CHAPTER 3

ARM Instruction Set Architecture

This chapter gives an overview of ARM assembly instructions and presents basic instruction formats.

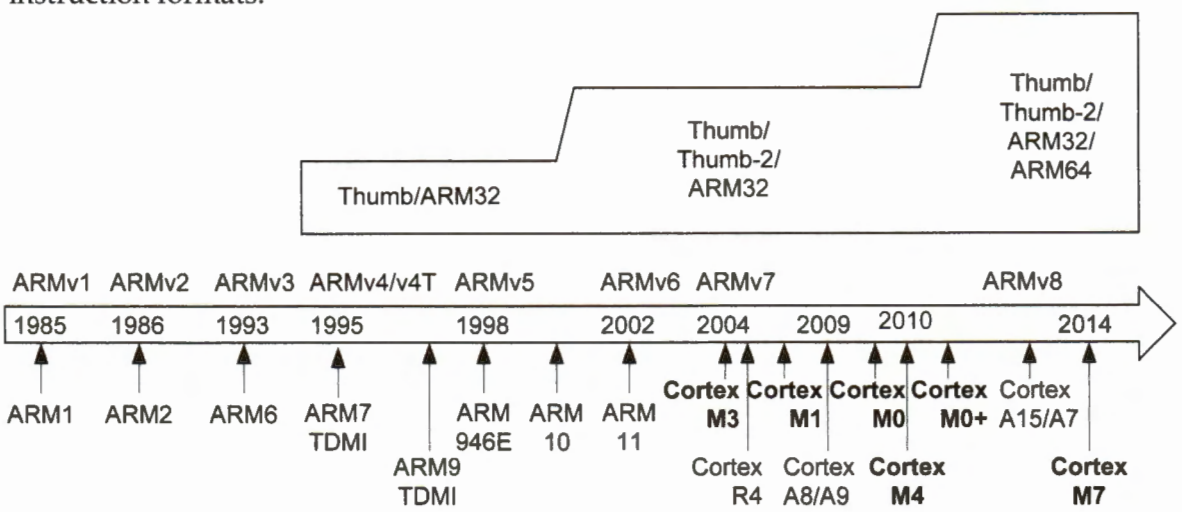

Figure 3-1. History of ARM architecture and instruction sets

3.1 ARM Assembly Instruction Sets

ARM processors support mainly four different assembly instruction sets: Thumb, Thumb-2, ARM32, and ARM64. Figure 3-1 shows their history.

- ***Thumb***. The objective of the Thumb instruction set is to improve the code density. Because an instruction in Thumb has only 16 bits in length, the size of their executable files is small. The space saving is achieved by reducing the possibilities of operands and limiting the number of registers that are accessible by an

instruction. Reducing the size of instruction memory benefits many embedded systems demanding for low cost and long battery life.

- *ARM32*. Each instruction in ARM32 has 32 bits and provides more coding flexibility than a Thumb instruction. More operand options, more flexible memory addressing schemes, larger immediate numbers, and more addressable registers can be encoded in a 32-bit word. Furthermore, ARM32 instructions run faster than Thumb because an instruction can perform more operations or include more operands. However, the disadvantage is its code density.
- *Thumb-2*. It provides an outstanding compromise between ARM32 and Thumb. It optimizes the tradeoff between code density and processor performance. It consists of 16-bit Thumb instructions and a subset of 32-bit ARM32 instructions. The goal of Thumb-2 is to achieve higher code density like Thumb and fast performance comparable to ARM32.
- *ARM64*. ARM 64-bit processors are often used in desktops and servers. These processors have a set of 64-bit assembly instructions.

One prominent ARM family is Cortex processors, which have three groups:

- *Cortex-M* series for microcontrollers (*M* stands for microcontroller),
- *Cortex-R* series for real-time embedded systems (*R* stands for real-time), and
- *Cortex-A* series for high-performance applications (*A* stands for application).

Cortex-A processors are specially designed based on the ARMv7-A or ARMv8-A architecture to provide fast performance for sophisticated devices, such as smartphones and tablets. They often support full-fledged operating systems such as Linux, iOS, and Android.

Cortex-R processors are designed for mission-critical real-time systems that require high reliability, fault-tolerance, and most importantly, deterministic real-time responsiveness. Example systems include factory automation and automobile engine control. In real-time systems, the correctness of computation is determined not only by the logical correctness but also by whether it is consistently completed within certain time constraints.

Cortex-M processors offer an excellent tradeoff between performance, cost, and energy efficiency. Therefore, they are suitable for a broad range of microcontroller applications, such as home appliances, robotics, industrial control, smart watch, and internet-of-things (IoT). In contrast to general-purpose processors in desktops, a microcontroller is a small processor with a processor core, memory, and many integrated I/O peripherals such as timers, analog-to-digital converter, serial communications, and LCD driver.

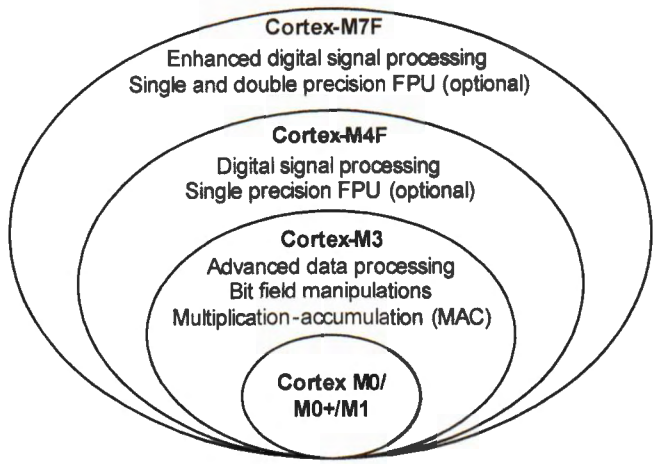

Figure 3-2. ARM Cortex-M family

The Cortex-M family includes Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4, and Cortex-M7. The former three are Von Neumann architecture, and the latter three are Harvard architecture. Moreover, Cortex-M0/M0+/M1 are ARMv6-M, and Cortex-M3/M4/M7 are ARMv7-M.

Cortex-M processors are backward compatible, and Figure 3-2 compares the instructions supported by each processor group. For example, a binary program compiled for Cortex-M3 can run on Cortex-M4 without any modification.

The floating-point unit (FPU), which is a coprocessor for floating-point operations, is optional on Cortex-M4 and Cortex-M7. Cortex-M4 and M7 also provide single-instruction multiple-data (SIMD) and multiply-and-accumulate (MAC) instructions for digital signal processing applications (DSP).

This book focuses on the ARMv7-M architecture, including Cortex-M3, Cortex-M4, and Cortex-M7. ARMv7-M only supports the Thumb-2 instruction set and is not compatible with ARM32. Conventional ARM processors are required to switch to the Thumb state to execute a 16-bit instruction and to the ARM state to run a 32-bit instruction. Cortex-M processors can run a mix of 16-bit and 32-bit Thumb-2 instructions without changing the processor state, thus eliminating the overhead of state switching.

Thumb-2 optimizes the tradeoff between code density and application speed.

This book also presents assembly instructions for floating-point operations and digital signal processing, which are available on Cortex-M4 and M7 but not available on the other Cortex-M processors.

3.2 ARM Cortex-M Organization

An ARM Cortex-M processor chip consists of a Cortex-M core licensed by ARM, on-chip peripheral devices implemented by chip manufacturers, and buses and bridges for the communication between the core and peripheral devices.

Examples of peripheral devices integrated into a Cortex-M chip are LCD controllers, serial communication (I^2C, SPI, and USART), USB, digital-to-analog converters (DAC), and analog-to-digital converters (ADC). Different manufacturers may add various peripheral devices to the chip.

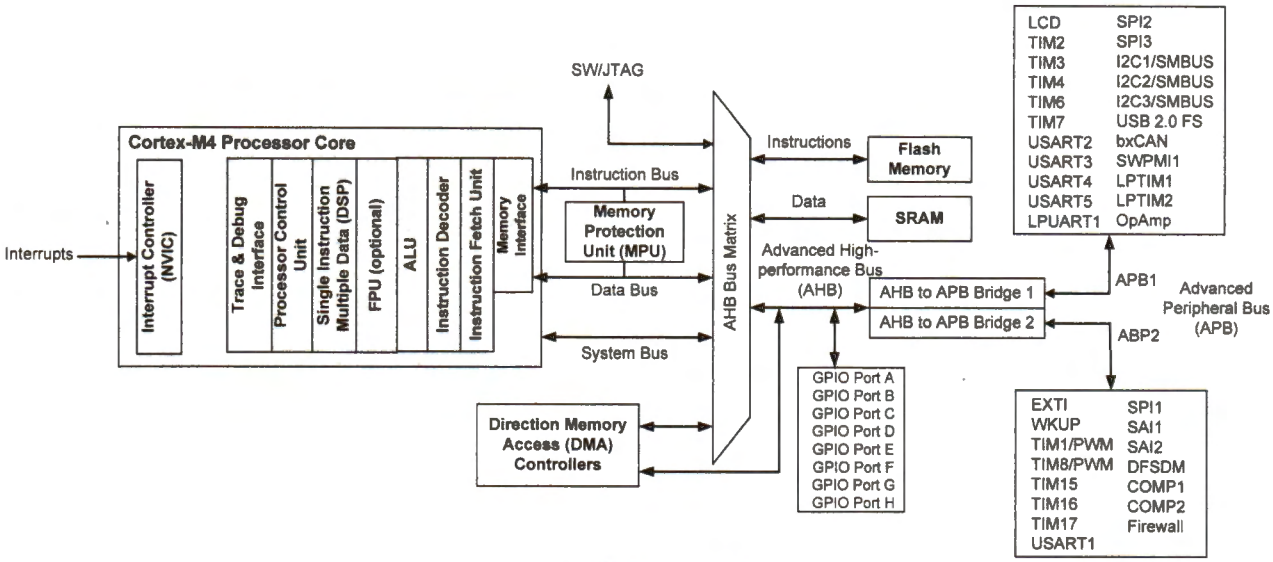

Figure 3-3. Organization of STM32L4 ARM Cortex-M4 processor

Figure 3-3 shows the core and peripheral devices integrated into the STM32L4 Cortex-M4 processor chip.

- The core processor communicates with the flash memory (typically used as instruction memory), SRAM (generally used as data memory), Direct Memory Access (DMA) controller, and general-purpose input/output (GPIO) ports via a bus matrix (also called crossbar switch).
- The bus matrix is an interconnection scheme, which allows concurrent data streams between components connected to the bus matrix, thus providing a high communication bandwidth. The bus matrix connects high-speed components, such as the processor core, Flash, SRAM, DMA controllers, and GPIO ports.
- Peripheral devices are connected to the bus matrix via the bus bridges that links the advanced high-performance bus (AHB) and the advanced peripheral bus

(APB). Generally, AHB is for high-bandwidth communication, and APB is for low-bandwidth communication. AHB and APB are connected via bridges, which buffers data and control signals to fill the bandwidth gap between these two buses, and ensure that there is no data loss.

- Each GPIO pin has multiple functions usually. Software can change its function, even at runtime. We can use a pin simply for digital input or digital output, or we can use it for more advanced functions such as analog-to-digital conversion (ADC), serial communication, timer functions, and so on. Different SoC chips may have different GPIO functions, depending on the chip manufacturers.
- Most peripheral components, such as timers, ADC and I2C, are connected to APB.

A bus is a set of physical wires for transferring data or control signals between two or more hardware components. A communication protocol or agreement must be in place to coordinate the use of a bus. The bandwidth of a bus depends on the width of the bus (usually specified in bits) and the clock speed supported. A processor has various buses for communicating internal and external hardware components. A bus bridge connects two different buses together.

Fundamental components of a Cortex-M processor core include the arithmetic logic unit (ALU), the processor control unit, the interrupt controller (NVIC), the instruction fetching and decoding unit, and the interfaces for memory and debug.

- ALU carries out logical (such as logic `AND`), and integer arithmetic operations (such as add). ALU has two data inputs (called operands) and one data output.
- The processor control unit generates control signals for internal digital circuits (such as the selection signal of the multiplexers, the control signals of the ALU) and coordinates all components of the processor core.
- The interrupt controller (NVIC) allows the processor core to stop the execution of the current task and immediately respond to special events or signals generated by software or by peripheral devices. Chapter 11 introduces interrupts.
- The instruction fetching and decoding unit reads one machine instruction from the instruction memory address pointed by the program counter and decodes the instruction to figure out what operations the processor core should perform. The processor control unit then generates corresponding control signals based on the decoding result. Chapter 13 introduces how to encode and decode an instruction.
- The memory interface supports the access to memory devices (such as SRAM and flash).

- The debug interface allows a programmer to use a host computer to start or stop a software program on a Cortex-M processor, and monitor or modify processor registers, peripheral registers, and memory in real-time.
- Cortex-M4 supports digital signal processing (DSP) and can optionally have a single-precision floating processing unit (FPU). Cortex-M0/M0+/M1/M3 has no support to DSP and FPU. Compared with Cortex-M4, the optional FPU on Cortex-M7 can support both single-precision and double-precision operations.

3.3 Going from C to Assembly

Before we study the syntax (grammar) and semantics (meaning) of assembly instructions, let us first examine the key differences between C and assembly.

C, like many other high-level programming languages, makes powerful abstraction of computer hardware to hide from programmers the details of how computation is implemented. High-level languages make program codes more concise, more portable, and easier to develop and debug.

However, the assembly language, a low-level programming language, offers programmers not only almost complete and fine-grained control of the underlying hardware, but also the flexibility of specifying how a computation should be carried out. Hence, it is often that a well-written assembly program is more efficient than its C counterpart is. Besides abstracting a microprocessor at different levels, some assembly instructions have no equivalent implementation in C, as we shall see later in this book.

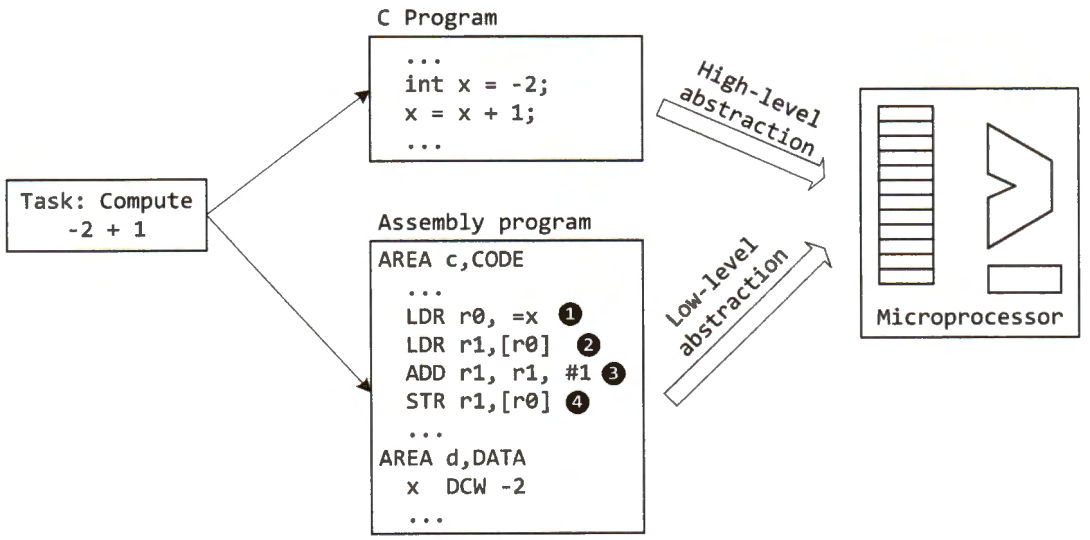

Figure 3-4. Comparison of C and assembly in abstracting microprocessors

In Figure 3-4, we use a simple example, which computes the sum of two signed integers (1 and -2), to compare the hardware abstraction of C and assembly. We assume the variable *x* is stored in memory. Note that a variable may be stored in a register instead of in memory to improve the computation speed.

- C abstracts away much detail of complex low-level computing operations. Accordingly, C provides a friendly and convenient programming interface to programmers. Because of strong abstraction, the same C program can be recompiled for two different hardware platforms, as given below:

	Platform 1	Platform 2
Signed integer representation	Two's complement	One's complement
Size of an integer (bits)	32	16
Operate on immediate numbers?	No	Yes
Data endian (see Chapter 5.1)	Big endian	Little endian

Table 3-1. A C program can be compiled for two different hardware platforms.

- In contrast, assembly language requires programmers to understand low-level details of the instruction set that this specific microprocessor supports. For example, how many bits does an integer take in memory? What is the data layout of the signed integer x in memory? How are memory locations specified? How is the integer x retrieved from memory? How many operands can an addition support? How is an overflow or carry handled on an addition?

In general, there are three types of instruction set architecture (ISA).

- **Accumulator-based instruction set.** One of the ALU source operands is implicitly stored in a special register called accumulator, and the ALU result is saved into the accumulator. The programmer does not have to specify this operand and the destination register in the program. The accumulator-based instruction set was popular in the 1950s.
- **Stack-based instruction set.** All ALU operands are assumed to be on top of the stack, and the ALU result is also placed on top of the stack. The stack is a special region of memory. Thus, programmers need to push the value of operands onto the stack before an ALU operation is called. The stack-based instruction set was used in the 1960s.
- **Load-store instruction set.** ALU source or destination operands can be any general-purpose registers. ALU cannot directly use data stored in memory as operands. ALU can only access data in memory by using load or store instructions. Most modern processors are based on a load-store instruction set.

In the load-store instruction set, many arithmetic and logic instructions typically support two source operands that are stored in registers. The second operand of some instructions can also be a constant number, encoded directly in the instruction.

Load-store instruction set allows effective use of registers.

Compared with the other two types of instruction sets, the load-store instruction set is faster in performance. The accumulator-based instruction set must make an extra copy to store one of the source operands in the accumulator. The performance of a stack-based instruction set is undermined by the performance of memory because ALU must access the memory repeatedly. However, because there are many general-purpose registers available, the load-store instruction set can take full advantage of temporal locality exhibited in almost all applications, effectively reducing the number of accesses to slow memory.

In a load-store instruction set, data stored in memory cannot be ALU operands directly. Therefore, if we want to change some data in memory, software needs to perform a sequence of ***load-modify-store*** operations. Software (1) loads target data from memory to a register, (2) modifies the value in the register, and (3) stores the new value in the register back to the memory.

As Figure 3-5 shows, to increment the value of variable x stored in memory by one, a load-modify-store sequence is carried out in four steps in a sequential order: (1) set up the memory address, (2) load data from memory, (3) perform addition, and (4) store new value back to memory.

While we will examine the detailed syntax of assembly instruction later, we can briefly show the assembly program to illustrate the load-modify-store concept.

```
LDR r0, =x      ; Step 1: Set up address (Load memory address of x into r0)
LDR r1, [r0]    ; Step 2: Load (Register r0 holds the memory address of x)
ADD r1, r1, #1  ; Step 3: Modify (Increase the value in register r1 by 1)
STR r1, [r0]    ; Step 4: Store (Save the content in register r1 into memory)
```

Note an integer takes four bytes in memory. The 32-bit two's complement of -2 is `0xFFFFFFFE`. Assume this number is stored in contiguous memory locations, starting at `0x20000000`. The second LDR instruction loads this 32-bit integer into register `r0` (LDR stands for load register). The last step is to save the 32-bit result (`0xFFFFFFFF`, *i.e.*, `-1`) back to the memory region. After these four steps, the byte stored at memory location `0x20000003`, `0x20000002`, `0x20000001` and `0x20000000` is `0xFF`, `0xFF`, `0xFF`, and `0xFF`.

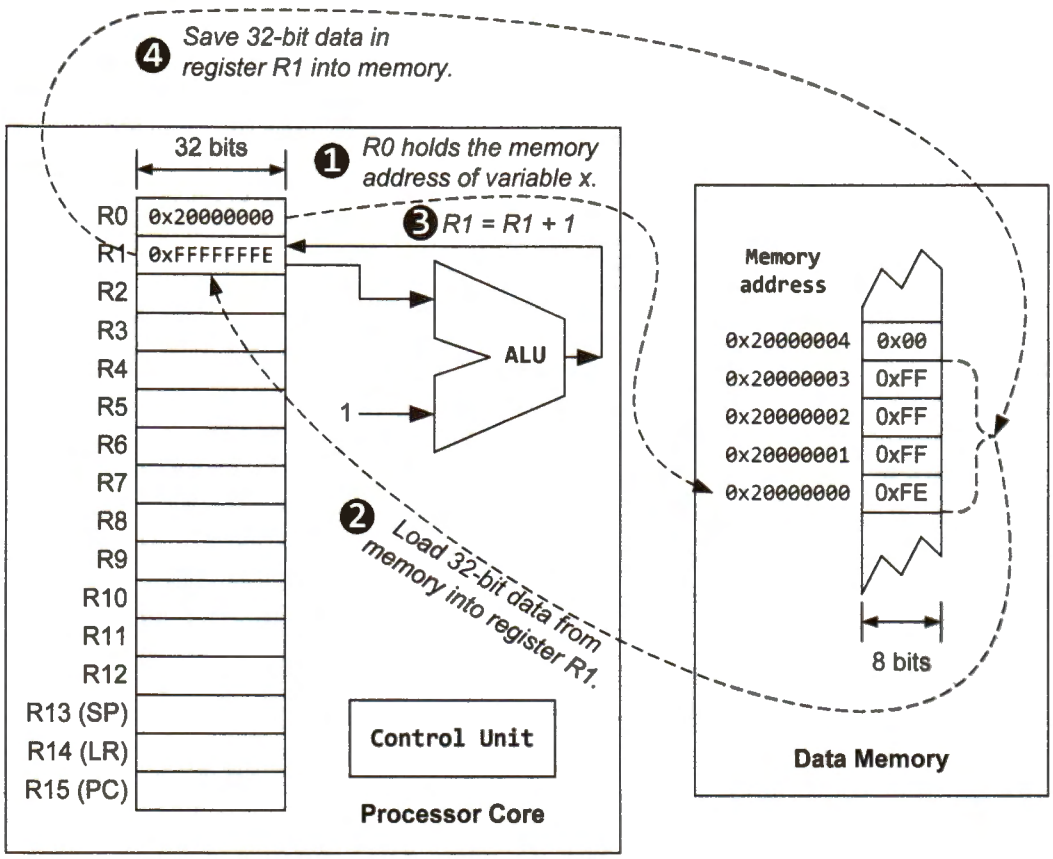

Figure 3-5. A sequence of load-modify-store in assembly equivalent to "`x = x + 1;`" in C. The variable *x* has an initial value of -2 (*i.e.*, `0xFFFFFFFE`) in two's complement.

3.4 Assembly Instruction Format

A machine instruction consists of:

- a binary operation code (opcode) denoting a specific operation to be carried out
- zero or more operands specifying the inputs of the operation

In an assembly program, each binary opcode is replaced by its symbolic abbreviation, called instruction *mnemonic*. Using human-readable mnemonics instead of binary opcode makes developing an assembly program simpler and more convenient.

The general format of an assembly instruction for ARM Keil compilers is as follows:

```
label        mnemonic operand1, operand2, operand3    ; comments
```

- The ***label*** in the above instruction is a reference to the memory address of this instruction. The assembler either replaces the label with the actual numeric memory address or memory address offset when generating the binary executable. The label is optional and must be unique within the same assembly program file. The label should start at the beginning of a line, without any leading whitespace. The instruction can start a new line, as shown below:

```
label
        mnemonic operand1, operand2, operand3     ; comments
```

- The ***mnemonic*** represents the operation to be performed.
- The number of ***operands*** varies, depending on each instruction. Some instructions have no operands. The comma "," is used to separate operands. Some instruction allows constant numbers (also called immediate numbers) as operands.
- Typically, the first operand (operand1) is the ***destination register***, and operand2 and operand3 are ***source operands***. The second operand (operand2) is usually a register. The last operand (operand3) may be a register, an immediate number, a register shifted by a constant amount of bits (using the Barrel shifter introduced in Chapter 4.5), or a register plus an offset (used for memory access).
- Everything after the semicolon ";" is a ***comment***, which is an annotation explicitly declaring programmers' intentions or assumptions.

For GNU compilers, the instruction format is slightly different. All assembly instructions presented in this book follow the ARM format. The GNU format is shown below.

```
label:  mnemonic operand1, operand2, operand3     /* comment */
```

The following gives five examples of ARM assembly instructions.

Example 1: Adding two registers

```
    ADD r0, r2, r3      ; r0 = r2 + r3
```

"ADD" is a mnemonic for arithmetic addition, register `r0` is the destination operand, and registers `r2` and `r3` are two source operands. Register names are case-insensitive. We can also write `r0` as `R0`, `r1` as `R1`, and so on.

Example 2: Subtracting an immediate number

```
    SUB r3, r0, #3      ; r3 = r0 - 3
```

"SUB" is mnemonic for subtraction, register `r3` is the destination operand, register `r0` is the minuend, and the immediate number 3 is the subtrahend.

Example 3: Setting the value of a register

```
MOV r0, #'M'          ; r0 = ASCII value of 'M' (i.e., 0x4D)
```

"MOV" instruction sets the value of r0 to the ASCII value of character M. A constant number has the prefix '#'.

Example 4: Variants of the ADD instruction

```
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r3          ; r1 = r1 + r3
ADD r1, r2, #4      ; r1 = r2 + 4
ADD r1, #15         ; r1 = r1 + 15
```

The number of operands in an instruction varies. If the destination operand (operand1) is the same as the first source operand (operand2), the destination operand can be omitted. The second operand (operand2) can have some variations (such as using Barrel shifter, see Chapter 4.5, and memory addressing, see Chapter 5.4), and it is often written as Op2 in the instruction description. For example, the add instruction is described as follows:

```
ADD {Rd,} Rn, Op2    ; Rd = Rn + Op2
```

The curly brackets "{ }" mean the destination operand Rd is optional if Rn is the same as Rd.

Example 5: Inline Barrel shifter

```
ADD r0, r2, r1, LSL #2  ; r0 = r2 + r1 << 2 = r2 + 4 × r1
MOV r0, r2, ASR #2      ; r0 = r2/4 (signed division)
MOV r0, r0, ROR #16     ; Swap the top and bottom halfword
```

In many instructions, the last operand (operand2 or operand3) can have different formats. We can use the Barrel shifter to shift or rotate the last operation. Refer to Chapter 4.5 for details.

3.5 Anatomy of an Assembly Program

Let us take a quick look at a complete assembly program, as shown in Figure 3-6. The program copies a string to another string. An assembly program includes labels, directives, assembly instructions, and program comments.

1. A ***label***, such as *strcpy*, *stop*, *srcStr*, and *dstStr*, represents the memory address of the data or instruction marked with that label. The assembler replaces each label with its memory address or its memory address offset when generating the

executable. *A label must start with the beginning of a line without any leading space.* A label can be a function name (such as "`__main`" as in the example), which is the memory address of the first instruction of a function. The "`__main`" label is exported to allow the linker to find it and resolve this label.

2. The ***directives*** provide valuable information for assisting the assembler. The example in Figure 3-6 uses the following directives. `PROC` and `ENDP` declare the start and the end of a function (or called a subroutine). `END` indicates the end of an assembly program file. `AREA` defines code or data regions. `ENTRY` designates the initial entry into the program. `ALIGN` specifies the requirement of memory address alignment. `DCB` allocates and defines data.
3. An ***assembly instruction*** is a machine command that controls the program flow or manipulates data. Some instructions are pseudo instructions, which are not real machine commands but are allowed in assembly language code. The assembler translates a pseudo instruction, such as "`LDR r1, =srcStr`" in the example code, into a real instruction. Pseudo instructions make the job of writing assembly language programs easier.
4. A ***comment*** is a text annotation that explains the programmer's intentions or assumptions. It aims to improve inter-programmer communication and code readability. A comment in an assembly program starts with a semicolon. Assemblers ignore everything after the semi-colon in that line.

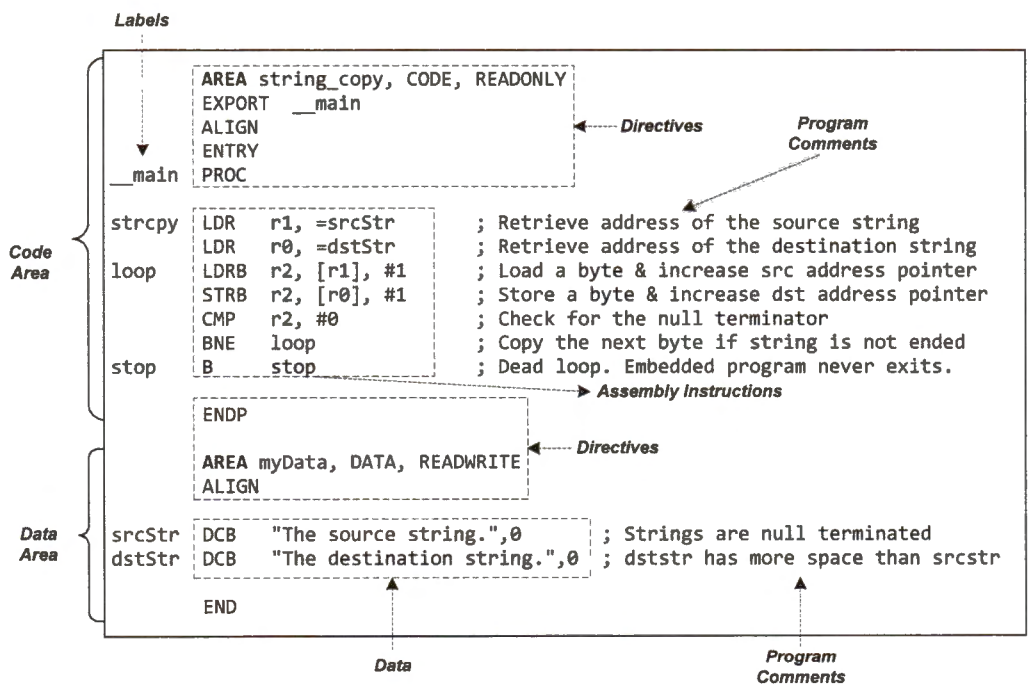

Figure 3-6. An example assembly program

The example assembly program in Figure 3-6 has two areas: a data area and a code area.

- The ***data area*** defines two strings: *srcStr* and *dstStr*. The program allocates memory space for both strings and gives them initial values. The `NULL` character terminates a string.
- The ***code area*** includes a function named `__main`, which is equivalent to the `main()` function in a C program. This program copies string *srcStr* to string *dstStr*.

Most assembly instructions of Cortex-M3 can be classified into the following four categories:

- arithmetic, shift, and logic instructions (see Chapter 4),
- data movement instructions (see Chapter 5),
- compare and branch instructions (see Chapter 6), and
- miscellaneous instructions for various functions such as debugging.

In addition to these instructions, Cortex-M4 and M7 also support

- digital signal processing instructions (see Chapter 24), and
- floating-point instructions (see Chapter 12.4).

(1) Arithmetic, shift, and logic instructions

Shift, logic, and bit instructions	*Shift*: **LSL** (logic shift left), **LSR** (logic shift right), **ASR** (arithmetic shift right), **ROR** (rotate right), **RRX** (rotate right with extend)
	Logic: **AND** (bitwise and), **ORR** (bitwise or), **EOR** (bitwise exclusive or), **ORN** (bitwise or not), **MVN** (move not)
	Bit set/clear: **BFC** (bit field clear), **BFI** (bit field insert), **BIC** (bit clear), **CLZ** (count leading zeros)
	Bit/byte reordering: **RBIT** (reverse bit order in a word), **REV** (reverse byte order in a word), **REV16** (reverse byte order in each halfword independently), **REVSH** (reverse byte order in the bottom halfword, and sign extend to 32 bits)

Arithmetic instructions	*Addition*: **ADD**, **ADC** (add with carry)
	Subtraction: **SUB**, **RSB** (reverse subtract), **SBC** (subtract with carry)
	Multiplication: **MUL** (multiply), **MLA** (multiply with accumulate), **MLS** (multiply with subtract), **SMULL** (signed long multiply), **UMULL** (unsigned long multiply), **SMLAL** (signed long multiply, with accumulate), **UMLAL** (unsigned long multiply, with subtract)

	Division: **SDIV** (signed), **UDIV** (unsigned)
	Saturation: **SSAT** (signed), **USAT** (unsigned)
	Extension: **SXTB** (sign-extend a byte), **SXTH** (sign-extend a halfword), **UXTB** (zero-extend a byte), **UXTH** (zero-extend a halfword)
	Bit field extract: **SBFX** (signed extraction), **UBFX** (unsigned extraction)

(2) Data movement instructions

Memory access instructions	*Read data memory*: **LDRB** (load byte), **LDRH** (load halfword), **LDR** (load word), **LDRD** (load double-word), **LDRSB** (load signed byte), **LDRSH** (load signed halfword), **LDM, LDMDB, LDMFD** (load multiple words) **LDREXB, LDREXH, LDREX** (load register exclusive with a byte, halfword, and word), **LDRT** (load in privileged modes), **POP** (load from stack)
	Write data memory: **STRB** (store byte), **STRH** (store halfword), **STR** (store word), **STRD** (store double-word), **STRSB** (store signed byte), **STRSH** (store signed halfword), **STM, STMDB, STMFD** (store multiple words), **STREXB, STREXH, STREX** (store register exclusive with a byte, halfword, and word), **STRT** (store in privileged modes), **PUSH** (store into stack)
Data copy instructions	**MOV** (move), **MOVT** (move top), **MOVW** (move halfword), **MRS** (move from coprocessor), **MSR** (move to coprocessor)

(3) Compare and branch instructions

Data compare instructions	**CMP** (compare), **CMN** (compare negative), **TST** (test), **TEQ** (test equivalent), **IT** (if-then)
Branch instructions	**B** (branch), **CBZ** (compare and branch on zero), **CBNZ** (compare and branch on non-zero), **TBB** (table branch byte), **TBH** (table branch halfword)
Subroutine instructions	**BL** (branch with link), **BLX** (branch with link and exchange), **BX** (branch and exchange)

(4) Miscellaneous instructions

Miscellaneous instructions	**BKPT** (breakpoint), **NOP** (no operation), **SEV** (set event), **WFE** (wait for event), **WFI** (wait for interrupt), **CPSID** (interrupt disable), **CPSIE** (interrupt enable), **DMB** (data memory barrier), **DSB** (data synchronization barrier), **ISB** (instruction synchronization barrier)

3.6 Assembly Directives

In assembly programs, directives are not actual commands. Instead, they are used to provide key information to compile the source program, such as declaring constants and symbolic names, defining data layout, allocating memory space, and specifying the program structure and entry point. Table 3-2 lists some commonly used directives.

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more halfwords (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
DCFS	Allocate single-precision (32 bits) floating-point numbers
DCFB	Allocate double-precision (64 bits) floating-point numbers
SPACE	Allocate a zeroed block of memory
FILL	Allocate a block of memory and fill with a given value
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Table 3-2 Directives commonly used in ARM assembly language

Table 3-3 gives a typical skeleton frame of an assembly program.

```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY  ; Define a code section
        EXPORT  __main               ; Make __main visible to the linker
        ENTRY                        ; Mark the entrance to the entire program
__main  PROC                         ; PROC marks the beginning of subroutine
        ...                          ; Assembly program starts here.
        ENDP                         ; Mark the end of a subroutine
        END                          ; Mark the end of a program
```

Table 3-3. Skeleton of an ARM assembly program.

(1) AREA

An application consists of one or multiple data and code areas. The AREA directive indicates to the assembler the start of a new data or code section. A code section contains a list of instructions, and a data section includes the declaration and initialization of variables.

An area is a basic independent and indivisible unit processed by the linker. Each area should have a name, and areas within the same source file cannot share the same name. An assembly program must have at least one code area. By default, a code area can only be read (READONLY), and a data area may be read from and written to (READWRITE).

(2) ENTRY

The ENTRY directive marks the first instruction to be executed within an application. There must be one and only one entry directive in an application, no matter how many source files the application has. When there is no entry directive, the linker generates an error message. When there are multiple entry directives, the assembler gives an error message.

There should be only one entry for the whole application, even if it has multiple source files.

For applications written in C or C++, the entry point is in the C library's initialization function, not directly visible to programmers.

(3) END

The END directive indicates the end of a source file. Each assembly program file must end with this directive. Suppose we have two assembly source files *A* and B. When *A* uses either GET or INCLUDE to include *B*, the assembler returns to *A* after reaching END in *B*, and continues to assemble the rest of *A*. The END directive of the top-level file informs the assembler to complete the application.

(4) Function or subroutine definition: PROC and ENDP

PROC and ENDP mark the beginning and the end of a function (also called a subroutine or procedure), respectively. PROC stands for "procedure" and ENDP means "end of procedure."

A single source file can contain multiple subroutines. However, PROC and ENDP cannot be nested. We cannot define a subroutine within another subroutine.

A C program must have at least one function named `main()`. Similarly, an assembly program must have at least one subroutine named `__main`.

(5) Data allocation directive: DCB, DCW, DCD, DCQ, SPACE, and FILL

An assembly program needs to reserve space in the data memory for variables and set their initial contents. Table 3-4 lists commonly used data allocation directives.

Directive	Description	Memory Space
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant Doubleword	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve some zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Table 3-4. Directives for data allocation and initialization

Example 3-1 shows how to declare an initialized string, initialized integer arrays, a zeroed memory region, and a few variables in different formats.

```
        AREA  myData, DATA, READWRITE
hello   DCB   "Hello World!",0  ; Allocate a string that is null-terminated
dollar  DCB   2,10,0,200        ; Allocate integers ranging from -128 to 255
scores  DCD   2,3.5,-0.8,4.0    ; Allocate 4 words containing decimal values
miles   DCW   100,200,50,0      ; Allocate integers between -32768 and 65535
p       SPACE 255               ; Allocate 255 bytes of zeroed memory space
f       FILL  20,0xFF,1         ; Allocate 20 bytes and set each byte to 0xFF
binary  DCB   2_01010101        ; Allocate a byte in binary
octal   DCB   8_73              ; Allocate a byte in octal
char    DCB   'A'               ; Allocate a byte initialized to ASCII of 'A'
```

Example 3-1. Data definition by using data allocation directive

(6) The EQU and RN directive

EQU and RN are to make an assembly program easier to understand. The EQU directive associates a symbolic name to a numeric constant. Like "#define" in a C program, EQU can be used to define a constant in an assembly code.

```
; Interrupt Number Definition (IRQn)
BusFault_IRQn    EQU   -11      ; Cortex-M Bus Fault Interrupt
SVCall_IRQn      EQU   -5       ; Cortex-M Supervisor Call (SVC) Interrupt
PendSV_IRQn      EQU   -2       ; Cortex-M Pend SVC Interrupt
SysTick_IRQn     EQU   -1       ; Cortex-M System Tick Interrupt
```

Example 3-2. EQU is equivalent to "define" in a C program.

The RN directive gives a symbolic name to a register.

```
Dividend         RN    6        ; Defines dividend for register 6
Divisor          RN    5        ; Defines divisor for register 5
```

Example 3-3. RN gives a special, meaningful name to a register.

(7) ALIGN

To improve performance, many processors require that the starting memory address of an instruction or a variable must be a multiple of 2^n. For example, an address aligned to a word boundary must be divisible by 4 (*i.e.*, 2^2). If instructions or data are not appropriately aligned in memory, some processors generate a misalignment fault signal and abort the memory access. Cortex-M processors allow unaligned memory accesses at the sacrifice of performance. Multiple memory accesses may be required to fetch a misaligned data item or instruction. Chapter 10.1 introduces alignment in detail.

The following shows an example usage of `ALIGN` and its layout of the data area.

```
    AREA myCode, CODE, ALIGN = 3  ; Memory address begins at a multiple of 8
    ADD r0, r1, r2                ; Instructions start at a multiple of 8

    AREA myData, DATA, ALIGN = 2  ; Address begins at a multiple of 4
a   DCB 0xFF                      ; The first byte of a word (4 bytes)
    ALIGN 4, 3                    ; Align to the last byte of a word
b   DCB 0x33                      ; Set the fourth byte of a 4-byte word
c   DCB 0x44                      ; Add a byte to make next data misaligned
    ALIGN                         ; Force the next data to be aligned
d   DCD 0x12345                   ; Skip three bytes and store the word
```

Example 3-4. Data alignment in assembly language

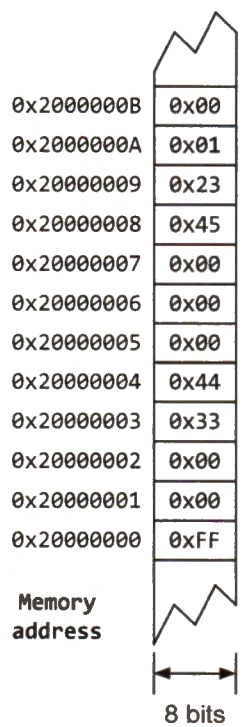

Figure 3-7. Data layout of Example 3-4. Assume the myData area starts at `0x20000000`.

(8) EXPORT and IMPORT

EXPORT and IMPORT define and locate symbols externally defined in different source files. The EXPORT declares a symbol and makes this symbol visible to the linker. The IMPORT gives the assembler a symbol that is not defined locally in the current assembly file. The IMPORT is like the "extern" keyword in C.

(9) INCLUDE or GET

The INCLUDE or GET directive is to include an assembly source file within another source file. It is useful to include constant symbols defined by using EQU and stored in a separate source file. In Example 3-5, all constants are defined by using EQU directives and are stored in a separate assembly file called "*constants.s*". To include these constants, we can use a simple statement "INCLUDE constants.s".

```
        INCLUDE constants.s       ; Load Constant Definitions
        AREA myCode, CODE, READONLY
        EXPORT  __main
        ENTRY
__main  PROC
        ...
        ENDP
        END
```

Example 3-5. Using INCLUDE to load constants defined in a separate file

3.7 Exercises

1. Find five devices that use an ARM processor. Identify the instruction set they support (such as ARM32, Thumb, Thumb-2, or ARM64).

2. Identify two ARM Cortex-M processors and find what I/O peripherals are built into the processor chip.

3. Identify key differences between Cortex-M3 and Cortex-M4.

4. Compared with accumulator-based and stack-based instruction set, what are the advantages and disadvantages of the load-store instruction set?

5. The C language standard (C99 standard) specifies the minimum field width of each variable type. The actual size of a variable type varies by implementations. Find out the minimum size of the following variable types in terms of bytes.

   1) char
   2) short

```
3) signed short int
4) int
5) long
6) unsigned long int
7) long long
8) unsigned long long
9) float
10)double
```

6. An assembly program must have a subroutine named `__main`. Find why it must be named as `__main`. (Hints: Look at the assembly source code of the boot loader, which initializes the processor when the processor starts.)

7. What does "`ALIGN 8, 5`" mean? Draw the data memory layout if the data memory starts at `0x20000000`.

```
    AREA  myData, Data
    ALIGN 4
a   DCB   1
b   DCB   2
c   DCB   3

    ALIGN 8,5
d   DCB   5
```

8. What are incorrect in the following assembly program?

```
        AREA myData, DATA, READWRITE
        String DCB "ABCDE"
        Array  DCD 1, 2, 3, 4, 5
        END

        AREA myCode, CODE, READONLY
        EXPORT  __main2
__main  PROC
        ...

sum     PROC
        ...
        ENDP

        ENDP
        END
```

9. How does an assembly program define a float or double variable? How is a float or double array defined?

CHAPTER 4

Arithmetic and Logic

Data processing instructions can be classified into seven categories: arithmetic instructions, reorder instructions, extension instructions, bitwise logic instructions, shift instructions, comparison instructions, and data copy instructions. This chapter focuses on assembly instructions for arithmetic and logic operations.

4.1 Program Status Register

Cortex-M processors have five status flags: negative (**N**), zero (**Z**), overflow (**V**), carry (**C**), and saturation (**Q**).

- The negative flag (**N**) is set if the result of ALU is negative (*i.e.*, bit[31] is **1**), and is cleared otherwise.
- The zero flag (**Z**) is set if the ALU result is zero, and is cleared otherwise.
- The carry flag (**C**) is set if a carry occurs in unsigned addition, and is cleared otherwise. For unsigned subtraction, it is set if no borrow has occurred, and is cleared otherwise.
- The overflow flag (**V**) is set if an overflow takes place when performing a signed addition or subtraction, and is cleared otherwise.
- The saturation flag (**Q**) is set if an `SSAT` or `USAT` instruction causes saturation, and is cleared otherwise.

Most data processing instructions of Cortex-M processors have an option to update these ALU status flags. These flags are stored in the program status register (`PSR`). The program status register is a combination of three special registers: the application program status register (`APSR`), the interrupt program status register (`IPSR`), and the execution program status register (`EPSR`).

Because `APSR`, `IPSR`, and `EPSR` have no overlap in bit fields, the processor combines them into one register `PSR`, or called `xPSR`, as shown in Figure 4-1, to allow convenient accesses.

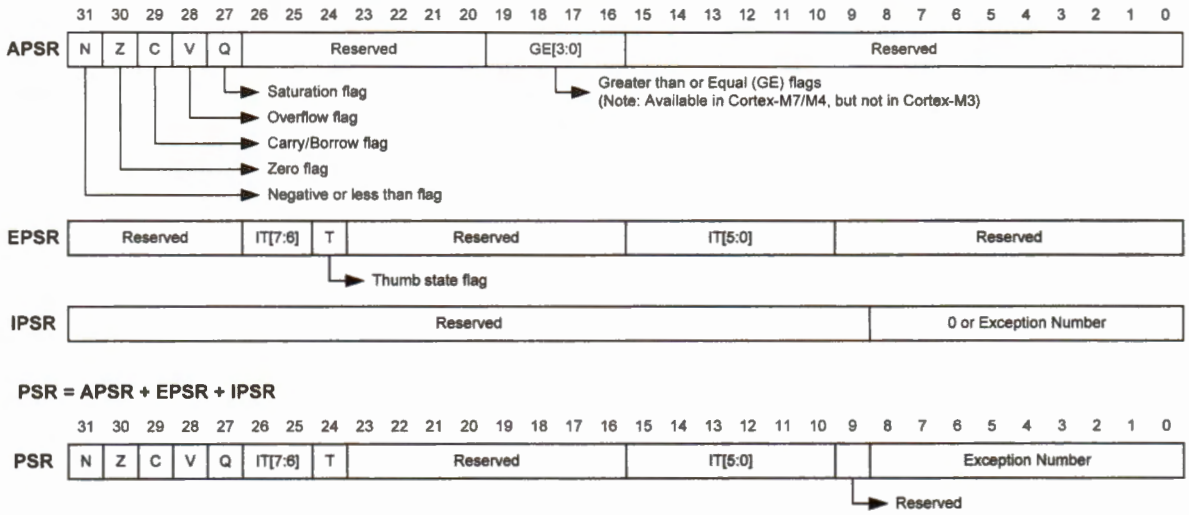

Figure 4-1. Program status register APSR, EPSR, and IPSR

In APSR, the GE flags indicate whether the corresponding results are greater than or equal to zero (see Chapter 24.7). The GE flags are only available on Cortex-M4 and M7.

In EPSR, the T flag indicates whether the processor is in Thumb state or ARM32 state. Since Cortex-M processors only support Thumb-2/Thumb instructions, the T flag has a fixed value of 1 in Cortex-M. Additionally, the IT bit fields (IT[7:6] and IT[5:0]) in EPSR hold the condition states associated with the current IF-THEN (IT) block. An IT block is a convenient approach to implementing conditionally executed instructions.

In IPSR, the least significant 9 bits are zero if the processor is in the thread mode, or the exception or interrupt number if the processor is in the handler mode. On reset, the processor is in the thread mode. Chapter 11 introduces the concept of interrupts.

These special registers can only be accessed by using two special instructions:

- MRS (move from a special register to a general register) and
- MSR (move from a general register to a special register).

Specifically, MRS reads these registers, and MSR writes to these registers. The following gives a few examples.

```
MRS r0, apsr          ; Read APSR
MRS r0, ipsr          ; Read IPSR
MRS r0, epsr          ; Read EPSR
MRS r0, xpsr          ; Read APSR, IPSR, and EPSR
MSR apsr_nzcvq, r0    ; Change N,Z,C,V,Q flags in APSR
MSR apsr_g, r0        ; Copy r0[19:16] to GE[3:0] in APSR (Not on Cortex-M3)
MSR apsr_nzcvqg, r0   ; Change N,Z,C,V,Q and GE flags (Not on Cortex-M3)
```

4.2 Updating Program Status Flags

ADD *vs* ADDS

It is an option for an arithmetic or logic instruction to set the processor status flags. If the S suffix is appended to an instruction mnemonic, the processor modifies the status flags based on the computation result. For example, the ADDS instruction changes the N, Z, C, and V flags when performing addition. On the contrary, ADD cannot change these flags. If an instruction does not update these flags, the existing value of each flag, set by a previous instruction, is preserved.

Data comparison instructions (introduced in Chapter 4.9), such as CMP (compare), CMN (compare negative), TST (test), and TEQ (test equivalence), set these flags even though they do not have the S suffix.

Let us look at ADD instructions with and without the S suffix.

```
ADD  r1, r2, r3  ; r1 = r2 + r3, but won't update N, Z, C, and V flags
ADDS r1, r2, r3  ; r1 = r2 + r3, and update N, Z, C, and V flags
```

While the first instruction ADD does not change the N, Z, C, and V flags, the second instruction ADDS modifies the flags in the following ways:

(1) the overflow flag by assuming that r2 and r3 hold signed integers represented in two's complement,
(2) the carry flag by assuming that r2 and r3 hold unsigned integers,
(3) the zero flag by checking whether the result saved in the destination register r1 is zero or not, and
(4) the negative flag by checking the sign bit of r1 (the most significant bit of r1).

If the Barrel shifter is used, the source operand may update the program status flags. Chapter 4.5 introduces the Barrel shifter. For example, the bitwise logical ANDS instruction can update the N, Z, and C flags. In the following instruction, the N flag is set if the most significant bit of r1 is 1, and the Z flag is set if r1 equals 0.

```
ANDS r1, r2, r3          ; r1 = r2 AND r3
```

It is easy to understand that most logical instructions do not update the overflow flag. How does a logical instruction update the carry flag? The answer lies in the second operand of a logical instruction. If the second operand uses the Barrel shifter, then the processor updates the carry flag based on the shift or rotation result.

```
ANDS r1, r2, r3, LSL #3    ; r1 = r2 AND (r3 << 3)
```

When MOVS uses the Barrel shifter, the processor also updates the Z, N, and C flags.

```
MOVS r2, r1, LSR #3        ; r2 = r1 << 3
```

However, the Barrel shifter does not change the flags if it is employed in an arithmetic instruction. For example, in the following instruction, the flags depend on the result of addition, instead of logical shift left.

```
ADDS r1, r2, r3, LSL #3    ; r1 = r2 AND (r3 << 3)
```

If the program is written in assembly, it is the programmer's responsibility to interpret and use these flags correctly. For programs written in high-level languages, compilers automatically interpret these flags. As introduced in Chapter 2.4.3, if the ALU is to update the status flags when performing an arithmetic addition or subtraction, the processor updates both the carry flag and the overflow flag. It must be clear to programmers whether the numbers stored in the registers are signed or unsigned.

4.3 Shift and Rotate

As shown in Figure 4-2, the second ALU operand is equipped with a Barrel shifter, which is a special digital circuit for quick shift and rotation. Barrel shifters are usually not available on other processors such as PIC and AVR.

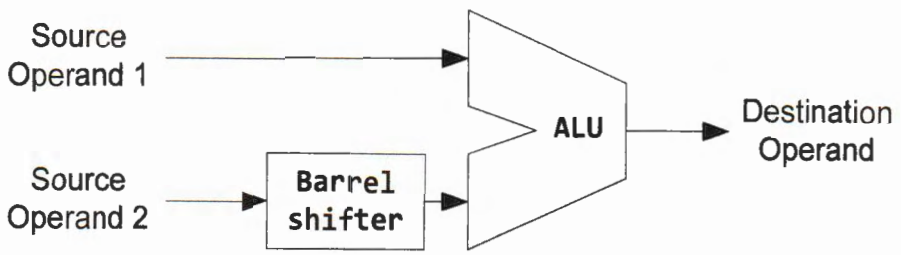

Figure 4-2. Barrel shifter is special hardware that performs quick shift and rotate operations on the second source operand.

There are five types of shift and rotate operations: LSL, LSR, ASR, ROR, and RRX, as shown in Figure 4-3.

- **LSL** (logical shift left) moves all bits of a register value left by n bits and zeros are shifted in at the right end. LSL is equivalent to multiplication by 2^n ("<<" operation in C).

- **LSR** (logical shift right) moves all bits of a register value right by n bits and zeros are shifted in at the left end. LSR is equivalent to unsigned division by 2^n (">>" operation on unsigned numbers in C).
- **ASR** (arithmetic shift right) moves all bits right by n bits and copies of the left most bit (the sign bit) are shifted in at the left end. ASR is equivalent to signed division by 2^n (">>" operation on signed numbers in C).
- **ROR** (rotate right) is the circular shift, in which all 32 bits are shifted right simultaneously as if the right end of the register is joined with its left end. The bit shifted out from the right end of the register is copied into the carry bit. The carry bit can be optionally used to update the carry flag of the processor status register.
- **RRX** (rotate right with extend) works similarly to ROR except that the carry bit joins the rotating circle, and RRX can rotate the data by only one bit.

Below gives a few examples of shift and rotate instructions.

```
LSL r1, r2       ; r1 = r1 << r2
LSL r1, #3       ; r1 = r1 << 3
LSL r1, r2, #3   ; r1 = r2 << 3
LSL r1, r2, r3   ; r1 = r2 << r3
ROR r1, r2       ; r1 = rotate r1 by r2 bits
RRX r1, r2       ; rotate r2 right by one bit (with extension)
```

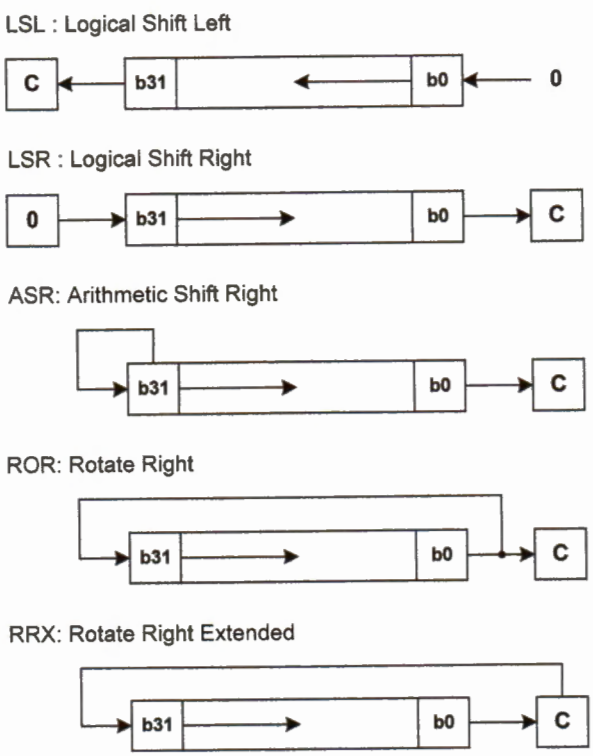

Figure 4-3. Shift and rotate operations. Note that the carry (C) is not APSR's carry flag.

The C language does not provide rotate operations (`ROR` and `RRX`). The compiler automatically uses a rotation instruction if it can improve the performance. Besides, ARM assembly language does not provide a rotate left assembly instruction. However, a rotate left by n bits can be replaced with a rotate right by $32 - n$ bits. For example, rotating left by 6 bits has the same result as rotating right by 26 bits.

Note the carry bit shown in Figure 4-3 is not the carry flag of the processor status register. Therefore, none of these shift and rotate instructions updates the status flags by default. If these flags need to be updated, a shift or rotate instruction must have the suffix `S` specified. What's more, these instructions cannot modify the overflow flags.

```
LSL  r1, #3        ; r1 = r1 << 3, but won't update the flags
LSLS r1, #3        ; r1 = r1 << 3, and update the N, Z, C flags
                   ; LSLS does not update the V flag
```

Programs often use the Barrel shifter to replace slow multiplication and division instructions to improve the speed, as shown below.

```
ADD r0, r2, r1, LSL #1   ; r0 = r2 + r1 << 1 = r2 + 2 × r1
ADD r1, r0, r0, LSR #3   ; r1 = r0 + r0 >> 3 = r0 + r0/8
```

The Barrel shifter used in a move (`MOVS` and `MVNS`), and logical/bitwise instruction with the `S` suffix (such as `ANDS`, `ORRS`, `EORS`, `BICS`) updates the carry flag. This chapter gives detailed discussions later.

4.4 Arithmetic Instructions

Table 4-1 lists arithmetic instructions that produce 32-bit results.

ADD {Rd,} Rn, Op2	Add. *Rd* ← *Rn* + *Op2*
ADC {Rd,} Rn, Op2	Add with carry. *Rd* ← *Rn* + *Op2* + *Carry*
SUB {Rd,} Rn, Op2	Subtract. *Rd* ← *Rn* - *Op2*
SBC {Rd,} Rn, Op2	Subtract with carry. *Rd* ← *Rn* - *Op2* + *Carry* - 1
RSB {Rd,} Rn, Op2	Reverse subtract. *Rd* ← *Op2* - *Rn*
MUL {Rd,} Rn, Rm	Multiply. *Rd* ← (*Rn* × *Rm*)[31:0]
MLA Rd, Rn, Rm, Ra	Multiply with accumulate. *Rd* ← (*Ra* + (*Rn* × *Rm*))[31:0]
MLS Rd, Rn, Rm, Ra	Multiply and subtract. *Rd* ← (*Ra* – (*Rn* × *Rm*))[31:0]
SDIV {Rd,} Rn, Rm	Signed divide. *Rd* ← *Rn* / *Rm*
UDIV {Rd,} Rn, Rm	Unsigned divide. *Rd* ← *Rn* / *Rm*
SSAT Rd,#n,Rm{,shift #s}	Signed saturate
USAT Rd,#n,Rm{,shift #s}	Unsigned saturate

Table 4-1. Arithmetic instructions with 32-bit results

4.4.1 Addition and Subtraction Instructions

Most of these instructions take two source operands, and the 32-bit result is saved in a destination register. While the first source operand is a register, the second source operand is flexible and can be a register, an immediate constant, or an inline Barrel shifter.

Examples of three register operands:

```
SUB r3, r2, r1      ; r3 = r2 - r1
SBC r3, r2, r1      ; r3 = r2 - r1 + Carry - 1
RSB r3, r2, r1      ; r3 = r1 - r2
```

Examples of an immediate number operand:

```
SUB r3, r2, #987    ; r3 = r2 - 987
RSB r3, r2, #987    ; r3 = 987 - r2
```

Examples of inline Barrel shifter:

```
RSB r0, r0, r0, LSL #5      ; r0 = r0 << 5 - r0 = 31 × r0
ADD r0, r0, r0, LSL #3      ; r0 = r0 + r0 << 3 = 9 × r0
```

The next section introduces the Barrel shifter in detail.

If an instruction has three operands, the second operand cannot be a constant number in most instructions (except SSAT and USAT). For example, the SUB instruction below has a syntax error.

```
SUB r0, #1, r3      ; Not allowed, causing a syntax error.
RSB r0, r3, #1      ; r0 = 1 - r3. RSB is for reverse subtraction.
```

Example 4-1 given below shows the implementation of subtracting two 96-bit integers by using SUB and SBC. A 96-bit integer is saved in three registers.

$$C(r8{:}r7{:}r6) = A(r2{:}r1{:}r0) - B(r5{:}r4{:}r3)$$

The program uses the LDR instruction (see Chapter 5.1). The LDR instruction sets a register to a constant value. A constant value is also called an immediate number.

```
; C = A - B
; Subtracting two 96-bit integers A (r2:r1:r0) and B (r5:r4:r3).
; Three registers to hold a 96-bit integer: upper word : middle word : lower word
; Result C (r8:r7:r6)
; A = 00001234,00000002,FFFFFFFF
; B = 12345678,00000004,00000001

LDR r0, =0xFFFFFFFF    ; A's lower 32 bits (See LDR in Chapter 5.1)
LDR r1, =0x00000002    ; A's middle 32 bits
LDR r2, =0x00001234    ; A's upper 32 bits
```

```
LDR r3, =0x00000001   ; B's lower 32 bits
LDR r4, =0x00000004   ; B's middle 32 bits
LDR r5, =0x12345678   ; B's upper 32 bits

; Subtract A from B
SUBS r6, r0, r3  ; C[31:0] = A[31:0] - B[31:0], update carry

; Carry flag is 1 if no borrow has occurred in the previous subtraction
SBCS r7, r1, r4  ; C[64:32] = A[64:32] - B[64:32] + carry - 1, update carry
SBC  r8, r2, r5  ; C[96:64] = A[96:64] - B[96:64] + carry - 1
```

Example 4-1. Subtracting two 96-bit integers. Each integer is stored in three registers.

4.4.2 Short Multiplication and Division Instructions

The result of a multiplication may have more than 32 bits. However, the destination register only holds the least significant 32 bits (LSB32) of the result.

```
MUL  r6, r4, r2       ; signed multiply, r6 = LSB32( r4 × r2 )
UMUL r6, r4, r2       ; unsigned multiply, r6 = LSB32( r4 × r2 )
MLA  r6, r4, r1, r0   ; r6 = LSB32( r4 × r1 ) + r0
MLS  r6, r4, r1, r0   ; r6 = LSB32( r4 × r1 ) - r0
SDIV r3, r2, r1       ; signed divide, r3 = r2/r1
UDIV r3, r2, r1       ; unsigned divide, r3 = r2/r1
```

4.4.3 Long Multiplication Instructions

Table 4-2 presents long multiplication instructions that produce 64-bit results.

UMULL RdLo,RdHi,Rn,Rm	Unsigned long multiply, *RdHi,RdLo* ← *unsigned(Rn × Rm)*
SMULL RdLo,RdHi,Rn,Rm	Signed long multiply, *RdHi,RdLo* ← *signed(Rn × Rm)*
UMLAL RdLo,RdHi,Rn,Rm	Unsigned multiply with accumulate, *RdHi,RdLo* ← *unsigned(RdHi,RdLo + Rn × Rm)*
SMLAL RdLo,RdHi,Rn,Rm	Signed multiply with accumulate, *RdHi,RdLo* ← *signed(RdHi,RdLo + Rn × Rm)*

Table 4-2. Long multiplication instructions

Two registers are used to store a 64-bit result, with the high register (RdHi) holding the most significant 32 bits, and the low register (RdLo) holding the least significant 32 bits.

UMULL and UMLAL assume that the operands Rn and Rm, and the 64-bit multiplication result is an unsigned integer. On the other hand, SMULL and SMLAL treat the operands as signed integers. UMLAL and SMLAL also perform accumulation.

```
UMULL r3, r4, r0, r1   ; r4:r3 = r0 × r1, r4 = MSB bits, r3 = LSB bits
SMULL r3, r4, r0, r1   ; r4:r3 = r0 × r1
UMLAL r3, r4, r0, r1   ; r4:r3 = r4:r3 + r0 × r1
SMLAL r3, r4, r0, r1   ; r4:r3 = r4:r3 + r0 × r1
```

4.4.4 Saturation Instructions

The saturation instructions limit a given input to a configurable signed or unsigned range. When the input value exceeds the specified range, its output is then set as the maximum or minimum value of the selected range. Otherwise, the output is equal to the input. The saturation instructions take one immediate source operand and one register source operand.

- SSAT saturates a signed integer x to the signed range $-2^{n-1} \le x \le 2^{n-1}-1$.

$$SSAT(x) = \begin{cases} 2^{n-1}-1 & if\ x > 2^{n-1}-1 \\ -2^{n-1} & if\ x < 2^{n-1} \\ x & otherwise \end{cases}$$

- USAT saturates a signed integer x to the unsigned range $0 \le x \le 2^n - 1$.

$$USAT(x) = \begin{cases} 2^{n}-1 & if\ x > 2^{n}-1 \\ x & otherwise \end{cases}$$

The following gives two examples in which n is 11. Note the second operand is an immediate number in SSAT and USAT.

```
SSAT    r2, #11, r1     ; output range: -2^10 ≤ r2 ≤ 2^10
USAT    r2, #11, r3     ; output range: 0 ≤ r2 ≤ 2^11
```

4.5 Barrel Shifter

The key advantage of Barrel shifters is that it can shift or rotate a register by a specified number of bits in one clock cycle. Typically, a Barrel shifter is implemented as a cascade of parallel 2-to-1 multiplexers. Figure 4-4 gives an example implementation of a four-bit Barrel shifter that performs rotate right. The S_1S_0 indicates the amount of rotation. The implementation of logic shift is similar, except that a zero bit is shifted in either from the right end or the left end.

As shown in Figure 4-2, the Barrel shifter is special hardware that can perform shift and rotation on the second ALU source operand. Therefore, not only can a shift and rotate instruction be used as a standalone assembly instruction, but it can also be utilized in other instructions to make changes to the second source operand.

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	D_3	D_2	D_1	D_0
0	1	D_0	D_3	D_2	D_1
1	0	D_1	D_0	D_3	D_2
1	1	D_2	D_1	D_0	D_3

Table 4-3. Truth table of rotation right

ASR and LSR differ on whether the sign is preserved. For example,

```
ADD r1, r0, r0, LSL #3   ; r1 = r0 + r0 << 3 = r0 + 8 × r0
ADD r1, r0, r0, LSR #3   ; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)
ADD r1, r0, r0, ASR #3   ; r1 = r0 + r0 >> 3 = r0 + r0/8 (signed)
```

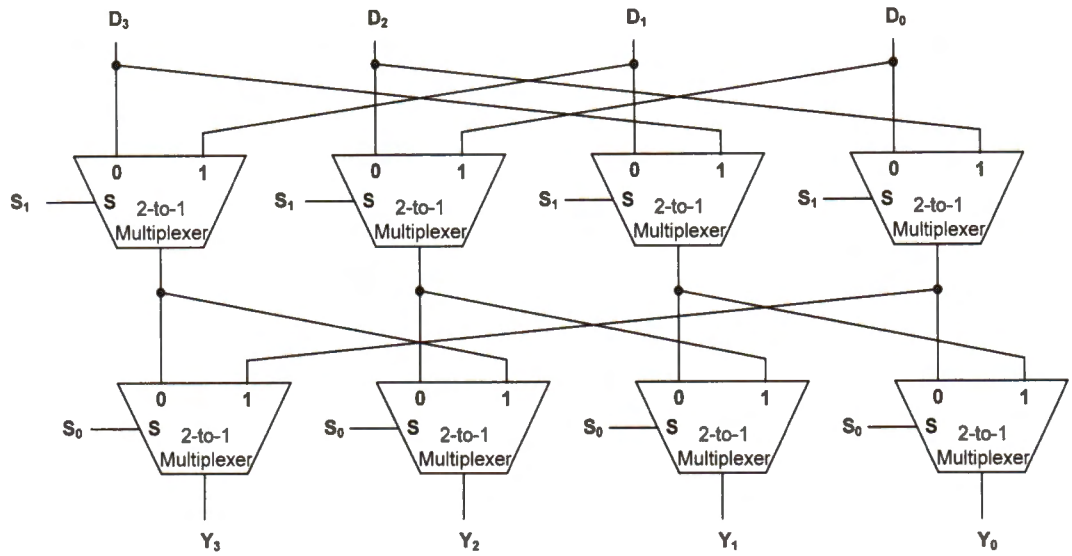

Figure 4-4. Example four-bit Barrel shifter that performs rotate right

We can leverage Barrel shifter to speed up the application.

- Without Barrel shifter, two separate instructions would be required to carry out each of the above instructions. This would not only increase the size of a binary program but also take more processor cycles to complete the same task.
- Barrel shifter can also replace slow multiplication instructions, as shown in the following example.

```
ADD r1, r0, r0, LSL #3   ⟺   MOV r2, #9      ; r2 = 9
                              MUL r1, r0, r2  ; r1 = r0 * 9
```

4.6 Bitwise Logic Operations

Bitwise operations treat input operands as a sequence of binary bits, rather than as integer numbers. The computation is carried out at the bit level. For example, we can reset a specific bit of a register to zero or set a specific bit a register to one, leaving the other bits unchanged.

There are four commonly used bitwise Boolean operators: AND, OR, Exclusive OR (⊕), and negation (NOT). The output of the Exclusive OR is true only when the input bits differ (*i.e.*, one is true, and the other is false). Table 4-4 shows their truth table.

a	*b*	*a and b*	*a or b*	$a \oplus b$	*not a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 4-4. Truth table of logic operations

Table 4-5 shows the bitwise assembly instructions supported in Cortex-M.

`AND {Rd,} Rn, Op2`	Bitwise logic AND. *Rd* ← *Rn* & *operand2*
`ORR {Rd,} Rn, Op2`	Bitwise logic OR. *Rd* ← *Rn* | *operand2*
`EOR {Rd,} Rn, Op2`	Bitwise logic exclusive OR. *Rd* ← *Rn* ^ *operand2*
`ORN {Rd,} Rn, Op2`	Bitwise logic NOT OR. *Rd* ← *Rn* | *(NOT operand2)*
`BIC {Rd,} Rn, Op2`	Bit clear. *Rd* ← *Rn* & *NOT operand2*
`BFC Rd, #lsb, #width`	Bit field clear. *Rd[(width+lsb–1):lsb]* ← *0*
`BFI Rd, Rn, #lsb, #width`	Bit field insert. *Rd[(width+lsb–1):lsb]* ← *Rn[(width-1):0]*
`MVN Rd, Op2`	Logically negate all bits. *Rd* ← *0xFFFFFFFF EOR Op2*

Table 4-5. Bitwise Logic Instructions

These instructions operate at the bit level. They perform logic operations for each pair of bits that are at the same position of inputs. For example, suppose `r0 = 0xD5755755` and `r1 = 0xAABAAAA9`, the following shows the result of various bitwise logic operations.

```
AND r2, r0, r1   ; r2 = r0 bitwise AND r1 ⟹ r2 = 0x80300201

          r0   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
          r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
          r2   1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

ORR r2, r0, r1   ; r2 = r0 bitwise OR r1 ⟹ r2 = 0xFFFFFFFD

          r0   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
          r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
          r2   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1

EOR r2, r0, r1   ; r2 = r0 bitwise Exclusive OR r1 ⟹ r2 = 0x7FCFFDFC

          r0   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
          r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
          r2   0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0

ORN r2, r0, r1   ; r2 = r0 bitwise NOT OR r1 ⟹ r2 = 0xD5755757

          r0   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
          r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
       NOT r1  0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0
          r2   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1
```

```
BIC r2, r0, r1   ; r2 = bit clear r0 according to r1 ⟹ r2 = 0x55455554
```

```
        r0   1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
        r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
    NOT r1   0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0
        r2   0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0
```

```
MVN r2, r1       ; r2 = NOT r1 ⟹ r2 = 0x55455556
```

```
        r1   1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
        r2   0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0
```

Bit mask

We often use bit masks to manipulate a particular subset of binary bits in a single bitwise operation conveniently. For an integer *N*, its bit mask is constructed as follows:

- The mask has the same number of bits in binary as the integer *N*.
- Bit mask(*i*) is set if bit *N*(*i*) is to be operated; otherwise, mask(*i*) is `0`.
- If *N*(*i*) is `1`, we say bit *N*(*i*) is masked.

The mask can separate the binary bits of an integer into two parts. The part selected by the bit mask is examined or modified, and the other part is ignored. If a bit in the bit mask is 1, the corresponding bit in the target variable is chosen. For example, a mask of `0b00110100` (`0x34`) selects bits 2, 4, and 5 of the target variable. The following gives C and assembly example programs to set, clear, toggle and check bits in a variable.

- `N = 0xA2 = 0b10100010`
- `Mask = 0x34 = 0b00110100`

Bitwise Operators	Symbol	Example
AND	&	`C = N & Mask; // C = 0b00100000 = 0x20`
OR	|	`C = N | Mask; // C = 0b10110110 = 0xB6`
EXCLUSIVE-OR (EOR)	^	`C = N ^ Mask; // C = 0b10010111 = 0x97`
NOT	~	`C = ~N; // C = 0b01011101 = 0x5D`
SHIFT RIGHT	>>	`C = N >> 2; // C = 0b00101000 = 0x28`
SHIFT LEFT	<<	`C = N << 2; // C = 0b10001000 = 0x88`

Example 4-2. The mask selects bit 2, 4, and 5.

Checking a bit via bitwise AND (&)

C Program	Assembly Program 1	Assembly Program 2

`char a = 0x34;`
`char mask = 1<<5;`
`char b;`
`// Check bit 5`
`b = a & mask;`
	`LDR r0,#0x34 ; r0 = a`
	`LDR r1,#(1<<5) ; r1 = mask`
	`ANDS r2,r0,r1 ; r2 = b`
		`LDR r0,#0x34 ; r0 = a`
		`ANDS r2,r0,#(1<<5)`

We can check whether a bit is 1 by performing bitwise AND operation with the corresponding mask. In this example, register r2, representing variable b, is non-zero only when bit 5 in register r0 is 1.

Setting a bit via bitwise OR (|)

C Program	Assembly Program 1	Assembly Program 2

`char a = 0x34;`
`char mask = 1<<5;`
`// Set bit 5`
`a |= mask;`
	`LDR r0,#0x34 ; r0 = a`
	`LDR r1,#(1<<5) ; r1 = mask`
	`ORR r0,r0,r1`
		`LDR r0,#0x34 ; r0 = a`
		`ORR r0,r0,#(1<<5)`

ORR a bit with 1 sets this bit. ORR a bit with 0 does not change it. Therefore, ORR a variable with the mask sets all bits marked by the mask, while keeping all the other bits unchanged.

Clearing a bit via bitwise AND (&)

C Program	Assembly Program 1	Assembly Program 2

`char a = 0x34;`
`char mask = 1<<5;`
`// Reset bit 5`
`a &= ~mask;`
	`LDR r0,#0x34 ; r0 = a`
	`LDR r1,#(1<<5) ; r1 = mask`
	`MVN r1,r1 ; NOT`
	`EOR r0,r0,r1`
		`LDR r0,#0x34 ; r0 = a`
		`BIC r0,#(1<<5)`

AND a bit with 0 clears this bit. AND a bit with 1 does not change it. Therefore, AND a variable with the negation of the mask clears all data bits marked by the mask.

Toggling a bit via bitwise EOR (^)

C Program	Assembly Program 1	Assembly Program 2

`char a = 0x34;`
`char mask = 1<<5;`
`// Toggle bit 5`
`a ^= mask;`
	`LDR r0,#0x34 ; r0 = a`
	`LDR r1,#(1<<5) ; r1 = mask`
	`EOR r0,r0,r1`
		`LDR r0,#0x34 ; r0 = a`
		`EOR r0,r0,#(1<<5)`

As illustrated by the truth table given in Table 4-6, exclusive OR (EOR) between 1 and a bit inverts this bit, and exclusive OR between 0 and a bit keeps the bit unchanged. Therefore, exclusive OR between a data and its mask toggles all data bits masked.

Data bit	Mask bit	Data bit ⊕ Mask bit
0	1	1
1	1	0
0	0	0
1	0	1

Table 4-6. Truth table of Exclusive OR. Use bitwise EOR with a 1 to toggle a bit.

In C, the Boolean operations are **A && B** (Boolean and), **A||B** (Boolean or), and **!B** (Boolean not), which are different from the above bitwise operations.

- The Boolean operators perform word-wide operations, not bitwise. For example, "`0x10 & 0x01`" equals `0x00`, but "`0x10 && 0x01`" equals `0x01`.
- The bitwise negation expression "`~0x01`" equals `0xFFFFFFFE`, but Boolean `NOT` expression "`! 0x01`" equals `0x00`.

Using EQU to define a mask in assembly

To make programs easier to read, we often give a name to a mask. For example, we define the bit masks for the clock enable and disable bits for GPIO ports.

```
RCC_AHB2ENR_GPIOAEN  EQU  (0x00000001)  ; GPIO port A clock enable
RCC_AHB2ENR_GPIOBEN  EQU  (0x00000002)  ; GPIO port B clock enable
RCC_AHB2ENR_GPIOCEN  EQU  (0x00000004)  ; GPIO port C clock enable

LDR r7, =RCC_BASE                  ; Address of reset and clock control (RCC)
LDR r1, [r7, #RCC_AHB2ENR]         ; Load AHB2ENR from memory into r1
ORR r1, r1, #RCC_AHB2ENR_GPIOAEN ; Enable clock of GPIO port A
ORR r1, r1, #RCC_AHB2ENR_GPIOBEN ; Enable clock of GPIO port B
ORR r1, r1, #RCC_AHB2ENR_GPIOCEN ; Enable clock of GPIO port C
STR r1, [r7, #RCC_AHB2ENR]         ; Save to RCC->AHB2ENR
```

By using `EQU`, the program defines three constants (such as `RCC_AHB2ENR_GPIOAEN`). These constants are bit masks, which make it easier to manipulate individual bits. It is not a good programming style to set or clear bits directly by using constants instead of a named mask, such as the following instruction.

```
ORR r1, r1, #0x7      ; Set bits 0, 1, and 2
```

Updating program status flags in assembly

The logic operations with `S` suffix, including `ANDS`, `ORRS`, `EORS`, `ORNS`, and `MVNS` update the `N`, `Z`, `C` flags in `APSR`. None of them affects the `V` flag. Neither `BFC` nor `BFI` updates these four flags.

It is understandable that a logical instruction with `S` suffix can update the negative and zero flags in `APSR`. You may wonder how a logic operation can change the carry flag. The carry flag is updated when the second source operand uses the Barrel shifter. For example,

```
ANDS r0, r1, r2, LSL #3    ; Update N, Z, C flags. (V is unchanged.)
```

The carry flag of the above `ANDS` operation is, in fact, the carry of the "`LSLS r2, #3`" operation.

4.7 Reversing the Order of Bits and Bytes

Instructions for reversing the bit or byte orders are useful, particularly when data exchanged between two systems have different formats. For example, the REV instruction is useful to convert data that are exchanged between different endian systems.

RBIT Rd, Rn	Reverse bit order in a word. *for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]*
REV Rd, Rn	Reverse byte order in a word. *Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8],* *Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]*
REV16 Rd, Rn	Reverse byte order in each halfword. *Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],* *Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]*
REVSH Rd, Rn	Reverse byte order in bottom halfword and sign extend. *Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],* *Rd[31:16] ← Rn[7] & 0xFFFF*

Table 4-7. Instructions for changing the order of bits or bytes

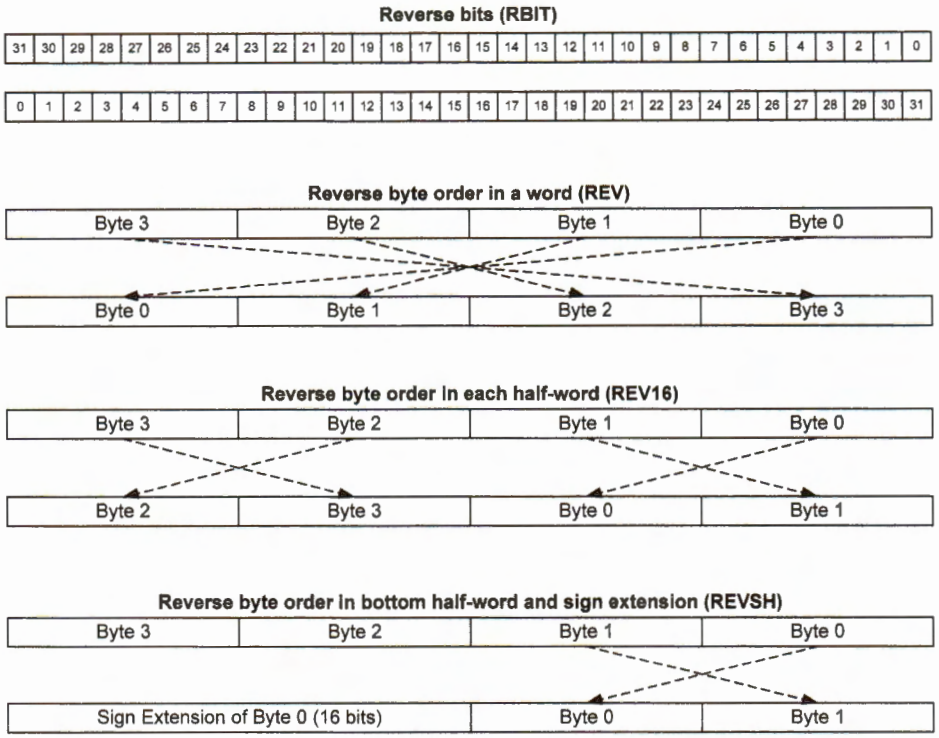

Figure 4-5. Reverse bit or byte order

The following gives a few examples of changing the bit or the byte order of a value stored in register r0.

```
LDR   r0, =0x12345678  ; r0 = 0x12345678
RBIT  r1, r0           ; Reverse bits, r1 = 0x1E6A2C48

LDR   r0, =0x12345678  ; r0 = 0x12345678
REV   r1, r0           ; Reverse byte order, r1 = 0x78563412
REV16 r2, r0           ; Reserve byte order in halfwords, r2 = 0x34127856

LDR   r0, =0x33448899  ; r0 = 0x33448899
REVSH r1, r0           ; Reverse bytes in lower halfword and extend sign
                       ; r0 = 0xFFFF9988
```

Example 4-3. Assembly codes to change the order of bits or bytes.

4.8 Sign and Zero Extension

Most computers represent signed integers in two's complement. When a signed integer is converted to another signed integer with more bits, the sign bit (*i.e.*, the most significant bit or the leftmost bit) should be duplicated to maintain the integer's sign. Duplicating the sign bit is called ***sign extension***.

When an unsigned integer is converted to another unsigned integer with more bits, ***zero extension*** is deployed to place zeros in the upper bits of the output.

In Example 4-4, when signed variable *a* and *b* are assigned to variable *c*, sign extension is performed. However, when unsigned *d* is assigned to *e*, zero-extension is performed.

- The `int_8` (signed char), `int_16` (signed short), and `int_32` (signed integer) are standard integer data types defined in the header file *stdint.h*. They define 8-, 16- and 32-bit signed integers, respectively.
- The unsigned integer definition includes `uint_8` (unsigned char), `uint_16` (unsigned short), and `uint32_t` (unsigned integer).

```
int_8   a = -1;  // a signed 8-bit integer,  a = 0xFF
int_16  b = -2;  // a signed 16-bit integer, b = 0xFFFE
int_32  c;       // a signed 32-bit integer
c = a;           // sign extension, c = 0xFFFFFFFF
c = b;           // sign extension, c = 0xFFFFFFFE

uint_8  d = 1;   // an unsigned 8-bit integer, d = 0x01
uint_32 e;       // an unsigned 32-bit integer
e = d;           // zero extension, e = 0x00000001
```

Example 4-4. Example of sign and zero extension performed in a C program

Table 4-8 shows assembly instructions that perform sign and zero extension.

`SXTB {Rd,} Rm {,ROR #n}`	Sign extend a byte. *Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[7:0])*
`SXTH {Rd,} Rm {,ROR #n}`	Sign extend a halfword. *Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[15:0])*
`UXTB {Rd,} Rm {,ROR #n}`	Zero extend a byte. *Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[7:0])*
`UXTH {Rd,} Rm {,ROR #n}`	Zero extend a halfword. *Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[15:0])*

Table 4-8. Instructions for zero and sign extension

The following program gives a few examples of sign and zero extension. Assume the value of register `r0` is `0x11228091`.

```
; r0 = 0x11228091
SXTB r1, r0  ; r1 = 0xFFFFFF91, sign extend a byte
SXTH r1, r0  ; r1 = 0xFFFF8091, sign extend a halfword
UXTB r1, r0  ; r1 = 0x00000091, zero extend a byte
UXTH r1, r0  ; r1 = 0x00008091, zero extend a halfword
```

Example 4-5. Example code of sign and zero extension

4.9 Data Comparison

There are four different data comparison instructions.

`CMP Rn, Op2`	Compare	Set NZCV flags on *Rn – Op2*
`CMN Rn, Op2`	Compare negative	Set NZCV flags on *Rn + Op2*
`TST Rn, Op2`	Test	Set NZCV flags on *Rn* AND *Op2*
`TEQ Rn, Op2`	Test equivalence	Set NZCV flags on *Rn* EOR *Op2*

Table 4-9. Data comparison instructions

- The `CMP` instruction subtracts the value of `Op2` from the value in Rn. It is the same as a `SUBS` instruction, except that the processor discards the result. `CMP` updates the `N`, `Z`, `C`, and `V` flags per the subtraction result.
- The `CMN` instruction adds the value of `Op2` to the value in `Rn`. "`CMN Rn, Op2`" is like "`ADDS Rn, Op2`" except that the result is discarded. `CMN` updates `N`, `Z`, `C`, and `V`.
- The instruction "`TST Rn, Op2`" performs a bitwise `AND` operation on `Rn` and `Op2`. Different from "`ANDS Rn, Op2`", the `TST` instruction discards the result. `TST`

updates the N and Z flags. If Op2 uses the Barrel shifter, TST also updates the C flag during the calculation of Op2. However, it does not affect the V flag.

- The TEQ instruction performs a bitwise exclusive OR operation on Rn and Op2. "TEQ Rn, Op2" is the same as "EOR Rn, Op2" except that the result is discarded. TEQ updates the N, Z, and C flags.

TEQ and TST have different usages.

- TEQ is to check whether two values are equal, and TST is to exam whether target bits set by the second operand are clear. After TEQ completes, the zero flag is set if two operands are equal; otherwise, the zero flag is clear.
- TST cannot check the equivalence of two operands. For example, when r0 = 0b1010 and r1 = 0b0101, the instruction "TST r0, r1" sets the zero flag because the result of AND is 0. However, these two operands are not equal.

	r0	r1	Action
TST r0, r1	**0b1010**	**0b0101**	**Set flag Z**
TEQ r0, r1	**0b1010**	**0b0101**	**Clear flag Z**
TST r0, r1	**0b1010**	**0b1010**	**Clear flag Z**
TEQ r0, r1	**0b1010**	**0b1010**	**Set flag Z**

The following gives a few examples of data comparison.

```
CMP r0, #3          ; Compare r0 with 3
CMN r0, #10         ; Compare r0 with -10
CMP r0, r1          ; Compare r0 and r1
TEQ r0, #'?'        ; Compare r0 with ASCII value of '?' (0x3F)

MOV r1, #(1<<31)    ; r1 = 0x80000000
TST r0, r1          ; check whether the sign bit is 1.
```

4.10 Data Movement between Registers

We can classify instructions for moving data between registers into two categories:

- Move data between two general-purpose registers (r0 – r12)
- Move data between a general-purpose register and a special-purpose register

MOV (move) and MVN (move not) are used to copy data between two general-purpose registers. MRS and MSR move content between special registers and general registers.

Special registers include APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, and CONTROL.

MOV	*Rd* ← *operand2*
MVN	*Rd* ← NOT *operand2*
MRS Rd, spec_reg	Move from special register to general register
MSR spec_reg, Rm	Move from general register to special register

Table 4-10. Data copy instructions

MOV and MVN can also load an immediate number into a register, as introduced in Chapter 5.4.4. The following are a few examples of MOV and MVN.

```
MOV r4, r5          ; Copy r5 to r4
MVN r4, r5          ; r4 = bitwise logical NOT of r5
MOV r1, r2, LSL #3  ; r1 = r2 << 3
MOV r0, PC          ; Copy PC (r15) to r0
MOV r1, SP          ; Copy SP (r14) to r1
```

The following instructions copy a special-purpose register to a general-purpose register.

```
MRS r0, APSR        ; Read flag state into r0
MRS r0, IPSR        ; Read exception/interrupt state into r0
MRS r0, EPSR        ; Read execution state into r0
MRS r0, PSR         ; Copy combined CPSR, EPSR, and SPSR into r0
```

The following shows how to copy a general-purpose register to a special-purpose register.

```
MSR APSR, r0        ; Write flag state
MSR BASEPRI, r0     ; Write to base priority mask register; Disable
                    ; exceptions with same or lower priority level
```

4.11 Bit Field Extract

Table 4-11 shows two instructions that extract adjacent bits from one register.

- The #lsb parameter, ranging from 0 to 31, specifies the starting position.
- The #width parameter, ranging from 1 to (32 – #lsb), indicates the number of contiguous bits to be extracted.

SBFX Rd, Rn, #lsb, #width	Signed Bit Field Extract *Rd[(width–1):0]* ← *Rn[(width+lsb–1):lsb]* *Rd[31:width]* ← *Replicate(Rn[width+lsb–1])*
UBFX Rd, Rn, #lsb, #width	Unsigned Bit Field Extract *Rd[(width–1):0]* ← *Rn[(width+lsb–1):lsb]* *Rd[31:width]* ← *Replicate(0)*

Table 4-11. Bit field extract instructions

UBFX simply places zero in the upper bits, while SBFX duplicates the sign bit. The sign bit, in this case, is not the most significant bit; instead, it is the bit at the position of #width + #lsb - 1.

The following shows two examples of extracting 8 bits from register r3, starting at bit 4. One has no sign extension, and the other has sign extension.

```
; Assume r3 = 0x1234CDEF
UBFX r4, r3, #4, #8    ; r4 = 0x000000DE (zero extension)
SBFX r4, r3, #4, #8    ; r4 = 0xFFFFFFDE (sign extension)
```

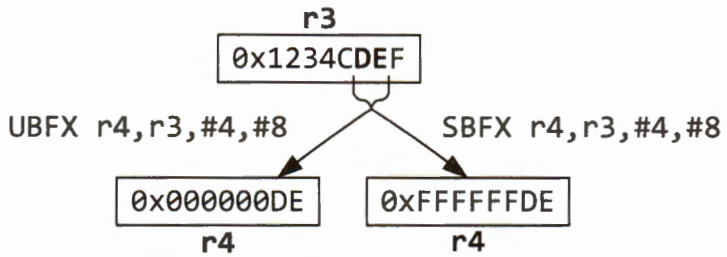

Figure 4-6. Extracting 8 bits starting at bit position 4, *i.e.*, r4 = r3[11:4]

4.12 Exercises

1. LSL (logic shift left) can speed up some special multiplication because it runs much faster than MUL. Use LSL to implement the following C statements.

   (1) $x = 31 * x$;
   (2) $x = 38 * x$;
   (3) $x = 17 * x$;

2. Suppose r0 = 0x0F0F0F0F, and r1 = 0xFEDCBA98, find the result of the following operations.

```
(1) EOR r3, r1, r0
(2) ORR r3, r1, r0
(3) AND r3, r1, r0
(4) BIC r3, r1, r0
(5) BFI r3, r1, #4, #8
(6) MVN r3, r1
(7) MVN r3, r0
(8) MVN r3, r0
    ADD r3, r1, r3
```

3. Suppose r0 = 0x56789ABC, find the result of the following operations. These operations run independently.

```
(1) RBIT  r1, r0
(2) REV   r1, r0
(3) REV16 r1, r0
(4) REVSH r1, r0
```

4. Translate the following C statement into an assembly program, assuming 16-bit signed integers x, y and z (*i.e.*, signed short) are stored in 32-bit register r0, r1, and r2, respectively.

$$x = x * y + z - x;$$

5. Translate the following C statement into an assembly program, assuming 16-bit unsigned integers x and y (*i.e.*, unsigned short) are stored in register r0, and r1, respectively.

$$x = x \,\%\, y;$$

6. Write an assembly program that calculates the value of the following given polynomial, assuming signed integers x and y are stored in register r0 and r1, respectively.

$$y = 3x^3 - 7x^2 + 10x - 11.$$

7. Write an assembly program that calculates the remainder of the division between two unsigned 32-bit integers.

8. Explain why Cortex-M processors do not provide any left rotation instructions. They only offer ROR (rotate right) and RRX (rotate right extended).

9. Explain the difference between the Barrel shifter's role in the following instructions:

```
(1) ANDS r1, r2, r3, LSL #3
(2) ADDS r1, r2, r3, LSL #3
```

10. Write an assembly program that reverses the byte order of a register without using the REV instruction.

11. Write an assembly program that swaps the upper halfword and the lower halfword of a register.

12. Implement the BIC (bitwise clear) instruction by using other assembly instructions.

13. Suppose Mask = 0x00000F0F and P = 0xABCDABCD. What are the results of the following bitwise operations?

```
(1) Q = P & Mask;
(2) Q = P | Mask;
(3) Q = P ^ Mask;
(4) Q = ~Mask;
(5) Q = P & ~Mask;
```

14. Suppose `r0 = 0xFFFFFFFF`, `r1 = 0x00000001`, and `r2 = 0x00000000`. Initially the `N`, `Z`, `C`, and `V` flags are zero. Find the value of the `N`, `Z`, `C`, and `V` flags of the following instructions. (Assume each instruction runs individually, *i.e.*, these instructions are not part of a program.)

```
(1) ADD  r3, r0, r2
(2) SUBS r3, r0, r0
(3) ADDS r3, r0, r2
(4) LSL  r3, r0, #1
(5) LSRS r3, r1, #1
(6) ANDS r3, r0, r2
```

15. Suppose we have a hypothetical processor, of which each register has only five bits. `r0 = 0b11101` and `r1 = 0b10110`. What are the `N`, `Z`, `C`, and `V` flags of the following instructions? Assume initially `N = 0`, `Z = 0`, `C = 1`, `V = 0`, and these instructions are executed independently (*i.e.*, they are NOT part of a program)

```
(1) ADDS r3, r0, r1
(2) SUBS r3, r0, r1
(3) EOR  r3, r0, r1
(4) ANDS r3, r1, r1, LSL #3
```

16. What is the value in register `r1`? Assume `r0 = 0x00001016`

```
(1) USAT r1, #8, r0
(2) SSAT r1, #8, r0
(3) USAT r1, #9, r0
(4) SSAT r1, #9, r0
```

17. Write short assembly programs to complete the following tasks.

    (1) Reset all even bits in register `r0` to zero and keep all odd bits unchanged
    (2) Set all odd bits in register `r0` to one and keep all even bits unchanged
    (3) Toggle all odd bits in register `r0` and keep all even bits unchanged