Unit – 2 10 Hours

Introduction to the ARM Instruction Set: Data Processing Instructions, Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants.

C Compilers and Optimization: Basic C Data Types, C Looping Structures, Register Allocation, Function Calls, Pointer Aliasing..

Table 3.1 ARM instruction set.

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

continued

ARM instruction set. (Continued)

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

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.

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 = ~N

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
```

Barrel Shifter :

In above example, MOV instruction has used N, which is a simple register. But it 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.
- Example : MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) .

Barrel shifter and ALU :

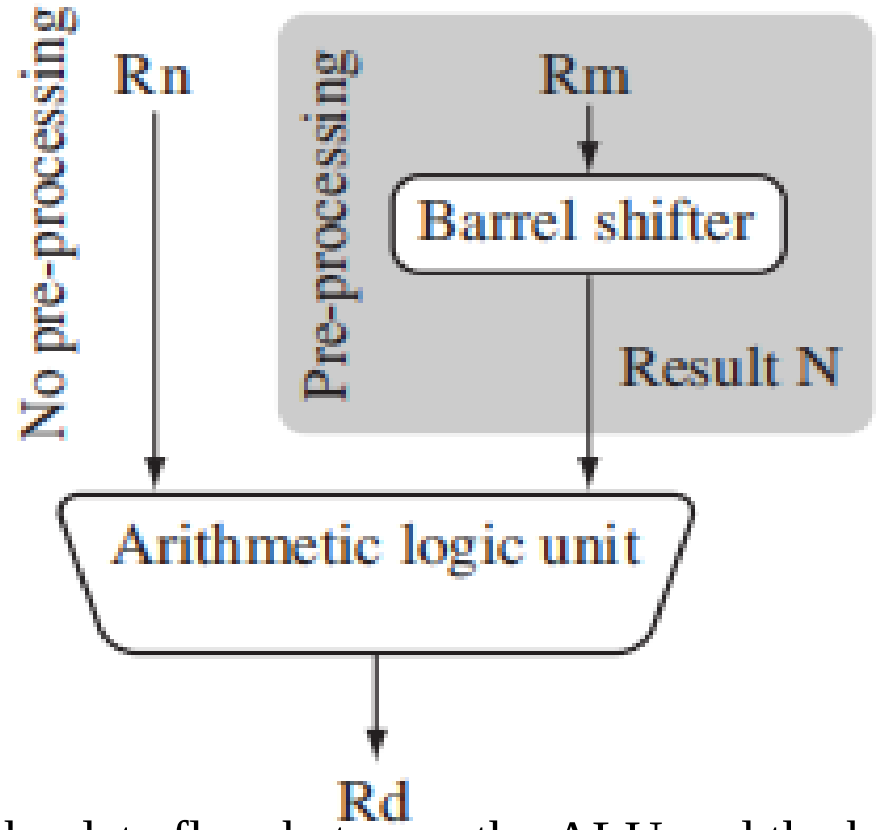


Figure shows the data flow between the ALU and the barrel shifter. It adds a shift operation to the move instruction and Register Rn enters the ALU without any preprocessing of registers.

Example : 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.

PRE r5 = 5
r7 = 8

MOV r7, r5, LSL #2 ; let r7 = r5*4 = (r5 << 2)

POST r5 = 5
r7 = **20**

Above example multiplies register r5 by four and then places the result into register r7.

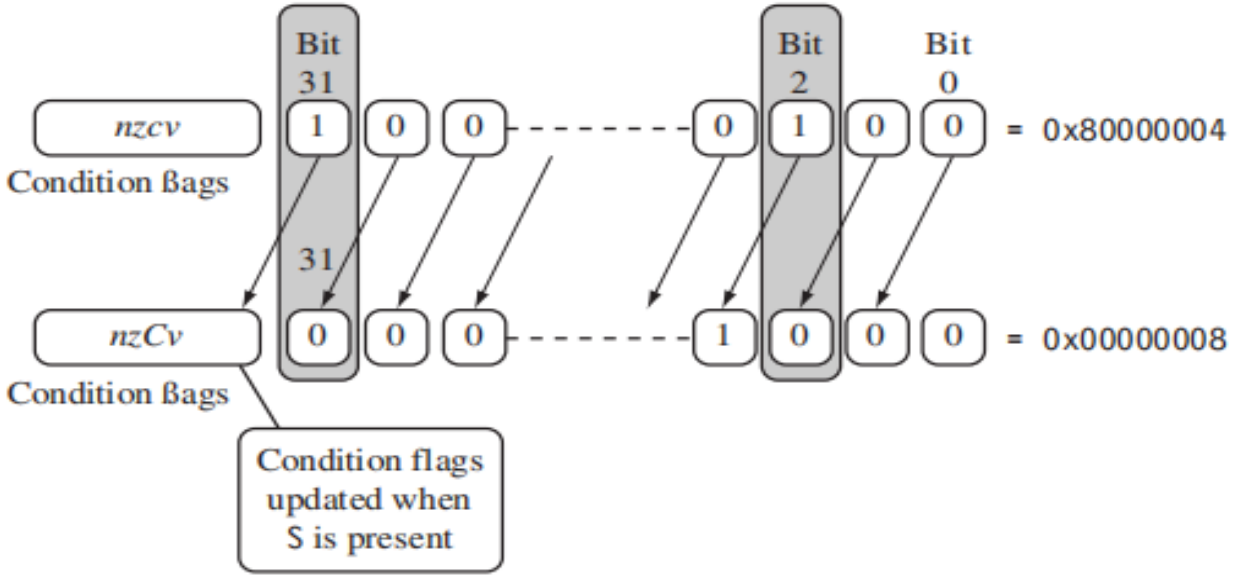
Five different shift operations that can be used within the barrel shifter are summarized below.

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*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

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

Logical shift left by one :



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

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

This example is a MOVS instruction that 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

Above Table lists the syntax for the different barrel shift operations available on data processing instructions.

- The second operand N can be an immediate constant preceded by #, a register value Rm, or the value of Rm processed by a shift.

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

PRE cpsr = nzcvqiFt_USER
r1 = 0x00000001
SUBS r1, r1, #1
POST cpsr = nZCvqiFt_USER
r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions

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

Example : 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

Logical Instructions:

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

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

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

Example : 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 : 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.

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. N is the result of the shifter operation

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*

Example : 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.

Multiply Instructions :

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	$[RdHi, RdLo] = Rm * Rs$

Example : This example shows a simple multiply instruction that multiplies registersr1 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: 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

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

```
            B forward
            ADD r1, r2, #4
            ADD r0, r6, #2
            ADD r3, r7, #4
forward
            SUB r1, r2, #4
backward
            ADD r1, r2, #4
             SUB r1, r2, #4
             ADD r4, r6, r7
             B backward
```

- 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, 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.

■

Software Interrupt Instruction

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

Syntax: SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

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

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

Example

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 = nzcVqi ft_USER
pc = 0x00008000
lr = 0x003fffff; lr = r14
r0 = 0x12
0x00008000 SWI 0x123456

POST

cpsr = **nzcVqIft_SVC**
spsr = **nzcVqi ft_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 : 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 fifirst 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.

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

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.

Loading Constants :

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 pseudo instructions 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 pseudo instruction 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 pseudo instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example

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.

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

Table 3.12 LDR pseudoinstruction conversion.

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

Example

Loading the constant 0xff00ffff using an MVN.

PRE

none...

MVN r0, #0x00ff0000

POST

r0 = **0xff00ffff**

There are no of alternatives to accessing memory, but they depend upon the constant you are trying to load. 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.

Above Table 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 pseudo instruction 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 pseudo instruction 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.

Overview of C Compilers and Optimization :

C compilers have to translate your C function literally into assembler so that it works for all possible inputs. In practice, many of the input combinations are not possible or won't occur. Let's start by looking at an example of the problems the compiler faces. The memclr function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{
for (; N>0; N--)
{
*data=0;
data++;
}
}
```

No matter how advanced the compiler, it does not know whether N can be 0 on input or not. Therefore the compiler needs to test for this case explicitly before the first iteration of the loop.
The compiler doesn't know whether the data array pointer is four-byte aligned or not.
If it is four-byte aligned, then the compiler can clear four bytes at a time using an int store rather than a char store. Nor does it know whether N is a multiple of four or not. If N is a multiple of four, then the compiler can repeat the loop body four times or store four bytes at a time using an int store.

Basic C Data Types

ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture. In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store a signed or unsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store a signed or unsigned 64-bit value

loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store.

C compiler datatype mappings.

C Data Type	Implementation
`char`	unsigned 8-bit byte
`short`	signed 16-bit halfword
`int`	signed 32-bit word
`long`	signed 32-bit word
`long long`	signed 64-bit double word

Local Variable Types

ARMv4-based processors can effiiciently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, int or long, for local variables wherever possible. Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value. The one exception is when you want wrap-around to occur. If you require modulo arithmetic of the form 255 + 1 = 0, then use the char type.

Example : The following code checksums a data packet containing 64 words. It shows why you should avoid using char for local variables.

```
int checksum_v1(int *data)
{
char i;
int sum = 0;
for (i = 0; i < 64; i++)
{
sum += data[i];
}
return sum;
}
```

At first sight it looks as though declaring i as a char is efficient. You may be thinking that a char uses less register space or less space on the ARM stack than an int. On the ARM, both these assumptions are wrong. All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255. Any attempt to increment 255 should produce the answer 0.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
        MOV r2,r0          ; r2 = data
        MOV r0,#0          ; sum = 0
        MOV r1,#0          ;i=0
checksum_v1_loop
        LDR r3,[r2,r1,LSL #2]    ; r3 = data[i]
        ADD r1,r1,#1             ; r1 = i+1
        AND r1,r1,#0xff          ; i = (char)r1
         CMP r1,#0x40            ; compare i, 64
         ADD r0,r3,r0            ; sum += r3
BCC checksum_v1_loop             ; if (i<64) loop
MOV pc,r14                       ; return sum
```

Now compare this to the compiler output where instead we declare i as an unsigned int.

```
checksum_v2
                MOV r2,r0                ; r2 = data
                 MOV r0,#0               ; sum = 0
                 MOV r1,#0               ;i=0
checksum_v2_loop
                 LDR r3,[r2,r1,LSL #2]   ; r3 = data[i]
                 ADD r1,r1,#1            ; r1++
                 CMP r1,#0x40            ; compare i, 64
                 ADD r0,r3,r0            ; sum += r3
BCC checksum_v2_loop                     ; if (i<64) goto loop
                 MOV pc,r14              ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce i to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum. It is tempting to write the following C code:

```
short checksum_v3(short *data)
{
unsigned int i;
short sum = 0;
for (i = 0; i < 64; i++)
{
sum = (short)(sum + data[i]);
}
return sum;
}
```

You may wonder why the for loop body doesn't contain the code

```
sum += data[i]
```

The expression sum + data[i] is an integer and so can only be assigned to a short using an (implicit or explicit) narrowing cast. As you can see in the following assembly output, the compiler must insert extra instructions to implement the narrowing cast:

```
checksum_v3
                    MOV r2,r0           ; r2 = data
                    MOV r0,#0           ; sum = 0
                    MOV r1,#0           ;i=0
checksum_v3_loop
                    ADD r3,r2,r1,LSL #1 ; r3 = &data[i]
                    LDRH r3,[r3,#0]     ; r3 = data[i]
                    ADD r1,r1,#1        ; i++
                    CMP r1,#0x40        ; compare i, 64
                    ADD r0,r3,r0        ;  r0 = sum + r3
                    MOV r0,r0,LSL #16
                    MOV r0,r0,ASR #16   ; sum = (short)r0
BCC checksum_v3_loop ; if (i<64) goto loop
                    MOV pc,r14          ; return sum
```

There are two reasons for the extra instructions:

■ The LDRH instruction does not allow for a shifted address offset as the LDR instruction did in checksum_v2. Therefore the fifirst ADD in the loop calculates the address of item i in the array. The LDRH loads from an address with no offset. LDRH has fewer addressing modes than LDR as it was a later addition to the ARM instruction set. (See Table 5.1.)

■ The cast reducing total + array[i] to a short requires two MOV instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an int type variable to hold the partial sum.
We only reduce the sum to a short type at the function exit.

Example

The checksum_v4 code fixes all the problems we have discussed in this section. It uses int type local variables to avoid unnecessary casts. It increments the pointer data instead of using an index offset data[i].

```
short checksum_v4(short *data)
{
unsigned int i;
int sum=0;
for (i=0; i<64; i++)
{
sum += *(data++);
}
return (short)sum;
}
```

The *(data++) operation translates to a single ARM instruction that loads the data and increments the data pointer. Of course you could write sum += *data; data++; or even *data++ instead if you prefer. The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to checksum_v3.

```
checksum_v4
        MOV     r2,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v4_loop
        LDRSH   r3,[r0],#2          ; r3 = *(data++)
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r2,r3,r2            ; sum += r3
        BCC     checksum_v4_loop    ; if (sum<64) goto loop
        MOV     r0,r2,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)sum
        MOV     pc,r14              ; return r0
```

Function Argument Types :

converting local variables from types char or short to type int increases performance and reduces code size. The same holds for function arguments.

Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
return a + (b >> 1);
}
```

This function is a little artificial, but it is a useful test case to illustrate the problems faced by the compiler. The input values a, b, and the returnvalue will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of

a short type, that is, −32,768 to +32,767? Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fifill the 32-bit register? The compiler must make compatible decisions for the function caller and callee.

Signed versus Unsigned Types :

The previous sections demonstrate the advantages of using int rather than a char or short type for local variables and function arguments. This section compares the efficiencies of signed int and unsigned int.

If your code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations. However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```
int average_v1(int a, int b)
{
return (a+b)/2;
}
```

This compiles to average_v1

```
ADD r0,r0,r1              ; r0=a+b
ADD r0,r0,r0,LSR #31      ; if (r0<0) r0++
MOV r0,r0,ASR #1          ; r0 = r0 >> 1
 MOV pc,r14                ; return r0
```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces x/2 by the statement:

(x<0) ? ((x+1) >> 1): (x >> 1)

C Looping Structures

Loops with a Fixed Number of Iterations : Example

```
int checksum_v5(int *data)
{
                    unsigned int i;
                    int sum=0;
                    for (i=0; i<64; i++)
                    {
                        sum += *(data++);
                    }
                    return sum;
                    }
```

This compiles to

```
                    checksum_v5
                    MOV r2,r0       ; r2 = data
                    MOV r0,#0       ; sum = 0
                    MOV r1,#0       ; i=0
checksum_v5_loop
                    LDR r3,[r2],#4   ; r3 = *(data++)
                    ADD r1,r1,#1     ; i++
                    CMP r1,#0x40    ; compare i, 64
                    ADD r0,r3,r0     ; sum += r3
BCC checksum_v5_loop ; if (i<64) goto loop
                    MOV pc,r14       ; return sum
```

It takes three instructions to implement the for loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This is not efficient. On the ARM, a loop should only use 2 instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags. Since we are no longer using i as an array index, there is no problem in counting down rather than up.

Example

This example shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{
unsigned int i;
int sum=0;
for (i=64; i!=0; i--)
{
sum += *(data++);
}
return sum;
}
```

This compiles to checksum_v6

```
                MOV r2,r0       ; r2 = data
                MOV r0,#0       ; sum = 0
                MOV r1,#0x40    ; i = 64
checksum_v6_loop
                LDR r3,[r2],#4  ; r3 = *(data++)
                SUBS r1,r1,#1   ; i-- and set flags
                ADD r0,r3,r0    ; sum += r3
BNE checksum_v6_loop            ; if (i!=0) goto loop
                MOV pc,r14      ; return sum
```

The SUBS and BNE instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for checksum_v1 and eight for checksum_v3.

For an unsigned loop counter i we can use either of the loop continuation conditions i!=0 or i>0. As i can't be negative, they are the same condition. For a signed loop counter, it is tempting to use the condition i>0 to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

```
SUBS r1,r1,#1 ; compare i with 1, i=i-1
BGT loop ; if (i+1>1) goto loop
```

In fact, the compiler will generate

```
SUB r1,r1,#1 ; i--
CMP r1,#0 ; compare i with 0
BGT loop
; if (i>0) goto loop
```

The compiler is not being inefficient. It must be careful about the case when i = -0x80000000 because the two sections of code generate different answers in this case.

For the first piece of code the SUBS instruction compares i with 1 and then decrements i. Since -0x80000000 < 1, the loop terminates. For the second piece of code, we decrement i and then compare with 0. Modulo arithmetic means that i now has the value +0x7fffffff, which is greater than zero. Thus the loop continues for many iterations.

Loops Using a Variable Number of Iterations

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until N = 0 and don't require an extra loop counter i.

The checksum_v7 example shows how the compiler handles a for loop with a variable number of iterations N.

```
int checksum_v7(int *data, unsigned int N)
{
                    int sum=0;
                    for (; N!=0; N--)
                    {
                    sum += *(data++);
                    }
                    return sum;
}
```

This compiles to

```
                    checksum_v7
                    MOV r2,#0          ; sum = 0
                    CMP r1,#0           ; compare N, 0
BEQ checksum_v7_end                    ; if (N==0) goto end
```

```
checksum_v7_loop
                LDR r3,[r0],#4    ; r3 = *(data++)
                SUBS r1,r1,#1     ; N-- and set flags
                ADD r2,r3,r2      ; sum += r3
                BNE checksum_v7_loop   ; if (N!=0) goto loop
checksum_v7_end
                MOV r0,r2         ; r0 = sum
                MOV pc,r14        ; return r0
```

Loop Unrolling

Each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch.

These instructions are known as the loop overhead. On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by unrolling a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion. For example, let's unroll our packet checksum example four times.

Example

The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet N is a multiple of four.

```
int checksum_v9(int *data, unsigned int N)
{
        int sum=0;
do
{
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        Sum += *(data++);
        N -= 4;
} while ( N!=0);
        return sum;
}
```

This compiles to

```
checksum_v9
                    MOV r2,#0 ; sum = 0
checksum_v9_loop
                    LDR r3,[r0],#4 ; r3 = *(data++)
                    SUBS r1,r1,#4; N -= 4 & set flags
                    ADD r2,r3,r2       ; sum += r3
                    LDR r3,[r0],#4 ; r3 = *(data++)
                    ADD r2,r3,r2       ; sum += r3
                    LDR r3,[r0],#4 ; r3 = *(data++)
                    ADD r2,r3,r2       ; sum += r3
                    LDR r3,[r0],#4 ; r3 = *(data++)
                    ADD r2,r3,r2 ; sum += r3
BNE checksum_v9_loop   ; if (N!=0) goto loop
                    MOV r0,r2          ; r0 = sum
                    MOV pc,r14         ; return r0
```

Example

This example handles the checksum of any size of data packet using a loop that has been unrolled four times.

```
int checksum_v10(int *data, unsigned int N)
{
        unsigned int i;
        int sum=0;
        for (i=N/4; i!=0; i--)
        {
          sum += *(data++);
          sum += *(data++);
          sum += *(data++);
          sum += *(data++);
        }
        for (i=N&3; i!=0; i--)
        {
          sum += *(data++);
        }
        return sum;
}
```

Register Allocation :

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory (in a similar way virtual memory is swapped out to disk). Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, you need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

Table 5.3 C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

Pointer Aliasing

Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
*timer1 += *step;
*timer2 += *step;
}
```

This compiles to

timers_v1

```
LDR r3,[r0,#0]          ; r3 = *timer1
LDR r12,[r2,#0]         ; r12 = *step
ADD r3,r3,r12           ; r3 += r12
STR r3,[r0,#0]           ; *timer1 = r3
LDR r0,[r1,#0]          ; r0 = *timer2
LDR r2,[r2,#0]          ; r2 = *step
ADD r0,r0,r2            ; r0 += r2
STR r0,[r1,#0]          ; *timer2 = t0
MOV pc,r14              ; return
```

Note that the compiler loads from step twice. Usually a compiler optimization called common subexpression elimination would kick in so that *step was only evaluated once, nd the value reused for the second occurrence. The pointers timer1and step might alias one another. In other words, the compiler cannot be sure that the write to timer1 doesn't affect the read from step.

The following code also compiles inefficiently:

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;
void timers_v2(State *state, Timers *timers)
{
timers->timer1 += state->step;
timers->timer2 += state->step;
}
```

The compiler evaluates state->step twice in case state->step and timers->timer1 are at the same memory address. The fix is easy: Create a new local variable to hold the value of state->step so the compiler only performs a single load.