
After execution, register r0 now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.

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.



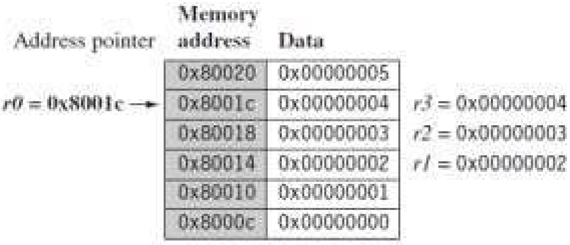
Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
r0 = **0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

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	

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	

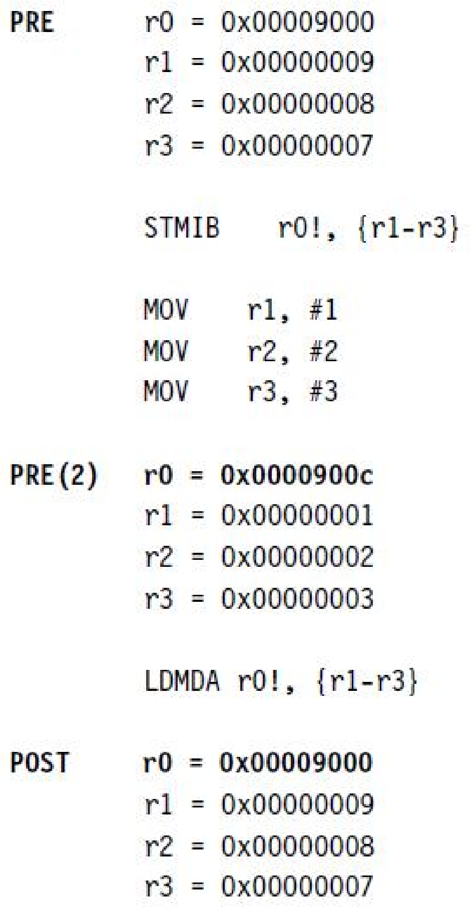
Post-condition for LDMIB instruction.

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.

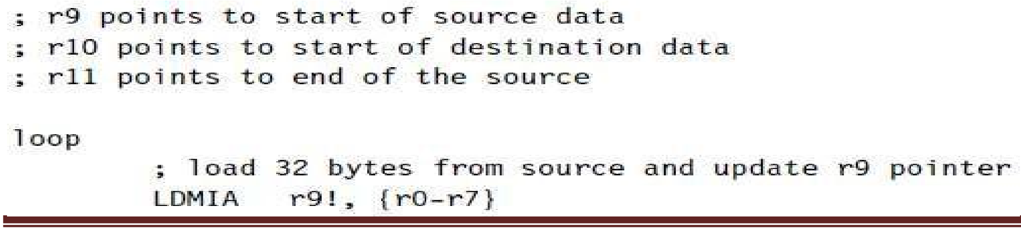
Example 3.18: This example shows an STM increment before instruction followed by an LDM decrement after instruction.



```
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}
```

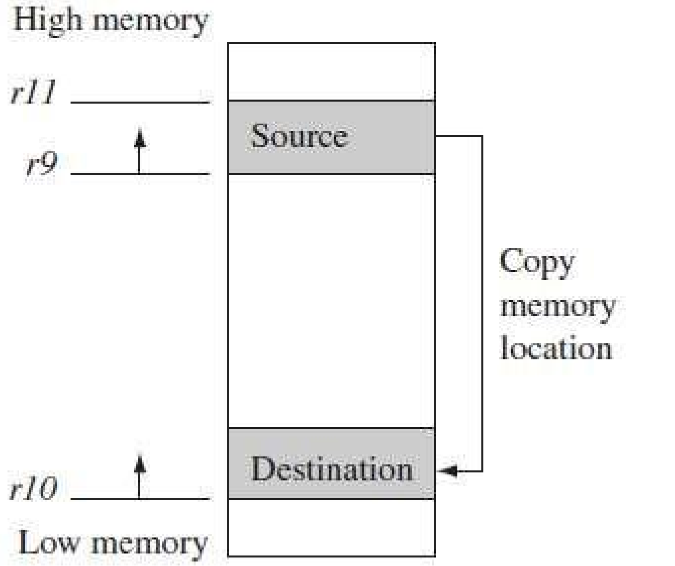
```
; store 32 bytes to destination and update r10 pointer
STMIA   r10!, {r0-r7} ; and store them

; 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.



Block memory copy in the memory map.

3.3.3.1 Stack Operations

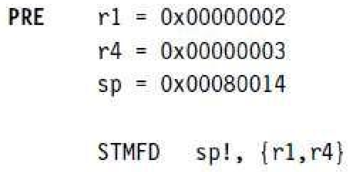
The ARM architecture uses the load-store multiple instructions to carry out stack operations. The pop operation (removing data from a stack) uses a load multiple instruction; similarly, the push operation (placing data onto the stack) uses a store multiple instruction. When using a stack you have to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

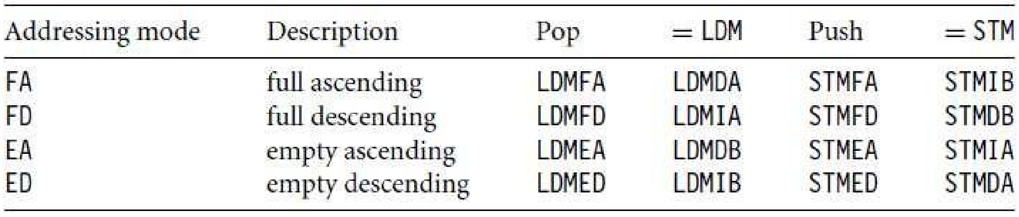
ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the pop and push functions, respectively.

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.

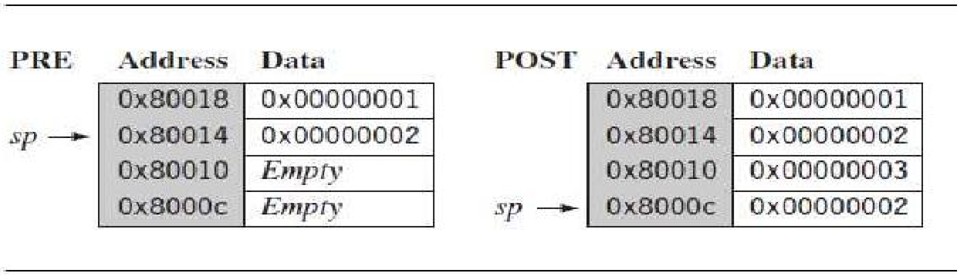


```
PRE    r1 = 0x00000002
       r4 = 0x00000003
       sp = 0x00080014

       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
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST

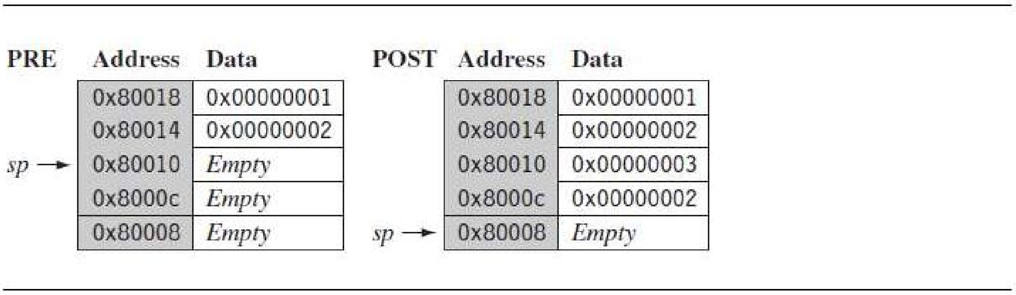
	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

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
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
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.

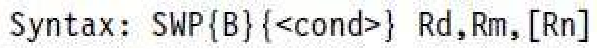
If the stack pointer passes the stack limit, then a stack overflow error has occurred. Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow
SUB sp, sp, #size
CMP sp, r10
BLLO _stack_overflow ; condition
```

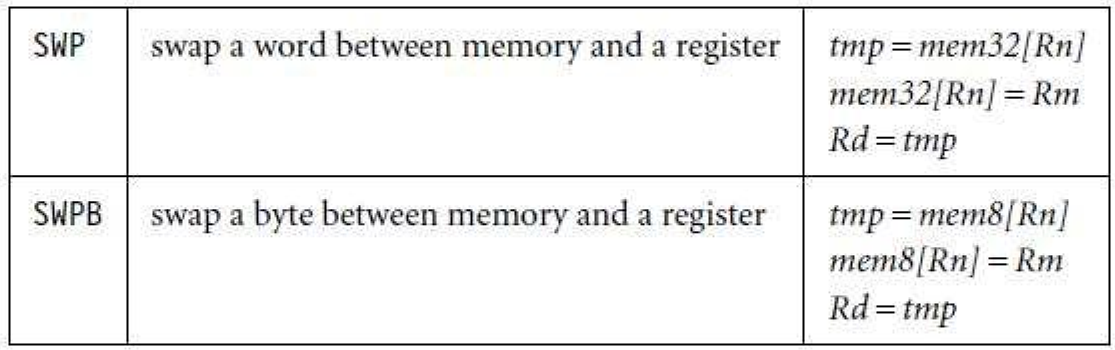
ATPCS defines register r10 as the stack limit or sl. This is optional since it is only used when stack checking is enabled. The BLLO instruction is a branch with link instruction plus the condition mnemonic LO. If sp is less than register r10 after the new items are pushed onto the stack, then stack 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]
```

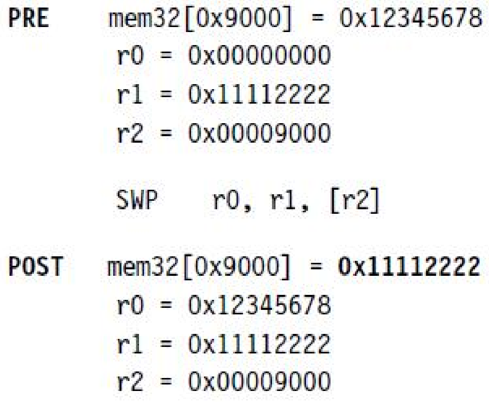


SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

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.

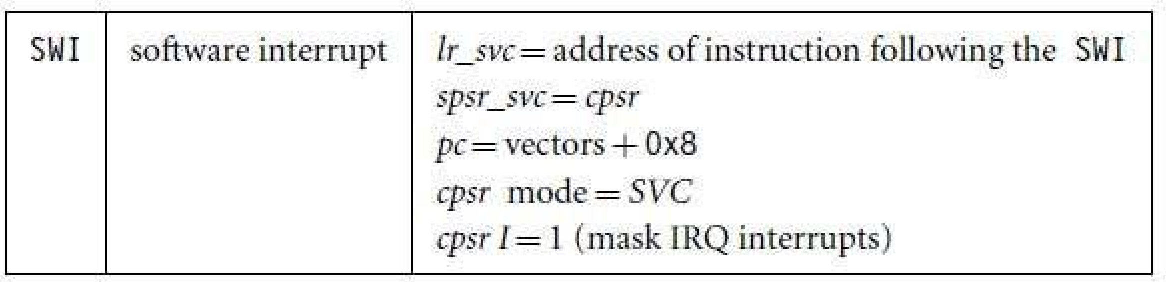
```
spin
MOV r1, =semaphore
MOV r2, #1
SWP r3, r2, [r1] ; hold the bus until complete
CMP r3, #1
BEQ spin
```

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

Asoftware 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}^
```

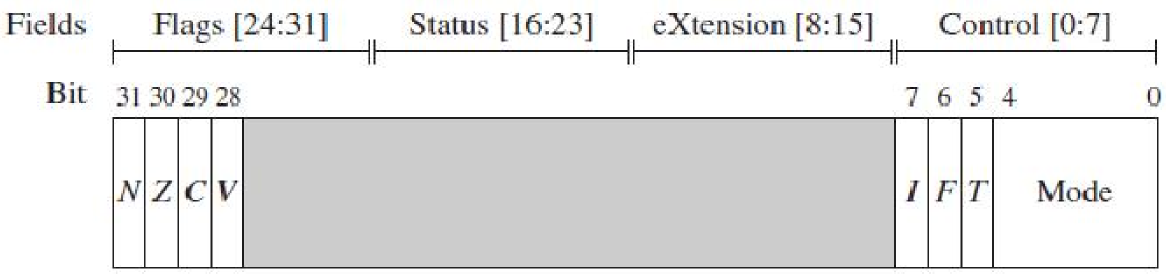
The number in register r10 is then used by the SWI handler to call the appropriate SWI service routine.

3.5 Program Status Register Instructions

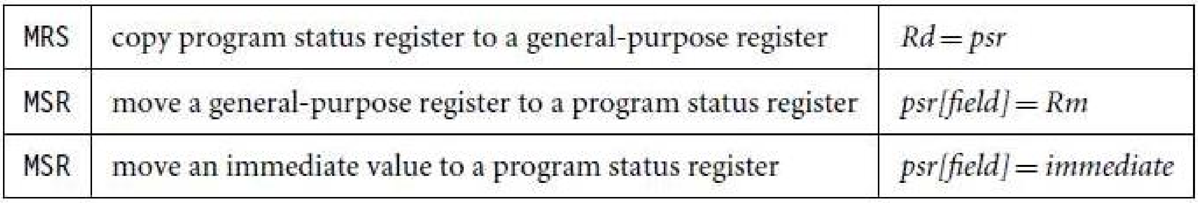
The ARM instruction set provides two instructions to directly control a program status register (psr). The MRS instruction transfers the contents of either the cpsr or spsr into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the cpsr or spsr. Together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



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
```

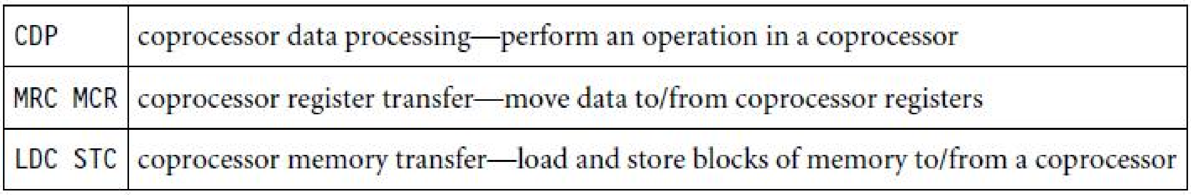
POST cpsr = nzcvqiFt_SVC

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
```

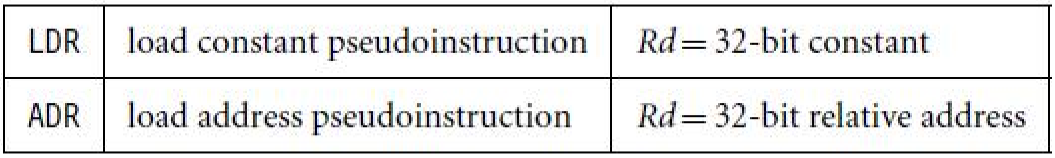
Here CP15 register-0 contains the processor identification number. This register is copied into the general-purpose register r10.

3.6 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.

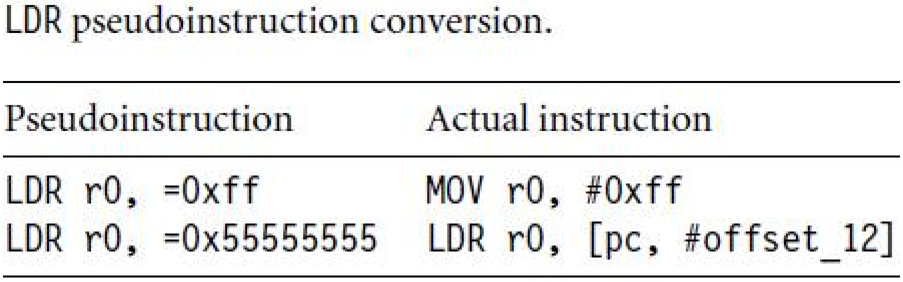
The second pseudoinstruction writes a relative address into a register, which will be encoded using a pc-relative expression.

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.



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.

6. ARM programming using Assembly language:

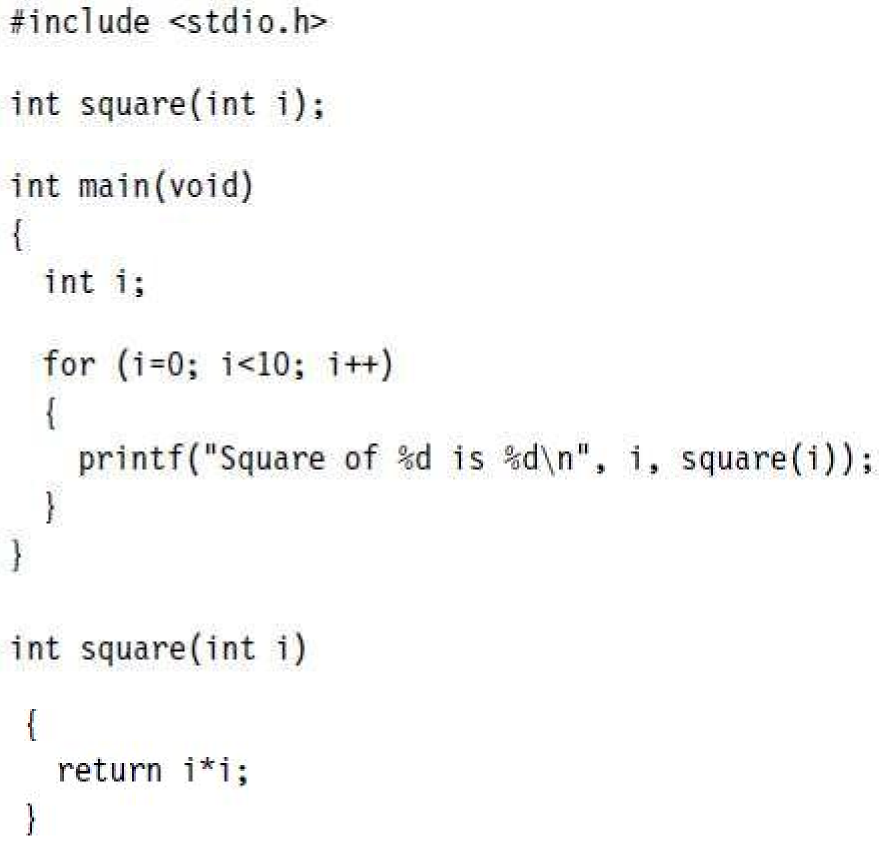
6.1 Writing and Optimizing ARM Assembly Code

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.

Writing assembly by hand gives you direct control of three optimization tools that you cannot explicitly use by writing C source:

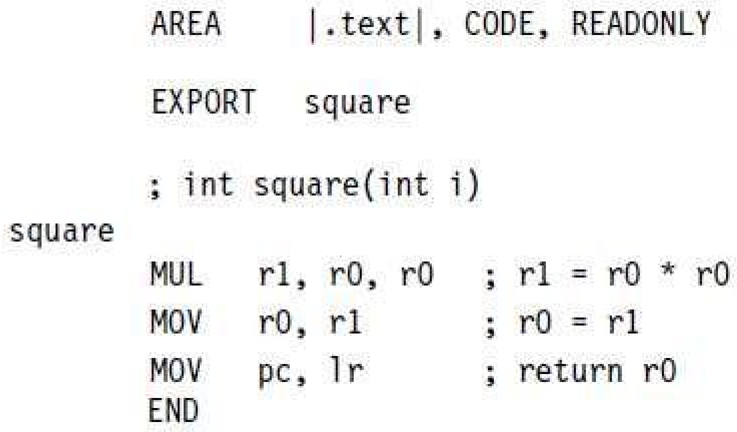
- **Instruction scheduling**: Reordering the instructions in a code sequence to avoid processor stalls. Since ARM implementations are pipelined, the timing of an instruction can be affected by neighboring instructions.
- **Register allocation:** Deciding how variables should be allocated to ARM registers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

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.

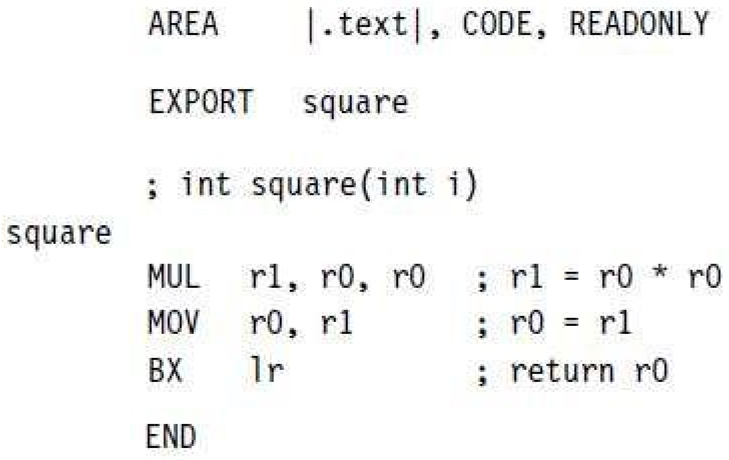
The EXPORT directive makes the symbol square available for external linking. At line six we define the symbol square as a code label. Note that armasm treats nonindented text as a label definition. When square is called, the parameter passing is defined by the ATPCS (see Section 5.4). The input argument 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:



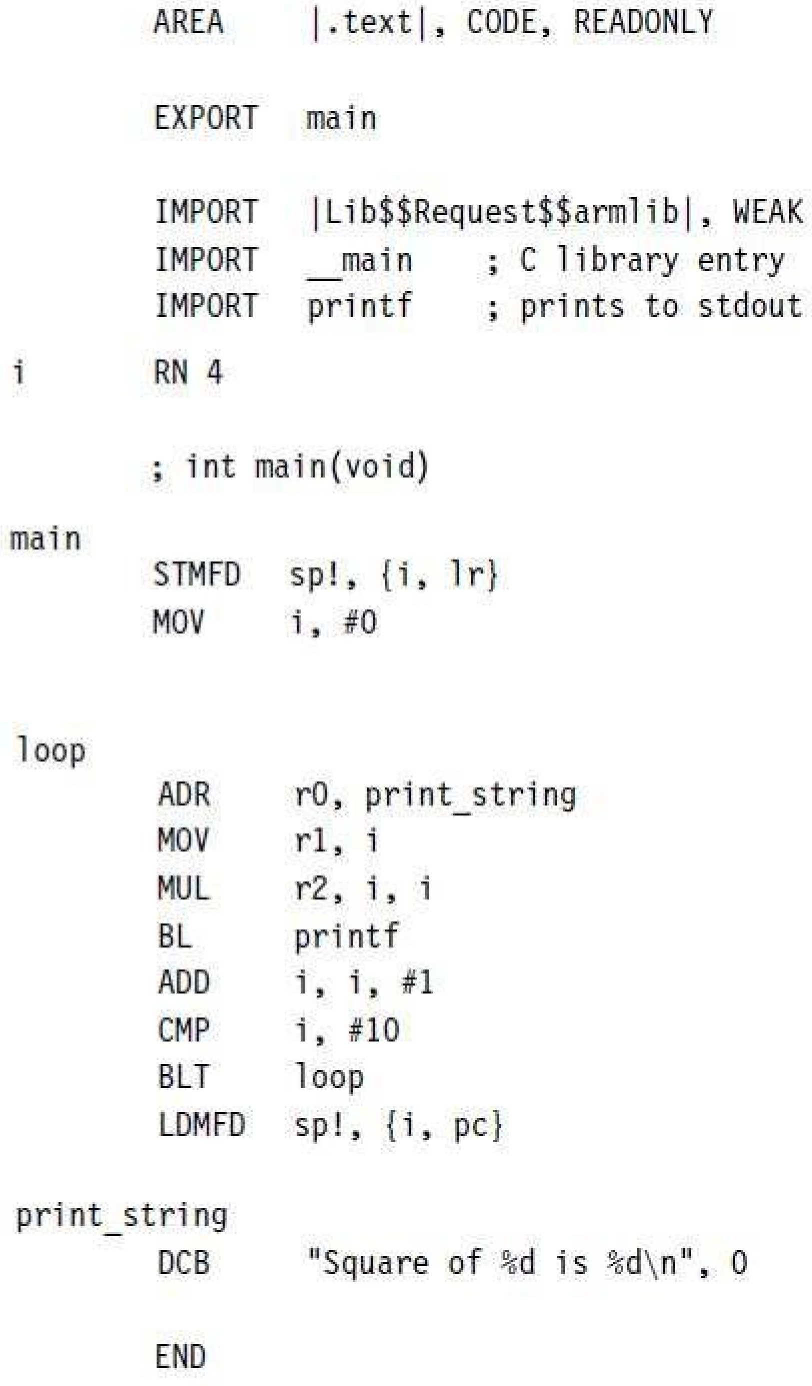
```
        AREA    |.text|, CODE, READONLY

        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

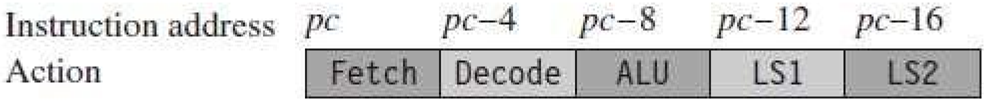
The time taken to execute instructions depends on the implementation pipeline. For this chapter, we assume ARM9TDMI pipeline timings. You can find these in Section D.3 of Appendix D. The following rules summarize the cycle timings for common instruction classes on the ARM9TDMI.
Instructions that are 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 writesto the pc, then add two cycles.
- Load instructions that loadN32-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 foran 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.



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 wasoriginally 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.

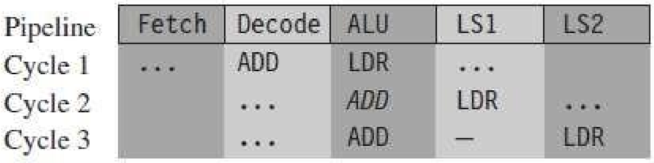
To work out how many cycles a block of code will take, use the tables in Appendix D that summarize the cycle timings and interlock cycles for a range of ARM cores.

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 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 these stalls by 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.

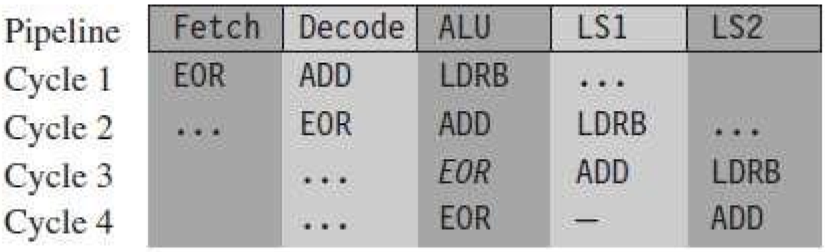


Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	*ADD*	LDR	...
Cycle 3		...	ADD	—	LDR

Figure 6.2 One-cycle interlock caused by load use.

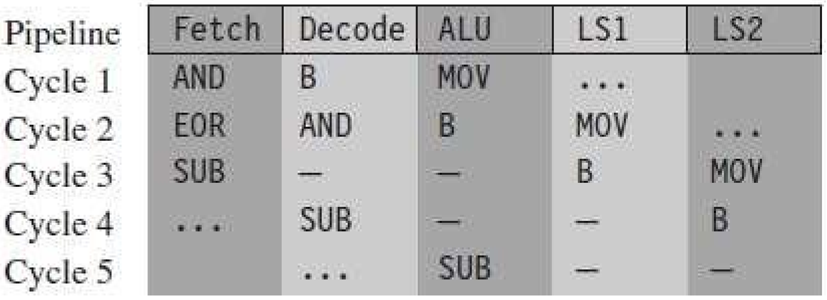
Example 6.7
This example shows a one-cycle interlock caused by delayed load use.
LDRB r1, [r2, #1]
ADD r0, r0, r2
EOR r0, r0, r1
This instruction triplet takes four cycles. Although the ADD proceeds on the cycle following the load byte, the EOR instruction cannot start on the third cycle. The r1 value 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 onethird 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);
}
```

```
        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
```

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

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.

- 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.
- 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 callingsubroutines. 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 ≤ 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 2k . 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}
```

Nowto 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 RN2
k RN3
```

For the load multiple to work correctly, we must assign x0 through x7 to sequentially increasing register numbers, and similarly for y0 through y7. Notice that we finish with x0 before starting with y1. In general, we can assign xn to the same register as yn+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 y0 and then move it to y1 when x0 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 3B{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
```

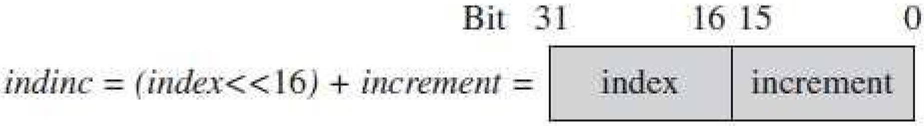
6.4.3 Making the Most of Available Registers

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:



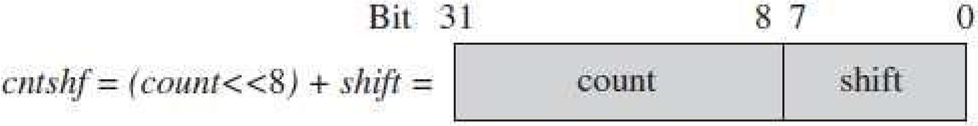
The C code translates into assembly code using a single register to hold indinc:

```
        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:



```
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 xn, yn, and zn denote the nth 8-bit pixel in these images, respectively. Let 0 ≤ a ≤ 256 be a scaling factor. To merge the images, we set

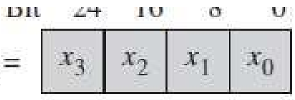
$$z_n = (ax_n + (256 - a)y_n)/256 \qquad (6.1)$$

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

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

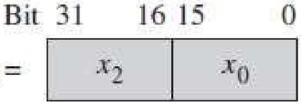
Therefore each pixel requires a subtract, a multiply, a shifted add, and a right shift. To process multiple pixels at a time, we load four pixels at once using a word load. We use a bracketed notation to denote several values packed into the same word:

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



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

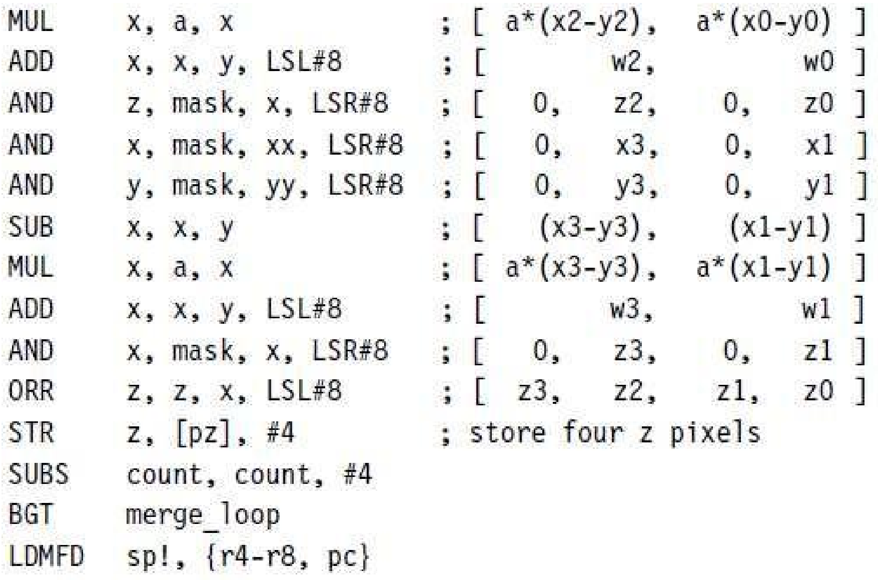
$[x2, x0] = x_2 2^{16} + x_0 =$



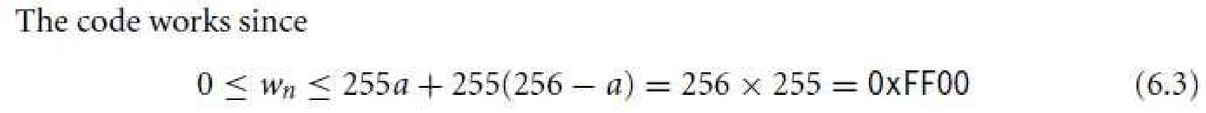
Note that even for signed values $[a, b] + [c, d] = [a + b, c + d]$ if we interpret $[a, b]$ using the mathematical equation $a2^{16} + b$. Therefore we can perform SIMD operations on these values using normal arithmetic instructions.

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 [$w2$, $w0$] into $w2$ and $w0$ 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 ≤ i ≤ 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 ≤ c ≤ 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.
- You can implement if statements with the logical AND or OR of several similar conditions using compare instructions that are themselves conditional.

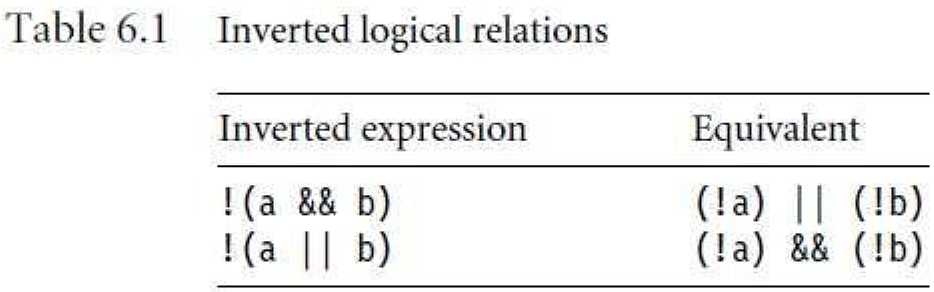


Table 6.1 Inverted logical relations

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

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 fromN to 1 inclusive. The loop terminates with i = 0. An efficient implementationis

```
          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 T1 and only bother to align the array when N ≥ T1. Clearly T1 ≥ 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 ≥ T2 ≥ 128, where T2 is another threshold to be determined later on.

Finally, we are left with N < T2 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 T1 and T2.

```
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
```

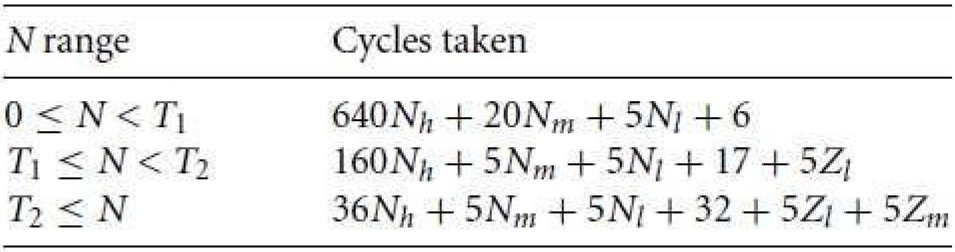
It remains to find the best values for the thresholds T1 and T2. To determine these we need to analyze the cycle counts for different ranges of N. Since the algorithm operates on blocks of size 128 bytes, 4 bytes, and 1 byte, respectively, we start by decomposing N with respect to these block sizes:

$N = 128Nh + 4Nm + Nl$, where $0 \leq Nm < 32$ and $0 \leq Nl < 4$

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 $0 \leq N < T1$: The routine takes $5N + 6$ cycles on an ARM9TDMI including the return.
- Case $T1 \leq N < T2$: 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 block takes $5(32Nh + Nm) + 5(Nl + Zl) + 2$ cycles, where Zl is 1 if $Nl = 0$, and 0 otherwise. The total cycles for this case is $5(32Nh + Nm + Nl + Zl) + 17$.
- Case $N \geq T2$: As in the previous case, the first algorithm block averages 9 cycles. The second algorithm block takes $36Nh + 21$ cycles. The final algorithm block takes $5(Nm + Zm + Nl + Zl) + 2$ cycles, where Zm is 1 if Nm is 0, and 0 otherwise. The total cycles for this case is $36Nh + 5(Nm + Zm + Nl + Zl) + 32$.

Table 6.2 Cycles taken for each range of *N* values.



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

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 Nm ≥ 1, unless Nm = 1 and Nl = 0. We set T1 = 5 to choose the best cycle counts from rows one and two. The third row wins over the second row as soon as Nh ≥ 1. Therefore take T2 = 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

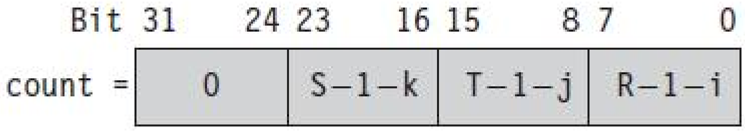
&A[i,j] = a + 4*(i*T+j)

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 216 to clear bits 16 to 31. Then we subtract 28 to decrement the count stored at bits 8 to 15, implementing the j loop. We can encode the constant 216 − 28 = 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 2N 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 over unroll 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.

Module 2 Question Bank

1. Explain conditional execution with an example.
2. Explain MAC unit with an example.
3. Explain Barrel shifter with a neat sketch.

4. Explain 5 different shift operations that can be used with Barrel shifter.
5. List compare instructions & Write the useful of AND, ORR, EOR instructions
6. Describe the difference between ADR & ADRL
7. List the data processing instructions with one example each.
8. Explain stack operation using STM & LDM instructions.
9. Explain SWAP & SWI instructions with example
10. Explain AND & EOR instructions with example
11. Explain TST & TEQ instructions with example
12. Write a note on software interrupt instruction.
13. Predict the operation performed by the execution of each compare instruction
14. Calculate the effective address of the following instructions if registerR3=0x4000 and register R4=0x20

    (i) STRH R9,[R3,R4]
    (ii) LDRB R8,[R3,R4,LSL #3]
    (iii)LDR R7,[R3],R4
    (iv) STRB R6,[R3],R4,ASR #2
    (v) LDR R0,[R3,-R4,LSL #3]

15. Test whether the following instruction are pre or post indexed addressing mode

    (i) STR R6,[R4,#4]
    (ii) LDR R3,[R12],#6
    (iii)LDRB R4,[R3,R2]
    (iv)LDR R6,[R0,R1,ROR #6]
    (v) STR R3,[R0,R5,LSL #3]

16. Write the instruction to perform the following operations.

    (i) Add number 256 to R1, place the sum in register R2
    (ii) Place a 2‟s complement of -1 into register R3
    (iii) ANDing , R1 content with the complement of 256,place the result in register R2
    (iv) To returning from subroutine
    (v) Copy a complement of 4 into R1

17. Write a note on Profiling and Cycle Counting.
18. Brief about the categories of Load-Store instructions used with ARM.
19. Explain the ARM Single-Register and Multiple-Register load-store addressing modes with example.
20. Explain Co-Processor instructions of ARM Processor.
21. Explain the MOV instruction set provided by ARM7 with the example for each.