8

ARM Architecture

In this chapter, we take a detailed look at the ARM architecture. Unlike the other architectures we have seen so far, ARM provides only 16 general-purpose registers. Even though it follows the RISC principles, it provides a large number of addressing modes. Furthermore, it has several complex instructions. The format of this chapter is similar to that we used in the last few chapters. We start with a brief background on the ARM architecture. We then present details on its registers and addressing modes. Following this, we give details on some sample ARM instructions. We conclude the chapter with a summary.

Introduction

The ARM architecture was developed in 1985 by Acorn Computer Group in the United Kingdom. Acorn introduced the first RISC processor in 1987, targeting low-cost PCs. In 1990, Acorn formed Advanced RISC Machines. ARM, which initially stood for Acorn RISC Machine but later changed to Advanced RISC Machine, defines a 32-bit RISC architecture.

ARM's instruction set architecture has evolved over time. The first two versions had only 26-bit address space. Version 3 extended it to 32 bits. This version also introduced separate program status registers that we discuss in the next section. Version 4, ARMv4, is the oldest version supported today. Some implementations of this version include the ARM7™ core family and Intel StrongARM™ processors.

ARM architecture has been extended to support cost-sensitive embedded applications such as cell phones, modems, and pagers by introducing the Thumb instruction set. The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. To reduce the memory requirements, the instructions are compressed into 16-bit wide encodings. To execute these 16-bit instructions, they are decompressed transparently to full 32-bit ARM instructions in real-time. The ARMv4T architecture has added these 16-bit Thumb instructions to produce compact code for handheld devices.

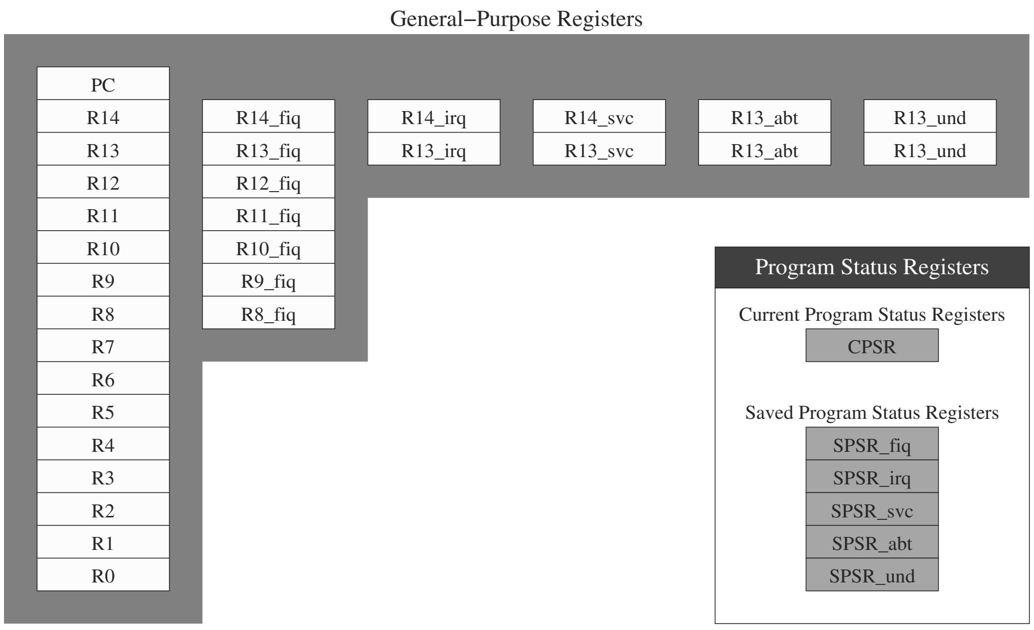


Figure 8.1 ARM register set consists of 31 general-purpose register and six status registers. All registers are 32 bits wide.

In 1999, the ARMv5TE architecture introduced several improvements to the Thumb architecture. In addition, several DSP instructions have been added to the ARM ISA. In 2000, the ARMv5TEJ architecture added the Jazelle extension to support Java acceleration technology for small memory footprint designs. In 2001, the ARMv6 architecture was introduced. This version provides support for multimedia instructions to execute in Single Instruction Multiple Data (SIMD) mode.

Like the MIPS, the ARM is dominant in the 32-bit embedded RISC microprocessor market. ARM is used in several portable and handheld devices including Dell's AXIM X5, Palm Tungsten T, RIM's Blackberry, HP iPaq, Nintendo's Gameboy Advance, and so on [4]. The ARM Web site claims that, by 2002, they shipped over 1 billion of its microprocessor cores [3].

In the remainder of this chapter we look at the ARM architecture in detail. After reading this chapter, you will notice that this architecture is somewhat different from the other processors we have seen so far. It shares some features with the Itanium architecture discussed in the last chapter. However, you should note that the ARM architecture development started in 1985.

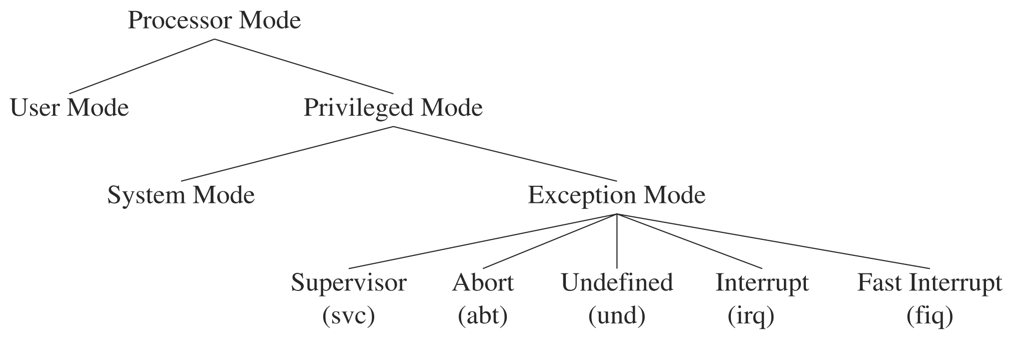


Figure 8.2 ARM processor modes.

Registers

ARM architecture has a total of 37 registers as shown in Figure 8.1. These registers are divided into two groups: general-purpose registers and program status registers.

- *General-purpose registers:* There are 31 general-purpose registers. All these registers are 32 bits wide. At any time, the user can access only 16 of these registers. The actual set of registers visible depends on the processor mode. For example, in the User and System modes, the leftmost 16 registers (R0–R14 and PC) shown in Figure 8.1 are visible.
- *Program status registers:* ARM has six status registers that keep the program status. These registers are also 32 bits wide. The Current Program Status Register (CPSR) is available in all processor modes. The visibility of the other registers depends on the processor mode.

The 16 general-purpose registers are divided into three groups: unbanked, banked, and program counter. The first eight registers (R0–R7) are unbanked registers. These registers are available in all processor modes. (We discuss the processor modes shortly.) The next seven registers (R8–R14) are banked registers. With a few exceptions, most instructions allow the banked registers to be used wherever a general-purpose register is allowed. The actual physical register used in these banks depends on the current processor mode, as shown in Figure 8.1. The last register (R15) is used as the Program Counter (PC) register.

The ARM has seven processor modes, which are divided into User and Privileged modes. The Privileged modes are further divided into System and Exception modes as shown in Figure 8.2.

Application programs typically run in the User mode. This is a nonprivileged mode, which restricts application programs from accessing protected resources. Furthermore, an exception is generated if a mode switch is attempted.

The remaining six modes are privileged modes. These modes allow access to all system resources; furthermore, we can freely switch modes. This group consists of the

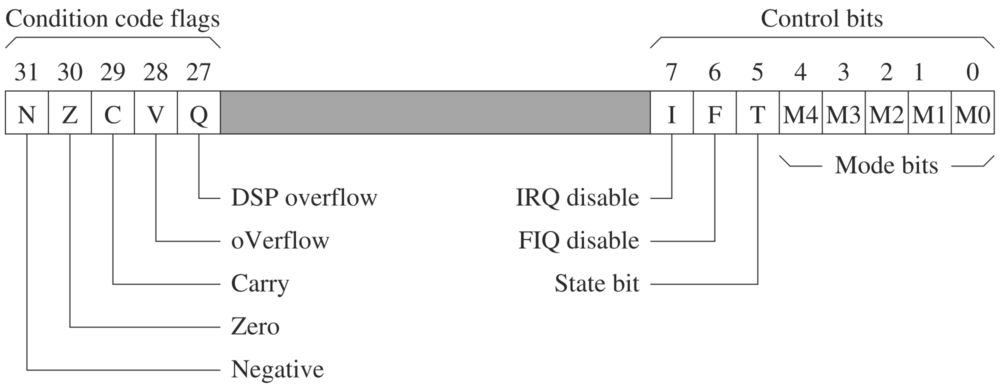


Figure 8.3 CPSR register keeps the control code flags and control bits.

System and Exception modes. The System mode has access to the same set of registers as the User mode; however, because this is a privileged mode, the User mode restrictions are not applicable. This mode is used by the operating system.

The Exception modes are entered when a specific exception occurs. The reset and software interrupts are executed under the Supervisor mode. When a Fast Interrupt request is externally asserted on the FIQ pin of the processor, an FIQ exception is generated. In response to the exception, the processor enters the Fast Interrupt mode. This mode responds to the exception with minimum context switch overhead. This mode is typically used for DMA-type data transfers. When a lesser priority interrupt request is asserted on the IRQ pin, the processor enters the IRQ mode. This mode is used for general interrupt processing.

The Undefined mode is entered when an Undefined exception is generated. This exception occurs if an attempt is made to execute an undefined instruction. The Abort mode is entered when a prefetch abort or a data abort exception occurs. The ARM manual has more details on these exceptions [1].

The general-purpose registers shown on the left in Figure 8.1 are available in the User and System modes. As mentioned before, the first eight registers (R0–R7) and the PC register are available in all modes.

- In the Fast Interrupt mode, the registers R8–R14 are replaced by the FIQ registers. By providing these banked registers, FIQ exceptions can be processed faster by minimizing the context switch overhead.
- In the remaining modes, only the two registers (R13 and R14) are replaced by the corresponding banked registers, as shown in Figure 8.1.

Each exception mode has access to a Saved Program Status Register (SPSR). This register is used to preserve the CPSR contents when the associated exception occurs. The CPSR keeps condition code information, the current processor mode, interrupt disable bits, and so on (see Figure 8.3). The CPSR is available in all modes. One of the SPSR is also available in each of the five exception modes; the actual register depends on the mode. For example, SPSR_fiq is available in FIQ mode, SPSR_irq in IRQ mode, and so on.

The condition code flags N, Z, C, and V are used to record information about the result of an operation. For example, if an instruction produces a zero result, the zero (Z) flag is set (Z = 1). The Q flag is used in DSP instructions and we do not discuss it here. The four condition code flags are modified by arithmetic, logic, and compare instructions. If you have read the previous chapters, you know what they stand for. Here is a brief description of these flags.

The N (Negative) flag indicates that the result of the operation is negative. Note that, as in the other architectures, 2's complement representation is used to represent signed numbers. This flag is essentially a copy of the sign bit of the result. The zero (Z) flag is set if the result is zero.

The remaining two flags record overflow conditions. The carry (C) flag indicates an overflow/underflow on unsigned numbers. For example, when we add two unsigned numbers, if the result does not fit the destination register, this flag is set to indicate an overflow. Similarly, in a subtract operation, a result that is less than zero indicates an underflow. The C flag records this underflow condition. The overflow (V) flag records similar information for signed numbers.

The five mode bits (M0–M4) determine the processor mode. As mentioned before, the processor operates in one of the seven modes. Thus, we do not use all combinations of these five mode bits: only the specified combinations are useful. For example, for the System mode, the mode bits are 11111.

Addressing Modes

In the last four chapters we looked at the addressing modes of four different RISC architectures. Most of these architectures provide very few memory addressing modes. At one extreme, the MIPS architecture supports only a single addressing mode. At the other end, the Itanium supports three addressing modes, some of them with an update facility. PowerPC also supports similar addressing modes. All these architectures support a single register addressing mode for instructions other than load and store.

The ARM architecture goes a few steps further. It supports effectively nine addressing modes for the load and store instructions. In addition, several other addressing modes are provided for the other instructions. In fact, the *ARM Reference Manual* devotes an entire chapter of about 65 pages to describe its addressing modes [1]. In this section, we limit our discussion to the memory addressing modes supported by the ARM architecture. Of course, full details on the addressing modes are available in [1].

In the load and store instructions, the memory address is formed by adding two components as in the other processors:

Effective address = Contents of the base register + offset.

The base register can be any general-purpose register, including the PC. When the PC is used, it allows us to use PC-relative access. Such an access facilitates position-independent code. So far, it looks as if ARM also supports addressing modes similar to those supported by the processors discussed in the previous chapters. However, the flexibility of the ARM addressing modes is due to (i) how the offset can be specified, and (ii) how the base register and offset are used to form the effective address.

Offset Specification

The offset component can be specified in one of three ways: an immediate value, a register value, or a scaled register value.

Immediate Value The offset can be an unsigned constant that is either 8 or 12 bits long. Recall that the other architectures allow this to be a signed number. However, to compensate for this, ARM allows this number to be either added or subtracted from the contents of the base register to form the effective address. For the unsigned byte and word instructions, the immediate value is a 12-bit number. For the signed byte and halfword instructions, it is an 8-bit value. This is similar to the “Register Indirect with Immediate Addressing” mode of the Itanium and other architectures.

Register Value In this mode, the offset can be in a general-purpose register. The effective address is formed by combining (either adding or subtracting) the contents of the base register and the offset register. This register is referred to as the index register in the previous chapters. For the index register, we can specify any general-purpose register other than the PC register. This is similar to the “Register Indirect with Index Addressing” mode discussed in the last chapter.

Scaled Register Value This addressing mode is similar to the last one except that the index register contents are shifted before combining the contents of the base and index registers. That is, the offset is formed by taking the value from a general-purpose register and shifting it by a number of positions specified as an immediate value. As in the last mode, the PC cannot be specified as the index register. We can use any of the shift operations that we use on other operands (left-shift, right-shift, and so on). However, the left-shift operation is commonly used to scale the index register.

This addressing mode is useful to access arrays. For example, assume that the array elements are 8-byte doubles and the array index value is in the index register. In this case, we left-shift the array index in the index register by three bit positions (i.e., multiplying it by 8) so that the index value is converted to the corresponding byte displacement. If you are familiar with the Intel IA-32 architecture, you will recognize that this ARM addressing mode is similar to one of the IA-32 addressing modes.

Address Computation Mode

The effective address computation is done in one of three ways: offset, pre-indexed, and post-indexed modes.

Offset Mode This is the basic mode of address computation and is similar to the addressing modes we have seen in previous chapters. In this mode, the address is computed as the sum of the base register value and the offset. Because we can specify the offset in three ways, the following addressing modes are available.

Effective address = Contents of base register `Rb` $\pm$ `imm`.

Effective address = Contents of base register `Rb` $\pm$ Contents of index register `Rx`.

Effective address = Contents of base register `Rb` $\pm$ Shifted contents of index register `Rx`.

In all the addressing modes, we can use any general-purpose register as the base register. In the first addressing mode, the immediate value `imm` is either 8 or 12 bits long. In the last two addressing modes, the index can be any general-purpose register except the PC register.

Pre-Index Mode In this mode, the memory address is computed as in the last mode. However, the computed effective address is loaded into the base register. It is similar to the update addressing modes we have seen in the PowerPC and Itanium architectures. The ARM refers to this as *write-back* rather than update. As in the last addressing mode, the following three modes are available.

Effective address = Contents of base register `Rb` $\pm$ `imm`,
`Rb` = Effective address.

Effective address = Contents of base register `Rb` $\pm$ Contents of index register `Rx`,
`Rb` = Effective address.

Effective address = Contents of base register `Rb` $\pm$ Shifted contents of index register `Rx`,
`Rb` = Effective address.

Post-Index Mode In this mode, the contents of the base register are used as the memory address. However, the effective address, computed as in the last addressing mode, is loaded into the base register. The three post-index addressing modes are shown below:

Effective address = Contents of base register `Rb`,
`Rb` = Contents of base register `Rb` $\pm$ `imm`.

Effective address = Contents of base register `Rb`,
`Rb` = Contents of base register `Rb` $\pm$ Contents of index register `Rx`.

Effective address = Contents of base register `Rb`,
`Rb` = Contents of base register `Rb` $\pm$ Shifted contents of index register `Rx`.

The addressing modes we discussed here are available to the load and store instructions. However, ARM architecture provides several addressing modes for other instructions. We mention some of these modes in later sections. The ARM reference manual gives a complete list of all addressing modes available [1].

Instruction Format

The ARM instruction execution model is different from the MIPS, PowerPC, and SPARC models. It is somewhat similar to that of the Itanium in that most ARM instructions are conditionally executed. In the Itanium architecture, we used predicate registers to determine if an instruction should be executed. If the specified predicate register is 1, the instruction is executed; otherwise, it is treated as a `nop` (no operation).

The ARM architecture uses a similar scheme. However, ARM uses a 4-bit condition code field to express the condition under which the instruction should be executed. In each instruction, the most significant four bits (bits 28–31) are reserved for this purpose (see Figure 8.4).

These four bits specify a variety of conditions, as shown in Table 8.1. This table gives the condition code field value and the mnemonic used in the instructions along with a description and the condition tested. As shown in this table, the four condition code flags are used to test for the condition specified in the condition field of the instructions.

The mnemonic given in the second column is appended to the instruction mnemonic to indicate the condition. For example, to branch on equal, we use `beq`, whereas to branch on the less than condition, we use `blt`.

Out of the 16 values, two are unconditional. In particular, if the condition field is 1110, the instruction is always executed. This is the default in the sense that if no condition is specified in the instruction, it is assumed to be AL and the instruction is unconditionally executed. Thus, to branch unconditionally, we can specify either `b` or `bal`. We give more examples later on.

The last condition is used as "Never" and the instruction is never executed, essentially making it a `nop`. However, recent versions use it for other purposes (e.g., in DSP instructions) and, therefore, this condition should not be used.

The ARM instruction format is not as simple as that of the MIPS architecture. ARM supports a wide variety of instruction formats. What we have shown in Figure 8.4 is a small sample of the formats used by ARM. The A encoding shown in this figure is used by most arithmetic and logical instructions. The `opcode` field specifies the operation such as `add` and `cmp`. The `Rd` field identifies the destination register that receives the result. One of the source operands is in the source register `Rn`. Because we have 16 general-purpose registers available, each of these fields is 4 bits long. The second source operand is provided by the `shifter-operand` field. This field takes the least significant 12 bits and is used to specify the second source operand. How the second source operand is specified depends on the addressing mode and whether the operand is to be manipulated prior to using it. The I bit distinguishes between the immediate and register forms used

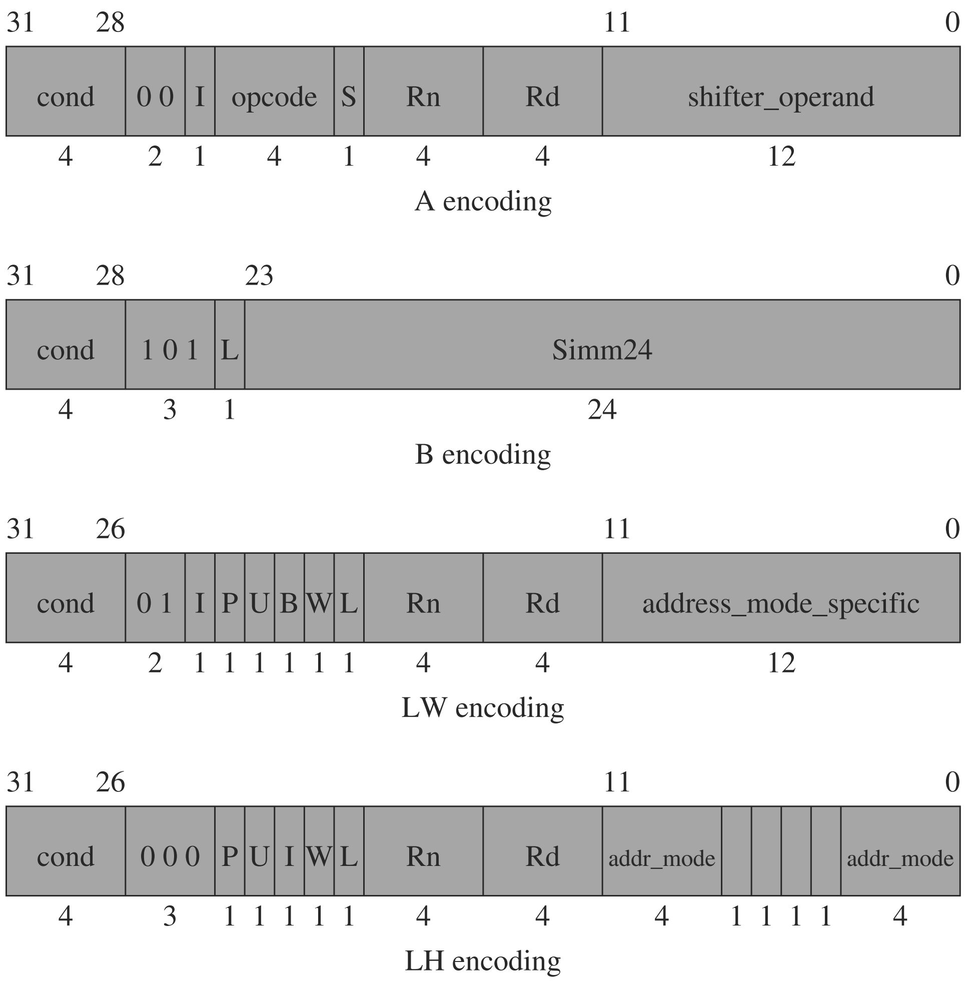


Figure 8.4 Selected ARM instruction formats.

to specify the second source operand. We defer a discussion of this topic until the next section.

Most ARM instructions can optionally update the condition code flags (N, Z, C, and V). The S bit indicates whether the condition code flags should be updated (S = 1) or not (S = 0) by the instruction.

The branch instructions use the B encoding, which can take a 24-bit signed number to specify the branch target. The L bit is used to indicate whether to store the return address in the link register. More information is given later when we discuss the branch instructions.

Table 8.1 ARM condition codes

cond	Mnemonic	Description	Condition tested
0000	EQ	Equal	Z = 1
0001	NE	Not equal	Z = 0
0010	CS/HS	Carry set/unsigned higher or same	C = 1
0011	CC/LO	Carry clear/unsigned lower	C = 0
0100	MI	Minus/negative	N = 1
0101	PL	Plus/positive or zero	N = 0
0110	VS	Overflow	V = 1
0111	VC	No overflow	V = 0
1000	HI	Unsigned higher	C = 1 AND Z = 0
1001	LS	Unsigned lower or same	C = 0 OR Z = 1
1010	GE	Signed greater than or equal	N = V
1011	LT	Signed less than	$N \neq V$
1100	GT	Signed greater than	Z = 0 AND N = V
1101	LE	Signed less than or equal	Z = 1 OR $N \neq V$
1110	AL	Always (unconditional)	—
1111	(NV)	Never (unconditional)	—

The load and store instructions use the last two formats (LW and LH encodings). As in the A encoding, the destination for load (source for store) is given by the `Rd` field. The `Rn` and the least significant 12 bits are used to give address mode specific information. The L bit indicates whether the instruction is a load (L = 1) or a store (L = 0). The B bit specifies whether the load operation is on an unsigned byte or word. The I, P, U, and W bits specify the addressing mode and other information as described here:

P bit	P = 0 indicates post-indexed addressing. P = 1 indicates either the offset addressing or pre-indexed addressing. If W = 0, offset addressing is used; W = 1 indicates the pre-indexed addressing.
U bit	This bit indicates whether the offset value is added (U = 1) or subtracted (U = 0).
I bit	I = I indicates the offset is immediate value. I = 0 indicates index register-based offset.

Instruction Set

This section presents some sample instructions of the ARM instruction set. If you have read the previous chapters, you will notice that the ARM instruction semantics are somewhat different from the other architectures.

Data Transfer Instructions

We discuss three types of data transfer instructions: instructions that move data between registers, load instructions, and store instructions.

Move Instructions Two instructions are available to copy data between registers: `mov` and `mvn`. The `mov` instruction syntax is

```
mov{cond}{s}   Rd,shifter_operand
```

It copies the value of `shifter_operand` into the destination register `Rd`. It uses the A encoding format shown in Figure 8.4. The fields shown in curly brackets are optional. If `s` is specified, it makes the S bit a 1. Note that this bit determines whether the condition code flags are updated (S = 1) or not (S = 0). The `cond` field takes any of the two-letter mnemonics given in Table 8.1. Also note that omitting `cond` implies unconditional execution as `AL` (ALways) is the default. Thus the instruction

```
mov   R1,R2
```

copies the `R2` value to `R1` unconditionally. We can also use this instruction to load constants, as in the following example.

```
mov   R3,#0
```

This instruction loads zero into `R3`. Because the PC register is also a general-purpose register, we can specify PC as the destination register to cause a branch. The instruction

```
mov   PC,LR
```

can be used to return from a procedure. Remember that the link register LR (R14) generally keeps the return address.

Now is a good time to unravel the mystery associated with the `shifter_operand`. This operand can take one of three formats.

Immediate Value: The immediate value is not given directly as in the other architectures. Instead, it takes two numbers: an 8-bit constant and a 4-bit rotate count. The 8-bit constant is rotated by rotate count bit positions to derive the constant. The 4-bit shift count can only specify 16 bit positions, therefore a zero is appended to it to make it 5 bits long. For example, if the 4-bit rotate count is `1001`, it is converted to five bits as `10010`. Thus, the rotate count is always an even number.

A limitation of this process is that not all constants can be expressed. For example, `0xFF00` is a valid constant as we can specify this value with the 8-bit constant taking the `0xFF` value and using 8 as the rotate count. If we change the rotate count to 28, we get `0xF000000F` as another valid constant. However, `0x7F8` is not a valid constant as it requires an odd number of rotations (three in this example). Thus, the move instruction

```
    mov    R3,#0xFF00
```

is a valid one as `0xFF00` is a valid constant.

Register Value: A register can be specified to provide the second operand. This is the format used in the following instructions.

```
    mov    R1,R2
    mov    PC,LR
```

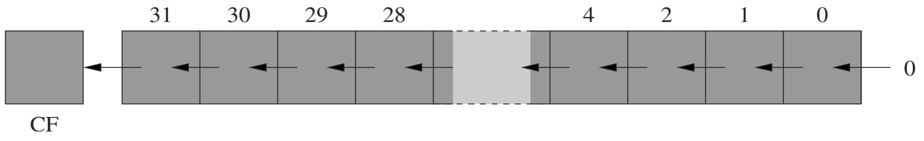
Shifted Register Value: This is a novel way of expressing the second operand, special to the ARM architecture. Before the value is used, the operand is shifted/rotated. In this format, three components are used to form the final value:

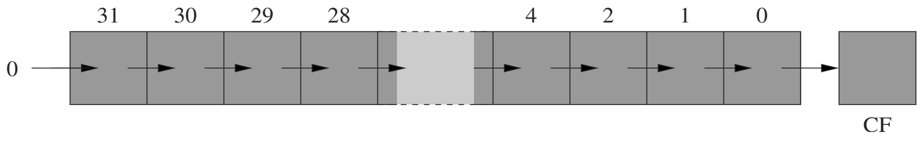
```
    Rm, shift_op SHFT_CNT
```

The value in `Rm` is shifted/rotated `SHFT_CNT` times. The type of shift or rotate operation is specified by `shift_op`. The `shift_op` can be any of the following.

`lsl`	Logical Shift Left
`lsr`	Logical Shift Right
`asr`	Arithmetic Shift Right
`ror`	ROtate Right
`rrx`	Rotate Right with eXtend

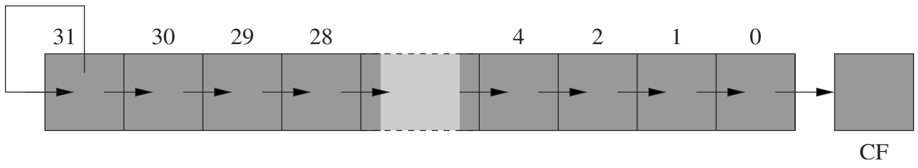
The `lsl` specifies a left-shift operation whereas the `lsr` is used for the right-shift operation. The last bit shifted out will be captured in the carry flag as shown here:





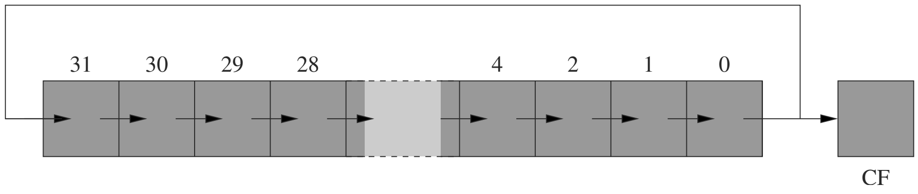
In both cases, the shifted-out bits receive zeros. These are called the logical shifts.

When we right-shift an operand, we have two choices: replace the vacated bits with zeros or with the sign bit. As we have seen, zeros replace the vacated bits in the logical right shift. On the other hand, the sign bit is copied into the vacated bits in the arithmetic right-shift `asr` as shown below:

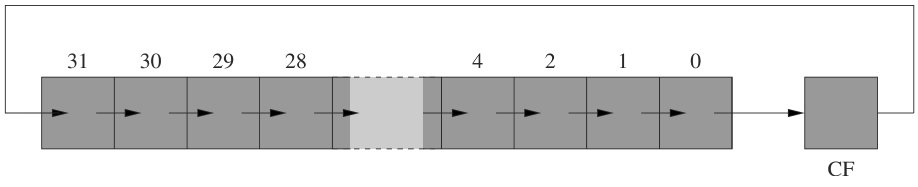


For details on the differences and the need for logical and arithmetic shift operations and why we need them only for the right-shifts, see our discussion in Chapter 15.

The rotate-right `ror` does not throw away the shifted-out bits as does the shift operation. Instead, these bits are fed at the other end as shown here.



The extended version of rotate right `rrx` works on 33 bit numbers by including the carry flag as the 33rd bit as shown below:



The `SHFT_CNT` can be an immediate value or it can be specified via a register. Here is an example that uses this format:

```
mov   R1,R2,lsl #3
```

This instruction multiplies the contents of `R2` by 8 and stores the value in the `R1` register. The `shift_operand` is also used in arithmetic and logical instructions that we discuss later.

The second move instruction `mvn` (MoVe Negative) has the same syntax as the `mov` except that it moves the 1's complement (i.e., bitwise `not`) of the `shifter_operand` value into `Rd`. For example, the instruction

```
mvn   R1,#0
```

loads `0xFFFFFFFF` into `R1` register.

Load Instructions The ARM instruction set provides load instructions for words (32 bits), halfwords (16 bits), and bytes (8 bits). The `ldr` (load register) instruction has the following syntax.

```
    ldr{cond}   Rd,addr_mode
```

It loads a word from the memory into the `rd` register using the addressing mode `addr_mode`. This instruction uses the LW encoding shown in Figure 8.4. The `addr_mode` can be specified in one of the following nine addressing modes.

Addressing mode	Format
Immediate offset	`[Rn,#±offset12]`
Register offset	`[Rn,±Rm]`
Scaled register offset	`[Rn,±Rm, shift_op #shift_imm]`
Immediate pre-indexed	`[Rn,#±offset12]!`
Register pre-indexed	`[Rn,±Rm]!`
Scaled register pre-indexed	`[Rn,±Rm, shift_op #shift_imm]!`
Immediate post-indexed	`[Rn],#±offset12`
Register post-indexed	`[Rn],±Rm`
Scaled register post-indexed	`[Rn],±Rm, shift_op #shift_imm`

The `shift_op` can take `lsl`, `lsr`, `asr`, and so on as discussed before (see page 132). Here are some example `ldr` instructions to illustrate the various formats.

Addressing mode	Example
Immediate offset	`ldr   R0,[R1,#-4]`
Register offset	`ldr   R0,[R1,-R2]`
Scaled register offset	`ldr   R0,[R1,-R2, lsl #3]`
Immediate pre-indexed	`ldr   R0,[R1,#4]!`
Register pre-indexed	`ldr   R0,[R1,R2]!`
Scaled register pre-indexed	`ldr   R0,[R1,R2, lsl #3]!`
Immediate post-indexed	`ldr   R0,[R1],#4`
Register post-indexed	`ldr   R0,[R1],R2`
Scaled register post-indexed	`ldr   R0,[R1],R2, lsl #3`

When we load halfwords and bytes, we need to extend them to 32 bits. This can be done in one of two ways: zero-extension or sign-extension (see our discussion of these two schemes in Appendix A). Corresponding to these two extensions, two instructions are available:

`ldrh`	Load halfword (zero-extended)
`ldrsh`	Load halfword (sign-extended)
`ldrb`	Load byte (zero-extended)
`ldrsb`	Load byte (sign-extended)

The `ldrb` instruction uses the LW encoding whereas the other three instructions use the LH encoding shown in Figure 8.4.
Here is an example:

```
cmp     R0,#0
ldrneb  R1,[R0]
```

The first instruction tests if `R0` is zero (i.e., NULL pointer). If not, we load the byte at the address given by `R0`. We look at the compare (`cmp`) instructions later.

The instruction set also provides multiple loads through the `ldm` instruction. This instruction could be used to load all or a nonempty subset of the general-purpose registers. The register list can be specified as part of the instruction. The format of `ldm` is

```
ldm{cond}addr_mode  Rn{!}, register_list
```

The optional `!` is used to indicate the write-back operation. The `register_list` is a list of resisters in curly brackets. The `addressing_mode` can be one of the following four addressing modes.

- Increment After (mnemonic `ia`);
- Increment Before (mnemonic `ib`);
- Decrement After (mnemonic `da`);
- Decrement Before (mnemonic `db`).

Here are some examples:

```
ldmia   R0,{R1-R12}
ldmia   R0!,{R4-R12,LR}
```

The first instruction loads registers `R1` to `R12` from the memory address in `R0`. The register `R0` is not updated. The second instruction loads registers `R4` to `R12` and the link register (LR) as does the first one. However, because we specified `!`, the register `R0` is updated with the address after loading the registers. We can also specify a condition as in the following example.

```
cmp      R0,#0
ldmeqia  R0!,{R1,R2,R4}
```

Store Instructions The store instruction moves data from a register to the memory. They are similar to the load instructions except for the direction of data transfer, therefore our discussion of these instructions is rather brief.

To store a word, we use the `str` instruction. It uses encoding similar to that of the `ldr` instruction. The halfword and byte versions are also available. Unlike the load instructions, signed versions are not needed as we store exactly 16 or 8 bits with these instructions. We use `strh` to store the lower halfword of the specified source register. Similarly, we use the `strb` instruction to store the least significant byte of the source register. Multiple store instruction `stm` is also available.

The following example illustrates how we can use `ldm` and `stm` instructions to do memory-to-memory block copying. This example copies 50 elements from a source array to a destination array. Each element is a double, which takes 64 bits. We assume that `R0` points to the source array and `R1` to the destination array. The number of elements to be copied is maintained in `R2` as shown in the following code.

```
          mov    R2,#50        ; number of doubles to copy
    copyLoop
          ldmia  R0!,{R3-R12}
          stmia  R1!,{R3-R12}
          subs   R2,R2,#5
          bne    copyLoop
```

The copy loop uses `ldmia` to copy five elements from the source array into registers `R3` to `R12`. The `stmia` copies these elements into the destination array. Because both the load and store instructions use the write-back option (`!`), the pointers are automatically updated to point to successive elements. The loop iterates 10 times to copy the 50 elements.

Arithmetic Instructions

As in the Itanium, the ARM supports add, subtract, and multiply instructions. There is no divide instruction. These instructions use the A encoding shown in Figure 8.4. All instructions in the group follow the same syntax. We use the `add` instruction to illustrate their syntax.

Add Instructions The instruction set provides two add instructions: `add` and `adc`. The `add` instruction has the following syntax.

```
    add{cond}{s}   Rd,Rn,shifter_operand
```

The register `Rn` supplies one of the operands. The `shifter_operand` provides the second operand. As discussed in the move instructions, the `shifter_operand` pre-processes (shift/rotate) the second operand before using it in the arithmetic operation. Here are some examples of this instruction. The instruction

```
    add   R0,R0,#1
```

increments `R0`. However, it does not update the condition code flags. The instruction

```
adds   R0,R1,R2
```

adds the contents of `R1` and `R2` and places the result in `R0`. Because we specified the `s` option, it updates the condition codes. Especially interesting are the carry and overflow flags. If we treat the numbers in `R1` and `R2` as representing unsigned numbers, we use the carry flag to see if there is an overflow (i.e., if a carry is generated out of bit 31). On the other hand, if `R1` and `R2` have signed numbers, we should look at the overflow flag to find if there is an overflow. If you are familiar with the Intel IA-32 architecture, you will see that these two flags are similar to the carry and overflow flags in that architecture.

Often we need to include the carry flag in the addition operation. This is particularly useful in multiword arithmetic operations. The `adc` is like the `add` except that it also adds the carry flag. If the carry flag is zero, it behaves like the `add` instruction.

Here is an example of how we can use `add` and `adc` to perform 64-bit additions. In this example, we assume that the first 64-bit number is in the `R0` and `R1` register pair with `R0` holding the lower 32 bits. The second 64-bit number is in the `R2` and `R3` register pair (`R2` has the lower 32 bits). The following two-instruction sequence

```
adds   R0,R0,R2
adcs   R1,R1,R3
```

adds these two 64-bit numbers and places the result in the `R0` and `R1` register pair. The conditional code flags are updated as we use the `adcs` instruction. It is interesting to see what these flags indicate: the N flag indicates that the 64-bit result is negative; the C flag indicates unsigned overflow as explained before; the V flag indicates signed overflow; and the Z flag indicates that the upper 32 bits of the result are zero.

Subtract Instructions The instruction set provides a total of four subtract operations. The first two that we present are the `add` and `adc` counterparts. The `sub` instruction subtracts the value of `shifter_operand` from that in `Rn`. The `sbc` is like the `sub` instruction but it also subtracts the carry (the `adc` counterpart). These two instructions can be used to perform multiword subtract operations. Assuming that the two 64-bit numbers are in the `R1:R0` and `R3:R2` register pairs as in the last 64-bit addition example, the following two instructions

```
subs   R0,R0,R2
sbcs   R1,R1,R3
```

produce the difference (`R1:R0` − `R3:R2`) in `R1:R0` and set the condition code flags appropriately.

In addition to these two standard subtract instructions, which are provided by almost all architectures, ARM provides two reverse subtract operations. In the reverse subtract instructions, the value in `Rn` is subtracted from the `shifter_operand` value. The reverse instructions are not required in other architectures as the operands are in registers (we can simply switch the registers). However, in the ARM architecture, because one operand is special (given by `shifter_operand`), reverse subtract is useful. The instruction `rsb` performs the subtract operation without the carry flag and the `rsc` subtracts the carry flag as well. Here is an example that multiplies `R3` by 3:

```
rsb   R3,R3,R3,lsl #2
```

The shifter operand left-shifts `R3` by two bit positions, effectively multiplying it by 4. The `rsb` performs (4 ∗ `R3` − `R3`), producing the desired result.

Multiply Instructions The ARM instruction set supports several multiplication instructions. We can divide these instructions into short and long multiply instructions. Both versions multiply two 32-bit numbers. Note that we get a 64-bit result when we multiply two 32-bit numbers. The short multiply instructions produce only the lower 32 bits of the 64-bit result whereas the long versions generate the full 64-bit results. Let us first look at the short multiply instruction. The syntax of the `mul` instruction is

```
mul{cond}{s}    Rd,Rm,Rs
```

It multiplies the contents of `Rm` and `Rs` to produce the lower 32 bits of the 64-bit result. This value is placed in the `Rd` register. This instruction can be used with both signed and unsigned numbers. As usual, the `s` bit can be used to specify whether the condition code flags are to be updated.

The long versions have separate instructions for unsigned and signed numbers. Because both instructions use the same syntax, we illustrate the syntax using the unsigned multiply long (`umull`) instruction.

```
umull{cond}{s}    Rdlo,Rdhi,Rm,Rs
```

This instruction takes four registers: two registers specify where the 64-bit product should go (`Rdlo` and `Rdhi` with `Rdlo` getting the lower 32 bits) and the two numbers to be multiplied as in the `mul` instruction. The signed multiply long (`smull`) uses exactly the same format except that it treats the numbers in `Rm` and `Rs` as signed numbers.

The instruction set also supports an accumulator version of these three instructions. The `mla` (multiply accumulate) instruction has the following syntax.

```
mla{cond}{s}    Rd,Rm,Rs,Rn
```

It adds contents of `Rn` to the lower 32 bits of `Rm` ∗ `Rs` and places the result in the `Rd` register.

The long versions have separate instructions for unsigned and signed numbers. The unsigned version has the same syntax as the `umull` instruction as shown below:

```
umlal{cond}{s}    Rdlo,Rdhi,Rm,Rs
```

It adds the 64-bit value in `Rdhi:Rdlo` to the 64-bit result of `Rm` ∗ `Rs`. The result is stored back in the `Rdhi:Rdlo` register pair. The signed version (`smlal`) has a similar syntax.

Logical and Bit Instructions

The instruction set supports three logical operations: AND (`and`), OR (`orr`), and exclusive-or (`eor`). These instructions follow the same syntax as the `add` group of instructions discussed before. However, they do not affect the overflow (V) flag.

These instructions perform bitwise logical operations. The truth tables for these operations are given on pages 170 and 271. The usage of these logical operations is discussed in Chapter 15. Here we give some examples of these instructions. The `and` instruction

```
    ands     R0,R0,#1
```

can be used to determine if the number in `R0` is odd or even. A number is even if its least significant bit is zero; otherwise, it is odd. Essentially, this instruction isolates the least significant bit. If this bit is 1, the result of this operation is 1; else it is zero. Thus, the zero flag is set if the number is even; it is cleared if odd.

The `or` instruction

```
    orr    R0,R0,#0x20
```

can be used to convert the uppercase letter in `R0` to lowercase by forcing the 5th bit to 1. See Appendix B for details on why this conversion works.

The exclusive-or instruction

```
    eor    R0,R0,#0x20
```

flips the case. That is, if it is a lowercase letter, it is converted to the uppercase and vice versa. We use the `eor` to flip the 5th bit while leaving the rest the bits unchanged. See Chapter 15 (page 275) for an explanation of why this is so.

There is no logical NOT instruction. However, note that the `mvn` instruction effectively performs the NOT operation. In addition, the instruction set has a bit clear (`bic`) instruction.

```
    bic{cond}{s}   Rd,Rn,shifter_operand
```

It complements the value of `shifter_operand` and performs bitwise `and` of this value with the value in `Rn`. The result is placed in the `Rd` register as in the other instructions.

The last instruction we discuss in the section is the `clz` (count leading zeros). The syntax is

```
    clz{cond}   Rd,Rm
```

It counts the number of leading zeros in `Rm`. It starts counting from the most significant bit and counts zeros until it encounters a 1 bit. This count is placed in the `Rd` register. For example, if the value in `R1` is zero, the instruction

```
    clz   R0,R1
```

places 32 in the `R0` register.

Test Instructions

The ARM instruction set has two test instructions that perform logical `and` and exclusive-OR operations. Like the logical instructions, these instructions do not affect the V flag. The test instruction format is

```
tst{cond}   Rd,shifter_operand
```

It performs the logical bitwise `and` operation like the `and` instruction. However, the result of the operation is not stored. Instead, the condition code flags are updated based on this result. This instruction is useful to test the selected bits without destroying the original value. For example, we have written the following instruction to test if `R0` has an even or odd value.

```
ands    R0,R0,#1
```

This instruction writes the result of the `and` operation back in `R0`. If we use the `tst` instruction

```
tst    R0,#1
```

we don't modify the contents of `R0`.

The second test instruction `teq` performs the exclusive-OR operation instead of `and`. The instruction

```
teq    R0,R1
```

can be used to compare if the contents of `R0` and `R1` are equal.

Shift and Rotate Operations

The ARM instruction set does not provide any shift or rotate instructions. There is really no need for this. As we have seen, every instruction has access to the shifter through the `shifter_operand`. If we want to perform a shift operation, we can use the `mov` instruction as in the following example.

```
mov   R0,R0,asr #5
```

This instruction right-shifts the contents of `R0` by five bit positions. This is an arithmetic right-shift, which means the vacated bits on the left are replaced by the sign bit. Similarly, we can use any of the other shift and rotate operations allowed on `shifter_operand`.

Comparison Instructions

The instruction set provides two compare instructions:`cmp` (compare) and `cmn` (compare negative). These instructions can be used to compare two numbers for their relationship (less than, equal to, greater than, and so on). Because both instructions use the same syntax, we look at the compare instruction in detail. The compare instruction takes two operands:

```
    cmp{cond}   Rn,shifter_operand
```

It performs `Rn` − `shifter_operand` and updates the four condition flags (N, Z, C, and V) based on the result. These flag values can be used to conditionally execute subsequent instructions. Here is an example.

Suppose we want to find the minimum of two numbers stored in registers `R0` and `R1`. The following statements

```
    cmp    R1,R0
    movlt  R0,R1
```

move the minimum number to `R0`. The `cmp` instruction compares the number in `R1` with that in `R0`. If the number in `R1` is greater than or equal to the other number, we do nothing. In this case, because the "less than" condition is not true, the `movlt` instruction acts as a `nop`. Otherwise, we copy the value from `R1` to `R0`.

This code assumes that the numbers in the two registers are treated as signed numbers as we used "`lt`" condition. If they are unsigned numbers, we have to use "`lo`" as in the following code.

```
    cmp    R1,R0
    movlo  R0,R1
```

As a second example, look at the following C code.

```
    if (x > y)
        x = x + 4*y;
    else
        y = 3*x;
```

Assuming that `x` and `y` are signed numbers, the following ARM code implements this C code.

```
    ;x is in R0 and y in R1
    cmp    R0,R1
    addgt  R0,R0,R1,lsl #2
    addle  R1,R1,R1,lsl #1
```

In the last instruction, we compute `3x` as `x` + `2x` with the `shifter_operand` supplying `2x`. We can also get `3x` as `4x` − `x`, which can be implemented by the following instruction.

```
    rsble  R0,R0,R0,lsl #2
```

The `cmn` instruction updates the flags by adding the `shifter_operand` value to the value in the `Rn` register. In case you are wondering what is "negative" about this instruction, think of it as subtracting the negative value of the second operand.

Branch Instructions

The branch instructions are used to alter control flow. The branch instruction can also be modified for procedure invocation. These instructions use the B encoding format shown in Figure 8.4. We first discuss the branch instruction. The syntax is

```
b{l}{cond}  target_address
```

For now, ignore the `l` option. We discuss this in the next section. This instruction transfers control to `target_address` if the `cond` is true. As you can see from Figure 8.4, the target address is specified as a 24-bit value. The target instruction address is PC-relative and is computed as follows.

- The 24-bit `target_address` is sign-extended to 32 bits.
- This value is left-shifted by two bit positions. This is because ARM instructions are word aligned. The shift operation adds two zeros to the 24-bit `target_address`.
- The value computed in the last step is added to the PC register to form the branch target. Note that the PC register contains the following address:

  PC = branch instruction address + 8

  You might have thought that this is due to the delayed branching that many RISC processors implement (see our discussion in Chapter 2). Not so! ARM does not implement delayed branching. The reason for the "plus 8" is the three-stage pipeline used in the ARM cores up to ARM7. If you are not familiar with pipelining, here is a brief explanation of how this three-stage pipeline works. Each instruction that runs on this pipeline goes through three stages: fetch, decode, and execute. The pipeline consists of three units corresponding to these three stages. The pipeline works on three instructions concurrently, with each unit working on one instruction. While the first instruction is being executed by the execution unit, the next instruction is being decoded by the decode unit and a third instruction is being fetched. Thus, the PC points to this third instruction, which is 8 bytes from the first instruction. Even though the ARM9 versions use a five-stage pipeline, the "plus 8" still holds for compatibility and other reasons.

If no condition is specified, it becomes an unconditional branch as in the following example.

```
b   repeat
```

This is also equivalent to

```
bal  repeat
```

Here is an example that illustrates how the branch instruction can be used to implement a countdown loop that iterates 50 times.

```
; loop count is in R0
    mov    R0,#50      ; init loop count to 50
loop
        . . .
        . . .
    subs   R0,R0,#1    ; decrement loop count
    bne    loop        ; if R0 != 0, loop back
```

The last instruction `bne` can be considered as “branch on not equal to zero” as it branches to the target if the zero flag is 0.

Procedures

The branch instruction of the last section can be modified to invoke a procedure. A procedure call is like a branch with the provision that it remembers the return address. Thus, if we can make the branch instruction store the return address, it can be used as a procedure call instruction. This is what the `l` field does:

```
bl{cond}  target_address
```

This branch and link instruction places the return address in the link register LR, which is R14, before transferring control to the target. The return address stored is the address of the instruction following the `bl` instruction. For example, if we want to invoke the `findMin` procedure, we do so by

```
bl  findMin
```

How do we return from the procedure? It is simple: all we have to do is to copy the return address from the LR to the PC. We can do this by using the following move instruction.

```
mov  PC,LR
```

Here is an example that shows the structure of an ARM procedure:

```
; ARM code to find minimum of two signed integers
; The two input values are in R0 and R1
; The minimum is returned in R2
findMin
     cmp    R1,R0
     movge  R2,R0     ; R1 >= R0? Min = R0
     movlt  R2,R1     ; R1 < R0? Min = R1
     mov    PC,LR     ; return
```

This procedure receives two signed numbers in `R0` and `R1` and returns the minimum of these two values in `R2`.

Stack Operations

As in the MIPS, there is no special stack pointer register in the ARM architecture. By convention, register R13 is used as the stack pointer. There is another similarity with MIPS: ARM does not provide any special instructions to manipulate the stack. That is, there are no push and pop instructions. We have to implement these stack operations by using load and store instructions. This means we have complete freedom as to how the stack should behave. A stack implementation is characterized by two attributes:

- **Where the stack pointer points**
  - *Full stack:* The stack pointer points to the last full location;
  - *Empty stack:* The stack pointer points to the first empty location.
- **How the stack grows**
  - *Descending stack:* The stack grows downward (i.e., toward lower memory addresses);
  - *Ascending stack:* The stack grows upward (i.e., toward higher memory addresses).

These two attributes define the four types of stacks:

- Full Descending stack (mnemonic `fd`);
- Full Ascending stack (mnemonic `fa`);
- Empty Descending stack (mnemonic `ed`);
- Empty Ascending stack (mnemonic `ea`).

These mnemonics can be used in `ldm` and `stm` instructions to manipulate the stack. For example, we can use instructions such as `ldmfd`, `stmfd`, and so on. Most implementations, including MIPS and SPARC, prefer descending stacks for reasons discussed in Chapter 4 (page 53). Furthermore, the stack pointer points to the last item pushed on to the stack.

Using the `ldmfd` and `stmfd` instructions, we can write procedure entry and exit code as follows.

```
procName
     stmfd  R13!,{R4-R12,LR}    ; save registers and
                                ; return address
              . . .
         <procedure body>
              . . .
     ldmfd  R13!,{R4-R12,PC}    ; restore registers and
                                ; return address to PC
```

This code assumes that we do not have to preserve the first four registers as they are often used to pass parameters. Furthermore, we assume that the stack pointer (R13) points to a full descending stack. The `stmfd` instruction stores registers R4 to R12 and LR on the stack before executing the procedure body. Notice that we use `R13!` (i.e., the write-back option) so that the stack pointer is updated to point to the last full location of the stack. Before returning from the procedure, we use the `ldmfd` instruction to restore the registers and to store the return address in the PC. Because we store the return address in the PC we do not need a separate return from the procedure.

Summary

The ARM architecture is remarkably different from the other architectures we have seen in this part. It shares some features with the Itanium architecture. One feature of the ARM instruction set that sets it apart is that its instructions are executed conditionally. If the specified condition is not true, the instruction acts as a `nop`. Although Itanium also uses conditional instruction execution, the ARM uses a 4-bit condition code as opposed to the predicate bits of Itanium. Another interesting feature of ARM is that arithmetic and logical instructions can pre-process (shift/rotate) one of the input operands before operating on it.

At the beginning of this chapter, we mentioned that Thumb instructions are 16 bits long. To achieve this reduction from the 32-bit ARM instructions, several significant changes have been made. These include:

- The 4-bit `cond` field is eliminated. As a result, most Thumb instructions are executed unconditionally. In contrast, ARM instructions are executed conditionally.
- Most Thumb arithmetic and logical instructions use the 2-address format. In this format, as discussed in Chapter 2, the destination register also supplies a source operand.

Even though the ARM is a RISC architecture, it does not strictly follow the RISC principles as does the MIPS. For example, some of the ARM instructions such as `ldm` and `stm` are not simple instructions. In addition, it provides a large number of addressing modes and uses a somewhat complex instruction format. However, having looked at five different RISC designs, you will also see a lot of commonality among these architectures and the RISC principles given in Chapter 3. In the next part, we take a detailed look at the MIPS assembly language.

Web Resources

Most of the ARM documentation is available from their Web site `www.arm.com`. The *ARM Architecture Reference Manual* is published by Addison-Wesley [1]. The reference manual is also distributed by ARM on their documentation CD. Unfortunately, this information is not available from their Web site; you have to request this CD from ARM.

PART III

MIPS Assembly Language