ARM Assembly Language Programming & Architecture

Muhammad Ali Mazidi
Sarmad Naimi
Sepehr Naimi
Janice Mazidi

ARM Assembly Language Programming & Architecture

Muhammad Ali Mazidi

Sarmad Naimi

Sepehr Naimi

Janice Mazidi

“Regard man as a mine rich in gems of inestimable value. Education can, alone, cause it to reveal its treasures, and enable mankind to benefit therefrom.”

Baha’u’llah

Dedication

To the faculty, staff, and students of BIHE university for their dedication and steadfastness.

Preface

The ARM processor is becoming the dominant CPU architecture in the computer industry. It is already the leading architecture in cell phones and tablet computers. With such a large number of companies producing ARM chips, it is certain that the architecture will move to the laptop, desktop and high-performance computers currently dominated by x86 architecture from Intel and AMD. Currently the PIC and AVR microcontrollers dominate the 8-bit microcontroller market. The ARM architecture will have a major impact in this area too as designers become more familiar with its architecture. This book is intended as an introduction to ARM assembly language programming and architecture. We assume no prior background in assembly language programming with other CPUs. However, we urge you to study Chapter 0 covering the fundamentals of digital systems such as hexadecimal numbers, various types of memory, memory and I/O interfacing, and memory address decoding. Chapter 0 is available free of charge on our website http://www.MicroDigitalEd.com/ARM/ARM_books.htm

Universities and colleges

This book is intended for both academic and industry readers. The answers to review questions at end of each section are provided at end of the chapter. If you are using this book for a university course there are end-of-chapter problems that can be found on www.MicroDigitalEd.com/ARM/ARM_books.htm

Our upcoming books in the ARM series

This book covers the Assembly language programming of the ARM chip. The ARM Assembly language is standard regardless of who makes the chip. The ARM licensees are free to implement the on-chip peripheral (ADC, Timers, I/O, …) as they choose. Since the ARM peripherals are not standard among the various vendors, we have dedicated a separate book to each vendor. The following books are planned in this series:

TI ARM Peripheral Programming and Interfacing

Freescale ARM Peripheral Programming and Interfacing

NXP ARM Peripheral Programming and Interfacing

Atmel ARM Peripheral Programming and Interfacing

The above books use C language to program the peripherals.

Finally, we would like to thank professors Shujen Chen, Rabah Aoufi, and Clyde Knight for their reading of the book before publication. We sincerely appreciate their insights and inputs. We would also like to thank Fada Mahmoudi, Mozhde Amiri, Keyvan Roshani, Azalia Yaghini, Arash Zamani, Parham Fazlali, and Ashkan Farivar for sharing their comments on the prepublication version of this e-book.

Contact us at the following email address:

mdebooks@yahoo.com

and please place ARM book in subject line of your email.

Table of Contents

Chapter 1: The History of ARM and Microcontrollers

In Section 1.1 we look at the history of microcontrollers then we introduce some of the available microcontrollers. The history of ARM is provided in Section 1.2.

Section 1.1: Introduction to Microcontrollers

The evolution of Microprocessors and Microcontrollers

In early computers, CPUs were designed using a number of vacuum tubes. The vacuum tube was bulky and consumed a lot of electricity. The invention of transistors, followed by the IC (Integrated Circuit), provided the means to put a CPU on printed circuit boards. The advances in IC technology allowed putting the entire CPU on a single IC chip. This IC was called a *microprocessor*. Some of the microprocessors are the x86 family of Intel used widely in desktop computers, and the 68000 of Motorola. The microprocessors do not contain RAM, ROM, or I/O peripherals. As a result, they must be connected externally to RAM, ROM and I/O, as shown in Figure 1-1.

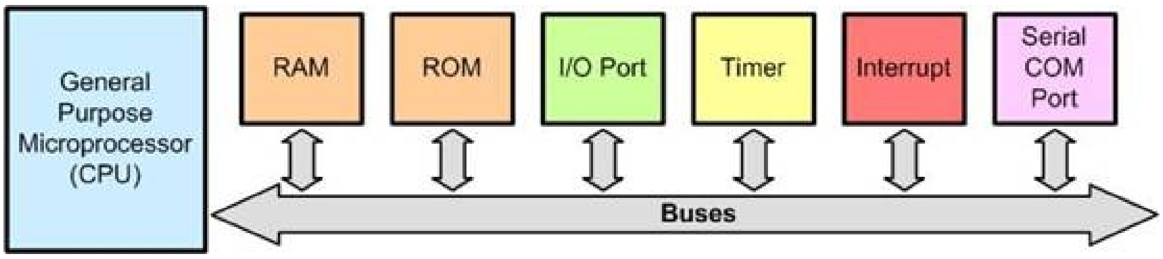


Figure 1- 1: A Computer Made by General Purpose Microprocessor

In the next step, the different parts of a system, including CPU, RAM, ROM, and I/Os, were put together on a single IC chip and it was called *microcontroller*. SOC (System on Chip) and MCU (Micro Controller Unit) are other names used to refer to microcontrollers. Figure 1-2 shows the simplified view of the internal parts of microcontrollers.

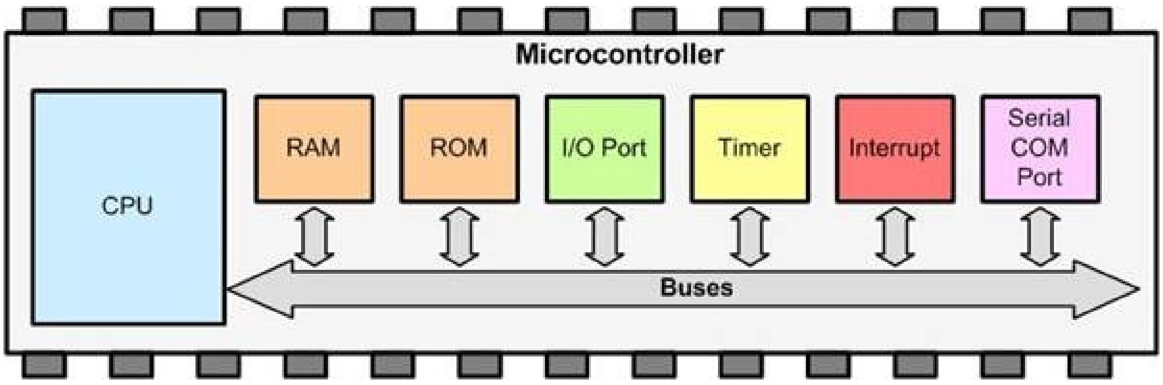


Figure 1- 2: Simplified View of the Internal Parts of Microcontrollers (SOC)

Since the microcontrollers are cheap and small, they are widely used in many devices.

Types of Computers

Typically, computers are categorized into 3 groups: desktop computers, servers, and embedded systems.

Desktop computers, including PCs, tablets, and laptops, are general purpose computers. They can be used to play games, read and edit articles, and do any other task just by running the proper application programs. The desktop computers use microprocessors.

In contrast, embedded systems are special-purpose computers. In embedded system devices, the software application and hardware are embedded together and are designed to do a specific task. For example, the Kindle, digital camera, vacuum cleaner, mp3 player, mouse, keyboard, and printer, are some examples of embedded systems. In most cases embedded systems run a fixed program and contain a microcontroller. It is interesting to note that embedded systems are the largest class of computers though they are not normally considered as computers by the general public.

Servers are the fast computers which might be used as web hosts, database servers, and in any application in which we need to process a huge amount of data such as weather forecasting. Similar to desktop computers, servers are made of microprocessors but, multiple processors are usually used in each server. Both servers and desktop computers are connected to a number of embedded system devices such as mouse, keyboard, disk controller, Flash stick memory and so on.

A Brief History of the Microcontrollers

In the 1980s and 1990s, Intel and Motorola dominated the field of microprocessors and microcontrollers. Intel had the x86 (8088/86, 80286, 80386, 80486, and Pentium). Motorola (now Freescale) had the 68xxx (68000, 68010, 68020, etc.). Many embedded systems used Intel's 32-bit chips of x86 (386, 486, Pentium) and Motorola's 32-bit 68xxx for high-end embedded products such as routers. For example, Cisco routers used 68xxx for the CPU. At the low end, the 8051 from Intel and 68HC11 from Motorola were the dominant 8-bit microcontrollers. With the introduction of PIC from Microchip and AVR from Atmel, they became major players in the 8-bit market for microcontroller. At the time of this writing, PIC and AVR are the leaders in terms of volume for 8-bit microcontrollers. In the late 1990s, the ARM microcontroller started to challenge the dominance of Intel and Motorola in the 32-bit market. Although both Intel and Motorola used RISC features to enhance the performance of their microprocessors, due to the need to maintain compatibility with legacy software, they could not make a clean break and start over. Intel used massive amounts of gates to keep up the performance of x86 architecture and that in turn increased the power consumption of the x86 to a level unacceptable for battery-powered embedded products. Meanwhile Freescale (Motorola) streamlined the instructions of the 68xxx CPU and created a new line of microprocessors called ColdFire, while at the same time worked with IBM to design a new RISC processor called PowerPC. While both PowerPC and Coldfire are still alive and being used in the 32-bit market, it is ARM which has become the leading microcontroller in the 32-bit market.

Currently Available Microcontrollers

There are many microcontrollers available in the market. Some of them are listed in Table 1-1.

32-bit
ARM, AVR32 (Atmel), ColdFire (Freescale), MIPS32, PIC32 (Microchip), PowerPC, TriCore (Infineon), SuperH

16-bit
MSP430 (TI), HCS12 (Freescale), PIC24 (Microchip), dsPIC (Microchip)
8-bit
8051, AVR (Atmel), HCS08 (Freescale), PIC16, PIC18

Table 1- 1: Some Microcontrollers

Introduction to some 32-bit microcontrollers

x86: The x86 and Pentium processors are based on the 32-bit architecture of the 386. Although both Intel and AMD are pushing the x86 into the embedded market, due to the high power consumption of these chips, the embedded market has not embraced the x86. Intel is working hard to make a low-power version of the 386 called Atom available for the embedded market.

PIC32: It is based on the MIPS architecture and is getting some attention due to the fact it shares some of the peripherals with the PIC24/PIC18 chips and also using the MPLAB for IDE. Microchip hopes the free MPLAB IDE and engineers' knowledge of the 8-bit PIC will attract embedded developers to the PIC32 as they move to 32-bit systems for their high end embedded products.

ColdFire: The Freescale (formerly Motorola) is based on the venerable 680x0 (68000, 68010) so popular in the 1980s and 1990s. They streamlined the 68000 instructions to make it more RISC-type architecture and is the top seller of 32-bit processors from the Freescale. In recent years Freescale revamped and redesigned the 8-bit HCS08 (from the 6808) to share some of the peripherals with ColdFire and are pushing them under the name Flexis. They hope engineers use the HCS08 at the low-end and move to Coldfire for high-end of the embedded products with minimum learning curve.

PowerPC: This was developed jointly by IBM and Freescale. It was used in the Apple Mac for a few years. Then Apple switched to x86 for a while and currently is using ARM in all their products. Nowadays, both Freescale and IBM market the PowerPC for the high-end of the embedded systems.

How to choose a microcontroller

The following two factors can be important in choosing a microcontroller:

- ***Chip characteristics:*** Some of the factors in choosing a microcontroller chip are clock speed, power consumption, price, and on-chip memories and peripherals.
- ***Available resources:*** Other factors in choosing a microcontroller include the IDE compiler, legacy software, and multiple sources of production.

Review Questions

1. True or false. Microcontrollers are normally less expensive than microprocessors.
2. When comparing a system board based on a microcontroller and a general- purpose microprocessor, which one is cheaper?

3. A microcontroller normally has which of the following devices on-chip?

   (a) RAM (b) ROM (c) I/O (d) all of the above

4. A general-purpose microprocessor normally needs which of the following devices to be attached to it?

   (a) RAM (b) ROM (c) I/O (d) all of the above

5. An embedded system is also called a dedicated system. Why?
6. What does the term embedded system mean?
7. Why does having multiple sources of a given product matter?

Section 1.2: The ARM Family History

In this section, we look at the ARM and its history.

A brief history of the ARM

The ARM came out of a company called Acorn Computers in United Kingdom in the 1980s. Professor Steve Furber of Manchester University worked with Sophie Wilson to define the ARM architecture and instructions. The VLSI Technology Corp. produced the first ARM chip in 1985 for Acorn Computers and was designated as Acorn RISC Machine (ARM). Unable to compete with x86 (8088, 80286, 80386, …) PCs from IBM and other personal computer makers, the Acorn was forced to push the ARM chip into the single-chip microcontroller market for embedded products. That is when Apple Corp. got interested in using the ARM chip for the PDA (personal digital assistants) products. This renewed interest in the chip led to the creation of a new company called ARM (Advanced RISC Machine). This new company bet its entire fortune on selling the rights to this new CPU to other silicon manufacturers and design houses. Since the early 1990s, an ever increasing number of companies have licensed the right to make the ARM chip. See Table 1-2 for the major milestones of the ARM.

Also see *http://www.arm.com/about/company-profile/milestones.php* for the list.

Table 1- 2: ARM Company milestones (www.ARM.com)

1982

- Acorn produced a computer for BBC named BBC micro. Good sales of the computer motivated Acorn to decide to make its own microprocessor.

1983

- Acorn and VLSI began designing the ARM microprocessor.

1985

- Acorn Computer Group developed the world's first commercial RISC processor. The ARMv1 had 2500 transistors, and worked with a frequency of 4MHz.

1987

- Acorn's ARM processor debuts as the first RISC processor for low-cost PCs

1989

- Acorn introduced ARMv3 with a frequency of 25MHz. It had a 4KB cache as well.

1990

- Advanced RISC Machines (ARM) spins out of Acorn and Apple Computer’s collaboration efforts with a charter to create a new microprocessor standard. VLSI Technology becomes an investor and the first licensee.

1991

- ARM introduced its first embeddable RISC core, the ARM6 solution using ARMv3 architecture.

1992

- GEC Plessey and Sharp licensed ARM technology

1993

- Texas Instruments licensed ARM technology
- ARM introduced the ARM7 core.

1995

- ARM announced the Thumb architecture extension, which gives 32-bit RISC performance at 16-bit system cost and offers industry-leading code density
- ARM launched Software Development Toolkit

1996

- ARM and VLSI Technology introduced the ARM810 microprocessor
- ARM and Microsoft worked together to extend Windows CE to the ARM architecture

1997

- Hyundai, Lucent, Philips, Rockwell and Sony licensed ARM technology
- ARM9TDMI family announced

1998

- HP, IBM, Matsushita, Seiko Epson and Qualcomm licensed ARM technology
- ARM developed synthesizable version of the ARM7TDMI core
- ARM Partners shipped more than 50 million ARM-powered products

1999

- LSI Logic, STMicroelectronics and Fujitsu licensed ARM technology
- ARM announced synthesizable ARM9E processor with enhanced signal processing

2000

- Agilent, Altera, Micronas, Mitsubishi, Motorola, Sanyo, Triscend and ZTEIC licensed ARM technology
- ARM launched SecurCore family for smartcards
- TSMC and UMC became members of ARM Foundry Program

2001

- ARM's share of the 32-bit embedded RISC microprocessor market grew to 76.8 per cent
- ARM announced new ARMv6 architecture
- Fujitsu, Global UniChip, Samsung and Zeevo licensed ARM technology
- ARM acquired key technologies and an embedded debug design team from Noral Micrologics Ltd

2002

- ARM announced that it had shipped over one billion of its microprocessor cores to date
- ARM technology licensed to Seagate, Broadcom, Philips, Matsushita, Micrel, eSilicon, Chip Express and ITRI
- ARM launched the ARM11 micro-architecture
- ARM launches its RealView family of development tools
- Flextronics became the first ARM Licensing Partner program member, allowing it to sub-license ARM technology to its own customers

2004

- The ARM Cortex family of processors, based on the ARMv7 architecture, is announced. The ARM Cortex-M3 is announced in conjunction, as the first of the new family of processors
- ARM Cortex-M3 processor announced, the first of a new Cortex family of processor cores
- MPCore multiprocessor launched, the first integrated multiprocessor
- OptimoDE technology launched, the groundbreaking embedded signal processing core

2005

- ARM acquired Keil Software
- ARM Cortex-A8 processor announced

2007

- Five billionth ARM Powered processor shipped to the mobile device market
- ARM Cortex-M1 processor launched – the first ARM processor designed specifically for implementation on FPGAs

- RealView Profiler for Embedded Software Analysis introduced
- ARM unveils Cortex-A9 processors for scalable performance and low-power designs

2008

- ARM announces 10 billionth processor shipment
- ARM Mali-200 GPU Worlds First to achieve Khronos Open GL ES 2.0 conformance at 1080p HDTV resolution

2009

- ARM announces 2GHz capable Cortex-A9 dual core processor implementation
- ARM launches its smallest, lowest power, most energy efficient processor, Cortex-M0

2010

- ARM launches Cortex-M4 processor for high performance digital signal control
- ARM together with key Partners form Linaro to speed rollout of Linux-based devices
- Microsoft becomes an ARM Architecture Licensee
- ARM & TSMC sign long-term agreement to achieve optimized Systems-on-Chip based on ARM processors, extending down to 20nm
- ARM extends performance range of processor offering with the Cortex-A15 MPCore processor
- ARM Mali becomes the most widely licensed embedded GPU architecture
- ARM Mali-T604 Graphics Processing Unit introduced providing industry-leading graphics performance with an energy-efficient profile

2011

- Microsoft unveils Windows on ARM at CES 2011
- IBM and ARM collaborate to provide comprehensive design platforms down to 14nm
- ARM and UMC extend partnership into 28nm
- Cortex-A7 processor launched
- Big-Little processing announced, linking Cortex-A15 and Cortex-A7 processors
- ARMv8 architecture unveiled at TechCon
- AMP announce license and plans for first ARMv8-based processor
- ARM Mali-T658 GPU launched
- ARM expands R&D presence in Taiwan with Hsinchu Design Center
- ARM and Avnet launch Embedded Software Store (ESS)
- ARM, Cadence and TSMC tape out first 20nm Cortex-A15 multicore processor

2012

- ARM, Gemalto and G&D form joint venture to deliver next-generation mobile security
- First Windows RT (Windows on ARM) devices revealed
- ARM, AMD, Imagination, MediaTek and Texas Instruments founding members of Heterogeneous System Architecture (HAS) Foundation
- ARM and TSMC work together on FinFET process technology for next-generation 64-bit ARM processors
- ARM forms first UK forum to create technology blueprint "Internet of Things" devices
- ARM named one of Britain's Top Employers
- MIT Technology Review named ARM in its list of 50 Most Innovative Companies

Currently the ARM Corp. receives its entire revenue from licensing the ARM to other companies since it does not own state of the art chip fabrication facility. This business model of making money from selling IP (intellectual property) has made ARM one of the most widely used CPU architectures in the world. Unlike Intel or Freescale who define the architecture and fabricate the chip, hundreds of companies who have licensed the ARM IP feel a level playing field when it comes to competing with the originator of the chip.

ARM and Apple

When Steve Jobs came back to run the Apple in 1996, the company was in decline. It had lost the personal computer race that had started 20 years earlier. The introduction of iPod in 2001 changed the fortune of that company more than anything else. Apple had tried to sell a PDA called Newton in the 1990s but was not successful. The Newton was using the ARM processor and it was too early for its time. The iPod used an enhanced version of ARM called ARM7 and became an instant success. iPod brought the attention to the ARM chip that it deserved. Since then Apple has been using the ARM chip in iPhones and iPads. Today, the ARM microcontroller is the CPU of choice for designing cell phone and other hand-held devices. In the future, ARM will make further in-roads into the tablet and laptop PC market now that Microsoft Corp has introduced the ARM version of its Windows operating system.

ARM family variations

Although the ARM7 family is the most widely used version, ARM is determined to push the architecture into the low end of the microcontroller market where 8- and 16-bit microcontrollers have been traditionally dominating. For this reason they have come up with a microcontroller version of ARM called Cortex. As we will see in future chapters, the Cortex family of ARM microcontrollers maintains compatibility with the ARM7 without sacrificing performance. The ARM architecture is also being pushed into high-performance systems where multicore chips such as Intel Xeon dominate.

Figure 1-3 shows some of the most widely used ARM processors. It should be emphasized that we cannot use the terms ARM family and ARM architecture interchangeably. For example, ARM11 family is based on ARMv6 architecture and

ARMv7A is the architecture of Cortex-A family.

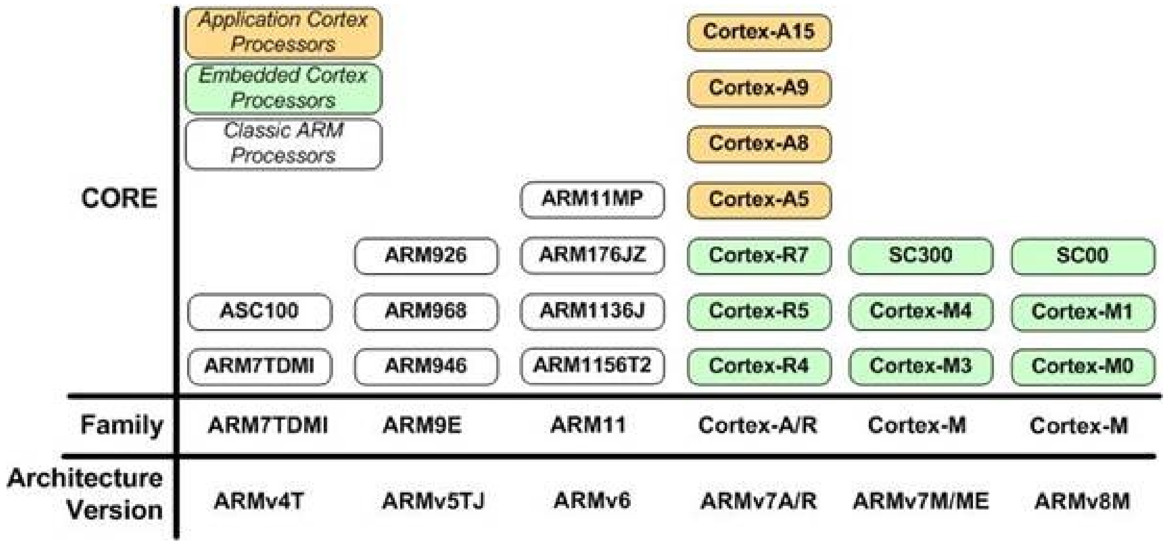


Figure 1- 3: ARM Family and Architecture

One CPU, many peripherals

ARM has defined the details of architecture, registers, instruction set, memory map, and timing of the ARM CPU and holds the copyright to it. The various design houses and semiconductor manufacturers license the IP (intellectual property) for the CPU and can add their own peripherals as they please. It is up to the licensee (design houses and semiconductor manufactures) to define the details of peripherals such as I/O ports, serial port UART, timer, ADC, SPI, DAC, I2C, and so on. As a result while the CPU instructions and architecture are same across all the ARM chips made by different vendors, their peripherals are not compatible. That means if you write a program for the serial port of an ARM chip made by TI (Texas Instrument), the program might not necessarily run on an ARM chip sold by NXP. This is the only drawback of the ARM microcontroller. The good news is the IDE (integrated development environment) such as Keil (see www.keil.com) or IAR (see www.IAR.com) do provide peripheral libraries for chips from various vendors and make the job of programming the peripherals much easier. It must be noted that in recent years ARM provides the IP for some peripherals such as UART and SPI, but unlike the CPU architecture, its adoption is not mandatory and it is up to the chip manufacturer whether to adopt it or not. This is in contrast to the Coldfire microcontroller from Freescale, in which the Freescale defines the architecture and peripherals, fabricates, sells, and supports the chip. Figure 1-4 shows the ARM simplified block diagram and Table 1-3 provides a list of some ARM vendors.

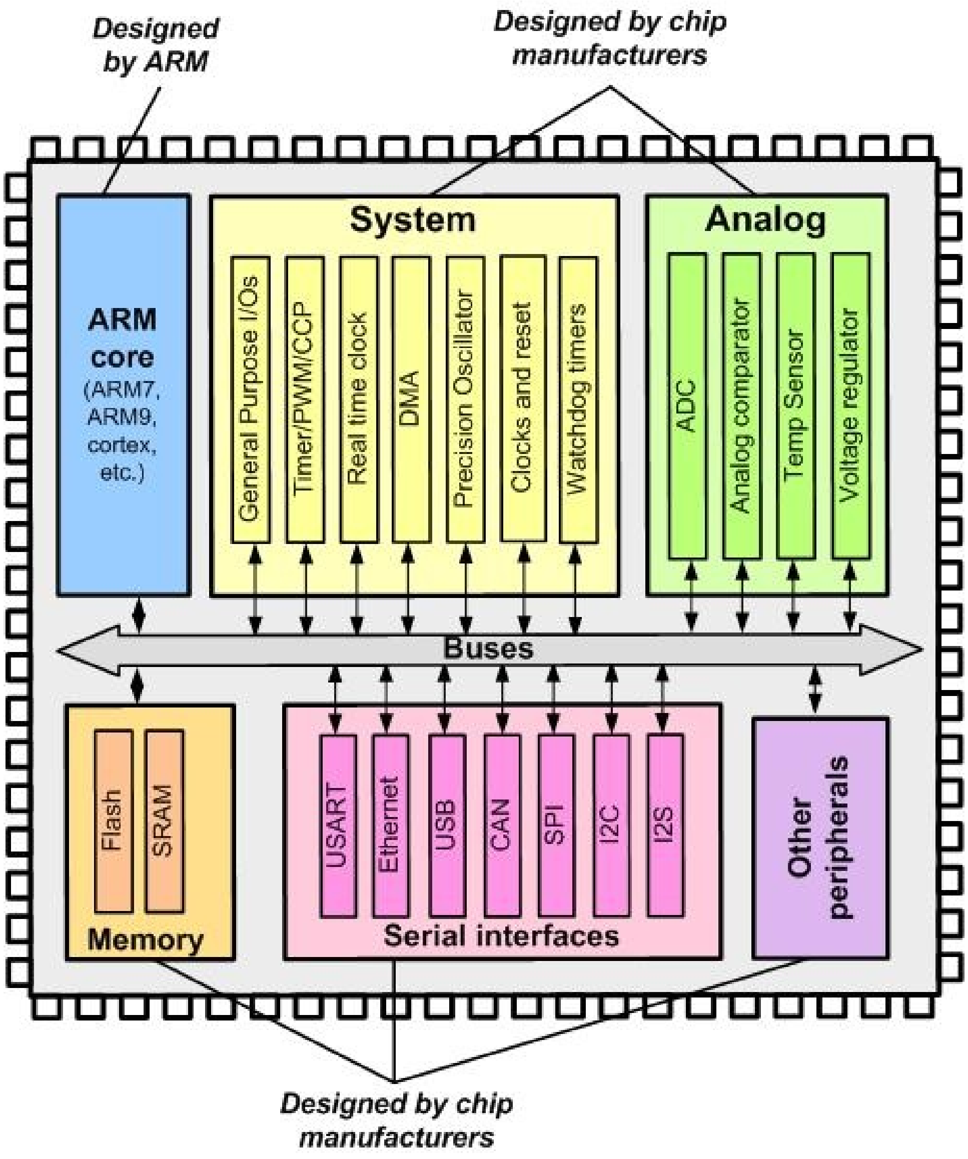


Figure 1- 4: ARM Simplified Block Diagram

Actel	Analog Devices	Atmel
Broadcom	Cypress	Ember
Dust Networks	Energy	Freescale
Fujitso	Nuvoton	NXP
Renesas	Samsung	ST
Toshiba	Texas Instruments	Triad Semiconductor

Table 1- 3: ARM Vendors

Review Questions

1. True or false. The ARM CPU instructions are universal regardless of who makes the chip.
2. True or false. The peripherals of ARM microcontroller are standardized regardless of

who makes the chip.

3. An ARM microcontroller normally has which of the following devices on-chip?

   (a) RAM (b) Timer (c) I/O (d) all of the above

4. For which of the followings, ARM has defined standard?

   (a) RAM size (b) ROM size (c) instruction set (d) all of the above

See the following websites for ARM microcontrollers and ARM trainers:

http://www.ARM.com

http://www.MicroDigitalEd.com

Problems

Section 1.1: Introduction to Microcontrollers

1. True or False. A general-purpose microprocessor has on-chip ROM.
2. True or False. Generally, a microcontroller has on-chip ROM.
3. True or False. A microcontroller has on-chip I/O ports.
4. True or False. A microcontroller has a fixed amount of RAM on the chip.
5. What components are usually put together with the microcontroller onto a single chip?
6. Intel's Pentium chips used in Windows PCs need external ______ and _____ chips to store data and code.
7. List three embedded products attached to a PC.
8. Why would someone want to use an x86 as an embedded processor?
9. Give the name and the manufacturer of some of the most widely used 8-bit microcontrollers.
10. In Question 9, which one has the most manufacture sources?
11. In a battery-based embedded product, what is the most important factor in choosing a microcontroller?
12. In an embedded controller with on-chip ROM, why does the size of the ROM matter?
13. In choosing a microcontroller, how important is it to have multiple sources for that chip?
14. What does the term "third-party support" mean?

Section 1.2: The ARM Family History

15. What does ARM stand for?
16. True or false. In ARM, architectures have the same names as families.
17. True or false. In 1990s, ARM was widely used in microprocessor world.
18. True or false. ARM is widely used in Apple products, like iPhone and iPod.
19. True or false. Currently the Microsoft Windows does not support ARM products.
20. True or false. All ARM chips have standard instructions.
21. True or false. All ARM chips have standard peripherals
22. True or false. The ARM corp. also manufactures the ARM chip.
23. True or false. The ARM IP must be licensed from ARM corp.
24. True or false. A given serial communication program is written for TI ARM chip. It

should work without any modification on Freescale ARM chip

25. True or false. A given Assembly language program is written for a given family of ARM Cortex chip. Any other Cortex ARM chip can execute the program.
26. True or false. At the present time, ARM has just one manufacturer.
27. What is the difference between the ARM products of different manufacturers?
28. Name some 32-bit microcontrollers.
29. What is Intel’s challenge in decreasing the power consumption of the x86?

Answers to Review Questions

Section 1.1

1. True
2. A microcontroller-based system
3. d
4. d
5. It is dedicated because it does only one type of job.
6. Embedded system means that the application (software) and the processor (hardware such as CPU and memory) are embedded together into a single system.
7. Having multiple sources for a given part means you are not hostage to one supplier. More importantly, competition among suppliers brings about lower cost for that product.

Section 1.2

1. True
2. False
3. d
4. c

Chapter 2: ARM Architecture and Assembly Language Programming

CPUs use registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data. In Section 2.1 we look at the general purpose registers (GPRs) of the ARM. We demonstrate the use of GPRs with simple instructions such as MOV and ADD. Memory map and memory access of the ARM are discussed in Sections 2.2 and 2.3, respectively. In Section 2.4 we discuss the status register's flag bits and how they are affected by arithmetic instructions. In Section 2.5 we look at some widely used Assembly language directives, pseudo-code, and data types related to the ARM. In Section 2.6 we examine Assembly language and machine language programming and define terms such as mnemonics, opcode, operand, and so on. The process of assembling and creating a ready-to-run program for the ARM is discussed in Section 2.7. Step-by-step execution of an ARM program and the role of the program counter are examined in Section 2.8. Section 2.9 examines some ARM addressing modes. The merits of RISC architecture are examined in Section 2.10. Section 2.11 discusses the Keil IDE.

Section 2.1: The General Purpose Registers in the ARM

CPUs use registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data. In this section we look at the general purpose registers (GPRs) of the ARM and we demonstrate the use of GPRs with simple instructions such as MOV and ADD.

ARM microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. All of ARM registers are 32-bit wide. The 32 bits of a register are shown in Figure 2-1. These range from the MSB (most-significant bit) D31 to the LSB (least-significant bit) D0. With a 32-bit data type, any data larger than 32 bits must be broken into 32-bit chunks before it is processed. Although the ARM default data size is 32-bit many assemblers also support the single bit, 8-bit, and 16-bit data types, as we will see in future chapters. The 32-bit data size of the ARM is often referred as word. This is in contrast to x86 CPU in which word is defined as 16-bit. In ARM the 16-bit data is referred to as half-word. Therefore ARM supports byte, half-word (two byte), and word (four bytes) data types.

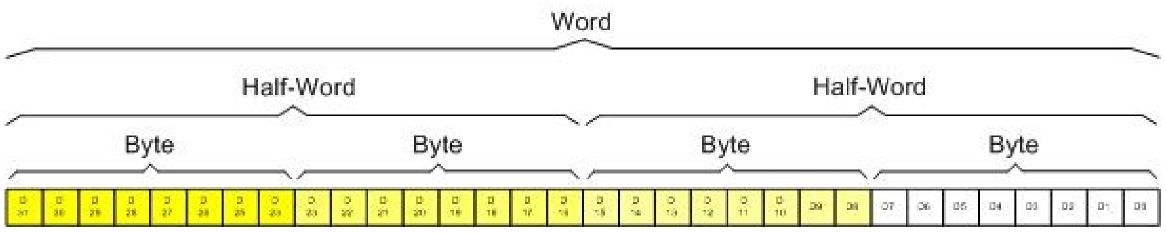

Word
Half-Word
Half-Word
Byte
Byte
Byte
Byte
D31 D30 D29 D28 D27 D26 D25 D24 D23 D22 D21 D20 D19 D18 D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Figure 2- 1: ARM Registers Data Size

In ARM there are 13 general purpose registers. They are R0–R12. See Figure 2-2. All of these registers are 32 bits wide.

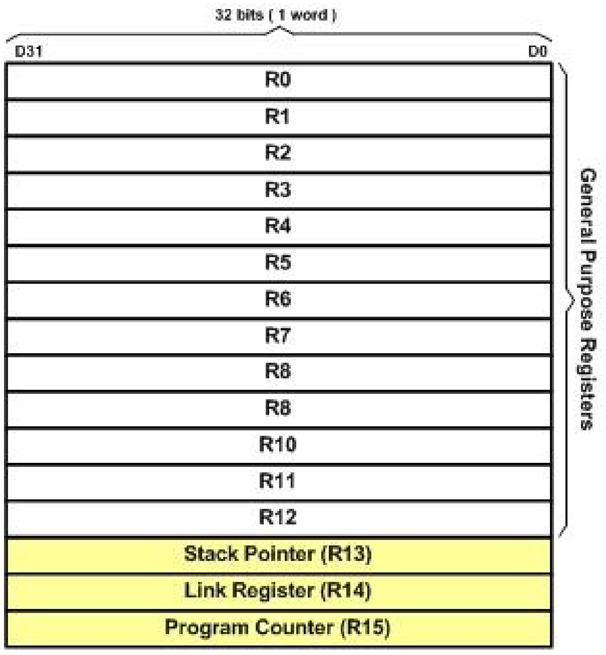


Figure 2- 2: ARM Registers

The general purpose registers in ARM are the same as the accumulator in other microprocessors. They can be used by all arithmetic and logic instructions. To understand the use of the general purpose registers, we will show it in the context of three simple instructions: MOV, ADD, and SUB. The ARM core has three special function registers of R13, R14, and R15. We will examine their use in the next section. In some ARM processors, we also have shadow registers in various operating modes designed to speed up the program execution when CPU switches task.

ARM Instruction Format

The ARM CPU uses the tri-part instruction format for most instructions. One of the most common format is:

```
instruction  destination,source1,source2
```

Depending on the instruction the source2 can be a register, immediate (constant) value, or memory. The destination is often a register or read/write memory.

MOV instruction

Simply stated, the MOV instruction copies data into register or from register to register. It has the following formats:

```
MOV     Rn,Op2 ;load Rn register with Op2 (Operand2).
;Op2 can be immediate
```

Op2 can be an immediate (constant) number #K which is an 8-bit value that can be 0–255 in decimal, (00–FF in hex). Op2 can also be a register Rm. Rn or Rm are any of the registers R0 to R15. If we see the word “immediate”, we are dealing with a constant value that must be provided right there with the instruction. Notice the # before immediate values. The following instruction loads the R2 register with a value of 0x25 (25 in hex).

```
MOV R2,#0x25  ;load R2 with 0x25 (R2 = 0x25)
```

The following instruction loads the R1 register with the value 0x87 (87 in hex).

```
MOV R1,#0x87  ;copy 0x87 into R1  (R1 = 0x87)
```

The following instruction loads R5 with the value of R7.

```
MOV R5,R7        ;copy contents of R7 into R5 (R5 = R7)
```

Notice the position of the source and destination operands. As you can see, the MOV loads the right operand into the left operand. In other words, the destination comes first.

To write a comment in Assembly language we use ‘;’. It is the same as ‘//’ in C language, which causes the remainder of the line of code to be ignored. For instance, in the above examples the expressions mentioned after ‘;’ just explain the functionality of the instructions to you, and do not have any effects on the execution of the instructions.

When programming the registers of the ARM microcontroller with an immediate value, the following points should be noted:

1. We put # in front of every immediate value.
2. If we want to present a number in hex, we put a 0x in front of it. If we put nothing in front of a number, it is in decimal. For example, in “MOV R1,#50”, R1 is loaded with 50 in decimal, whereas in “MOV R1,#0x50”, R1 is loaded with 50 in hex (80 in decimal).
3. If values 0 to FF are moved into a 32-bit register, the rest of the bits are assumed to be all zeros. For example, in “MOV R1,#0x5” the result will be R1=0x00000005; that is, R1=00000000000000000000000000000101 in binary.
4. Moving an immediate value larger than 255 (FF in hex) into the register will cause an error.

Note!

We cannot load values larger than 0xFF (255) into registers R0 to R12 using the MOV instruction. For example, the following instruction is not valid:

```
MOV R5,#0x999999                              ;invalid instruction
```

The reason is the fact that although the ARM instruction is 32-bit wide, only 8 bits of MOV instruction can be used as an immediate value which can take values not larger than 0xFF (255).

ADD instruction

The ADD instruction has the following format:

```
ADD     Rd,Rn,Op2  ;ADD Rn to Op2 and store the result in Rd
;Op2 can be Immediate value #K  (K is between 0 and 255)
;or Register Rm
```

The ADD instruction tells the CPU to add the value of Rn to Op2 and put the result into the Rd (destination) register. As we mentioned before, Op2 can be an immediate value #K between 0–255 in decimal (00–FF in hex) or a register Rm. To add two numbers such as 0x25 and 0x34, one can do any of the following:

```
MOV    R1,#0x25 ;copy 0x25 into R1 (R1 = 0x25)
MOV    R7,#0x34 ;copy 0x34 into R1 (R7 = 0x34)
ADD     R5,R1,R7 ;add value R7 to R1 and put it in R5
  ;(R5 = R1 + R7)
```

or

```
MOV    R1,#0x25          ;load (copy) 0x25 into R1 (R1 = 0x25)
ADD     R5,R1,#0x34      ;add 0x34 to R1 and put it in R5
                        ;(R5 = R1 + 0x34)
```

Executing the above lines results in R5 = 0x59 (0x25 + 0x34 = 0x59)

SUB instruction

The SUB instruction is like ADD instruction format. It subtracts Op2 from Rn and put the result in Rd (destination)

```
SUB     Rd,Rn,Op2       ;Rd=Rn – Op2
```

To subtract two numbers such as 0x34 and 0x25, one can do the following:

```
MOV    R1,#0x34         ;load (copy) 0x34 into R1 (R1=0x34)
SUB     R5,R1,#0x25     ;R5 = R1 – 0x25 (R1 = 0x34 – 0x25)
```

The old format

Notice that in most of instructions like ADD and SUB, Rn can be omitted if Rd and Rn are the same. This format is no longer recommended by Unified Assembler Language.

For example, each pair of the following instructions are the same.

```
SUB     R1,R1,#0x25     ;R1=R1-0x25
SUB     R1,#0x25         ;R1=R1-0x25


SUB     R1,R1,R2         ;R1=R1-R2
SUB     R1,R2             ;R1=R1-R2


ADD     R1,R1,#0x25     ;R1=R1+0x25
ADD     R1,#0x25         ;R1=R1+0x25


ADD     R1,R1,R2         ;R1=R1+R2
ADD     R1,R2             ;R1=R1+R2
```

Figure 2-3 shows the general purpose registers (GPRs) and the ALU in ARM. The effect of arithmetic and logic operations on the status register will be discussed in Section 2.4. In Table 2-1 you see some of the ARM ALU instructions.

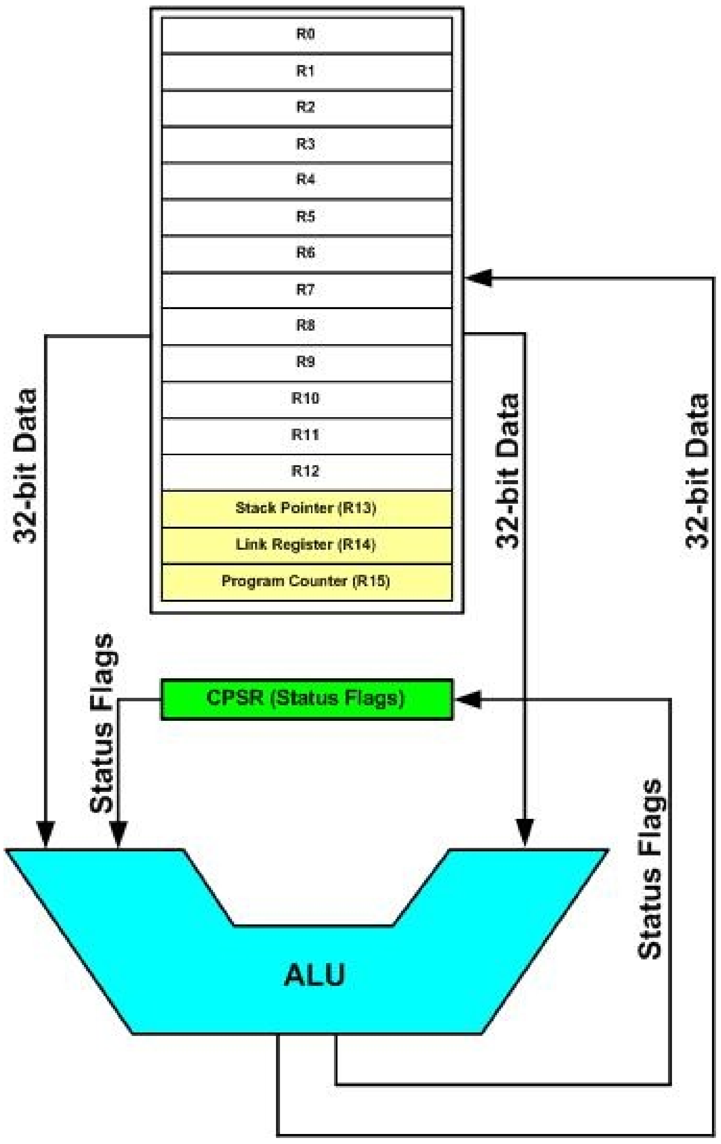


Figure 2- 3: ARM Registers and ALU

Instruction	Description
ADD Rd, Rn,Op2*	ADD Rn to Op2 and place the result in Rd
ADC Rd, Rn,Op2	ADD Rn to Op2 with Carry and place the result in Rd
AND Rd, Rn,Op2	AND Rn with Op2 and place the result in Rd
BIC Rd, Rn,Op2	AND Rn with NOT of Op2 and place the result in Rd
CMP Rn,Op2	Compare Rn with Op2 and set the status bits of CPSR**
CMN Rn,Op2	Compare Rn with negative of Op2 and set the status bits
EOR Rd, Rn,Op2	Exclusive OR Rn with Op2 and place the result in Rd
MVN Rd,Op2	Place NOT of Op2 in Rd

MOV Rd,Op2	MOVE (Copy) Op2 to Rd
ORR Rd, Rn,Op2	OR Rn with Op2 and place the result in Rd
RSB Rd, Rn,Op2	Subtract Rn from Op2 and place the result in Rd
RSC Rd, Rn,Op2	Subtract Rn from Op2 with carry and place the result in Rd
SBC Rd, Rn,Op2	Subtract Op2 from Rn with carry and place the result in Rd
SUB Rd, Rn,Op2	Subtract Op2 from Rn and place the result in Rd
TEQ Rn,Op2	Exclusive-OR Rn with Op2 and set the status bits of CPSR
TST Rn,Op2	AND Rn with Op2 and set the status bits of CPSR

***** *Op2 can be an immediate 8-bit value #K which can be 0–255 in decimal, (00–FF in hex). Op2 can also be a register Rm. Rd, Rn and Rm are any of the general purpose registers*

****** *CPSR is discussed later in this chapter*

******* *The instructions are discussed in detail in the next chapters*

Table 2- 1: ALU Instructions Using GPRs

Review Questions

1. Write instructions to move the value 0x34 into the R2 register.
2. Write instructions to add the values 0x16 and 0xCD. Place the result in the R1 register.
3. True or false. No value can be moved directly into the GPRs.
4. What is the largest hex value that can be moved into a 32-bit register using MOV instruction? What is the decimal equivalent of that hex value?
5. All of the registers in the ARM are _____-bit.

Section 2.2: The ARM Memory Map

In this section we discuss the memory map for ARM family members.

The Special Function Registers in ARM

In ARM the R13, R14, R15, and CPSR (current program status register) registers are called *SFRs (special function registers)* since each one is dedicated to a specific function. A given special function register is dedicated to specific function such as status register, program counter, stack pointer, and so on. The function of each SFR is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or keeping track of specific CPU status. The four SFRs of R13, R14, R15, and CPSR play an extremely important role in ARM. The R13 is set aside for stack pointer. The R14 is designated as link register which holds the return address when the CPU calls a subroutine and the R15 is the program counter (PC). The PC (program counter) points to the address of the next instruction to be executed as we will see in next section. The CPSR (current program status register) is used for keeping condition flags among other things, as we will see in Section 2.4. In contrast to SFRs, the GPRs (R0-R12) do not have any specific function and are used for storing general data. For this reason some ARM instruction formats (the Thumb) have only R0-7 but every variation of ARM chip has R13-R15 SFRs. The Thumb instruction format is designed to compete with the 8- and 16-bit microcontrollers and increase code density.

Program Counter in the ARM

One of the most important register in the ARM microcontroller is the PC (program counter) . As we mentioned earlier, the R15 is the program counter. The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program memory, the program counter is incremented automatically to point to the next instruction. The wider the program counter, the more memory locations a CPU can access. That means that a 32-bit program counter can access a maximum of 4G (2^{32} = 4G) bytes of program memory locations.

In ARM microcontrollers each memory location is a byte wide. In the case of a 32-bit program counter, the code space is 4G bytes (2^{32} = 4G), which occupies the 0x00000000–0xFFFFFFFF address range. The program counter in the ARM family is 32 bits wide. This means that the ARM family can access addresses 0x00000000 to 0xFFFFFFFF, a total of 4G bytes of locations (memory spaces). Although this 4G bytes of memory space can be allocated to on-chip or off-chip memory; however, at the time of this writing, none of the members of the ARM microcontroller family have the entire 4G bytes of on-chip memory populated. See Table 2-2.

Company	Device	Flash (K Bytes)	RAM (K Bytes)	I/O Pins
Atmel	AT91SAM7X512	512	128	62
NXP	LPC2367	512	58	70

ST	STR750FV2	256	16	72
TI	TMS470R1A256	256	12	49
Freescale	MK10DX256VML7	256	64	74

Table 2- 2: On-chip Memory Size for some ARM Chips

Memory space allocation in the ARM

The ARM has 4G bytes of directly accessible memory space. This memory space has addresses 0 to 0xFFFFFFFF. The 4G bytes of memory space can be divided into five sections. They are as follows:

1. **On-chip peripheral and I/O registers:** This area is dedicated to general purpose I/O (GPIO) and special function registers (SFRs) of peripherals such as timers, serial communication, ADC, and so on. In other words, ARM uses memory-mapped I/O. See Chapter 0 for discussion of memory-mapped I/O. The function and address location of each SFR is fixed by the chip vendor at the time of design because it is used for port registers of peripherals. The number of locations set aside for GPIO registers and SFRs depends on the pin numbers and peripheral functions supported by that chip. That number can vary from chip to chip even among members of the same family from the same vendor. Due to the fact that ARM does not define the type and number of I/O peripherals one must not expect to have same address locations for the peripheral registers among various vendors.

2. **On-chip data SRAM:** A RAM space ranging from a few kilobytes to several hundred kilobytes is set aside mainly for data storage. The data RAM space is used for data variables and stack and is accessed by the microcontroller instructions. The data RAM space is read/write memory used by the CPU for storage of data variables, scratch pad, and stack. The ARM microcontrollers' data SRAM size ranges from 2K bytes to several thousand kilobytes depending on the chip. Even within the same family, the size of the data SRAM space varies from chip to chip. A larger data SRAM size means more difficulties in managing these RAM locations if you use Assembly language programming. In today's high-performance microcontroller, however, with over a thousand bytes of data RAM, the job of managing them is handled by the C compilers. Indeed, the C compilers are the very reason we need a large data RAM since it makes it easier for C compilers to store parameters and allows them to perform their jobs much faster. The amount and the location of the SRAM space vary from chip to chip in the ARM chips. A section of the data RAM space is used by stack as we will see in Chapter 6. Although many of the ARM microcontrollers used in embedded products the SRAM space is used for data; one can also buy or design an ARM-based system in which the RAM space is used for both data and program codes. This is what we see in the x86 PCs. In such systems normally one connects the ARM CPU to external DRAM and the DRAM memory is used for both code and data. Microsoft Windows 8 uses such a system for ARM-based Tablet computers.

3. **On-chip EEPROM:** A block of memory from 1K bytes to several thousand bytes is set aside for EEPROM memory. The amount and the location of the EEPROM space vary from chip to chip in the ARM microcontrollers. Although in some applications the EEPROM is used for program code storage, it is used most often for saving critical data. Not all ARM chips have on-chip EEPROM.

4. **On-chip Flash ROM:** A block of memory from a few kilobytes to several hundred kilobytes is set aside for program space. The program space is used for the program code. In today's ARM microcontroller chips, the code ROM space is of Flash type memory. The amount and the location of the code ROM space vary from chip to chip in the ARM products. See Table 2-2 and Examples 2-1 and 2-2. The Flash memory of code ROM is under the control of the PC (program counter). The code ROM memory can also be used for storage of static fixed data such as ASCII data strings and look-up tables.

Example 2-1

A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.

(a) Address range of 0x00100000 – 0x00100FFF for EEPROM

(b) Address range of 0x40000000 – 0x40007FFF for SRAM

(c) Address range of 0x00000000 – 0x0007FFFF for Flash

(d) Address range of 0xFFFC0000 – 0xFFFFFFFF for peripherals

Solution:

(a) With address space of 0x00100000 to 00100FFF, we have 00100FFF – 00100000 = 0FFF bytes. Converting 0FFF to decimal, we get 4,095 + 1, which is equal to 4K bytes.

(b) With address space of 0x40000000 to 0x40007FFF, we have 40007FFF – 40000000 = 7FFF bytes. Converting 7FFF to decimal, we get 32,767 + 1, which is equal to 32K bytes.

(c) With address space of 0000 to 7FFFF, we have 7FFFF – 0 = 7FFFF bytes. Converting 7FFFF to decimal, we get 524,287 + 1, which is equal to 512K bytes.

(d) With address space of FFFC0000 to FFFFFFFF, we have FFFFFFFF–FFFC0000 = 3FFFF bytes. Converting 3FFFF to decimal, we get 262,143 + 1, which is equal to 256K bytes.

See Figure 2-4.

Example 2-2

Find the address space range of each of the following memory of an ARM chip:

(a) 2 KB of EEPROM starting at address 0x80000000

(b) 16 KB of SRAM starting at address 0x90000000

(c) 64 KB of Flash ROM starting at address 0xF0000000

Solution:

(a) With 2K bytes of on-chip EEPROM memory, we have 2048 bytes (2 × 1024 = 2048). This maps to address locations of 0x80000000 to 0x800007FF. Notice that 0 is always the first location.

(b) With 16K bytes of on-chip SRAM memory, we have 16,384 bytes (16 × 1024 = 16,384), and 16,384 locations gives 0x90000000–0x90003FFF.

(c) With 64K we have 65,536 bytes (64 × 1024 = 65,536), therefore, the memory space is 0xF0000000 to 0xF000FFFF.

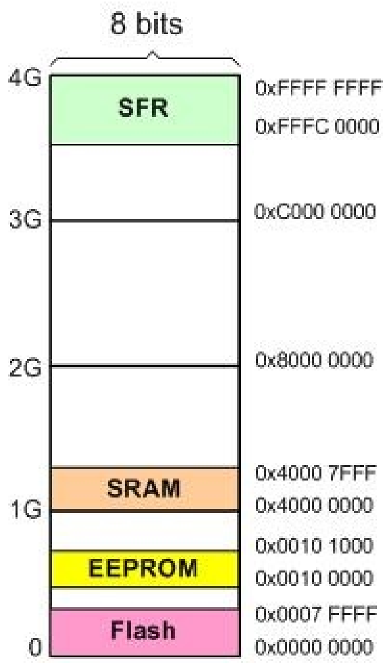


Figure 2- 4: An Example of ARM Memory Allocation

5. **Off-chip DRAM space:** A DRAM memory ranging from few megabytes to several hundred mega bytes can be implemented for external memory connection. Although many of the ARM microcontrollers used in embedded products use the on-chip SRAM for data, one can also design an ARM-based system in which the RAM is used for both data and program codes. This is what we see in the x86 PCs. In such systems one connects the ARM CPU to external DRAM and the DRAM memory is used for both code and data, just like the x86 PCs. Many ARM vendors

are pushing the ARM11 chip for the high-end of the market such as servers and database computers. In an ARM11-based server computers, the external (off-chip) DRAM is used and managed by the operating system while on-chip Flash, EEPROM, and SRAM memories are used for BIOS (basic input output system), POST (power on self test), and CPU scratch pad, respectively. In such cases the system is not different from x86 computers currently in use, except it uses ARM CPU instead of a Pentium chip from Intel or x86 from AMD. The Microsoft Windows 8 uses ARM motherboard with off-chip DRAM

Notice the following differences among the on-chip Flash ROM, data SRAM, and EEPROM memories in ARM microcontrollers used for embedded products:

a) The data SRAM is used by the CPU for data variables and stack, whereas the EEPROMs are considered to be memory that one can also add externally to the chip. In other words, while many ARM microcontroller chips have no EEPROM memory, it is very unlikely for an ARM microcontroller to have no on-chip data SRAM.

b) The on-chip Flash ROM is used for program code, while the EEPROM is used most often for critical system data that must not be lost if power is cut off. Remember that data SRAM is volatile memory and its contents are lost if the power to the chip is cut off. Since volatile data SRAM is used for dynamic variables (constantly changing data) and stack. We need EEPROM memory to secure critical system data that does not change very often and will not be lost in the event of power failure.

c) The on-chip Flash ROM is programmed and erased in block size. The block size is 8, 16, 32, or 64 bytes or more depending on the chip technology. That is not the case with EEPROM, since the EEPROM is byte programmable and erasable. Both the EEPROM and Flash memories have limited number of erase/write cycles, which can be 100,000 or more. While all semiconductor memories have unlimited number of reads (accesses), the number of times that we can erase and write to the Flash and EEPROM are limited to around 100,000 times at the time of this writing. As this number increases, the likelihood that we will have hard drive made out of Flash memory will also increase. We have already seen how the Flash stick memory has replaced the floppy drives which were so common into early 2000s.

Memory mapped I/O in the ARM

Traditional CPUs such as x86 had two distinct spaces: the I/O space and memory space. In x86, while all of the I/O ports are accessed using IN and OUT instructions, the memory address space is accessed using the MOV instruction. In the ARM CPU we have only one space and it is memory space and it can be as high as 4 Giga bytes. The ARM uses this 4 Giga bytes for both memory and I/O space. This mapping of the I/O ports to memory space is called memory mapped I/O and was discussed in Chapter 0.

Review Questions

1. True or false. The GPR registers are used for storing data.
2. The R0-R12 registers are called______.

3. The GPR registers in ARM are _____-bit.
4. The R13-R15 registers are called _____.
5. The SFR registers in ARM are ______ -bit.

Section 2.3: Load and Store Instructions in ARM

The instructions we have used so far worked with the immediate (constant) value of K and the GPRs. They also used the GPRs as their destination. We saw simple examples of using MOV and ADD earlier in Section 2.1. The ARM allows direct access to all locations in the memory. In this section we show the instructions accessing various locations of the memory. This is one of the most important sections in the book for mastering the topic of ARM Assembly language programming. Before we embark on studying the load and store instructions of the ARM, we must note the fact that all the instructions of the ARM are 32-bit wide. In other words, every instruction of ARM is fixed at 32-bit. As we will see in later section, the fixed size instruction is one of the most important characteristics of RISC architecture. In cases where there is no need for all the 32-bit, the ARM has added zero to make the instruction fixed at 32-bit.

LDR Rd, [Rx] instruction

```
LDR     Rd,[Rx] ;load Rd with the contents of location pointed
                        ;to by Rx register. Rx is an address between
                ;0x00000000 to 0xFFFFFFFF
```

The LDR instruction tells the CPU to load (bring in) one word (32-bit or 4 bytes) from a base address pointed to by Rx into the GPR. After this instruction is executed, the Rd will have the same value as four consecutive locations in the memory. Obviously since each memory location can hold only one byte (ARM is a byte addressable CPU), and our GPR is 32-bit, the LDR will bring in 4 bytes of data from 4 consecutive memory locations. The locations can be in the SRAM or a Flash memory. For example, the "LDR R2,[R5]" instruction will copy the contents of memory locations pointed to by R5 into register R2. Since the R2 register is 32-bit wide, it expects a 32-bit operand in the range of 0x00000000 to 0xFFFFFFFF. That means the R5 register gives the base address of the memory in which it holds the data. Therefore if R5=0x80000, the CPU will fetch into register R2, the contents of memory locations 0x80000, 0x80001,0x80002, and 0x80003.

The following instruction loads R7 with the contents of location 0x40000200. See Figure 2-5.

```
                ;assume R5 = 0x40000200
LDR     R7,[R5] ;load R7 with the contents of locations
                ;0x40000200-0x40000203
```

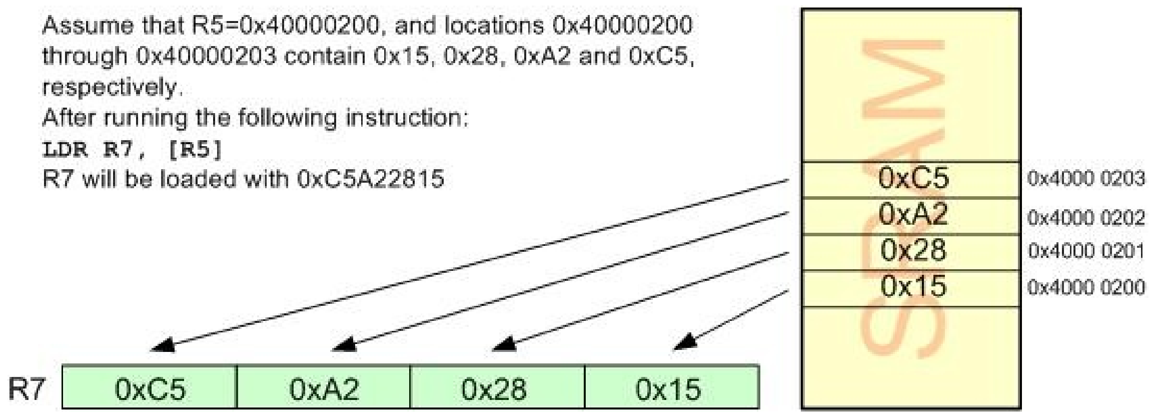

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x15, 0x28, 0xA2 and 0xC5, respectively.
After running the following instruction:
LDR R7, [R5]
R7 will be loaded with 0xC5A22815
SRAM
0xC5
0xA2
0x28
0x15
0x4000 0203
0x4000 0202
0x4000 0201
0x4000 0200
R7
0xC5
0xA2
0x28
0x15

Figure 2-5: Executing the LDR Instruction

STR Rx,[Rd] instruction

STR Rx,[Rd] ;store register Rx into locations pointed to by Rd

The STR instruction tells the CPU to store (copy) the contents of the GPR to a base address location pointed to by the Rd register. Notice that the source register of STR instruction is placed before the destination register. Obviously since GPR is 32-bit wide (4-byte) we need four consecutive memory locations to store the contents of GPR. The memory locations must be writable such as SRAM. See Figure 2-6. The "STR R3,[R6]" instruction will copy the contents of R3 into locations pointed to by R6. Locations 0x40000200 through 0x40000203 of the SRAM memory will have the contents of R3 since R6 = 0x40000200.

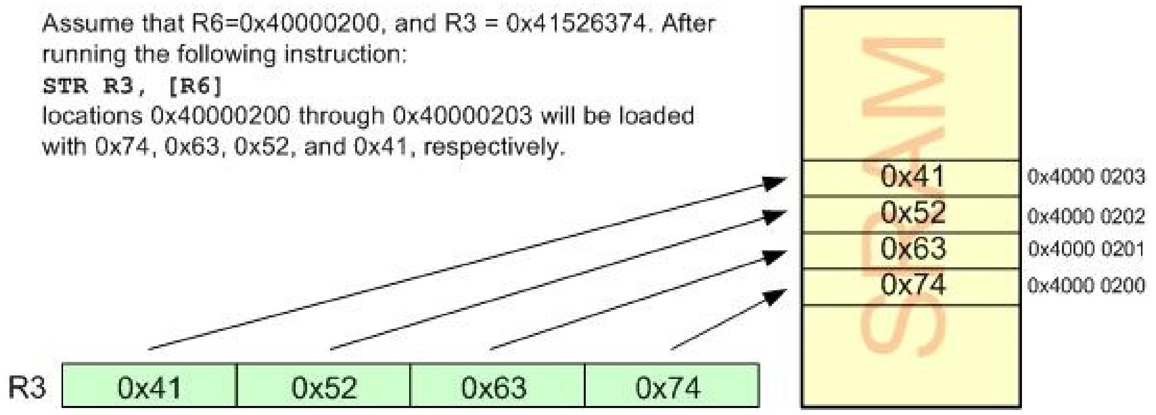

Assume that R6=0x40000200, and R3 = 0x41526374. After running the following instruction:
STR R3, [R6]
locations 0x40000200 through 0x40000203 will be loaded with 0x74, 0x63, 0x52, and 0x41, respectively.
SRAM
0x41
0x52
0x63
0x74
0x4000 0203
0x4000 0202
0x4000 0201
0x4000 0200
R3
0x41
0x52
0x63
0x74

Figure 2-6: Executing the STR Instruction

The following instruction stores the contents of R5 into locations pointed to by R1. Assume 0x40000340 is an address of internal RAM locations and held by register R1.

```
                    ;assume R1 = 0x40000340
STR     R5, [R1]    ;store R5 into locations pointed to by R1.
```

LDRB Rd, [Rx] instruction

```
LDRB    Rd, [Rx]    ;load Rd with the contents of the location
                    ; pointed to by Rx register.
```

The LDRB instruction tells the CPU to load (copy) one byte from an address pointed to by Rx into the lower byte of Rd. After this instruction is executed, the lower byte of Rd will have the same value as memory location pointed to by Rx. It must be noted that the unused portion (the upper 24 bits) of the Rd register will be all zeros, as shown in Figure 2-7.

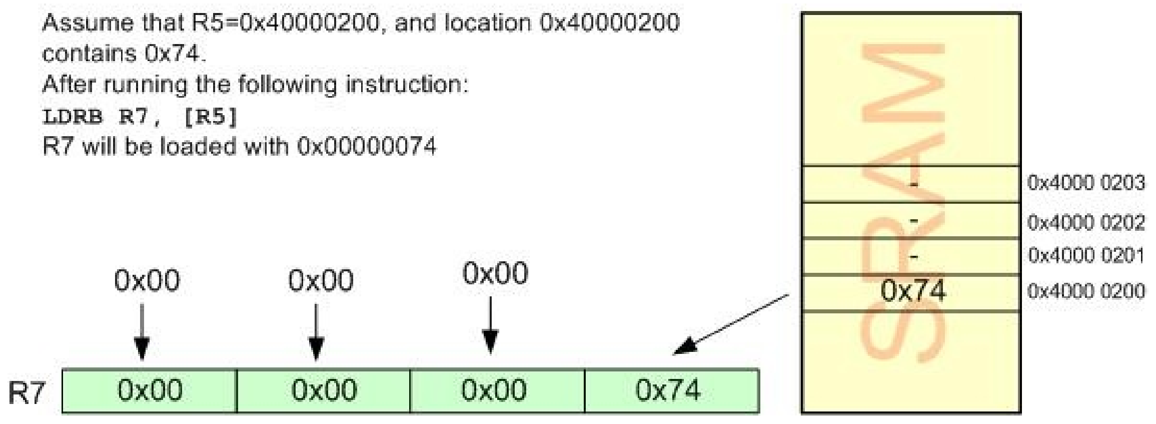

Assume that R5=0x40000200, and location 0x40000200 contains 0x74.
After running the following instruction:
LDRB R7, [R5]
R7 will be loaded with 0x00000074
SRAM
0x4000 0203
0x4000 0202
0x4000 0201
0x4000 0200
0x74
0x00
0x00
0x00
R7
0x00
0x00
0x00
0x74

Figure 2- 7

LDR vs. LDRB

As we mentioned earlier, we can use the LDR instruction to copy the contents of four consecutive memory locations into a 32-bit GPR. There are situations that we do not need to bring 4 bytes of data into GPR. An UART register is such a case. The UART registers are generally 8-bit and take only one memory space location (memory mapped I/O). Using LDRB, we can bring into GPR a single byte of data from UART registers. This is a widely used instruction for accessing the 8-bit I/O and peripheral ports.

STRB Rx,[Rd] instruction

```
STRB    Rx, [Rd]        ;store the byte in register Rx into
                         ;location pointed to by Rd
```

The STRB instruction tells the CPU to store (copy) the byte value in Rx to an address location pointed to by the Rd register. After this instruction is executed, the memory locations pointed to by the Rd will have the same byte as the lower byte of the Rx, as shown in Figure 2-8.

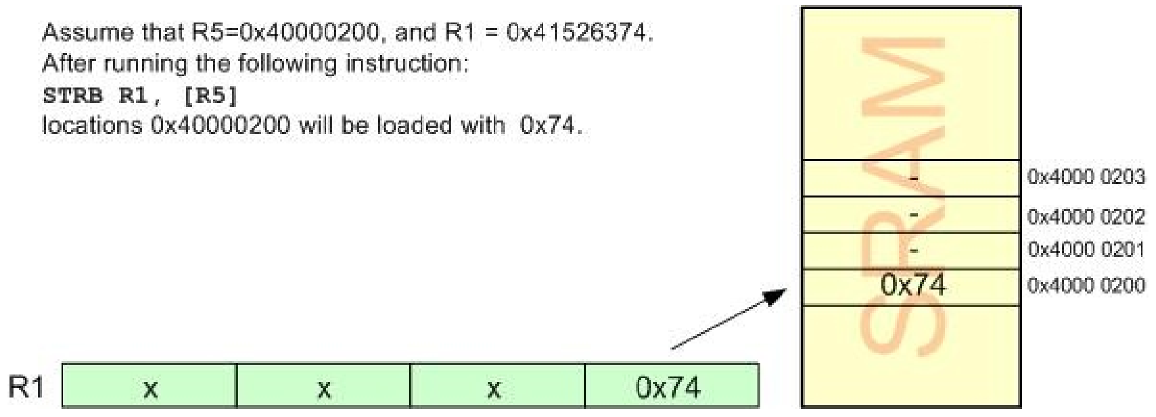

Assume that R5=0x40000200, and R1 = 0x41526374.
After running the following instruction:
STRB R1, [R5]
locations 0x40000200 will be loaded with 0x74.
SRAM
0x4000 0203
0x4000 0202
0x4000 0201
0x4000 0200
0x74
R1
x
x
x
0x74

Figure 2- 8

The following program first loads the R1 register with value 0x55, then stores this value into location 0x40000100:

```
                ;assume R5 = 0x40000100
```

```
MOV     R1,#0x55            ;R1 = 55 (in hex)
STRB     R1,[R5]             ;copy R1 location pointed to by R5
```

General purpose I/O (GPIO) are part of the special function registers in the memory-mapped I/O. They are connected to the I/O pins of the ARM microcontroller. See the datasheet of your ARM microcontroller. We can also store the contents of a GPR into any location in the SRAM region of the data space. See Examples 2-3 and 2-4.

Example 2-3

State the contents of RAM locations 0x92 to 0x96 after the following program is executed:

```
MOV     R1,#0x99            ;R1 = 0x99
MOV     R6,#0x92            ;R6 = 0x92
STRB     R1,[R6]             ;store R1 into location pointed to by R6
;(location 0x92)
ADD      R6,R6,#1            ;R6 = R6 + 1
MOV     R1,#0x85            ;R1 = 0x85
STRB     R1,[R6]             ;store R1 into location pointed to by R6
;(location 0x93)
ADD      R6,R6,#1            ;R6 = R6 + 1
MOV     R1,#0x3F            ;R1 = 0x3F
STRB     R1,[R6]             ;store R1 into location pointed to by R6
ADD      R6,R6,#1            ;R6 = R6 + 1
MOV     R1,#0x63            ;R1 = 0x63
STRB     R1,[R6]             ;store R1 into location pointed to by R6
ADD      R6,R6,#1            ;R6 = R6 + 1
MOV     R1,#0x12            ;R1 = 0x12
STRB     R1,[R6]
```

Solution:

After the execution of STRB R1,[R6] data memory location 0x92 has value 0x99;

after the execution of STRB R1,[R6] data memory location 0x93 has value 0x85;

after the execution of STRB R1,[R6] data memory location 0x94 has value 0x3F; and so on, as shown in the chart.

Address	Data

0x92	0x99
0x93	0x85
0x94	0x3F
0x95	0x63
0x96	0x12

Example 2-4

State the contents of R2, R1, and memory location 0x20 after the following program:

```
    MOV     R2,#0x5             ;load R2 with 5  (R2 = 0x05)
    MOV     R1,#0x2             ;load R1 with 2 (R1 = 0x02)
    ADD      R2, R1,R2           ;R2 = R1 + R2
    ADD      R2,R1,R2            ;R2 = R1 + R2
    MOV     R5,#0x20            ;R5 = 0x20
    STRB     R2,[R5]             ;store R2 into location pointed to by R5
```

Solution:

The program loads R2 with value 5. Then it loads R1 with value 2. Then it adds the R1 register to R2 twice. At the end, it stores the result in location 0x20 of memory.

After MOV R2,#0x05

Location	Data
R2	5
R1	
0x20	

After MOV R1,#0x02

Location	Data
R2	5
R1	2

0x20	

After ADD R2,R1,R1

Location	Data
R2	7
R1	2
0x20	

After ADD R2,R1,R1

Location	Data
R2	9
R1	2
0x20	

After STRB [R5],R2

Location	Data
R2	9
R1	2
0x20	9

STR vs. STRB

As we mentioned earlier, we can use the STR instruction to copy the contents of 32-bit GPR into four consecutive memory locations. The I/O ports are generally 8-bit and take only one memory space location (memory mapped I/O). Using STRB, we can send a byte of data from GPR to memory location such as an I/O port. Again, this is a widely used instruction for accessing the 8-bit I/O and peripheral ports.

LDRH Rd, [Rx] instruction

```
LDRH    Rd, [Rx]          ;load Rd with the half-word pointed
; to by Rx register
```

The LDRH instruction tells the CPU to load (copy) half-word (16-bit or 2 bytes) from a base address pointed to by Rx into the lower 16-bits of Rd Register. After this instruction is executed, the lower 16-bit of Rd will have the same value as two consecutive locations in the memory pointed to by base address of Rx. It must be noted that the unused portion (the upper 16 bits) of the Rd register will be all zeros, as shown in Figure 2-9.

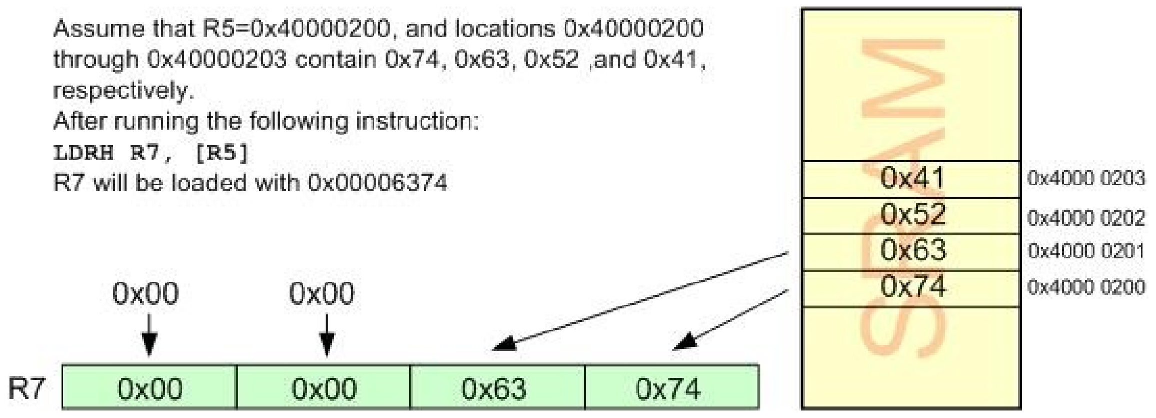

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.
After running the following instruction:
LDRH R7, [R5]
R7 will be loaded with 0x00006374
0x00
0x00
R7
0x00
0x00
0x63
0x74
SRAM
0x41
0x52
0x63
0x74
0x4000 0203
0x4000 0202
0x4000 0201
0x4000 0200

Figure 2- 9

Table 2-3 compares LDRB, LDRH, and LDR.

Data Size	Bits	Decimal	Hexadecimal	Load instruction used
Byte	8	0 – 255	0 - 0xFF	LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	LDRH
Word	32	0 – 2^{32}-1	0 - 0xFFFFFFFF	LDR

Table 2- 3: Unsigned Data Range in ARM and associated Load Instructions

STRH Rx,[Rd] instruction

```
STRH    Rx, [Rd]          ;store half-word (2-byte) in register Rx
                           ;into locations pointed to by Rd
```

The STRH instruction tells the CPU to store (copy) the lower 16-bit contents of the Rx to an address location pointed to by the Rd register. After this instruction is executed, the memory locations pointed to by the Rd will have the same value as the lower 16-bit of Rx Register. The locations are part of the data read/write memory space such as on-chip SRAM. For example, the "STRH R3,[R6]" instruction will copy the 16-bit lower contents of R3 into two consecutive locations pointed to by base register R6. As you can see in Figure 2-10, locations 0x2000 and 0x2001 of the SRAM memory will have the contents of the lower half word of R3 since R6 = 0x2000.

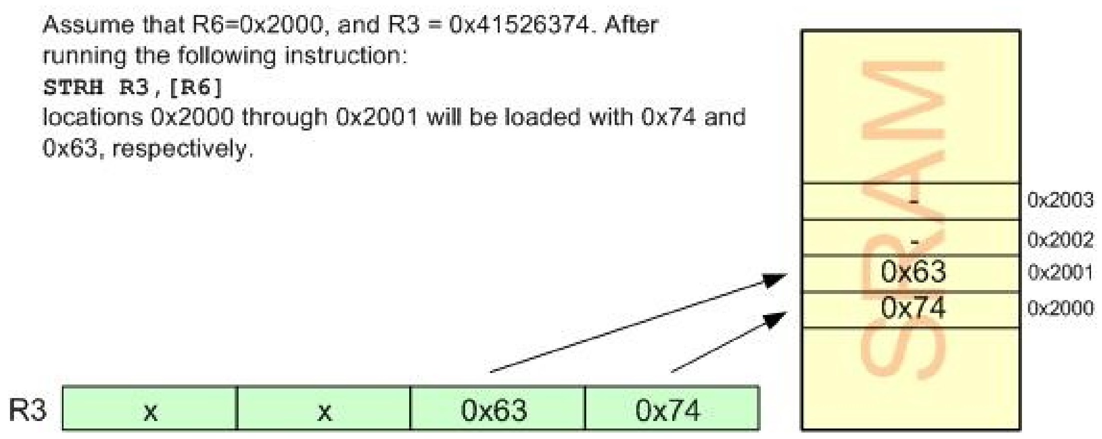

Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:
STRH R3, [R6]
locations 0x2000 through 0x2001 will be loaded with 0x74 and 0x63, respectively.
SRAM
-
-
0x63
0x74
0x2003
0x2002
0x2001
0x2000
R3
x
x
0x63
0x74

Figure 2- 10

In Table 2-4 you see a comparison between STRB, STRH, and STR.

Data Size	Bits	Decimal	Hexadecimal	Load instruction used
Byte	8	0 – 255	0 - 0xFF	STRB
Half-word	16	0 – 65535	0 - 0xFFFF	STRH
Word	32	0 – 2^{32}-1	0 - 0xFFFFFFFF	STR

Table 2-4: Unsigned Data Range in ARM and associated Store Instructions

Review Questions

1. True or false. No 32-bit value can be loaded directly into internal R0-R12.
2. Write instructions to load value 0x95 into location with address 0x20.
3. Write instructions to move the contents of R2 to memory location pointed to by R8.
4. Write instructions to load values from memory locations 0x20–0x23 to R4 register.
5. What is the largest hex value that can be moved into a single location in the data memory? What is the decimal equivalent of the hex value?
6. “LDR R6, [R3]” puts the result in _____ .
7. What does “STRB R1, [R2]” do?
8. What is the largest hex value that can be moved into four consecutive locations in the data memory? What is the decimal equivalent of the hex value?

Section 2.4: ARM CPSR (Current Program Status Register)

Like all other microprocessors, the ARM has a flag register to indicate arithmetic conditions such as the carry bit. The flag register in the ARM is called the *current program status register (CPSR)*. In this section, we discuss various bits of this register and provide some examples of how it is altered. Chapters 3 and 4 show how the flag bits of the status register are used.

ARM current program status register

The status register is a 32-bit register. See Figure 2-11 for the bits of the status register. The bits C, Z, N, and V are called conditional flags, meaning that they indicate some conditions that result after an instruction is executed. Each of the conditional flags can be used to perform a conditional branch (jump), as we will see in Chapter 4.

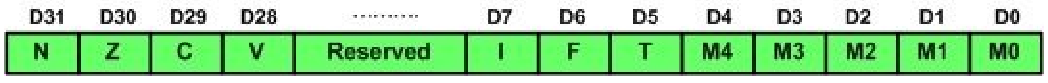

D31 D30 D29 D28 .... D7 D6 D5 D4 D3 D2 D1 D0
N Z C V Reserved I F T M4 M3 M2 M1 M0

Figure 2- 11: CPSR (Current Program Status Register)

The following is a brief explanation of the flag bits of the current program status register (CPSR). The impact of instructions on this register is then discussed.

C, the carry flag

This flag is set whenever there is a carry out from the D31 bit. This flag bit is affected after a 32-bit addition or subtraction. Chapter 4 shows how the carry flag is used.

Z, the zero flag

The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then Z = 1. Therefore, Z = 0 if the result is not zero. See Chapter 4 to see how we use the Z flag for looping.

N, the negative flag

Binary representation of signed numbers uses D31 as the sign bit. The negative flag reflects the result of an arithmetic operation. If the D31 bit of the result is zero, then N = 0 and the result is positive. If the D31 bit is one, then N = 1 and the result is negative. The negative and V flag bits are used for the signed number arithmetic operations and are discussed in Chapter 5.

V, the overflow flag

This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow flag is used to detect errors in signed arithmetic operations. The V and N flag bits are used for signed number arithmetic operations and are discussed in Chapter 5.

The T flag bit is used to indicate the ARM is in Thumb state. The I and F flags are used to enable or disable the interrupt. See the ARM manual.

S suffix and the status register

Most of ARM instructions can affect the status bits of CPSR according to the result. If we need an instruction to update the value of status bits in CPSR, we have to put S suffix at the end of instructions. That means, for example, ADDS instead of ADD is used.

ADD instruction and the status register

Next we examine the impact of the SUBS and ADDS instructions on the flag bits C and Z of the status register. Some examples should clarify their meanings. Although all the flag bits C, Z, V, and N are affected by the ADDS and SUBS instruction, we will focus on flags C and Z for now. The other flag bits are discussed in Chapter 5, because they relate only to signed number operations. Examine Example 2-5 to see the impact of the ADDS instruction on selected flag bits. See also Example 2-6 to see the impact of the SUBS instruction on selected flag bits.

Example 2-5

Show the status of the C and Z flags after the addition of

a) 0x0000009C and 0xFFFFFF64 in the following instruction:

```
;assume R1 = 0x0000009C and R2 = 0xFFFFFF64
ADDS    R2,R1,R2            ;add R1 to R2 and place the result in R2
```

b) 0x0000009C and 0xFFFFFF69 in the following instruction:

```
;assume R1 = 0x0000009C and R2 = 0xFFFFFF69
ADDS    R2,R1,R2            ;add R1 to R2 and place the result in R2
```

Solution:

a)

0x0000009C	0000 0000 0000 0000 0000 0000 1001 1100
+ 0xFFFFFF64	+ 1111 1111 1111 1111 1111 1111 0110 0100
0x100000000	1 0000 0000 0000 0000 0000 0000 0000 0000

C = 1 because there is a carry beyond the D31 bit.

Z = 1 because the R2 (the result) has value 0 in it after the addition.

b)

```
  0x0000009C      0000 0000 0000 0000 0000 0000 1001 1100
+ 0xFFFFFF69    + 1111 1111 1111 1111 1111 1111 0110 1001
 0x100000005    1 0000 0000 0000 0000 0000 0000 0000 0101
```

C = 1 because there is a carry beyond the D31 bit.

Z = 0 because the R2 (the result) does not have value 0 in it after the addition. (R2=0x00000005)

Example 2-6

Show the status of the Z flag during the execution of the following program:

```
MOV     R2,#4            ;R2 = 4
MOV     R3,#2            ;R3 = 2
MOV     R4,#4            ;R4 = 4
SUBS    R5,R2,R3         ;R5 = R2 - R3 (R5 = 4 - 2 = 2)
SUBS    R5,R2,R4         ;R5 = R2 - R4 (R5 = 4 - 4 = 0)
```

Solution:

The Z flag is raised when the result is zero. Otherwise, it is cleared (zero). Thus:

After	Value of R5	Z flag
SUBS R5,R2,R3	2	0
SUBS R5,R2,R4	0	1

Not all instructions affect the flags

Some instructions affect all the four flag bits C, Z, V, and N (e.g., ADDS). But some instructions affect no flag bits at all. The branch instructions are in this category. Some instructions affect only some of the flag bits. The logic instructions (e.g., ANDS) are in this category.

Table 2-5 shows the instructions and the flag bits affected by them. Appendix A provides a complete list of all the instructions and their associated flag bits.

Instruction	Flags Affected
ANDS	C, Z, N
ORRS	C, Z, N
MOVS	C, Z, N
ADDS	C, Z, N, V
SUBS	C, Z, N, V
B	No flags
Note that we cannot put S after B instruction.	

Table 2- 5: Flag Bits Affected by Different Instructions

Flag bits and decision making

There are instructions that will make a conditional jump (branch) based on the status of the flag bits. Table 2-6 provides some of these instructions. Chapter 4 discusses the conditional branch instructions and how they are used.

Instruction	Flags Affected
BCS	Branch if C = 1
BCC	Branch if C = 0
BEQ	Branch if Z = 1
BNE	Branch if Z = 0
BMI	Branch if N = 1
BPL	Branch if N = 0
BVS	Branch if V = 1
BVC	Branch if V = 0

Table 2- 6: ARM Branch (Jump) Instructions Using Flag Bits

Review Questions

1. The flag register in the ARM is called the ________.
2. What is the size of the flag register in the ARM?
3. Find the C and Z flag bits for the following code:

```
;assume R2 = 0xFFFFFF9F
;assume R1 = 0x00000061
ADDS    R2, R1, R2
```

4. Find the Z flag bit for the following code:

```
;assume R7 = 0x22
;assume R3 = 0x22
ADDS    R7, R3, R7
```

5. Find the C and Z flag bits for the following code:

```
;assume R2 = 0x67
;assume R1 = 0x99
ADDS    R2, R1, R2
```

Section 2.5: ARM Data Format and Directives

In this section we look at some widely used data formats and directives supported by the ARM assembler.

ARM data type

ARM has four data types. They are bit, byte (8-bit), half-word (16-bit) and word (32 bit). Due to the fact that ARM registers are 32-bit it is the job of programmer/compiler to break down data larger than 32 bits to be processed by the CPU. The data types used by the ARM can be positive or negative. A discussion of signed numbers is given in Chapter 5.

Data format representation

There are several ways to represent a byte of data in the ARM assembler. The numbers can be in hex, binary, decimal, or ASCII formats. The following are examples of how each works.

Hex numbers

To represent Hex numbers in an ARM assembler we put 0x (or 0X) in front of the number like this:

```
MOV     R1,#0x99
```

Here are a few lines of code that use the hex format:

```
MOV     R2,#0x75         ;R2 = 0x75
MOV     R1,#0x11
ADD      R2,R1,R2         ;R2 = R1 + R2 = 0x75 + 0x11 = 0x86
```

Decimal numbers

To indicate decimal numbers in some ARM assemblers such as Keil we simply use the decimal (e.g., 12) and nothing before or after it. Here are some examples of how to use it:

```
MOV   R7,#12     ;R7 = 00001100 or 0C in hex
MOV   R1,#32     ;R1 = 32 = 0x20
```

Binary numbers

To represent binary numbers in an ARM assembler we put 2_ in front of the number. It is as follows:

```
MOV    R6,#2_10011001 ;R6 = 10011001 in binary or 99 in hex
```

Numbers in any base between 2 and 9

To indicate a number in any base n between 2 and 9 in an ARM assembler we simply use the n_ in front of it. Here are some examples of how to use it:

```
MOV   R7,#8_33          ;R7 = 33 in base 8 or 011011 in binary format
MOV   R6,#2_10011001 ;R6 = 10011001 in base 2 or 99 in hex
```

ASCII characters

To represent ASCII data in an ARM assembler we use single quotes as follows:

```
LDR   R3,#'2'     ;R3 = 00110010 or 32 in hex (See Appendix F)
```

This is the same as other assemblers such as the 8051 and x86. Here is another example:

```
LDR   R2,#'9'     ;R2 = 0x39, which is hex number for ASCII '9'
```

To represent a string, double quotes are used; and for defining ASCII strings (more than one character), we use the DCB directive.

Assembler directives

While instructions tell the CPU what to do, directives (also called pseudo-instructions) give directions to the assembler. For example, the MOV and ADD instructions are commands to the CPU, but EQU, END, and ENTRY are directives to the assembler. The following section presents some widely used directives of the ARM and how they are used. The directives help us develop our program easier and make our program legible (more readable). Table 2-7 shows some assembler directives.

Directive	Description
AREA	Instructs the assembler to assemble a new code or data section
END	Informs the assembler that it has reached the end of a source file.
ENTRY	Declares an entry point to a program.
EQU	Gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.
INCLUDE	It adds the contents of a file to our program.

Table 2- 7: Some Widely Used ARM Directive

AREA

The AREA directive tells the assembler to define a new section of memory. The memory can be code (instruction) or data and can have attributes such as ReadOnly, ReadWrite, and so on. This is widely used to define one or more blocks of indivisible memory for code or data to be used by the linker. Every Assembly language program has at least one AREA. The following is the format:

```
AREA    sectionname, attribute, attribute, …
```

The following line defines a new area named MY_ASM_PROG1 which has CODE and READONLY attributes:

```
AREA    MY_ASM_PROG1, CODE, READONLY
```

Among widely used attributes are CODE, DATA, READONLY, READWRITE, COMMON, and ALIGN. The following describes these widely used attributes.

READWRITE is an attribute given to an area of memory which can be read from and written to. Since it is READWRITE section of the program it is by default for DATA. In ARM Assembly language we use this area to set aside SRAM memory for scratch pad and stack. The Assembler puts the READWRITE sections next to each other in the SRAM memory.

READONLY is an attribute given to an area of memory which can only be read from. Since it is READONLY section of the program it is by default for CODE. In ARM Assembly language we use this area to write our instructions for machine code execution. The READONLY sections are put next to each other in the flash memory.

Note

In Keil, The memory space of **READONLY** and **READWRITE** are defined in the ***Linker*** and ***Target*** tabs of the **Project\Options**. Keil sets the values according to the memory map of the chosen chip.

CODE is an attribute given to an area of memory used for executable machine instruction. Since it is used for code section of the program it is by default READONLY memory. In ARM Assembly language we use this area to write our instructions. The following line defines a new area for writing programs:

```
AREA    OUR_ASM_PROG, CODE, READONLY
```

DATA is an attribute given to an area of memory used for data and no instruction (machine instructions) can be placed in this area. Since it is used for data section of the program it is by default a READWRITE memory. In ARM Assembly language we use this area to set aside SRAM memory for scratch pad and stack. The following line defines a new area for defining variables:

```
AREA    OUR_VARIABLES, DATA, READWRITE
```

To define constant values in the flash memory we write the following:

```
AREA    OUR_CONSTS, DATA, READONLY
```

COMMON is an attribute given to an area of DATA memory section which can be used commonly by several program codes. We do not initialize the COMMON section of the memory since it is used by compiler exclusively. The compiler initializes the COMMON memory area with all zeros.

ALIGN is another attribute given to an area of memory to indicate how memory should be allocated according to the addresses. When the ALIGN is used for CODE and READONLY it aligned in 4-bytes address boundary by default since the ARM instructions are all 32-bit (4-bytes) word. The ALIGN attribute of AREA has a number after like ALIGN=3 which indicates the information should be placed in memory with addresses of

2^3, that is 0x50000, 0x50008, 0x50010, 0x50020, and so on. We will see more about that soon. The usage and importance of ALIGN attribute is discussed in Chapter 6.

ENTRY

Another important pseudocode is the ENTRY directive. This indicates to the assembler the beginning of the executable code. The ENTRY directive is the first line of the ARM Assembly language code section of the program, meaning that anything after the ENTRY directive in the source code is considered actual machine instruction to be executed by the CPU. For a given ARM Assembly language program we can have only one ENTRY point. Having multiple ENTRY directive in an Assembly language program will give you an error by assembler.

END directive

Another important pseudocode is the END directive. This indicates to the assembler the end of the source (asm) file. The END directive is the last line of the ARM Assembly language program, meaning that anything after the END directive in the source code is ignored by the assembler. Program 2-1 shows how the AREA, ENTRY and END directives are used.

Program 2-1

```
;ARM Assembly Language Program To Add Some Data and Store the SUM in R3.

        AREA   PROG_2_1, CODE, READONLY
  ENTRY
  MOV     R1, #0x25          ;R1 = 0x25
  MOV     R2, #0x34          ;R2 = 0x34
  ADD      R3, R2,R1          ;R3 = R2 + R1
HERE     B          HERE                ;stay here forever
        END
```

LDR

In the last section, we stated that one cannot load values larger than 0xFF into the 32-bit registers of ARM since only 8 bits are used for the immediate value. The ARM assembler provide us a pseudo-instruction of "LDR Rd,=32-bit_immidiate_vlaue" to load value greater than 0xFF. We will examine how this pseudo-instruction works in Chapter 6. For now, just notice the = sign used in the syntax. The following pseudo-instruction loads R7 with 0x112233.

```
  LDR      R7,=0x112233
```

We will use this pseudo-instruction to load 32-bit value into register extensively throughout the book. To load values less than 0xFF, we still use the "MOV Rd,#8-

bit_immidiate_value" instruction since it is a real instruction of ARM, therefore more efficient in code size.

EQU (equate)

This is used to define a constant value or a fixed address. The EQU directive does not set aside storage for a data item, but associates a constant number with a data or an address label so that when the label appears in the program, its constant will be substituted for the label. The following uses EQU for the counter constant, and then the constant is used to load the R2 register:

```
COUNT           EQU     0x25
…       …       ….
   MOV    R2, #COUNT     ;R2 = 0x25
```

When executing the above instruction "MOV R2, #COUNT", the register R2 will be loaded with the value 0x25. What is the advantage of using EQU? Assume that a constant (a fixed value) is used throughout the program, and the programmer wants to change its value everywhere. By the use of EQU, the programmer can change it once and the assembler will change all of its occurrences throughout the program. This allows the programmer to avoid searching the entire program trying to find every occurrence.

Using EQU for fixed data assignment

To get more practice using EQU to assign fixed data, examine the following:

```
DATA1  EQU     0x39               ;the way to define hex value
DATA2  EQU     2_00110101       ;the way to define binary value (35 in hex)
DATA3  EQU     39                 ;decimal numbers (27 in hex)
DATA4  EQU     '2'                 ;ASCII characters
```

Using EQU for SFR address assignment

EQU is also widely used to assign SFR addresses. Examine the following code:

```
PORTB  EQU     0xF0018            ;SFR Port B address
   MOV    R6,#0x01           ;R6 = 0x01
   LDR     R2,=PORTB        ;R2 = 0xF0018
   STRB    R6,[R2]            ;Port B now has 0x01
```

Using EQU for RAM address assignment

Another common usage of EQU is for the address assignment of the internal SRAM. It is exactly like using EQU for SFR address assignment. Examine the following code:

```
SUM     EQU     0x40000120       ;assign RAM loc to SUM
   MOV    R2,#5                ;load R2 with 5
```

```
    MOV     R1,#2               ;load R1 with 2
    ADD      R2, R2,R1          ;R2 = R2 + R1
    LDR      R3,=SUM            ;load R3 with 0x40000120
    STRB     R2,[R3]            ;store the result SUM
```

This is especially helpful when the address needs to be changed in order to use a different ARM chip for a given project. It is much easier to refer to a name than a number when accessing RAM address locations.

RN (equate)

This is used to define a name for a register. The RN directive does not set aside a seperate storage for the name, but associates a register with that name. It improves the clarity. Program 2-2 shows how we use SUM name for R3.

Program 2-2: An ARM Assembly Language Program Using RN Directive

```
    ;ARM Assembly Language Program To Add Some Data
   ;and store the SUM in R3.

VAL1      RN          R1          ;define VAL1 as a name for R1
VAL2      RN          R2          ;define VAL2 as a name for R2
SUM      RN          R3          ;define SUM as a name for R3

          AREA     PROG_2_2, CODE, READONLY
   ENTRY
   MOV     VAL1, #0x25                    ;R1 = 0x25
   MOV     VAL2, #0x34                    ;R2 = 0x34
   ADD       SUM, VAL1,VAL2              ;R3 = R2 + R1
HERE      B            HERE
          END
```

INCLUDE directive

The include directive tells the ARM assembler to add the contents of a file to our program (like the #include directive in C language).

Assembler data allocation directives

In most Assembly languages there are some directives to allocate memory and initialize its value. In ARM Assembly language DCB, DCD, and DCW allocate memory and initialize them. The SPACE directive allocates memory without initializing it.

DCB directive (define constant byte)

The DCB directive allocates a byte size memory and initializes the values.

```
MYVALUE      DCB     5        ;MYVALUE = 5
MYMSAGE      DCB     "HELLO WORLD"        ;string
```

DCW directive (define constant half-word)

The DCW directive allocates a half-word size memory and initializes the values.

```
MYDATA       DCW     0x20, 0xF230, 5000, 0x9CD7
```

DCD directive (define constant word)

The DCD directive allocates a word size memory and initializes the values.

```
MYDATA       DCD     0x200000, 0xF30F5, 5000000, 0xFFFF9CD7
```

See Tables 2-8 and 2-9.

Directive	Description
DCB	Allocates one or more bytes of memory, and defines the initial runtime contents of the memory
DCW	Allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.
DCWU	Allocates one or more halfwords of memory, and defines the initial runtime contents of the memory. The data is not aligned.
DCD	Allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.
DCDU	Allocates one or more words of memory and defines the initial runtime contents of the memory. The data is not aligned.

Table 2- 8: Some Widely Used ARM Memory Allocation Directives

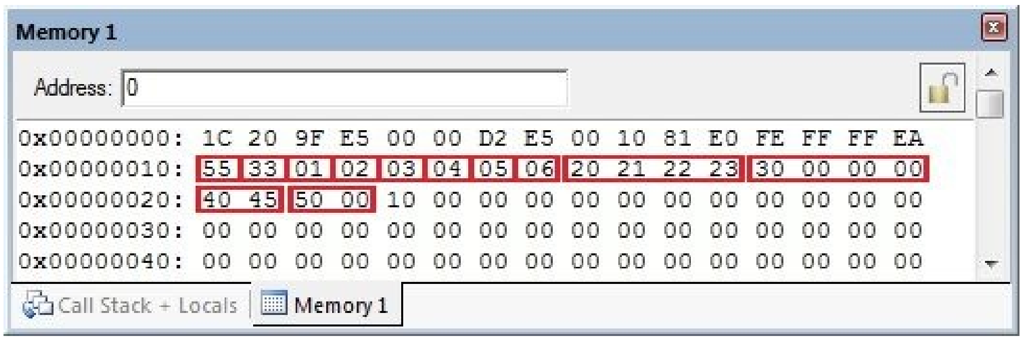
Data Size	Bits	Decimal	Hexadecimal	Directive	Instruction
Byte	8	0 – 255	0 - 0xFF	DCB	STRB/LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	DCW	STRH/LDRH
Word	32	$0 – 2^{32}-1$	0 - 0xFFFFFFFF	DCD	STR/LDR

Table 2- 9: Unsigned Data Range in ARM and associated Instructions

In Program 2-3A you see an example of storing constant values in the program memory using the directives. Figure 2-12 shows how the data is stored in memory. In the example, the program goes to locations 0x00 to 0x0F. The DCB directive stores data in addresses 0x10–0x17. As you see one byte is allocated for each data. The DCD allocates 4 bytes for each data. As a result the lowest byte of 0x23222120 (which is 0x20) is stored in location 0x18 and the next bytes are stored in the next locations.

Program 2-3A: Sample of Storing Fixed Data in Program Memory

```
    ;storing data in program memory.
    AREA    LOOKUP_EXAMPLE,READONLY,CODE
    ENTRY
    LDR     R2,=OUR_FIXED_DATA ;point to OUR_FIXED_DATA
    LDRB    R0,[R2]            ;load R0 with the contents
    ;of memory pointed to by R2
    ADD     R1,R1,R0           ;add R0 to R1
HERE    B       HERE               ;stay here forever
OUR_FIXED_DATA
    DCB     0x55,0x33,1,2,3,4,5,6
    DCD     0x23222120,0x30
    DCW     0x4540,0x50
    END
```



```
0x00000000: 1C 20 9F E5 00 00 D2 E5 00 10 81 E0 FE FF FF EA
0x00000010: 55 33 01 02 03 04 05 06 20 21 22 23 30 00 00 00
0x00000020: 40 45 50 00 10 00 00 00 00 00 00 00 00 00 00 00
0x00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2- 12: Memory Dump for Program 2-3A

The DCW directive allocates 2 bytes for each data. For example, the low byte of 0x4540 is located in address 0x20 and the high byte of it goes to address 0x21. Similarly the low byte of 0x50 is located in address 0x22 and the high byte of it in address 0x23.

In the program to access the data, first the R2 register is loaded with the address of OUR_FIXED_DATA. In this example, OUR_FIXED_DATA has address 0x10. So, R2 is loaded with 0x10. Then, the contents of location 0x10 is loaded into register R0, using the LDRB instruction.

SPACE directive

Using the SPACE directive we can allocate memory for variables. The following lines allocate 4 and 2 bytes of memory and name them as LONG_VAR and OUR_ALFA:

```
LONG_VAR        SPACE   4            ;Allocate 4 bytes
OUR_ALFA         SPACE   2            ;Allocate 2 bytes
```

In the following program 3 variables are defined: A, B, and C. Then A and B are initialized with 5 and 4, respectively. In the next step A and B are added together and the result is put in C:

Program 2-3B

```
   AREA OUR_PROG,CODE,READONLY
       ENTRY
   ;A = 5
   LDR        R0,=A    ;R0 = Addr. of A
           MOV     R1,#5     ;R1 = 5
           STR         R1,[R0] ;init. A with 5
   ;B = 4
       LDR        R0,=B    ;R0 = Addr. of B
       MOV     R1,#4     ;R1 = 4
       STR         R1,[R0]                 ;init. B with 4
       ;R1 = A
       LDR        R0,=A    ;R0 = Addr. of A
           LDR        R1,[R0]                 ;R1 = value of A
       ;R2 = B
       LDR        R0,=B    ;R0 = Addr. of A
           LDR        R2,[R0]                 ;R2 = value of A
       ;C = R1 + R2 (C = A + B)
       ADD       R3,R1,R2 ;R3 = A + B
       LDR        R0,=C    ;R0 = Addr. of C
           STR         R3,[R0]                 ;C = R3

loop       B   loop

   AREA     OUR_DATA,DATA,READWRITE
   ;Allocates the followings in SRAM memory
A             SPACE   4
```

```
B           SPACE   4
C           SPACE   4
   END
```

ADR directive

To load registers with the addresses of memory locations we can also use the ADR pseudo-instruction which has a better performance. See Chapter 6 for more. ADR has the following syntax:

```
   ADR      Rn,label
```

For example, in Program 2-3A we can load R2 with the address of OUR_FIXED_DATA using the following pseudo-instruction:

```
   ADR      R2, OUR_FIXED_DATA   ;point to OUR_FIXED_DATA
```

ALIGN

This is used to make sure data is aligned in 32-bit word or 16-bit half word memory address. The following uses ALIGN to make the data 32-bit word aligned:

```
   ALIGN   4           ;the next instruction is word (4 bytes) aligned
   …
   ALIGN   2           ;the next instruction is half-word (2 bytes) aligned
   …
```

Example 2-7 shows the result of using the ALIGN directive.

Example 2-7

Compare the result of using ALIGN in the following programs:

a)

```
   AREA     E2_7A,READONLY,CODE
   ENTRY
   ADR      R2,DTA
   LDRB      R0,[R2]
   ADD      R1,R1,R0
H1          B           H1


DTA         DCB         0x55
```

```
    DCB       0x22
    END
```

b)

```
    AREA    E2_7B,READONLY,CODE
    ENTRY
    ADR     R2,DTA
    LDRB     R0,[R2]
    ADD     R1,R1,R0
H1          B           H1

DTA         DCB       0x55
    ALIGN   2
    DCB       0x22
    END
```

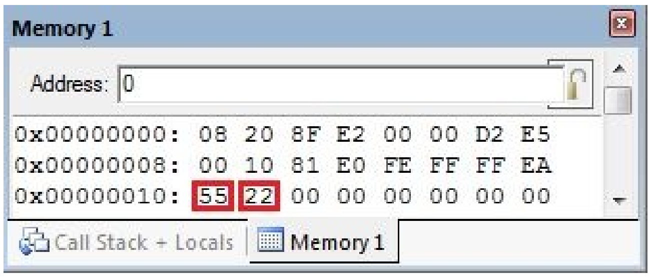
c)

```
    AREA    E2_7C,READONLY,CODE
    ENTRY
    ADR     R2,DTA
    LDRB     R0,[R2]
    ADD     R1,R1,R0
H1          B           H1

DTA         DCB       0x55
    ALIGN   4
    DCB       0x22
    END
```

Solution:

a)

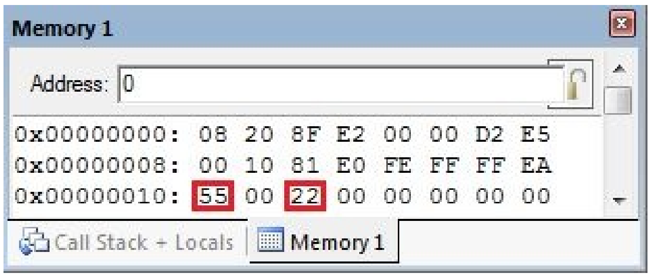
When there is no ALIGN directive the DCB directive allocates the first empty location for its data. In this example, address 0x10 is allocated for 0x55. So 0x22 goes to address 0x11.



```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 22 00 00 00 00 00 00
```

b)

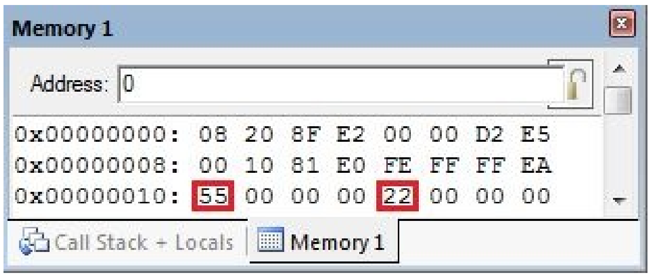
In the example the ALIGN is set to 2 which means the data should be put in a location with even address. The 0x55 goes to the first empty location which is 0x10. The next empty location is 0x11 which is not a multiple of 2. So, it is filled with 0 and the next data goes to location 0x12.



```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 00 22 00 00 00 00 00
```

c)

In the example the ALIGN is set to 4 which means the data should go to locations whose address is multiple of 4. The 0x55 goes to the first empty location which is 0x10. The next empty locations are 0x11, 0x12, and 0x13 which are not a multiple of 4. So, they are filled with 0s and the next data goes to location 0x14.



```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 00 00 00 22 00 00 00
```

Rules for labels in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. First, each label name must be unique. The names used for labels in Assembly language programming consist of alphabetic letters in both uppercase and lowercase, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign ($). The first character of the label must be an alphabetic character. In other words, it cannot be a number. Every assembler has some reserved words that must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions. For example, "MOV" and "ADD" are reserved because they are instruction mnemonics. In addition to the mnemonics there are some other reserved words. Check your assembler for the list of reserved words.

Review Questions

1. Give an example of hex data representation in the ARM assembler.
2. Show how to represent decimal 20 in formats of (a) hex, (b) decimal, and (c) binary in the ARM assembler.
3. What is the advantage in using the EQU directive to define a constant value?
4. Show the hex number value used by the following directives:

   (a) ASC_DATA EQU ‘4’ (b) MY_DATA EQU 2_00011111
5. Give the value in R2 for the following:

```
MYCOUNT        EQU      15
   MOV     R2, #MYCOUNT
```

6. Give the value in data memory location 0x200000 for the following:

```
MYCOUNT        EQU      0x95
MYMEM          EQU      0x200000
   MOV     R0, #MYCOUNT
   LDR      R2, =MYMEM
   STRB     R0, [R2]
```

7. Give the value in data memory 0x630000 for the following:

```
MYDATA         EQU      12
MYMEM          EQU      0x00630000
FACTOR         EQU      0x10
   MOV     R1, #MYDATA
   MOV     R2, #FACTOR
   LDR      R3, =MYMEM
   ADD      R1 R2,R1
   STRB     R1,[R3]
```

Section 2.6: Introduction to ARM Assembly Programming

In this section we discuss Assembly language format and define some widely used terminology associated with Assembly language programming.

While the CPU can work only in binary, it can do so at a very high speed. It is quite tedious and slow for humans, however, to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called machine language. In the early days of the computer, programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, Assembly languages were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error. The term mnemonic is frequently used in computer science and engineering literature to refer to codes and abbreviations that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an assembler. Assembly language is referred to as a low-level language because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know all the registers of the CPU and the size of each, as well as other details.

Today, one can use many different programming languages, such as BASIC, Pascal, C, C++, Java, and numerous others. These languages are called *high-level* languages because the programmer does not have to be concerned with the internal details of the CPU. Whereas an assembler is used to translate an Assembly language program into machine code (sometimes also called *object code* or *opcode* for operation code), high-level languages are translated into machine code by a program called a compiler. For instance, to write a program in C, one must use a C compiler to translate the program into machine language. Next we look at ARM Assembly language format.

Structure of Assembly language

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by two or three operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items. See Program 2-4.

Program 2-4: Sample of an ARM Assembly Language Program

```
;ARM Assembly language program to add some data and store the SUM in R3.

AREA    PROG_2_4, CODE, READONLY
ENTRY
MOV    R1, #0x25          ;R1 = 0x25
MOV    R2, #0x34          ;R2 = 0x34
```

```
  ADD     R3, R2,R1          ;R3 = R2 + R1
HERE    B         HERE
        END
```

An Assembly language program is a series of statements, or lines, which are either Assembly language instructions, such as ADD and MOV, or statements called directives. While instructions tell the CPU what to do, directives (also called pseudo-instructions) give directions to the assembler. For example, in Program 2-4, while the MOV and ADD instructions are commands to the CPU, ENTRY and END are directives to the assembler. The directive END tells the assembler that it is the end of the code, while ENTRY is beginning of the code.

An Assembly language instruction consists of four fields:

```
[label]  mnemonic  [operands]  [;comment]
```

Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted:

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.

2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

```
MOV    R3,#0x55
MOV    R2,#0x67
ADD     R2,R2,R3           ;R2 = R2 + R3
```

ADD and MOV are the mnemonics that produce opcodes; the “0x55” and “0x67” are the operands. Instead of a mnemonic and an operand, these two fields could contain assembler pseudo-instructions, or directives. Remember that directives do not generate any machine code (opcode) and are used only by the assembler, as opposed to instructions that are translated into machine code (opcode) for the CPU to execute. In Program 2-4 the commands END and ENTRY are examples of directives. Many of these pseudo-instructions were discussed in the last section.

3. The comment field begins with a semicolon comment indicator “;”. Comments may be at the end of a line or on a line by themselves. The assembler ignores comments, but they are indispensable to programmers. Although comments are optional, it is recommended that they be used to describe the program in a way that makes it easier for someone else to read and understand.

4. Notice the label “HERE” in the label field in Program 2-4. In the B (Branch) statement the ARM is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and should delete it from your

program. In the next section we will see how to create a ready-to-run program.

Note!

The first column of each line is always considered as label. Thus, be careful to press a Tab at the beginning of each line that does not have label; otherwise, your instruction is considered as a label and an error message will appear when compiling.

Review Questions

1. What is the purpose of pseudo-instructions?
2. _______________ are translated by the assembler into machine code, whereas _____________________ are not.
3. True or false. Assembly language is a high-level language.
4. Which of the following instructions produces opcode? List all that do.

   (a) MOV R6,#0x25 (b) ADD R2,R1,R3 (c) END (d) HERE B HERE
5. Pseudo-instructions are also called ____________.
6. True or false. Assembler directives are not used by the CPU itself. They are simply a guide to the assembler.
7. In Question 4, which one is an assembler directive?

Section 2.7: Assembling an ARM Program

Now that the basic form of an Assembly language program has been given, the next question is: How it is created, assembled, and made ready to run? The steps to create an executable Assembly language program (Figure 2-13) are outlined as follows:

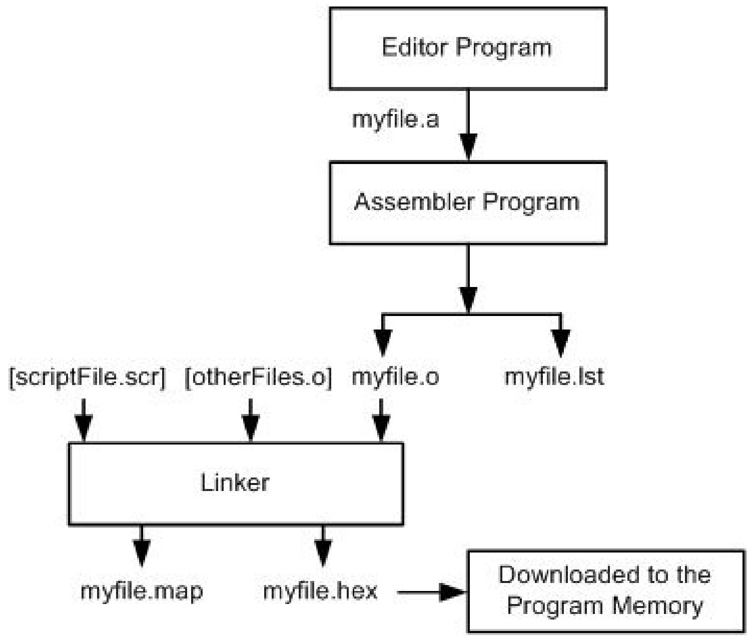


Figure 2- 13: Steps to Create a Program

1. First we use a text editor to type in a program similar to Program 2-4. In the case of ARM, we can use the Keil IDE, which has a text editor, assembler, simulator, and much more all in one software package. It is an excellent development software that supports all the ARM chips and is free for university students. See www.keil.com for evaluation version of the software for students. Many editors or word processors are also available that can be used to create or edit the program. A widely used editor is the Notepad in Windows, which comes with all Microsoft operating systems. Notice that the editor must be able to produce an ASCII file. For assemblers, the file names follow the usual DOS conventions, but the source file has the extension ".a" or ".asm". The "a" extension for the source file is used by an assembler in the next step.

2. The "a" source file containing the program code created in step 1 is fed to the ARM assembler. The assembler produces an object file, and a list file. The object file has the extension ".o", and the list file has ".lst" extension.

3. The object file plus a script file are used by the linker to produce map file and hex file. The map file has the extension ".map", and the hex file has ".hex" extension. The script file is optional and can be replaced with some command line options. After a successful link, the hex file is ready to be burned into the ARM's program ROM and is downloaded into the ARM chip.

More about asm and object files

The asm file is also called the source file and must have the "a" or "asm" extension. As mentioned earlier, this file is created with a text editor such as Windows Notepad.

Many assemblers come with a text editor. The assembler converts the asm file's Assembly language instructions into machine language and provides the o (object) file. The object file, as mentioned earlier, has an "o" as its extension. The object file is used as input to a simulator or an emulator.

Before we can assemble a program to create a ready-to-run program, we must make sure that it is error free. The Keil uVision IDE provides us error messages and we examine them to see the nature of syntax errors. The assembler will not assemble the program until all the syntax errors are fixed. A sample of an error message is shown in Figure 2-14.

```
Build target 'Target 1'
assembling a1.asm…a1.asm(7): error: A1163E: Unknown opcode MOVE, expecting opcode or Macro
Target not created
```

Figure 2- 14: Sample of an Error Message

"lst" and "map" files

The map file shows the labels defined in the program together with their values. Examine Figure 2-15. It shows the Map file of Program 2-4.

```
Memory Map of the image

 Image Entry point : 0x00000000

 Load Region LR_1 (Base: 0x00000000, Size: 0x00000010, Max: 0xffffffff, ABSOLUTE)

 Execution Region ER_RO (Base: 0x00000000, Size: 0x00000010, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
    0x00000000   0x00000010   Code   RO           1   * PROG_2_1           a2.o

    Execution Region ER_RW (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)
    **** No section assigned to this execution region ****


    Execution Region ER_ZI (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)
    **** No section assigned to this execution region ****
```

Figure 2- 15: Sample of a Map File

The lst (list) file, which is optional, is very useful to the programmer. The list shows the binary and source code; it also shows which instructions are used in the source code, and the amount of memory the program uses. See Figure 2-16.

```
ARM Macro Assembler          Page 1

1 00000000                   ;ARM Assembly Language Program To Add Some Data and Store the SUM in R3.

2
```

```
00000000

3 00000000                          AREA   PROG_2_4, CODE, READONLY

4 00000000                          ENTRY

5 00000000    E3A01025              MOV    R1, #0x25    ;R1 = 0x25

6 00000004    E3A02034              MOV    R2, #0x34    ;R2 = 0x34

7 00000008    E0823001              ADD    R3, R2,R1    ;R3 = R2 + R1

8 0000000C    EAFFFFFE    HERE      B      HERE

9 00000010                          END
```

Figure 2- 16: Sample of a List File for ARM

Many assemblers assume that the list file is not wanted unless you indicate that you want to produce it. These files can be accessed by a text editor such as Notepad and displayed on the monitor, or sent to the printer to get a hard copy. The programmer uses the list and map files to locate syntax error.

There are many different ARM assemblers available for evaluation nowadays. If you use the Windows operating system, ARM IAR IDE and Keil uVision can be used. They have great features and nice environments.

Review Questions

1. True or false. The ARM uVision IDE and Windows Notepad text editor both produce an ASCII file.
2. True or false. The extension for the source file is “a”.
3. Which of the following files can be produced by a text editor?

(a) myprog.a (b) myprog.obj (c) myprog.hex (d) myprog.lst

4. Which of the following files is produced by an assembler?

(a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst

Section 2.8: The Program Counter and Program ROM Space in the ARM

In this section we discuss the role of the program counter (PC) in executing a program and show how the code is fetched from ROM and executed. We will also discuss the program (code) ROM space for various ARM family members. Finally, we examine the Harvard architecture of the ARM.

Program counter in the ARM

The most important register in ARM is the PC (program counter). As we mentioned earlier the R15 is the program counter in ARM. The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction. The wider the program counter, the more memory locations a CPU can access. That means that a 32-bit program counter can access a maximum of 4G (2^{32} = 4G) bytes program memory locations.

In most microcontrollers each memory location is 1 byte wide. In the case of a 32-bit program counter, the memory space is 4G (2^{32} = 4G) bytes, which occupies the 0x00000000 –0xFFFFFFFF address range since it is byte-addressable. The program counter in the ARM family is 32 bits wide. This means that the ARM family can access addresses 00000000 to 0xFFFFFFFF, a total of 4G byte of memory space locations. The 4G bytes of memory space locations are allocated among the I/O peripherals, SRAM, and Flash ROM. However, at the time of this writing, none of the members of the ARM family have the entire 4G bytes of memory space populated with on-chip peripherals, SRAM, and Flash.

Power up location for ARM

One question that we must ask about any microcontroller (or microprocessor) is: “At what address does the CPU wake up when power is applied?” Each microprocessor is different. In the case of the ARM microcontrollers (that is, all members regardless of the family and variation), the microcontroller wakes up at memory address 0x00000000 when it is powered up. By powering up we mean applying VCC or activating RESET pin. In other words, when the ARM is powered up, the PC (program counter) has the value of 0x00000000 in it. This means that it expects the first opcode to be stored at ROM address 0x00000000. For this reason, in the ARM system, the first opcode must be burned into memory location 0x00000000 of program ROM because this is where it looks for the first instruction when it is booted. Next we discuss the step-by-step action of the program counter in fetching and executing a sample program.

Placing code in program ROM

To get a better understanding of the role of the program counter in fetching and executing a program, we examine the action of the program counter as each instruction is fetched and executed. First, we examine once more the list file of the sample program and show how the code is placed into the Flash ROM of the ARM chip. As we can see in Figure 2-16, the opcode and operand for each instruction are listed on the left side of the list file.

After the program is burned into ROM of an ARM chip, the opcode and operand are placed in ROM memory locations starting at 0x00000000 as shown in the Program 2-4 list file.

The list shows that address 00000000 contains E3A01025, which is the opcode for moving a value into register, and the operand (in this case 0x25) to be moved to another operand (in this case R1). Therefore, the instruction "MOV R1, #0x25" has a machine code of "E3A01025", where E3A is the opcode and 01025 is the operands. See Figure 2-16. Similarly, the machine code "E3A02034" is located in ROM memory location 00000004 and represents the opcode and the operands for the instruction "MOV R2, #0x34". In the same way, machine code "E0813002" is located in memory location 00000008 and represents the opcode and the operand for the instruction "ADD R3,R1,R2". The opcode for "HERE B HERE" and its target address are located in locations 0000000C. Notice that all the instructions in this program are 4-byte instructions.

Executing a program instruction by instruction

Assuming that the above program is burned into the ROM of an ARM chip, the following is a step-by-step description of the action of the ARM upon applying power to it:

1. When the ARM is powered up, the PC (program counter) has 00000000 and starts to fetch the first instruction from location 00000000 of the program ROM. In the case of the above program the first code is E3A01025, which is the code for moving operand 0x25 to R1. Upon executing the code, the CPU places the value of 25 in R1. Now one instruction is finished. The program counter is now incremented to point to 00000004 (PC = 00000004), which contains code E3A02034, the machine code for the instruction "MOV R2, #0x34".

2. Upon executing the machine code E3A02034, the value 0x34 is loaded to R2. The program counter is incremented to 00000008.

3. ROM location 00000008 has the machine code for instruction "ADD R3,R2,R1". This instruction is executed and now PC = 0000000C.

4. Now PC = 0000000C points to the next instruction, which is "HERE B HERE". After the execution of this instruction, PC = 0000000C. This keeps the program in an infinite loop.

The fact that the program counter points at the next instruction to be executed explains why some microprocessors (notably the x86) call the program counter the instruction pointer.

The steps of running a code in ARM is slightly different from what mentioned above because of the use of pipeline in ARM architecture. We will examine pipelines later in Chapter 7. Also in ARM Cortex M chips the power-on Reset location value is different as we will see in Chapter 8.

Instruction formation of the ARM

Recall that the ARM instructions are always 4-byte. Next we explore the instruction formation for a few of the instructions we have used in this chapter. This should give you some insight into the instructions of the ARM.

ADD instruction formation

The ADD is a 4-byte (32-bit) instruction. See Figure 2-17. Of the 32 bits, the first 4 bits are set aside for the condition field which will be discussed more in Chapter 4. Bits 26 and 27 are always 0 in ADD instruction. Bit 25 which is indicated by I defines the type of second operand. As we mentioned before, the second operand can be either a register or an immediate value between 0–255. If I = 1, the second operand is an immediate value otherwise it should be a register. Bits 24 to 21 are the operation code of ADD instruction. When these bits are 0100 the CPU knows that it should run the ADD instruction. Bit 20 which is indicated by S defines either the instruction should update the flag bits or not. In ADD instruction this bit is zero while in ADDS instruction it is one. Bits 19 to 16 define the first operand (Rn). It can be a register number between R0 to R15. Likewise, bits 15 to 12 define the destination register (Rd). Finally, bits 11 to 0 define the second operand. As we mentioned before, bit 25 (I) defines that either the second operand should be a register or an immediate value. We will discuss the bits of Operand 2 in more detail in Chapter 3.

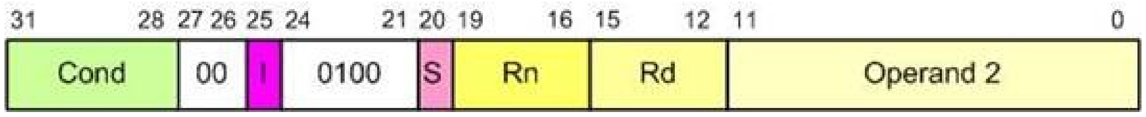


Figure 2- 17: ADD Instruction Formation

SUB instruction formation

The SUB is a 4-byte (32-bit) instruction. Of the 32 bits, the first 4 bits are set aside for the condition field. Bits 26 and 27 are always 0 in SUB instruction. Bit 25 which is indicated by I defines the type of second operand. If I = 1, the second operand is an immediate value otherwise it should be a register. Bits 24 to 21 are the operation code of SUB instruction. When these bits are 0010 the CPU knows that it should run the SUB instruction. Bit 20 which is indicated by S defines either the instruction should update the flag bits or not. In SUB instruction this bit is zero while in SUBS instruction it is one. Bits 19 to 16 define the first operand (Rn). It can be a register number between R0 to R15. Likewise, bits 15 to 12 define the destination register (Rd). Finally, bits 11 to 0 define the second operand. As we mentioned before, bit 25 (I) defines that either the second operand should be a register or an immediate value. The formation of SUB instruction is shown in Figure 2-18.

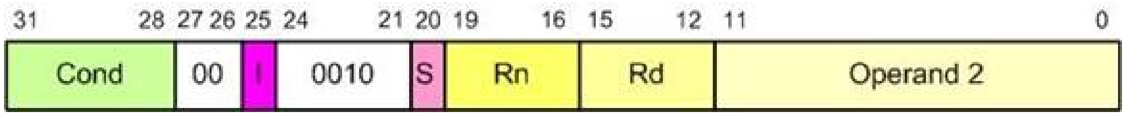


Figure 2- 18: SUB Instruction Formation

General formation of data processing instructions

As you may have noticed, the formation of ADD and SUB instructions are the same except bits 24 to 21 which are called the operation code and tells the CPU what instruction it should execute. In ARM, all of the data processing instructions have the same format. Figure 2-19 shows the general formation of data processing instructions. Each of the data processing instruction has a unique operation code. In Table 2-10 you see the list of all data processing instructions and their opcodes.

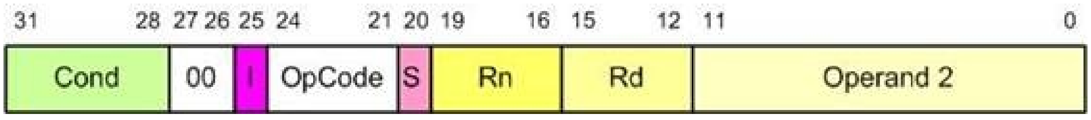


Figure 2- 19: General Formation of Data Processing Instructions

Branch instruction formation

The B is a 4-byte (32-bit) instruction. See Figure 2-20. Of the 32 bits, the first 4 bits are set aside for the condition field which will be discussed more in Chapter 4. Bits 27 to 25 are always 101 in B instruction. Bit 24 which is indicated by L is zero in B instruction and one in BL instruction. Bits 23 to 0 give us branch target location relative to the current address. These will be discussed further in Chapter 4.

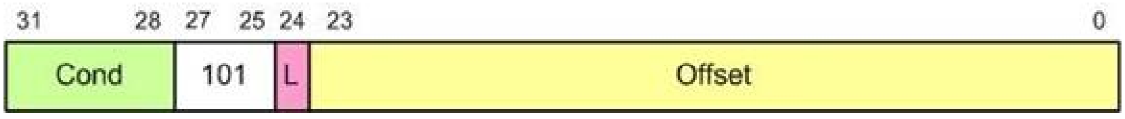


Figure 2- 20: Branch Instruction Formation

ROM width in the ARM

As we have seen so far in this section, each location of the address space holds 1 byte. If we have 32 address lines, this will give us 2^{32} locations, which is 4G bytes of memory location with an address map of 0x00000000–0xFFFFFFFF. To bring in more information (code or data) into the CPU in each bus cycle, ARM increased the width of the data bus to 32 bits. In other words, the ARM is word-addressable. In contrast, the 8051 CPU is byte-addressable only. In a sense, the data bus is like traffic lanes on the highway where each lane is 8 bits wide. The more lanes, the more information we can bring into the CPU for processing. For the ARM, the internal data bus between the code memory and the CPU is 32 bits wide, as shown in Figure 2-21. Therefore, the 4G memory space is shown as 1G × 32 using a 32-bit word data bus size. The widening of the data path between the program ROM and the CPU is another way in which the ARM designers increased the processing power of the ARM family. Another reason to make the code memory 32 bits wide is to match it with the instruction width of the ARM because all of the instructions are 4-byte wide. This way, the CPU brings in an instruction from memory every time it makes a trip to the program memory. That will make instruction fetch a single cycle, as we will see in the Chapter 4 when instruction timing is discussed.

The ARM designers have made all instructions fixed at 4-byte; there are no 1-byte, 2-byte, or 3-byte instructions, as is the case with the x86 and 8051 chips. This is part of the RISC architectural philosophy, which will be discussed later in this chapter. It must also be noted that the data memory SRAM in the ARM microcontroller are also 4-byte.

Harvard and von Neumann architectures in the ARM

In Chapter 0, we discussed Harvard and Von Neumann architecture. ARM 9 and newer architectures use Harvard architecture, which means that there are separate buses for the code and the data memory. See Figure 2-21. The program bus provides access to the program memory whereas the data bus is used for bringing data to the CPU.

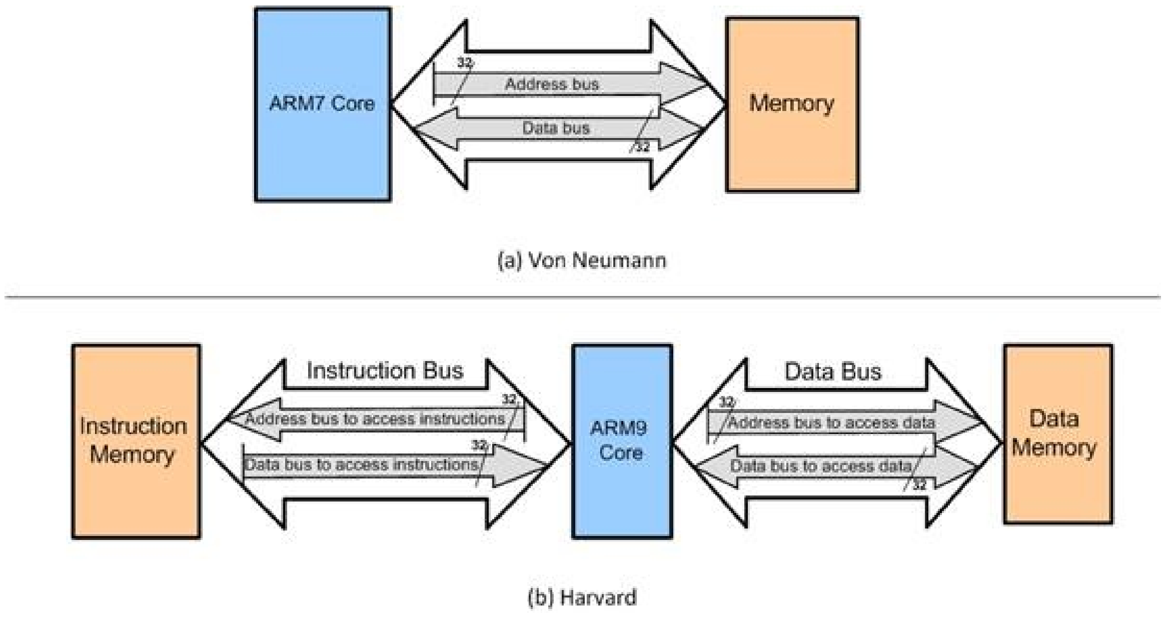


Figure 2- 21: Harvard vs. Von Neumann Architecture

As we can see in Figure 2-21, in the program bus, the data bus is 32 bits wide and the address bus is as wide as the PC register to enable the CPU to address the entire program memory.

In Sections 2-2 and 2-3, we learned about data memory space and how to use the STR and LDR instructions. When the CPU wants to execute the "LDR Rd,[Rx]" instruction, it puts Rx on the address bus of the data bus, and receives data through the data bus. For example, to execute "LDR R2,[R5]", assuming that R5 = 0x40000200, the CPU puts the value of R5 on the address bus. The location 0x40000200 is in the SRAM (see Figure 2-4). Thus, the SRAM puts the contents of location 0x40000200 on the data bus. The CPU gets the contents of location 0x40000200 through the data bus and brings into CPU and puts it in R2.

The "STR Rx,[Rd]" instruction is executed similarly. The CPU puts Rd on the address bus and the contents of Rx on the data bus. The memory location whose address is on the address bus receives the contents of data bus.

Little endian vs. big endian war

Examine the placing of the code in the ARM program memory, shown in Figure 2-22. The low byte goes to the low memory location, and the high byte goes to the high memory address. This convention is called little endian to contrast it with big endian. Figure 2-23 shows storing the same data using big endian convention. The origin of the terms big endian and little endian is from an argument in a Gulliver's Travels story over how an egg should be opened: from the big end or the little end. In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address. All Intel microprocessors and many microcontrollers use the little endian convention. Freescale (formerly Motorola) microprocessors, along with some mainframes, use big endian. The difference might seem as trivial as whether to break an egg from the big end or the little end, but it is a nuisance in converting software from one camp to be run on a computer of the other camp. Many microprocessors, including the ARM, let the software designer choose little endian or big endian convention.

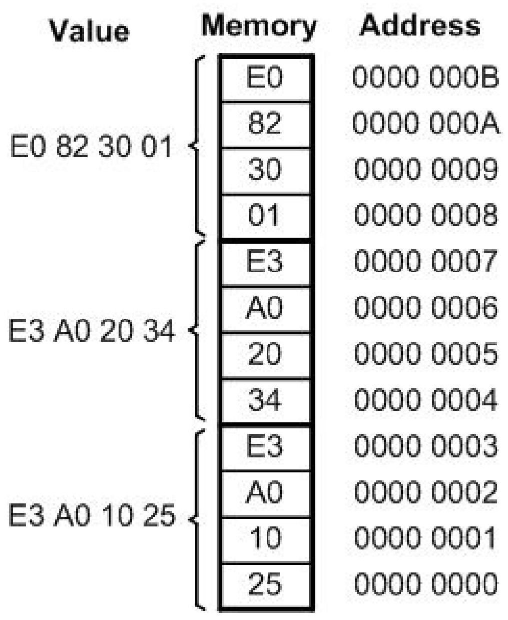


Figure 2- 22: ARM Program Memory Contents for Program 2-4 List File (Little Endian)

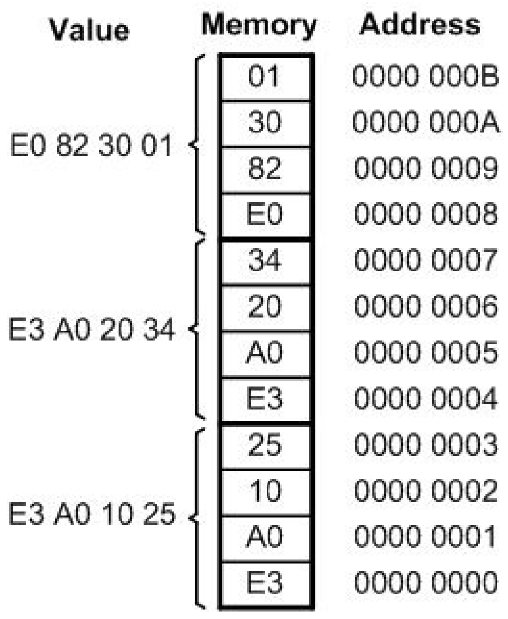


Figure 2- 23: Big Endian Convention

Review Questions

1. In the ARM, the program counter is ______ bits wide.
2. True or false. Every member of the ARM family wakes up at memory 0x00000000 when it is powered up.
3. At what ROM location do we store the first opcode of an ARM program?
4. True or false. All the instructions in the ARM are 4-byte instructions.

5. True or false. ARM9 and newer architectures use von Neumann architecture.
6. True or false. ARM7 and older architectures use von Neumann architecture.

Section 2.9: Some ARM Addressing Modes

The CPU can access operands (data) in various ways, called addressing modes. The number of addressing modes is determined when the microprocessor is designed and cannot be changed. Using advanced addressing modes the accessing of different data types and data structures (e.g. arrays, pointers, classes) are discussed in Chapter 6. Some of the simple ARM addressing modes are:

1. register
2. immediate
3. register indirect (indexed addressing mode)

Register addressing mode

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast. See Figure 2-24.

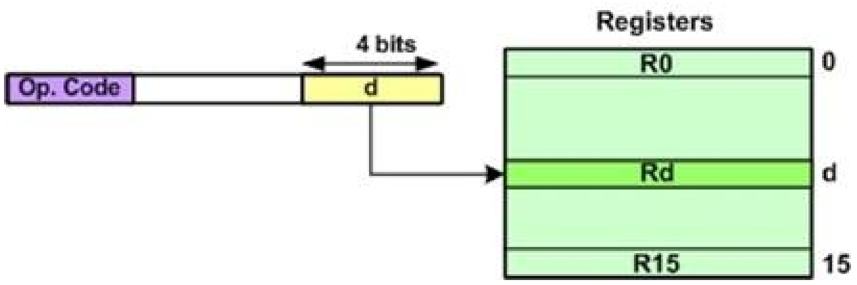


Figure 2- 24: Register Addressing Mode

Examples of register addressing mode are as follow:

```
MOV     R6,R2              ;copy the contents of R2 into R6
ADD      R1,R1,R3          ;add the contents of R3 to contents of R1
SUB      R7,R7,R2          ;subtract R2 from R7
```

Immediate addressing mode

In the immediate addressing mode, the source operand is a constant. In immediate addressing mode, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly. See Figure 2-25. Examples:

```
MOV     R9,#0x25          ;move 0x25 into R9
MOV     R3,#62            ;load the decimal value 62 into R3
ADD      R6,R6,#0x40      ;add 0x40 to R6
```

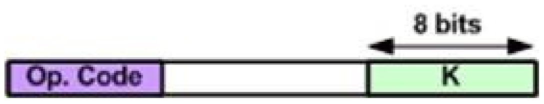


Figure 2- 25: Immediate Addressing Mode

In the first two addressing modes, the operands are either inside the microprocessor or tagged along with the instruction. In most programs, the data to be processed is often in some memory location outside the CPU. There are many ways of accessing the data in the data memory space. The following describes one of the methods.

Register Indirect Addressing Mode (Indexed addressing mode)

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. See Figure 2-26. For example:

```
STR     R5,[R6]          ;move R5 into the memory location
;pointed to by R6
LDR     R10,[R3]         ;move into R10 the contents of the
                          ;memory location pointed to by R3.
```

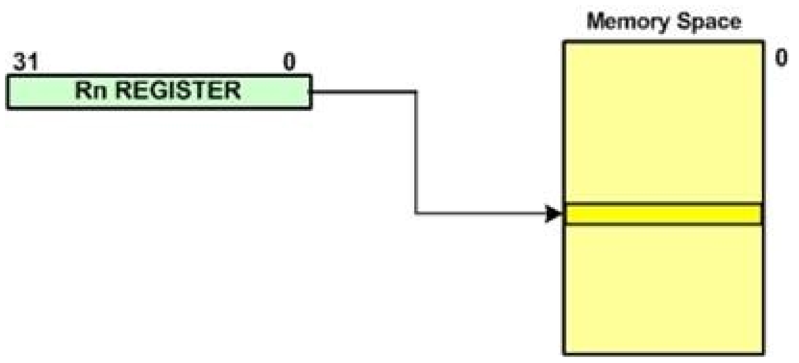


Figure 2- 26: Register Indirect Addressing Mode

Sample Usage: Register indirect addressing mode

Using register indirect addressing mode we can implement the different pointers. Since the registers are 32-bit they can address the entire memory space. Here you see a simple code in C and its equivalent in Assembly:

C Language:

```
char *ourPointer;
ourPointer = (char*) 0x12456; //Point to location 12456
*ourPointer = 25;   //store 25 in location 0x12456
ourPointer ++;     //point to next location
```

Assembly Language:

```
LDR     R2,=0x12456  ;point to location 0x12456
MOV     R0,#25     ;R0 = 25
STRB    R0,[R2]    ;store R0 in location 0x12456
ADD     R2,R2,#1  ;increment R2 to point to next location
```

Depending on the data type that the pointer points to, STR/LDR, STRH/LDRH, or STRB/LDRB might be used. In the above example, since it points to char (which is 8-bit) STRB is used.

See Chapter 6 for more advanced addressing modes.

Review Questions

1. Can the ARM programmer make up new addressing modes?
2. Which registers can be used for the register indirect addressing mode?
3. Where is the data located in immediate addressing mode?

Section 2.10: RISC Architecture in ARM

There are three ways available to microprocessor designers to increase the processing power of the CPU:

1. Increase the clock frequency of the chip. One drawback of this method is that the higher the frequency, the more power and heat dissipation. Power and heat dissipation is especially a problem for hand-held devices.
2. Use Harvard architecture by increasing the number of buses to bring more information (code and data) into the CPU to be processed. While in the case of x86 and other general purpose microprocessors this architecture is very expensive and unrealistic, in today's microcontrollers this is not a problem. As we saw in Section 2.8, some of the ARM chips have Harvard architecture.
3. Change the internal architecture of the CPU and use what is called RISC architecture.

ARM has used all three methods to increase the processing power of the ARM microcontrollers. In this section we discuss the merits of RISC architecture.

In the early 1980s, a controversy broke out in the computer design community, but unlike most controversies, it did not go away. Since the 1960s, in all mainframes and minicomputers, designers put as many instructions as they could think of into the CPU. Some of these instructions performed complex tasks. An example is adding data memory locations and storing the sum into memory. Naturally, microprocessor designers followed the lead of minicomputer and mainframe designers. Because these microprocessors used such a large number of instructions, many of which performed highly complex activities, they came to be known as CISC (complex instruction set computer) processors. According to several studies in the 1970s, many of these complex instructions etched into CPUs were never used by programmers and compilers. The huge cost of implementing a large number of instructions (some of them complex) into the microprocessor, plus the fact that a good portion of the transistors on the chip are used by the instruction decoder, made some designers think of simplifying and reducing the number of instructions. As this concept developed, the resulting processors came to be known as RISC (reduced instruction set computer).

Features of RISC

The following are some of the features of RISC as implemented by the ARM microcontroller.

Feature 1

RISC processors have a fixed instruction size. In a CISC microprocessors such as the x86, instructions can be 1, 2, 3, or even 5 bytes. For example, look at the following instructions in the x86:

```
CLR     C                     ;clear Carry flag, a 1-byte instruction
ADD     Accumulator, #mybyte  ;a 2-byte instruction
```

```
LJMP    target_address ;a 5-byte instruction
```

This variable instruction size makes the task of the instruction decoder very difficult because the size of the incoming instruction is never known. In a RISC architecture, the size of all instructions is fixed. Therefore, the CPU can decode the instructions quickly. This is like a bricklayer working with bricks of the same size as opposed to using bricks of variable sizes. Of course, it is much more efficient to use bricks of the same size. In the last section we saw how the ARM uses 4-byte instructions and if not all the 32 bits are needed to form the instruction it fills with zeros.

Feature 2

One of the major characteristics of RISC architecture is a large number of registers. All RISC architectures have at least 8 or 16 registers. Of these 16 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters. Although a stack is implemented on a RISC processor, it is not as essential as in CISC because so many registers are available. In ARM the use of a large number of general purpose registers (GPRs) satisfies this RISC feature. The stack for the ARM is covered in Chapter 6.

Feature 3

RISC processors have a small instruction set. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR, CALL, JUMP, and so on. The limited number of instructions is one of the criticisms leveled at the RISC processor because it makes the job of Assembly language programmers much more tedious and difficult compared to CISC Assembly language programming. This is one reason that RISC is used more commonly in high-level language environments such as the C programming language rather than Assembly language environments. It is interesting to note that some defenders of CISC have called it "complete instruction set computer" instead of "complex instruction set computer" because it has a complete set of every kind of instruction. How many of these instructions are used and how often is another matter. The limited number of instructions in RISC leads to programs that are large. Although these programs can use more memory, this is not a problem because memory is cheap. Before the advent of semiconductor memory in the 1960s, however, CISC designers had to pack as much action as possible into a single instruction to get the maximum bang for their buck. In the ARM we have around 50 instructions. We will examine more of the instruction set for the ARM in future chapters.

Feature 4

At this point, one might ask, with all the difficulties associated with RISC programming, what is the gain? The most important characteristic of the RISC processor is that more than 99% of instructions are executed with only one clock cycle, in contrast to CISC instructions. Even some of the 1% of the RISC instructions that are executed with two clock cycles can be executed with one clock cycle by juggling instructions around (code scheduling). Code scheduling is discussed in Chapter 7. We will examine the instruction cycle time and pipelining of the ARM in Chapter 7.

Feature 5

RISC processors have separate buses for data and code. In all the x86 processors, like all other CISC computers, there is one set of buses for the address (e.g., A0–A31 in the 80386) and another set of buses for data (e.g., D0–D31 in the 80386) carrying opcodes and operands in and out of the CPU. To access any section of memory, regardless of whether it contains code or data operands, the same address bus and data bus are used. In many RISC processors, there are four sets of buses: (1) a set of data buses for carrying data (operands) in and out of the CPU, (2) a set of address buses for accessing the data, (3) a set of buses to carry the opcodes, and (4) a set of address buses to access the opcodes. The use of separate buses for code and data operands is commonly referred to as Harvard architecture. We examined the Harvard architecture of the ARM in the previous section.

Feature 6

Because CISC has such a large number of instructions, each with so many different addressing modes, microinstructions (microcode) are used to implement them. The implementation of microinstructions inside the CPU employs more than 40–60% of transistors in many CISC processors. RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method. Hardwiring of RISC instructions takes no more than 10% of the transistors.

Feature 7

RISC uses load/store architecture. In CISC microprocessors, data can be manipulated while it is still in memory. For example, in instructions such as “ADD Reg, Memory”, the microprocessor must bring the contents of the external memory location into the CPU, add it to the contents of the register, then move the result back to the external memory location. The problem is there might be a delay in accessing the data from external memory. Then the whole process would be stalled, preventing other instructions from proceeding in the pipeline. In RISC, designers did away with these kinds of instructions. In RISC, instructions can only load from external memory into registers or store registers into external memory locations. There is no direct way of doing arithmetic and logic operations between a register and the contents of external memory locations. All these instructions must be performed by first bringing both operands into the registers inside the CPU, then performing the arithmetic or logic operation, and then sending the result back to memory. This idea was first implemented by the Cray 1 supercomputer in 1976 and is commonly referred to as load/store architecture. In the last section, we saw that the arithmetic and logic operations are between the GPRs registers, but none involves a memory location. For example, there is no “ADD R1, RAM-Loc” instruction in ARM.

In concluding this discussion of RISC processors, it is interesting to note that RISC technology was explored by the scientists at IBM in the mid-1970s, but it was

David Patterson of the University of California at Berkeley who in 1980 brought the merits of RISC concepts to the attention of computer scientists. It must also be noted that in recent years CISC processors such as the Pentium have used some RISC features in their design. This was the only way they could enhance the processing power of the x86 processors and stay competitive. Of course, they had to use lots of transistors to do the job, because they had to deal with all the CISC instructions of the x86 processors and the legacy software of DOS/Windows.

Review Questions

1. What do RISC and CISC stand for?
2. True or false. The CISC architecture executes the vast majority of its instructions in 2, 3, or more clock cycles, while RISC executes them in one clock.
3. RISC processors normally have a _____ (large, small) number of general-purpose registers.
4. True or false. Instructions such as "ADD R16, ROMmemory" do not exist in RISC microprocessors such as the ARM.
5. How many instructions of ARM are 32-bit wide?
6. True or false. While CISC instructions are of variable sizes, RISC instructions are all the same size.
7. Which of the following operations do not exist for the ADD instruction in RISC?

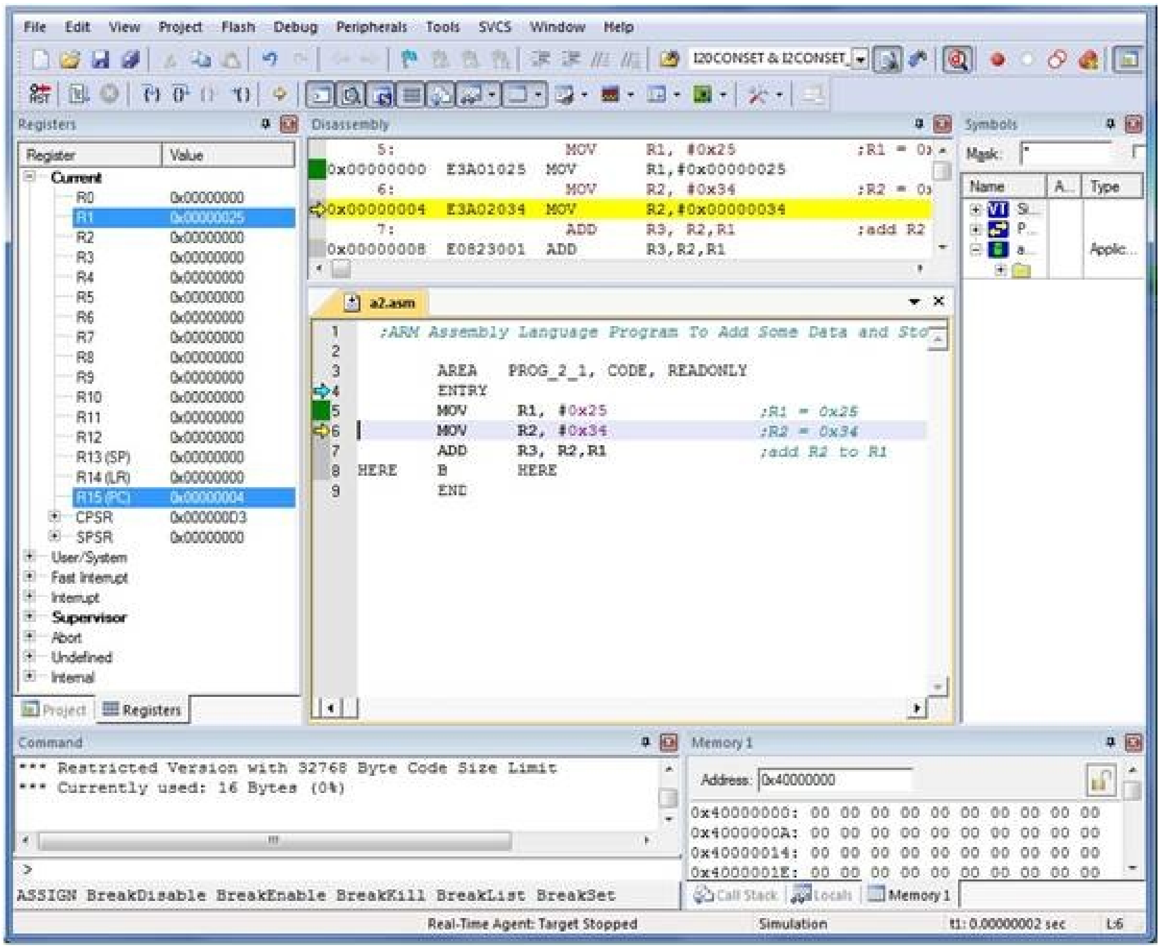
   (a) register to register (b) immediate to register (c) memory to memory
8. True or false. Harvard architecture uses the same address and data buses to fetch both code and data.

Section 2.11: Viewing Registers and Memory with ARM Keil IDE

The ARM microprocessor has great tools and support systems, many of them free or inexpensive. ARM Keil uVision is an assembler and simulator provided for free by Keil Corporation and can be downloaded from the www.keil.com website. See http://www.MicroDigitalEd.com for tutorials on how to use the Keil ARM uVision assembler and simulator.

Many assemblers and C compilers come with a simulator. Simulators allow us to view the contents of registers and memory after executing each instruction (single-stepping). It is strongly recommended to use a simulator to single-step some of the programs in this chapter and future chapters. Single-stepping a program with a simulator gives us a deeper understanding of microcontroller architecture, in addition to the fact that we can use it to find the errors in our programs.

Figures 2-27 and 2-28 show screenshots for ARM simulators from ARM Keil uVision.



Register	Value
Current	
R0	0x00000000
R1	0x00000025
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000004
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	

```
     5:                 MOV       R1, #0x25                  ;R1 = 0x
0x00000000  E3A01025  MOV       R1,#0x00000025
     6:                 MOV       R2, #0x34                  ;R2 = 0x
0x00000004  E3A02034  MOV       R2,#0x00000034
     7:                 ADD       R3, R2,R1                  ;add R2
0x00000008  E0823001  ADD       R3,R2,R1
```

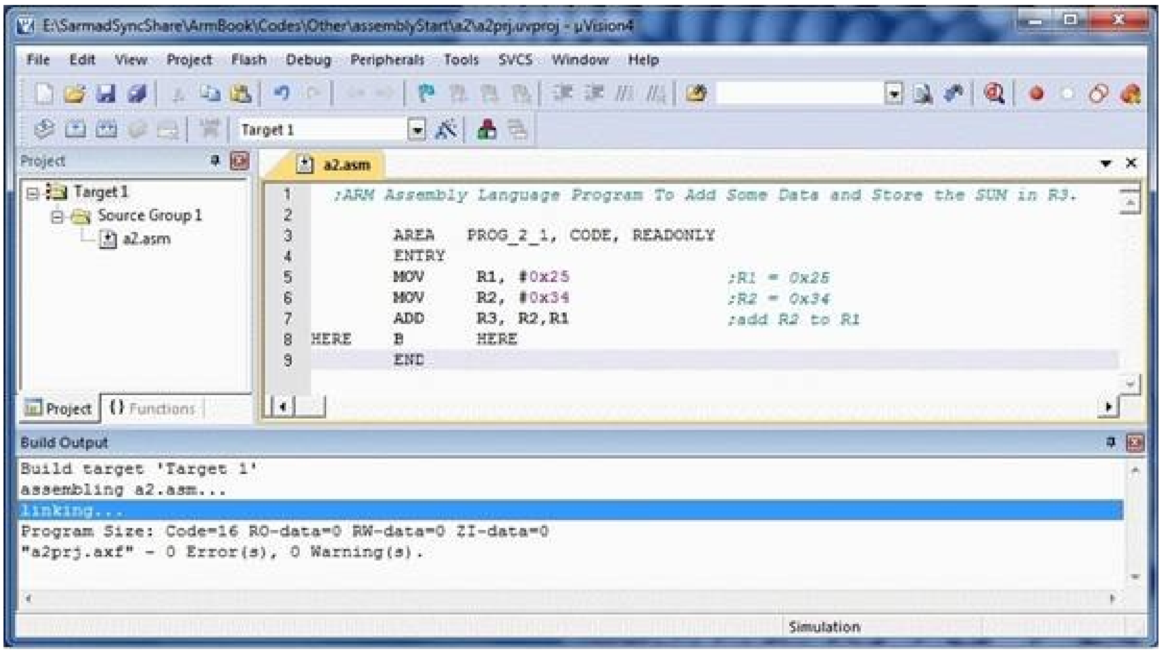
```
   ;ARM Assembly Language Program To Add Some Data and Sto

        AREA    PROG_2_1, CODE, READONLY
        ENTRY
        MOV      R1, #0x25               ;R1 = 0x25
        MOV      R2, #0x34               ;R2 = 0x34
        ADD      R3, R2,R1               ;add R2 to R1
HERE    B        HERE
        END
```

```
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 16 Bytes (0%)

>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

```
Address: 0x40000000
0x40000000: 00 00 00 00 00 00 00 00 00 00
0x4000000A: 00 00 00 00 00 00 00 00 00 00
0x40000014: 00 00 00 00 00 00 00 00 00 00
0x4000001E: 00 00 00 00 00 00 00 00 00 00
```

```
Real-Time Agent: Target Stopped     Simulation     t1: 0.00000002 sec     L:6
```

Figure 2- 27: Keil uVision Screenshot



```
;ARM Assembly Language Program To Add Some Data and Store the SUM in R3.

        AREA    PROG_2_1, CODE, READONLY
        ENTRY
        MOV     R1, #0x25               ;R1 = 0x25
        MOV     R2, #0x34               ;R2 = 0x34
        ADD     R3, R2,R1               ;add R2 to R1
HERE    B       HERE
        END
```

```
Build target 'Target 1'
assembling a2.asm...
linking...
Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0
"a2prj.axf" - 0 Error(s), 0 Warning(s).
```

Figure 2- 28: Keil uVision Screenshot

Problems

Section 2.1: The General Purpose Registers in the ARM

1. ARM is a(n) _____-bit microprocessor.
2. The general purpose registers are _____ bits wide.
3. The value in MOV R2,#value is _____ bits wide.
4. The largest number that an ARM GPR register can have is _____ in hex.
5. What is the result of the following code and where is it kept?

```
MOV    R2,#0x15
MOV    R1,#0x13
ADD     R2,R1,R2
```

6. Which of the followings is (are) illegal?

   (a) MOV R2,#0x50000 (b) MOV R2,#0x50   (c) MOV R1,#0x00

   (d) MOV R1,255   (e) MOV R17,#25   (f) MOV R23,#0xF5

   (g) MOV 123,0x50

7. Which of the following is (are) illegal?

   (a) ADD R2,#20,R1   (b) ADD R1,R1,R2   (c) ADD R5,R16,R3

8. What is the result of the following code and where is it kept?

```
MOV  R9,#0x25
ADD  R8,R9,#0x1F
```

9. What is the result of the following code and where is it kept?

```
MOV  R1,#0x15
ADD  R6,R1,#0xEA
```

10. True or false. We have 32 general purpose registers in the ARM.

Section 2.2: The ARM Memory Map

11. True or false. R13 and R14 are special function registers.
12. True or false. The peripheral registers are mapped to memory space.
13. True or false. The On-chip Flash is the same size in all members of ARM.
14. True or false. The On-chip data SRAM is the same size in all members of ARM.
15. What is the difference between the EEPROM and data SRAM space in the ARM?
16. Can we have an ARM chip with no EEPROM?

17. Can we have an ARM chip with no data RAM?

18. What is the maximum number of bytes that the ARM can access?

19. Find the address of the last location of on-chip Flash for each of the following, assuming the first location is 0:

    (a) ARM with 32 KB (b) ARM with 8 KB

    (c) ARM with 64 KB (d) ARM with 16 KB

    (e) ARM with 128 KB (f) ARM with 256 KB

20. Show the lowest and highest values (in hex) that the ARM program counter can take.

21. A given ARM has 0x7FFF as the address of the last location of its on-chip ROM. What is the size of on-chip Flash for this ARM?

22. Repeat Question 21 for 0x3FFF.

23. Find the on-chip program memory size in K for the ARM chip with the following address ranges:

    (a) 0x0000–0x1FFF (b) 0x0000–0x3FFF

    (c) 0x0000–0x7FFF (d) 0x0000–0xFFFF

    (e) 0x0000–0x1FFFF (f) 0x00000–0x3FFFF

24. Find the on-chip program memory size in K for the ARM chips with the following address ranges:

    (a) 0x00000–0xFFFFFF (b) 0x00000–0x7FFFF

    (c) 0x00000–0x7FFFFF (d) 0x00000–0xFFFFF

    (e) 0x00000–0x1FFFFF (f) 0x00000–0x3FFFFF

Section 2.3: Load and Store Instructions in ARM

25. Show a simple code to store values 0x30 and 0x97 into locations 0x20000015 and 0x20000016, respectively.

26. Show a simple code to load the value 0x55 into locations 0x20000030–0x20000038.

27. True or false. We cannot load immediate values into the data SRAM directly.

28. Show a simple code to load the value 0x11 into locations 0x20000010–0x20000015.

29. Repeat Problem 28, except load the value into locations 0x20000034–0x2000003C.

Section 2.4: ARM CPSR (Current Program Status Register)

30. The status register is a(n) ______ -bit register.
31. Which bits of the status register are used for the C and Z flag bits, respectively?
32. Which bits of the status register are used for the V and N flag bits, respectively?
33. In the ADD instruction, when is C raised?
34. In the ADD instruction, when is Z raised?
35. What is the status of the C and Z flags after the following code?

```
LDR     R0,=0xFFFFFFFF
LDR     R1,=0xFFFFFFF1
ADDS    R1,R0,R1
```

36. Find the C flag value after each of the following codes:

```
(a) LDR R0,=0xFFFFFF54   (b) MOV R3,#0            (c) LDR R3,=0xFFFFFFFF
LDR R5,=0xFFFFFFC4       LDR R6,=0xFFFFFFFF       LDR R8,=0xFFFFFF05
ADDS R2,R5,R0            ADDS R3,R3,R6            ADDS R2,R3,R8
```

37. Write a simple program in which the value 0x55 is added 5 times.

Section 2.5: ARM Data Format and Directives

38. State the value (in hex) used for each of the following data:

```
MYDAT_1 EQU 55
MYDAT_2 EQU 98
MYDAT_3 EQU ‘G’
MYDAT_4 EQU 0x50
MYDAT_5 EQU 200
MYDAT_6 EQU ‘A’
MYDAT_7 EQU 0xAA
MYDAT_8 EQU 255
MYDAT_9 EQU 2_10010000
MYDAT_10 EQU 2_01111110
MYDAT_11 EQU 10
MYDAT_12 EQU 15
```

39. State the value (in hex) for each of the following data:

    DAT_1 EQU 22

    DAT_2 EQU 0x56

    DAT_3 EQU 2_10011001

    DAT_4 EQU 32

    DAT_5 EQU 0xF6

    DAT_6 EQU 2_11111011

40. Show a simple code to load the value 0x10102265 into locations 0x40000030–0x4000003F.
41. Show a simple code to (a) load the value 0x23456789 into locations 0x40000060–0x4000006F, and (b) add them together and place the result in R9 as the values are added. Use EQU to assign the names TEMP0–TEMP3 to locations 0x40000060–0x4000006F.

Section 2.6: Introduction to ARM Assembly Programming and

Section 2.7: Assembling an ARM Program

42. Assembly language is a ________ (low, high)-level language while C is a ________ (low, high)-level language.
43. Of C and Assembly language, which is more efficient in terms of code generation (i.e., the amount of program memory space it uses)?
44. Which program produces the obj file?
45. True or false. The source file has the extension “asm”.
46. True or false. The source code file can be a non-ASCII file.
47. True or false. Every source file must have EQU directive.
48. Do the EQU and END directives produce opcodes?
49. Why are the directives also called pseudocode?
50. The file with the _______ extension is downloaded into ARM Flash ROM.
51. Give three file extensions produced by ARM Keil.

Section 2.8: The Program Counter and Program ROM Space in the ARM

52. Every ARM family member wakes up at address _________ when it is powered up.
53. A programmer puts the first opcode at address 0x100. What happens when the

microcontroller is powered up?

54. ARM instructions are _______ bytes.

55. Write a program to add each of your 5-digit ID to a register and place the result into memory location 0x4000100. Use the program listing to show the Flash memory addresses and their contents.

56. Show the placement of data in following code:

    LDR R1,=0x22334455

    LDR R2,=0x20000000

    STR R1,[R2]

    Use a) little endian and b) big endian.

57. Show the placement of data in following code:

    LDR R1,=0xFFEEDDCC

    LDR R2,=0x2000002C

    STR R1,[R2]

    Use a) little endian and b) big endian.

58. How wide is the memory in the ARM chip?

59. How wide is the data bus between the CPU and the program memory in the ARM7 chip?

60. In "ADD Rd,Rn,operand2", explain how many bits are set aside for Rd and how it covers the entire GPRs in the ARM chip.

Section 2.9: Some ARM Addressing Modes

61. Give the addressing mode for each of the following:

(a) MOV R5,R3	(b) MOV R0,#56
(c) LDR R5,[R3]	(d) ADD R9,R1,R2
(e) LDR R7,[R2]	(f) LDRB R1,[R4]

62. Show the contents of the memory locations after the execution of each instruction.

(a) LDR R2,=0x129F	(b) LDR R4,=0x8C63
LDR R1,=0x1450	LDR R1,=0x2400
LDR R2,[R1]	LDRH R4,[R1]
0x1450 = (…….)	0x2400 = (…….)

0x1451 = (…….) 0x2401 = (…….)

Section 2.10: RISC Architecture in ARM

63. What do RISC and CISC stand for?
64. In ______ (RISC, CISC) architecture we can have 1-, 2-, 3-, or 4-byte instructions.
65. In ______ (RISC, CISC) architecture instructions are fixed in size.
66. In ______ (RISC, CISC) architecture instructions are mostly executed in one or two cycles.
67. In ______ (RISC, CISC) architecture we can have an instruction to ADD a register to external memory.
68. True or false. Most instructions in CISC are executed in one or two cycles.

Answers to Review Questions

Section 2.1

1. MOV R2,#0x34
2.

```
MOV     R1,#0x16
MOV     R2,#0xCD
ADD     R1,R1,R2
```

or

```
MOV     R1,#0x16
ADD     R1,R1,#0xCD
```

3. False
4. FF in hex and 255 in decimal
5. 32

Section 2.2

1. True
2. general-purpose registers
3. 32
4. Special function registers (SFRs)
5. 32

Section 2.3

1. True
2.

```
MOV     R1,#0x20
MOV     R2,#0x95
STRB    R2,[R1]
```

3. STR R2,[R8]
4.

```
MOV     R1,#0x20
LDR     R4,[R1]
```

5. FF in hex or 255 in decimal
6. R6

7. It copies the lower 8 bits of R1 into location pointed to by R2.

8. FFFFFFFF in hex or 4,294,967,295 in decimal (2^{32}-1)

Section 2.4

1. CPSR (current program status register)
2. 32 bits
3.

Hex	Binary
FFFFFF9F	1111 1111 1111 1111 1111 1111 1001 1111
+00000061	+ 0000 0000 0000 0000 0000 0000 0110 0001
1 00000000	1 0000 0000 0000 0000 0000 0000 0000 0000

This leads to C = 1 and Z = 1.

4.

Hex	Binary
00000022	0000 0000 0000 0000 0000 0000 0010 0010
+00000022	+ 0000 0000 0000 0000 0000 0000 0010 0010
0 00000000	0000 0000 0000 0000 0000 0000 0100 0100

This leads to Z = 0.

5.

Hex	Binary
0000 0067	0000 0000 0000 0000 0000 0000 0110 0111
+ 0000 0099	+ 0000 0000 0000 0000 0000 0000 1001 1001
0000 0100	0000 0000 0000 0000 0000 0001 0000 0000

This leads to C = 0 and Z = 0.

Section 2.5

1. MOV R1,#0x20

2. (a) MOV R2,#0x14 (b) MOV R2,#20 (c) MOV R2,#2_00010100
3. If the value is to be changed later, it can be done once in one place instead of at every occurrence and the code becomes more readable, as well.
4. (a) 0x34 (b) 0x1F
5. 15 in decimal (0x0F in hex)
6. Value of location 0x00000200 = 0x95
7. 0x0C + 0x10 = 0x1C will be in data memory location 0x00000630.

Section 2.6

1. The real work is performed by instructions such as MOV and ADD. Pseudo-instructions, also called Assembly directives, direct the assembler in doing its job.
2. The instruction mnemonics, pseudo-instructions
3. False
4. All except (c)
5. Assembler directives
6. True
7. (c)

Section 2.7

1. True
2. True
3. (a)
4. (b) and (d)

Section 2.8

1. 32
2. True
3. 0x00000000
4. True
5. False
6. True

Section 2.9

1. No
2. The general purpose registers (R0 to R15)

3. It is a part of the instruction

Section 2.10

1. RISC is reduced instruction set computer; CISC stands for complex instruction set computer.
2. True
3. Large
4. True
5. All of them
6. True
7. (c)
8. False

Chapter 3: Arithmetic and Logic Instructions and Programs

In this chapter, most of the arithmetic and logic instructions are discussed and program examples are given to illustrate the application of these instructions. Unsigned numbers are used in this discussion of arithmetic and logic instructions. In Section 3.1 we examine the arithmetic instructions for unsigned numbers. The logic instructions and programs are covered in Section 3.2. Section 3.3 is dedicated to rotate and shift operations. We examine loading fixed (constant) values into registers using rotate options, as well. In Section 3.4 we discuss the ARM Cortex instructions for rotate and shift. Section 3.5 is dedicated to BCD and ASCII data conversion.

Section 3.1: Arithmetic Instructions

Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign. This means that the operand can be between 00 and 0xFF (0 to 255 decimal) for 8-bit data and between 0x0000 and 0xFFFF (0 to 65535 decimal) for 16-bit data. For the 32-bit operand it can be between 0 and 0xFFFFFFFF (0 to 2^{32} -1). See Table 3-1. This section covers the ADD, SUB, and multiply instructions for unsigned number.

Data Size	Bits	Decimal	Hexadecimal	Load instruction used
Byte	8	0 – 255	0 - 0xFF	STRB
Half-word	16	0 – 65535	0 - 0xFFFF	STRH
Word	32	0 – 2^{32}-1	0 – 0xFFFFFFFF	STR

Table 3-1: Unsigned Data Range Summary in ARM

Affecting flags in ARM instructions

A unique feature of the execution of ARM arithmetic instructions is that it does not affect (updates) the flags unless we specify it. This is different from other microcontrollers and CPUs such as 8051 and x86. In the x86 and 8051 the arithmetic instructions automatically change the Z and C flags regardless of we want it or not. This is not the case with ARM. The default for the ARM instruction is not to affect these flags after the execution of arithmetic instructions such as ADD and SUB. The ARM assembler gives us the option of telling the CPU to update the flag bits in the CSPR register to reflect the result. We override the default by having letter S in the instruction. For example, we must use SUBS instead of SUB since the SUB instruction will not update the flags. The SUBS means subtract and set the flags, while the SUB simply subtracts without having any effect on the flags. See Table 3-2 and Figure 3-1.

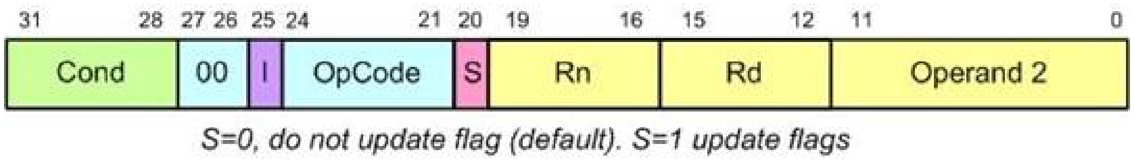


Figure 3- 1: General Formation of Data Processing Instruction

Instruction (Flags unchanged)		Instruction (Flags updated)	
ADD	Add	**ADDS**	Add and set flags
ADC	Add with carry	**ADCS**	Add with carry and set flags
SUB	SUBS	**SUBS**	Subtract and set flags
SBC	Subtract with carry	**SBCS**	Subtract with carry and set flags
MUL	Multiply	**MULS**	Multiply and set flags

UMULL	Multiply long	**UMULLS**	Multiply Long and set flags
RSB	Reverse subtract	**RSBS**	Reverse subtract and set flags
RSC	Reverse subtract with carry	**RSCS**	Reverse subtract with carry and set flags
Note: The above instruction affect all the Z, C, V and N flag bits of CPSR (current program status register) but the N and V flags are for signed data and are discussed in Chapter 5.			

Table 3-2: Arithmetic Instructions and Flag Bits for Unsigned Data

Addition of unsigned numbers

The form of the ADD instruction is

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

The instructions ADD and ADC are used to add two operands. The destination operand must be a register. The Op2 operand can be a register or immediate. Remember that memory-to-register or memory-to-memory arithmetic and logic operations are never allowed in ARM Assembly language since it is a RISC processor. The instruction could change any of the Z, C, N, or V bits of the status flag register, as long as we use the ADDS instead of ADD. The effect of the ADDS instruction on the overflow (V) and N (negative) flags is discussed in Chapter 5 since they are used in signed number operations. Look at Examples 3-1 and 3-2 for the effect of ADDS instruction on Z and C flags.

Example 3-1

Show the flag bits of status register for the following cases:

a)
```
LDR     R2,=0xFFFFFFF5          ;R2=0xFFFFFFF5 (notice the = sign)
MOV     R3,#0x0B
ADDS    R1,R2,R3                ;R1=R2 + R3 and update the flags
```

b)
```
LDR     R2,=0xFFFFFFFF
ADDS    R1,R2,#0x95             ;R1=R2 + 95 and update the flags
```

Solution:

a)

0xFFFFFFF5	1111 1111 1111 1111 1111 1111 1111 0101

```
          +        + 0000 0000 0000 0000 0000 0000
 0x0000000B                              0000 1011

 0x100000000       1  0000 0000 0000 0000 0000 0000
                                         0000 0000
```

First, notice how the “LDR R2,=0xFFFFFFF5” pseudo-instruction loads the 32-bit value into R2 register. Also notice the use of ADDS instruction instead of ADD since the ADD instruction does not update the flags. Now, after the addition, the R1 register (destination) contains 0 and the flags are as follows:

C = 1, since there is a carry out from D31

Z = 1, the result of the action is zero (for the 32 bits)

b)

```
 0xFFFFFFFF      1111 1111 1111 1111 1111 1111 1111 1111

          +       + 0000 0000 0000 0000 0000 0000 1001
 0x00000095                                       0101

 0x100000094      1  0000 0000 0000 0000 0000 0000 1001
                                                  0100
```

After the addition, the R1 register (destination) contains 0x94 and the flags are as follows:

C = 1, since there is a carry out from D31

Z = 0, the result of the action is not zero (for the 32 bits)

Example 3-2

Show the flag bits of status register for the following case:

```
LDR     R2,=0xFFFFFFF1          ;R2 = 0xFFFFFFF1
MOV     R3,#0x0F
ADDS    R3,R3,R2            ;R3 = R3 + R2 and update the flags
ADD     R3,R3,#0x7          ;R3 = R3 + 0x7 and flags unchanged
MOV     R1,R3
```

Solution:

```
 0xFFFFFFF1      1111 1111 1111 1111 1111 1111 1111 0001
```

```
+  0x0000000F        +  0000 0000 0000 0000 0000 0000 0000 1111

   0x100000000       1  0000 0000 0000 0000 0000 0000 0000 0000
```

After the ADDS addition, the R3 register (destination) contains 0 and the flags are as follows:

C = 1, since there is a carry out from D31

Z = 1, the result of the action is zero (for the 32 bits)

After the "ADD R3,R3,#0x7" addition, the R3 register (destination) contains 0x7 (0x + 07 = 07) and the flags are unchanged from previous instruction since we used ADD instead of ADDS. Therefore, the Z = 1 and C = 1. If we use "ADDS R3,R3,#0x7" instruction instead of "ADD R3,R3,#0x7", we will have Z = 0 and C = 0. Use the Keil ARM simulator to verify this.

Comment

Microsoft Windows comes with a calculator. Use it to verify the calculations in this and future chapters. The calculator supports data size of up to 64-bit

No increment instruction in ARM

There is no increment instruction in the ARM processor. Instead we use ADD to perform this action. The instruction "ADDS R4,R4,#1" will increment the R4 and places the result in R4 register. The RISC processors eliminate the unnecessary instructions and use an existing instruction to perform the operation.

ADC (add with carry)

This instruction is used for multiword (data larger than 32-bit) numbers. The form of the ADC instruction is

```
ADC     Rd,Rn,Op2       ;Rd = Rn + Op2 + C
```

In discussing addition, the following two cases will be examined:

1. Addition of individual word data
2. Addition of multiword data

CASE 1: Addition of individual word data

In previous examples, in programs regarding addition, the total sum was purposely kept less than 0xFFFFFFFF, the maximum value a 32-bit register can hold. In real world to calculate the total sum of any number of operands, the carry flag should be checked after the addition of each operand to see if the total sum is greater than 0xFFFFFFFF. See Example 3-3 and Program 3-1.

Example 3-3

Show the flag bits of status register for the following case:

```
LDR     R2,=0xFFFFFFF1          ;R2 = 0xFFFFFFF1
MOV     R3,#0x0F
ADDS    R3,R3,R2                ;R3 = R3 + R2 and update the flags
ADD     R3,R3,#0x7              ;R3 = R3 + 0x7 and flags unchanged
MOV     R1,R3
```

Solution:

```
  0xFFFFFFF1       1111 1111 1111 1111 1111 1111 1111 0001
+ 0x0000000F     + 0000 0000 0000 0000 0000 0000 0000 1111
  0x100000000    1 0000 0000 0000 0000 0000 0000 0000 0000
```

After the ADDS addition, the R3 register (destination) contains 0 and the flags are as follows:

C = 1, since there is a carry out from D31

Z = 1, the result of the action is zero (for the 32 bits)

After the “ADD R3,R3,#0x7” addition, the R3 register (destination) contains 0x7 (0x0 + 0x7 = 0x7) and the flags are unchanged from previous instruction since we used ADD instead of ADDS. Therefore, the Z = 1 and C = 1. If we use “ADDS R3,R3,#0x7” instruction instead of “ADD R3,R3,#0x7”, we will have Z = 0 and C = 0. Use the Keil ARM simulator to verify this.

Program 3-1

Write a program to calculate the total sum of five words of data. Each data value represents the mass of a planet in integer. The decimal data are as follow: 1000000000, 2000000000, 3000000000, 4000000000, and 4100000000.

```
   AREA    PROG3_1, CODE, READONLY
   ENTRY

   LDR   R1, =1000000000
   LDR   R2, =2000000000
   LDR   R3, =3000000000
   LDR   R4, =4000000000
   LDR   R5, =4100000000

   MOV    R8,#0                 ; R8 = 0 for saving the lower word
   MOV    R9,#0                 ; R9 = 0 for accumulating the carries

   ADDS    R8,R8,R1             ; R8 = R8 + R1
   ADC      R9,R9,#0            ; R9 = R9 + 0 + Carry
   ;(increment R9 if there is carry)
   ADDS    R8,R8,R2             ; R8 = R8 + R2
   ADC      R9,R9,#0            ; R9 = R9 + 0 + Carry
   ADDS    R8,R8,R3             ; R8 = R8 + R3
   ADC      R9,R9,#0            ; R9 = R9 + 0 + Carry
   ADDS    R8,R8,R4             ; R8 = R8 + R4
   ADC      R9,R9,#0            ; R9 = R9 + 0 + Carry
   ADDS    R8,R8,R5             ; R8 = R8 + R5
   ADC      R9,R9,#0            ; R9 = R9 + 0 + Carry
HERE     B          HERE
   END                     ; Mark end of file
```

CASE 2: Addition of multiword numbers

Assume a program is needed that will add the total U.S. budget for the last 100 years or the mass of all the planets in the solar system. In cases like this, the numbers being added could be up to 8 bytes wide or more. Since ARM registers are only 32 bits wide (4 bytes), it is the job of the programmer to write the code to break down these large numbers into smaller chunks to be processed by the CPU. If a 32-bit register is used and the operand is 8 bytes wide, that would take a total of two iterations. See Example 3-4. However, if a 16-bit register is used, the same operands would require four iterations. This obviously takes more time for the CPU. This is one reason to have wide registers in the design of the CPU.

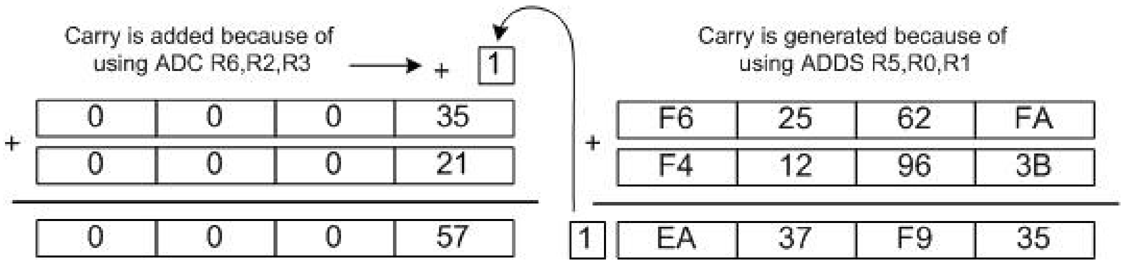
Example 3-4

Analyze the following program which adds 0x35F62562FA to 0x21F412963B:

```
LDR     R0,=0xF62562FA          ;R0 = 0xF62562FA
LDR     R1,=0xF412963B          ;R1 = 0xF412963B
MOV     R2,#0x35          ;R2 = 0x35
MOV     R3,#0x21          ;R3 = 0x21
ADDS    R5,R1,R0          ;R5 = 0xF62562FA + 0xF412963B
;now C = 1
ADC     R6,R2,R3          ;R6 = R2 + R3 + C
;   = 0x35 + 21 + 1 = 0x57
```

Solution:

After the R5 = R0 + R1 the carry flag is one. Since C = 1, when ADC is executed, R6 = R2 + R3 + C = 0x35 + 0x21 + 1 = 0x57.


Carry is added because of using ADC R6,R2,R3
+ 1
+
0 0 0 35
0 0 0 21
0 0 0 57
Carry is generated because of using ADDS R5,R0,R1
+
F6 25 62 FA
F4 12 96 3B
1 EA 37 F9 35

Microsoft Windows calculator support data size of up 64-bit (double word). Use it to verify the above calculations.

Subtraction of unsigned numbers

```
SUB     Rd,Rn,Op2          ;Rd = Rn - Op2
```

In subtraction, the ARM microprocessors (indeed, almost all modern CPUs) use the 2's complement method. Although every CPU contains adder circuitry, it would be too cumbersome (and take too many logic gates) to design separate subtractor circuitry. For this reason, the ARM uses internal adder circuitry to perform the subtraction operation. Assuming that the ARM is executing simple subtract instructions, one can summarize the steps of the hardware of the CPU in executing the SUB instruction for unsigned numbers as follows:

1. Take the 2's complement of the subtrahend (Op2 operand).
2. Add it to the minuend (Rn operand).
3. Place the result in destination Rd.
4. Set the carry flag if there is a carry.

These four steps are performed for every SUBS instruction by the internal hardware of the ARM CPU. It is after these four steps that the result is obtained and the flags are set. Examples 3-5 through 3-7 illustrates the four steps.

Example 3-5

Show the steps involved for the following cases:

a)

```
MOV     R2,#0x4F          ;R2 = 0x4F
MOV     R3,#0x39          ;R3 = 0x39
SUBS    R4,R2,R3          ;R4 = R2 – R3
```

b)

```
MOV     R2,#0x4F          ;R2 = 0x4F
SUBS    R4,R2,#0x05       ;R4 = R2 – 0x05
```

Solution:

a)

```
   0x4F        0000004F
 – 0x39     + FFFFFFC7   2's complement of 0x39
     16     1 00000016   (C = 1 step 4)
```

The flags would be set as follows: C = 1, and Z = 0. The programmer must look at the carry flag (not the sign flag) to determine if the result is positive or negative.

b)

```
 0x4F        0000004F
–0x05    +  FFFFFFFB   2's complement of 0x05
 0x4A      1 0000004A   (C=1 step 4)
```

Example 3-6

Analyze the following instructions:

```
LDR     R2,=0x88888888          ;R2 = 0x88888888
LDR     R3,=0x33333333          ;R3 = 0x33333333
SUBS    R4,R2,R3                ;R4 = R2 – R3
```

Solution:

Following are the steps for "SUB R4,R2,R3":

```
   88888888       88888888
–  33333333     + CCCCCCCD   (2's complement of
                              0x33333333)
   55555555     1 55555555   (C = 1 step 4) result is positive
```

Notice that, unlike x86 CPUs, ARM does not invert the carry flag after SUBS so C=0 when there is borrow and C=1 when there is no borrow. It means that after the execution of SUBS, if C=1, the result is positive; if C = 0, the result is negative and the destination has the 2's complement of the result. Normally, the result is left in 2's complement, but you can take the 2's complement of the result by inverting it and adding one to it.

Example 3-7

Analyze the following instructions:

```
MOV     R1,#0x4C          ;R1 = 0x4C
MOV     R2,#0x6E          ;R2 = 0x6E
SUBS    R0,R1,R2          ;R0 = R1 – R2
```

Solution:

Following are the steps for “SUB R0,R1,R2”:

4C	0000004C	
–6E	+ FFFFFF92	(2’s complement of 0x6E)
– 22	0 FFFFFFDE	(C = 0 step 4) result is negative

SBC (subtract with borrow)

```
SBC       Rd,Rn,Op2         ;Rd = Rn – Op2 – 1 + C
```

This instruction is used for subtraction of multiword (data larger than 32-bit) numbers. Notice that in some other architectures, the CPU inverts the C flag after subtraction so the content of carry flag is the borrow bit of subtract operation. In those architectures the subtract with borrow is implemented as “Rd = Rn – Op2 – C” but in ARM the carry flag is not inverted after subtraction and carry flag is invert of borrow. To invert the carry flag while running the subtract with borrow instruction it is implemented as “Rd = Rn – Op2 – 1 + C” See Example 3-8.

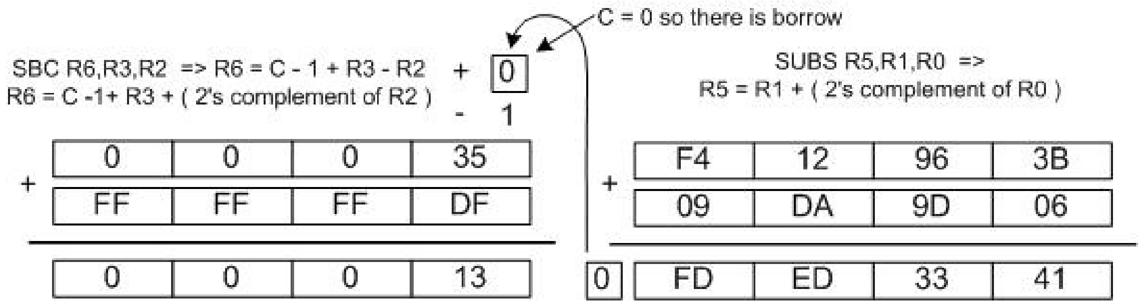
Example 3-8

Analyze the following program which subtracts 0x21F62562FA from 0x35F412963B:

```
LDR       R0,=0xF62562FA           ;R0 = 0xF62562FA,
; notice the syntax for LDR
LDR       R1,=0xF412963B           ;R1 = 0xF412963B
MOV     R2,#0x21             ;R2 = 0x21
MOV     R3,#0x35             ;R3 = 0x35
SUBS     R5,R1,R0             ;R5 = R1 – R0
;   = 0xF412963B – 0xF62562FA, and C = 0
SBC       R6,R3,R2             ;R6 = R3 – R2 – 1 + C
;  = 0x35 – 0x21 – 1 + 0 = 0x13
```

Solution:

After the R5 = R1 – R0 there is a borrow so the carry flag is cleared. Since C = 0, when SBC is executed, R6 = R3 – R2 – 1 + C = 0x35 – 0x21 – 1 + 0 = 0x35 – 0x21 – 1= 0x13.



No decrement instruction in ARM

There is no decrement instruction in the ARM processor. Instead we use SUB to perform the action. The instruction “SUB R4,R4,#1” will decrement one from R4 and places the result in R4 register. The RISC processors eliminate the unnecessary instructions and use an existing instruction to perform the desired operation.

RSB (reverse subtract)

The format for the RSB instruction is

```
RSB     Rd,Rn,Op2      ;Rd = Op2 – Rn
```

Notice the difference between the RSB and SUB instruction. They are essentially the same except the way the source operands are subtracted is reversed. This instruction can be used to get 2’s complement of a 32-bit operand. See Example 3-9.

Example 3-9

Find the result of R0 for the followings:

a)

```
MOV    R1,#0x6E        ;R1=0x6E
RSB    R0,R1,#0        ;R0= 0 – R1
```

b)

```
MOV    R1,#0x1         ;R1=1
RSB    R0,R1,#0        ;R0= 0 – R1 = 0 – 1
```

Solution:

a) Following are the steps for “RSB R0,R1,#0”:

```
  0      0000000
–6E   + FFFFFF92   (2’s complement)
```

```
 –6E       FFFFFF92   (C = 0) result is negative
```

b) This is one way to get a fixed value of 0xFFFFFFFF in a register. Therefore, we have R0=0xFFFFFFFF.

RSC (reverse subtract with carry)

The form of the RSC instruction is

```
RSC     Rd,Rn,Op2       ;Rd = Op2 – Rn – 1 + C
```

Notice the difference between the RSB and RSC instructions. They are essentially the same except the way the source operands are subtracted is reversed. This instruction can be used to get the 2's complement of the 64-bit operand. See Example 3-10.

Example 3-10

Show how to create 2's complement of a 64-bit data in R0 and R1 register. The R0 hold the lower 32-bit.

Solution:

```
LDR     R0,=0xF62562FA          ;R0 = 0xF62562FA
LDR     R1,=0xF812963B          ;R1 = 0xF812963B
RSB     R5,R0,#0           ;R5 = 0 – R0
;  = 0 – 0xF62562FA = 9DA9D06  and C = 0
RSC     R6,R1,#0           ;R6 = 0 – R1 – 1 + C
;  = 0 – 0xF812963B – 1 + 0 = 7ED69C4
```

Use Microsoft Windows calculator to verify the above calculations.

Multiplication and division of unsigned numbers

Not all CPUs have instructions for multiplication and division. All the ARM processors have a multiplication instruction but not the division. Some family members such as ARM Cortex have both the division and multiplication instructions. In this section we examine the multiplication of unsigned numbers. Signed numbers multiplication is treated in Chapter 5.

Multiplication of unsigned numbers in ARM

The ARM gives you two choices of unsigned multiplication: normal multiply and long multiply. The normal multiply instruction (MUL) is used when the result is less than 32-bit, while the long multiply (MULL) must be used when the result is greater than 32-bit. See Table 3-3. In this section we examine both of them.

Instruction	Source 1	Source 2	Destination	Result
MUL	Rn	Op2	Rd (32 bits)	Rd=Rn×Op2
UMULL	Rn	Op2	RdLo,RdHi (64 bits)	RdLo:RdHi=Rn×Op2
Note 1: *Using MUL for word × word multiplication provides only the lower 32-bit result in Rd and the rest are dropped if the result is greater than 32-bit. If the result is greater than 0xFFFFFFFF, then we must use UMULL (unsigned Multiply Long) instruction.* ***Note 2:*** *in word-by-word multiplication using MUL instruction, if the result is greater than 32-bit only the lower 32-bit is saved by ARM and the upper part is dropped without setting any flag. In some CPUs the C flag is used to indicate the result is greater than 32-bit but this is not the case with ARM.*				

Table 3- 3: Unsigned Multiplication (UMUL Rd,Rn,Op2) Summary

MUL (multiply)

```
MUL     Rd,Rn,Op2        ;Rd = Rn × Op2
```

In normal multiplication, the operands must be in registers. After the multiplication, the destination registers will contain the result. See the following example:

```
MOV    R1,#0x25          ;R1=0x25
MOV    R2,#0x65          ;R2=0x65
MUL     R3,R1,R2          ;R3 = R1 × R2 = 0x65 × 0x25
```

Note that in the case of half-word times half-word or smaller sources since the destination register is 32-bit there is no problem in keeping the result of 65,535 × 65,535, the highest possible unsigned 16-bit data. That is not the case in word times word multiplication because 32-bit × 32-bit can produce a result greater than 32-bit. If the MUL instruction is used, the destination register will hold the lower word (32-bit) and the portion beyond 32-bit is dropped. So it is not safe to use MUL for multiplication of numbers greater than 65,536. In many microprocessors when the result goes beyond the destination register size, the C flag is raised to indicate that. This is not the case with ARM. See the following example:

```
LDR      R1,=100000       ;R1=100,000
LDR      R2,=150000       ;R2=150,000
MUL     R3,R2,R1          ;R3 is not 15,000,000,000 because
                 ;it cannot fit in 32 bits.
```

For this reason we must use UMULL (unsigned multiply long) instruction if the result is going to be greater than 0xFFFFFFFF.

UMULL (unsigned multiply long)

```
UMULL  RdLo,RdHi,Rn,Op2
;RdHi:RdLoRd = Rn × Op2
```

In unsigned long multiplication, the operands must be in registers. After the multiplication, the destination registers will contain the result. Notice that the left most register, RdLo in our case, will hold the lower word and the higher portion beyond 32-bit is saved in the second register, RdHi. See the following example:

```
LDR      R1,=0x54000000 ;R1 = 0x54000000
LDR      R2,=0x10000002 ;R2 = 0x10000002
UMULL R3,R4,R2,R1       ;0x54000000 × 0x10000002
;  = 0x054000000A8000000
                        ;R3 = 0xA8000000, the lower 32 bits
             ;R4 = 0x05400000, the higher 32 bits
```

Notice that it is the job of programmer to choose the best type of multiplication depending on the size of operands and the result. See Example 3-11.

Example 3-11

Write a short program to multiply 0xFFFFFFFF by itself.

Solution:

```
MOV     R0,#1            ;R0 = 1
RSB      R1,R0,#0        ;R1 = 0 – R0
; = 0xFFFFFFFF
RSB      R2,R0,#0        ;R2 = 0xFFFFFFFF
UMULL R3,R4,R1,R2
```

Since 0xFFFFFFFF × 0xFFFFFFFF = 0xFFFFFFFE00000001, then R4=0xFFFFFFFE and R3=0x00000001. If we had used MUL instruction, then the 0xFFFFFFFF would have been dropped and only 0x00000001 would have been kept by the destination register.

Multiply and Accumulate Instructions in ARM

In some application such as digital signal processing (DSP) we need to multiply two registers and add the result with another register. The ARM has an instruction to do both jobs in a single instruction. The format of MLA (multiply and add) instruction is as follows:

```
MLA      Rd,Rm,Rs,Rn      ;Rd = Rm × Rs + Rn
```

In multiplication and add, the operands must be in registers. After the multiplication and add, the destination register will contain the result. See the following example:

```
MOV     R1,#100          ;R1 = 100
MOV     R2,#5            ;R2 = 5
MOV     R3,#40              ;R3 = 40
MLA     R4,R1,R2,R3         ;R4 = R1 × R2 + R3 = 100 × 5 + 40 = 540
```

Notice that multiply and add can produce a result greater than 32-bit, if the MLA instruction is used, the destination register will hold the lower word (32-bit) and the portion beyond 32-bit is dropped. For this reason we must use UMLAL (unsigned multiply and add long) instruction if the result is going to be greater than 0xFFFFFFFF. The format of UMLAL instruction is as follows:

```
UMLAL RdLo,RdHi,Rn,Op2        ;RdHi:RdLo = Rn × Op2 + RdHi:RdLo
```

In multiplication and add, the operands must be in registers. Notice that the addend and the high word of the destination use the same registers, the two left most registers in the instruction. It means that the contents of the registers which have the addend will be changed after execution of UMLAL instruction. See the following example:

```
LDR     R1,=0x34000000          ;R1 = 0x34000000
LDR     R2,=0x2000000           ;R2 = 0x2000000
EOR     R3,R3,R3                 ;R3 = 0x00
LDR     R4,=0x00000BBB          ;R4 = 0x00000BBB
UMLAL R4,R3,R2,R1       ;0x34000000×0x2000000+0xBBB
;  = 0x0680000000000BBB
```

Division of unsigned numbers in ARM

Some ARM families do not have an instruction for division of unsigned numbers since it takes too many gates to implement it. We can use SUB instruction to perform the division. In the next chapter, after explaining conditional branches, we will show an example of unsigned division using subtract operation.

Review Questions

1. Explain the difference between ADDS and ADD instructions.
2. The ADC instruction that has the syntax “ADC Rd, Rn, Op2” means ______________.
3. Explain why the Z=0 for the following:

```
MOV     R2,#0x4F
MOV     R4, #0xB1
ADDS    R2,R4,R2
```

4. Explain why the Z=1 for the following:

```
MOV     R2,#0x4F
LDR     R4,=0xFFFFFFB1
```

```
ADDS    R2,R4,R2
```

5. Show how the CPU would subtract 0x05 from 0x43.
6. If C = 1, R2 = 0x95, and R3 = 0x4F prior to the execution of “SBC R2,R2,R3”, what will be the contents of R2 after the subtraction?
7. In unsigned multiplication of “MUL R2,R3,R4”, the product will be placed in register ___________ .
8. In unsigned multiplication of “MUL R1,R2,R4”, the R2 can be maximum of ____________if R4 = 0xFFFFFFFF.

Section 3.2: Logic Instructions

In this section we discuss the logic instructions AND, OR, and Ex-OR in the context of many examples. Just like arithmetic instruction, we must use the S in the instruction syntax if we want to update the flags. If the S syntax is used the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result. The V flag in the CPSR will be unaffected, and the C flag will be set to the carry out from the barrel shifter which is discussed in the next section. See Table 3-4.

Instruction (Flags Unchanged)	Action	Instruction (Flags Changed)	Hexadecimal
AND	ANDing	**ANDS**	Anding and set flags
ORR	ORRing	**ORS**	Oring and set flags
EOR	Exclusive-ORing	**EORS**	Exclusive Oring and set flags
BIC	Bit Clearing	**BICS**	Bit clearing and set flags

Table 3-4: Logic Instructions and Flag Bits

The instruction format of logic instructions in ARM is similar to the format of other data processing instructions. See Figure 3-2.

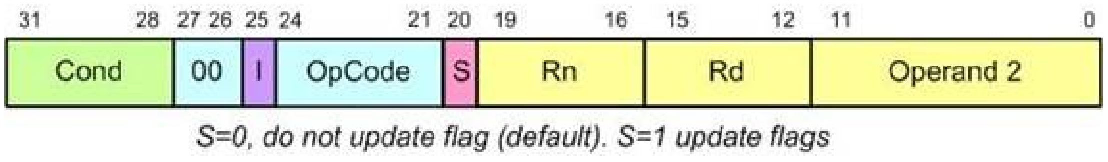
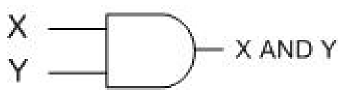


Figure 3-2: General Formation of Data Processing Instruction

AND

AND Rd, Rn, Op2 ;Rd = Rn ANDed Op2

Inputs		Output	Symbol
X	Y	X AND Y	
0	0	0	
0	1	0	
1	0	0	
1	1	1	

This instruction will perform a bitwise logical AND on the operands and place the result in the destination. The destination and the first source operands are registers. The second source operand can be a register or an immediate value of less than 0xFF.

If we use ANDS instead of AND it will change the C and Z flags according to the result. As seen in Example 3-12, AND can be used to mask certain bits of the operand.

Example 3-12

Show the results of the following cases

a)

```
MOV     R1,#0x35
AND      R2,R1,#0x0F       ;R2= R1 ANDed with 0x0F
```

b)

```
MOV     R0,#0x97
MOV     R1,#0xF0
AND      R2,R0,R1           ;R2= R0 ANDed with R1
```

Solution:

a)

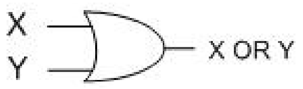
```
0x35      0 0 1 1 0 1 0 1
AND                 0x0F      0 0 0 0 1 1 1 1
0x05      0 0 0 0 0 1 0 1
```

b)

```
0x97      1 0 0 1 0 1 1 1
AND                 0xF0      1 1 1 1 0 0 0 0
0x90      1 0 0 1 0 0 0 0
```

ORR

```
ORR      Rd, Rn, Op2       ;Rd = Rn ORed Op2
```

Inputs		Output	Symbol
X	Y	X OR Y	
0	0	0	
0	1	1	
1	0	1	

1	1	1	

The operands are ORed and the result is placed in the destination. ORR can be used to set certain bits of an operand to one. The destination and the first source operands are registers. The second source operand can be either a register or an immediate value of less than 0xFF.

If we use ORRS instead of ORR, the flags will be updated, just the same as for the ANDS instruction. See Example 3-13.

Example 3-13

Show the results of the following cases:

a)

```
MOV     R1,#0x04            ;R1 = 0x04
ORRS     R2,R1,#0x68        ;R2= R1 ORed 0x68
```

b)

```
MOV     R0,#0x97
MOV     R1,#0xF0
ORR      R2,R0,R1             ;R2= R0 ORed with R1
```

Solution:

a)

```
0x04      0000 0100
OR        0x68      0110 1000          Flag will be:  Z = 0
0x6C      0110 1100
```

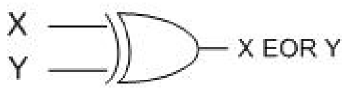
b)

```
0x97      1001 0111
OR        0xF0      1111 0000          Flag will be unchanged
0xF7      1111 0111
```

The ORR instruction can also be used to test for a zero operand. For example, “ORRS R2,R2,#0” will OR the register R2 with zero and make Z = 1 if R2 is zero.

EOR

EOR   Rd,Rn,Op2   ;Rd = Rn Ex-ORed with Op2

Inputs		Output	Symbol
X	Y	X EOR Y	
0	0	0	
0	1	1	
1	0	1	
1	1	0	

The EOR instruction will Exclusive-OR the operands and place the result in the destination register. EOR sets the result bits to 1 if they are not equal; otherwise, they are reset to 0. The flags are updated if we use EORS instead of EOR. The rules for the operands are the same as in the AND and OR instructions. See Examples 3-14 and 3-15.

Example 3-14

Show the results of the following:

MOV   R1,#0x54

EOR   R2,R1,#0x78   ;R2 = R1 ExOred with 0x78

Solution:

0x54   0 1 0 1 0 1 0 0

XOR   0x78   0 1 1 1 1 0 0 0

0x2C   0 0 1 0 1 1 0 0

Example 3-15

The EOR instruction can be used to clear the contents of a register by Ex-ORing it with itself.

Show how “EORS R1,R1,R1” clears R1, assuming that R1 = 0x45.

Solution:

0x45   0 1 0 0 0 1 0 1

```
XOR            0x45    0 1 0 0 0 1 0 1
0x00    0 0 0 0 0 0 0 0
```

EOR can also be used to see if two registers have the same value. "EORS R2,R3,R4" will make Z = 1 if both registers R4 and R3 have the same value, and if they do, the result (00000000) is saved in R2, the destination.

Another widely used application of EOR is to toggle bits of an operand. For example, to toggle bit 2 of register R2:

```
EOR     R2,R2,#0x04     ;EOR R2 with 0000 0100
```

This would cause bit 2 of R2 to change to the opposite value; all other bits would remain unchanged.

BIC (bit clear)

```
BIC     Rd,Rn,Op2       ;clear certain bits of Rn specified by
;the Op2 and place the result in Rd
```

Inputs		Output
X	Y	X AND (NOT Y)
0	0	0
0	1	0
1	0	1
1	1	0

The BIC (bit clear) instruction is used to clear the selected bits of the Rn register. The selected bits are held by Op2. The bits that are HIGH in Op2 will be cleared and bits with LOW will be left unchanged. For example, assuming that R3 = 00001000 the instruction "BIC R2,R2,R3" will clear bit 3 of R2 and leaves the rest of the bits unchanged. In reality, the BIC instruction performs AND operation on Rn register with the complement of Op2 and places the result in destination register. Look at the following example:

```
MOV     R1,#0x0F
MOV     R2,#0xAA
BIC     R3,R2,R1   ;now R3 = 0xAA ANDed with 0xF0 = 0xA0
```

If we want the flags to be updated, then we must use BICS instead of BIC.

MVN (move negative)

```
MVN     Rd, Rn   ;move negative of Rn to Rd
```

The MVN (move negative) instruction is used to generate one's complement of an operand. For example, the instruction "MVN R2,#0" will make R2=0xFFFFFFFF. Look at the following example:

```
LDR      R2,=0xAAAAAAAA        ;R2 = 0xAAAAAAAA
MVN     R2,R2                  ;R2 = 0x55555555
```

We can also use Ex-OR instruction to generate one's complement of an operand. Ex-ORing an operand with 0xFFFFFFFF will generate the 1's complement. See the following code:

```
LDR      R2,=0xAAAAAAAA        ;R2 = 0xAAAAAAAA
MVN     R0,#0                  ;R0 = 0xFFFFFFFF
EOR      R2,R2,R0               ;R2 = R2 ExORed with 0xFFFFFFFF
;  = 0x55555555
```

It must be noted that the instruction "MVN Rd,#0" is widely used to load the fixed value of 0xFFFFFFFF into destination register. We can use the "LDR Rd,=0xFFFFFFFF" pseudo-instruction to do the same thing, but the ARM assembler will substitute several real ARM instructions in its place and therefore it takes more code space.

Review Questions

1. Use operands 0x4FCA and 0xC237 to perform:

(a) AND (b) OR (c) XOR

2. ANDing a word operand with 0xFFFFFFFF will result in what value for the word operand? To set all bits of an operand to 0, it should be ANDed with ______.
3. To set all bits of an operand to 1, it could be ORed with _______.
4. XORing an operand with itself results in what value for the operand?
5. Write an instruction that sets bit 4 of R7.
6. Write an instruction that clears bit 3 of R5.

Section 3.3: Rotate and Barrel Shifter

Although ARM Cortex has shift and rotate instruction, for the ARM7 we can perform the shift and rotate operations as part of other instructions such as MOV. In previous sections we discussed that as the second argument of process instructions (arithmetic and logic instructions) we can use register or immediate values. In other words, the process instructions can be used in one of the following forms:

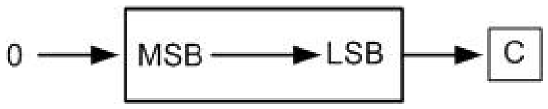
1. opcode Rd, Rn, Rs (e.g. ADD R1,R2,R3)
2. opcode Rd, Rn, immediateValue (e.g. ADD R2,R3,#5)

ARM is able to shift or rotate the second argument before using it as the argument. In this section we first discuss shifting and rotating on registers and then we cover shifting immediate values.

Barrel Shifter

There are two kinds of shifts: logical and arithmetic. The logical shift is for unsigned operands and the arithmetic shift is for signed operands. Logical shift will be discussed in this section and the discussion of arithmetic shift is covered in Chapter 5. Using MOV instructions in ARM one can shift the contents of a register right or left. The number of times (or bits) that the operand is shifted can be specified directly or through a register.

LSR



This is the logical shift right. The operand is shifted right bit by bit, and for every shift the LSB (least significant bit) will go to the carry flag (C) and the MSB (most significant bit) is filled with 0. One can use an immediate operand or a register to hold the number of times it is to be shifted. Examples 3-16 and 3-17 should help to clarify LSR.

Example 3-16

Show the result of LSR in the following:

```
MOV     R0,#0x9A           ;R0 = 0x9A
MOVS   R1,R0,LSR #3       ;shift R0 to right 3 times
;and then move (copy) the result to R1
```

Solution:

```
   0x9A =  00000000 00000000 0000000 00000000 10011010
first shift:        00000000 00000000 0000000 00000000 01001101  C = 0
second shift:     00000000 00000000 0000000 00000000 00100110  C = 1
third shift:       00000000 00000000 0000000 00000000 00010011  C = 0
```

After shifting right three times, R1 = 0x00000013 and C = 0. Another way to write the above code is:

```
    MOV     R0,#0x9A
    MOV     R2,#0x03
    MOV     R1,R0,LSR R2      ;shift R0 to right R2 times
    ;and move the result to R1
```

Example 3-17

Show the results of LSR in the following:

```
TIMES   EQU      0x4
    LDR       R1,=0x777          ;R1=0x777
    MOV     R2,#TIMES         ;R2=0x04
    MOVS   R3,R1,LSR R2      ;shift R1 right R2 number of times
    ;and place the result in R3
```

Solution:

After four shifts, the R3 will contain 0x77. The four LSBs are lost through the carry, one by one, and 0s fill the four MSBs.

Although LSR does affect the S and Z flags, they are not important in this case. But notice that if you want the flags to be set, use the MOVS instruction instead of MOV.

One can use the LSR to divide a number by 2. See Example 3-18.

Example 3-18

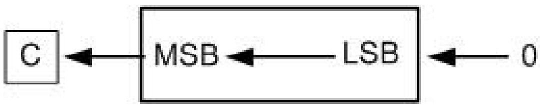
Show the results of LSR in the following:

```
    LDR       R0,=0x88              ;R0=0x88
    MOVS   R1,R0,LSR #3      ;shift R0 right three times (R1 = 0x11)
```

Solution:

After the three shifts, the R1 will contain 0x11. This divides the number by 8 since 2 to the power 3 is 8.

LSL



Shift left is also a logical shift. It is the reverse of LSR. After every shift, the LSB is filled with 0 and the MSB goes to C. All the rules are the same as for LSR. One can use an immediate operand or a register to hold the number of times it is to be shifted left. See Example 3-19. One can use the LSL to multiply a number by 2. See Example 3-20.

Example 3-19

Show the effects of LSL in the following:

```
    LDR     R1,=0x0F000006
    MOVS    R2,R1,LSL #8
```

Solution:

```
      00001111 00000000 00000000 00000110
C=0   00011110 00000000 00000000 00001100 (shifted left once)
C=0   00111100 00000000 00000000 00011000
C=0   01111000 00000000 00000000 00110000
C=0   11110000 00000000 00000000 01100000
C=1   11100000 00000000 00000000 11000000
C=1   11000000 00000000 00000001 10000000
C=1   10000000 00000000 00000011 00000000
C=1   00000000 00000000 00000110 00000000 (shifted eight times)
```

After eight shifts left, the R2 register has 0x00000600 and C = 1. The eight MSBs are lost through the carry, one by one, and 0s fill the eight LSBs. Another way to write the above code is:

```
    LDR     R1,=0x0F000006
    MOV     R0,#0x08
    MOV     R2,R1,LSL R0
```

Example 3-20

Show the results of LSL in the following:

```
TIMES   EQU      0x5
    LDR       R1,#0x7            ;R1=0x7
    MOV     R2,#TIMES         ;R2=0x05
    MOV     R1,R1,LSL R2       ;shift R1 left R2 number of times
    ;and place the result in R1
```

Solution:

After the five shifts, the R1 will contain 0x000000E0. 0xE0 is 224 in decimal. Notice that it multiplies number by power of 2. 7×32 = 224 = 0xE0 since 2 to the power of 5 is 32.

Table 3-5 lists the logical shift operations in ARM.

Operation	Destination	Source	Number of shifts
LSR (Shift Right)	Rd	Rn	Immediate value
LSR (Shift Right)	Rd	Rn	register Rm
LSL (Shift Left)	Rd	Rn	Immediate value
LSL (Shift Left)	Rd	Rn	register Rm
Note: Number of shift cannot be more than 32.			

Table 3- 5: Logic Shift operations for unsigned numbers in ARM

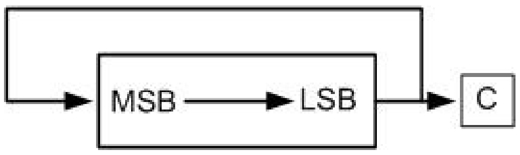
Arithmetic shift right ASR

The arithmetic shift ASR is used for signed numbers and is discussed in Chapter 5.

Rotating the bits of an operand right and left

There are two types of rotations. One is a simple rotation of the bits of the operand, and the other is a rotation through the carry. Each is explained below.

ROR (rotate right)



In rotate right, as bits are shifted from left to right they exit from the right end (LSB) and enter the left end (MSB). In addition, as each bit exits the LSB, a copy of it is given to the carry flag. In other words, in ROR the LSB is moved to the MSB and is also copied to C flag, as shown in the diagram. One can use an immediate operand or a register to hold the number of times it is to be rotated.

```
MOV     R1,#0x36
;R1 = 0000 0000 0000 0000 0000 0000 0011 0110
MOVS   R1,R1,ROR #1
;R1 = 0000 0000 0000 0000 0000 0000 0001 1011  C=0
MOVS   R1,R1,ROR #1
;R1 = 1000 0000 0000 0000 0000 0000 0000 1101  C=1
MOVS   R1,R1,ROR #1
;R1 = 1100 0000 0000 0000 0000 0000 0000 0110  C=1
```

or:

```
MOV     R1,#0x36
;R1 = 0000 0000 0000 0000 0000 0000 0011 0110
MOV     R0,#3
;R0 = 3 number of times to rotate
MOVS   R1,R1,ROR R0
;R1 = 1100 0000 0000 0000 0000 0000 0000 0110 C=1
```

also look at the following case:

```
LDR       R2,=0xC7E5
;R2 = 0000 0000 0000 0000 1100 0111 1110 0101
MOV     R4,#0x06
;R4 = 6 number of times to rotate
MOVS   R3,R2,ROR R4
;R3 = 1001 0100 0000 0000 0000 0011 0001 1111 C=1
```

Rotate left

There is no rotate left option in ARM7 since one can use the rotate right (ROR) to do the job. That means instead of rotating left n bits we can use rotate right 32–n bits to do the job of rotate left. Using this method does not give us the proper carry if actual instruction of ROL was available. Look at the following examples:

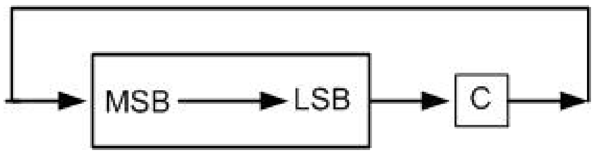
```
LDR       R0,=0x00000072
```

```
;R0 = 0000 0000 0000 0000 0000 0000 0111 0010
MOVS   R0,R0,ROR #31
;R0 = 0000 0000 0000 0000 0000 0000 1110 0100  C=0
MOVS   R0,R0,ROR #31
;R0 = 0000 0000 0000 0000 0000 0001 1100 1000  C=0
MOVS   R0,R0,ROR #31
;R0 = 0000 0000 0000 0000 0000 0011 1001 0000  C=0
MOVS   R0,R0,ROR #31
;R0 = 0000 0000 0000 0000 0000 0111 0010 0000  C=0
```

or:

```
MOV     R0,#0x72           ;R0 = 0111 0010
MOV     R1,#28             ;R1 = 32 – 4 = 28
MOVS   R0,R0,ROR R1        ;R0 = 0111 0010 0000  C=0
;assume R2 = 0x672A
LDR       R2,=0x671A
;R2 = 0000 0000 0000 0000 0110 0111 0010 1010
MOVS   R2,R2,ROR #27
;R2 = 0000 0000 0000 1100 1110 0101 0100 0000
;C = 0
```

RRX rotate right through carry



In RRX, as bits are shifted from left to right, they exit from the right end (LSB) to the carry flag, and the carry flag enters the left end (MSB). In other words, in RRX the LSB is moved to C and C is moved to the MSB. In reality, C flag acts as if it is part of the operand. That means the RRX is like 33-bit register since the C flag is 33th bit. The RRX takes no arguments and the number of times an operand to be rotated is fixed at one.

```
;assume C=0
MOV     R2,#0x26
;R2 = 0000 0000 0000 0000 0000 0000 0010 0110
MOVS   R2,R2,RRX
;R2 = 0000 0000 0000 0000 0000 0000 0001 0011 C=0
```

```
MOVS   R2,R2,RRX
;R2 = 0000 0000 0000 0000 0000 0000 0000 1001 C=1
MOVS   R2,R2,RRX
;R2 = 1000 0000 0000 0000 0000 0000 0000 0100 C=1
MOV    R2,#0x0F
;R2 = 0000 0000 0000 0000 0000 0000 0000 1111
MOVS   R2,R2,RRX
;R2 = 0000 0000 0000 0000 0000 0000 0000 0111 C=1
MOVS   R2,R2,RRX
;R2 = 1000 0000 0000 0000 0000 0000 0000 0011 C=1
MOVS   R2,R2,RRX
;R2 = 1100 0000 0000 0000 0000 0000 0000 0001 C=1
```

Table 3-6 lists the rotate instructions of the ARM.

Operation	Destination	Source	Number of Rotates
ROR (Rotate Right)	Rd	Rn	Immediate value
ROR (Rotate Right)	Rd	Rn	register Rm
RRX (Rotate Right Through Carry)	Rd	Rn	1 bit

Table 3- 6: Rotate operations for unsigned numbers in ARM

Shifting Immediate Arguments

We just examined the rotate and shift operations. One can use the rotate operation to load fixed (constant) values into ARM register, as well. Examine the MOV instruction bit assignment in Figure 3-3. Of the 32-bit opcode, the upper 12 bits (D31–D20) are used for the opcode itself. The lowest 8 (D7–D0) bits are used for fixed values and 4 bits (D11–D8) are used for the number of times rotate operation is performed before loaded into the register. The 8 bits for the fixed value gives us 00 to 255 (00 to 0xFF in hex) range and the 4 bits of the rotate number gives us 0 to 15 (0 to 0xF in hex) range. The MOV instructions can be constructed to rotate an 8-bit fixed value to the right a number of times and then loaded into the register. The number of times rotated right is always twice the number in the rotate portion of the instruction. Since rotate value can be 0–15 that gives number of rotations between 0–30. This means that whenever the second operand is an immediate value the number of rotation is an even number.

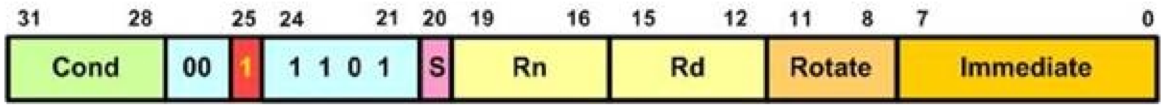


Figure 3- 3: MOV Instruction

As we mentioned earlier, the ARM supports the rotate right only and there is no rotate left option. So to do rotate left we must use 32-n for the number of rotation right. Therefore, "MOV R0,#0xFF,#30" is the same as rotating left 2 times and "MOV R0,#0xFF,#28" is the same as rotating left 4 times. See Examples 3-21 through 3-23.

Example 3-21

Show all the possible cases of using MOV instruction for 0xFF value and rotate options.

Solution:

```
MOV     R0, #0xFF, #0
;R0 = 0xFF is rotated right 0 times. R0 = 0x000000FF
MOV     R0, #0xFF, #2
;R0 = 0xFF is rotated right 2 times. R0 = 0xC000003F
MOV     R0, #0xFF, #4
;R0 = 0xFF is rotated right 4 times. R0 = 0xF000000F
MOV     R0, #0xFF, #6
;R0 = 0xFF is rotated right 6 times. R0 = 0xFC000003
MOV     R0, #0xFF, #8
;R0 = 0xFF is rotated right 8 times. R0 = 0xFF000000
MOV     R0, #0xFF, #10
;R0 = 0xFF is rotated right 10 times. R0 = 0x3FC00000
MOV     R0, #0xFF, #12
;R0 = 0xFF is rotated right 12 times. R0 = 0x0FF00000
MOV     R0, #0xFF, #14
;R0 = 0xFF is rotated right 14 times. R0 = 0x03FC0000
MOV     R0, #0xFF, #16
;R0 = 0xFF is rotated right 16 times. R0 = 0x00FF0000
MOV     R0, #0xFF, #18
;R0 = 0xFF is rotated right 18 times. R0 = 0x003FC000
MOV     R0, #0xFF, #20
;R0 = 0xFF is rotated right 20 times. R0 = 0x000FF000
```

```
MOV     R0, #0xFF, #22
;R0 = 0xFF is rotated right 22 times. R0 = 0x0003FC00
MOV     R0, #0xFF, #24
;R0 = 0xFF is rotated right 24 times. R0 = 0x0000FF00
MOV     R0, #0xFF, #26
;R0 = 0xFF is rotated right 26 times. R0 = 0x00003FC0
MOV     R0, #0xFF, #28
;R0 = 0xFF is rotated right 28 times. R0 = 0x00000FF0
MOV     R0, #0xFF, #30
;R0 = 0xFF is rotated right 30 times. R0 = 0x000003FC
```

Example 3-22

Using MOV instruction, show how to rotate left the fixed value of 0x99 total of (a) 4, (b) 8, and (c) 16 times. Also give the value in the register after the rotation.

Solution:

Since we do not have rotate left operation we must use rotate right 32–n times.

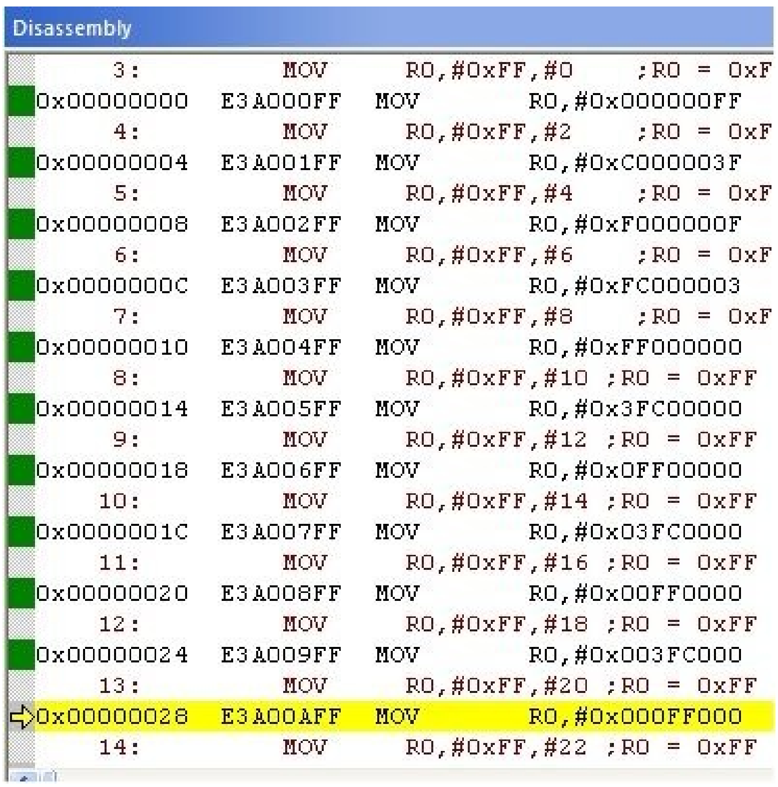
```
MOV     R1,#0x99,#28
;rotating right 28 times is the same as rotate left 4 times
MOV     R2,#0x99,#24
;rotating right 24 times is the same as rotate left 8 times
MOV     R3,#0x99,#16
;rotating right 16 times is the same as rotate left 16 times
```

Now, we have (a) R1 = 0x00000990, (b) R2 = 0x00009900, and (c) R3=0x00990000

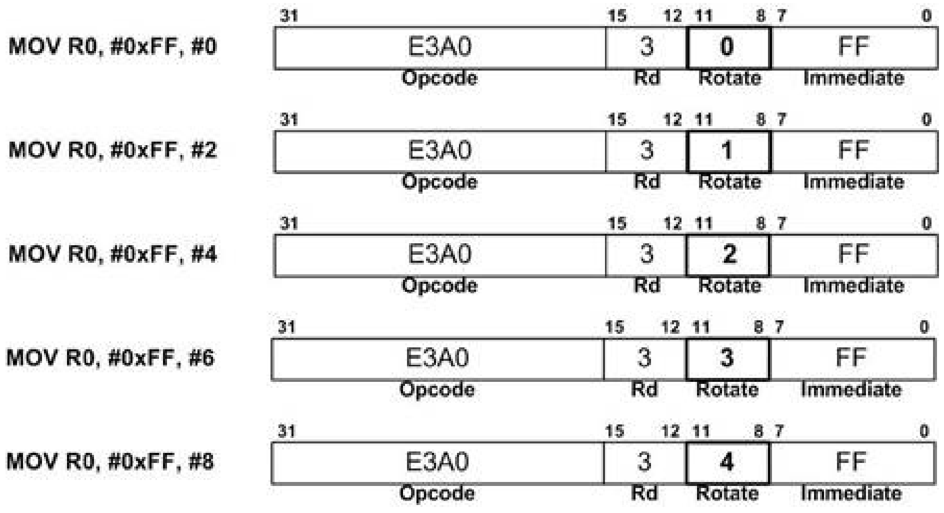
Example 3-23

Using the Keil IDE, assemble the program in Example 3-21 and examine the list file. Compare and contrast the count value in the instruction with count value of the opcodes.

Solution:



```
Disassembly
     3:           MOV    R0,#0xFF,#0    ;R0 = 0xF
0x00000000  E3A000FF  MOV        R0,#0x000000FF
     4:           MOV    R0,#0xFF,#2    ;R0 = 0xF
0x00000004  E3A001FF  MOV        R0,#0xC000003F
     5:           MOV    R0,#0xFF,#4    ;R0 = 0xF
0x00000008  E3A002FF  MOV        R0,#0xF000000F
     6:           MOV    R0,#0xFF,#6    ;R0 = 0xF
0x0000000C  E3A003FF  MOV        R0,#0xFC000003
     7:           MOV    R0,#0xFF,#8    ;R0 = 0xF
0x00000010  E3A004FF  MOV        R0,#0xFF000000
     8:           MOV    R0,#0xFF,#10 ;R0 = 0xFF
0x00000014  E3A005FF  MOV        R0,#0x3FC00000
     9:           MOV    R0,#0xFF,#12 ;R0 = 0xFF
0x00000018  E3A006FF  MOV        R0,#0x0FF00000
    10:           MOV    R0,#0xFF,#14 ;R0 = 0xFF
0x0000001C  E3A007FF  MOV        R0,#0x03FC0000
    11:           MOV    R0,#0xFF,#16 ;R0 = 0xFF
0x00000020  E3A008FF  MOV        R0,#0x00FF0000
    12:           MOV    R0,#0xFF,#18 ;R0 = 0xFF
0x00000024  E3A009FF  MOV        R0,#0x003FC000
    13:           MOV    R0,#0xFF,#20 ;R0 = 0xFF
0x00000028  E3A00AFF  MOV        R0,#0x000FF000
    14:           MOV    R0,#0xFF,#22 ;R0 = 0xFF
```



As expected, the number of times rotated right is twice the number of rotate field.

Also see Example 3-24 for further examples of rotate operation.

Example 3-24

Give the register value for each of the following instructions after it is executed.

```
MOV     R0,#0xAA,#2
MOV     R1,#0x20,#28
MOV     R4,#0x99,#6
MOV     R2,#0x55,#24
MOV     R3,#0x01,#20
MOV     R7,#0x80,#12
MOV     R10,#0x0F,#14
MOV     R5,#0x66,#2
```

Solution:

```
MOV     R0,#0xAA,#2
;R0 = 0xAA is rotated right 2 times. R0 = 0x8000002A
MOV     R1,#0x20,#28
;R1 = 0x20 is rotated right 28 times. R0 = 0x00000200
MOV     R4,#0x99,#6
;R4 = 0x99 is rotated right 6 times. R0 = 0x64000002
MOV     R2,#0x55,#24
;R2 = 0x55 is rotated right 24 times. R0 = 0x00005500
MOV     R3,#0x01,#20
;R3 = 0x01 is rotated right 20 times. R0 = 0x00001000
MOV     R7,#0x80,#12
;R7 = 0x80 is rotated right 12 times. R0 = 0x08000000
MOV     R10,#0x0F,#14
;R10 = 0x0F is rotated right 14 times. R0 = 0x003C0000
MOV     R5,#0x66,#2
;R5 = 0x66 is rotated right 2 times. R0 = 0x80000019
```

We can use the concept of rotate with other data processing instructions of the ARM to load fixed values into the ARM register. See Example 3-25.

Example 3-25

Give the register value for each of the following instructions after it is executed.

```
    MVN     R0,#0xAA,#2
    MVN     R1,#0x20,#28
    MVN     R4,#0x99,#6
    MVN     R2,#0x55,#24
    MVN     R3,#0x01,#20
    MVN     R7,#0x80,#12
    MVN     R10,#0x0F,#14
    MVN     R5,#0x66,#2
```

Solution:

```
    MVN     R0,#0xAA,#2
    ;0xAA rotated right 2 times = 0x8000002A; R0 = 0x7FFFFFD5
    MVN     R1,#0x20,#28
    ;0x20 is rotated right 28 times = 0x00000200; R1 = 0xFFFFFDFF
    MVN     R4,#0x99,#6
    ;0x99 is rotated right 6 times  = 0x64000002; R4 = 0x9BFFFFFD
    MVN     R2,#0x55,#24
    ;0x55 is rotated right 24 times = 0x00001A40; R2 = 0xFFFFAAFF
    MVN     R3,#0x01,#20
    ;0x01 is rotated right 20 times = 0x00001000; R3 = 0xFFFFEFFF
    MVN     R7,#0x80,#12
    ;0x80 is rotated right 12 times = 0x08000000; R7 = 0xF7FFFFFF
    MVN     R10,#0x0F,#14
    ;0x0F is rotated right 14 times = 0x003C0000; R10=0xFFC3FFFF
    MVN     R5,#0x66,#2
    ;0x66 is rotated right 2 times  = 0x80000019; R5 = 0x7FFFFFE6
```

General Formation of Process instruction

Next we will show how the different operands are supported by ARM. In Figure 3-4 you see the general formation of process instructions.

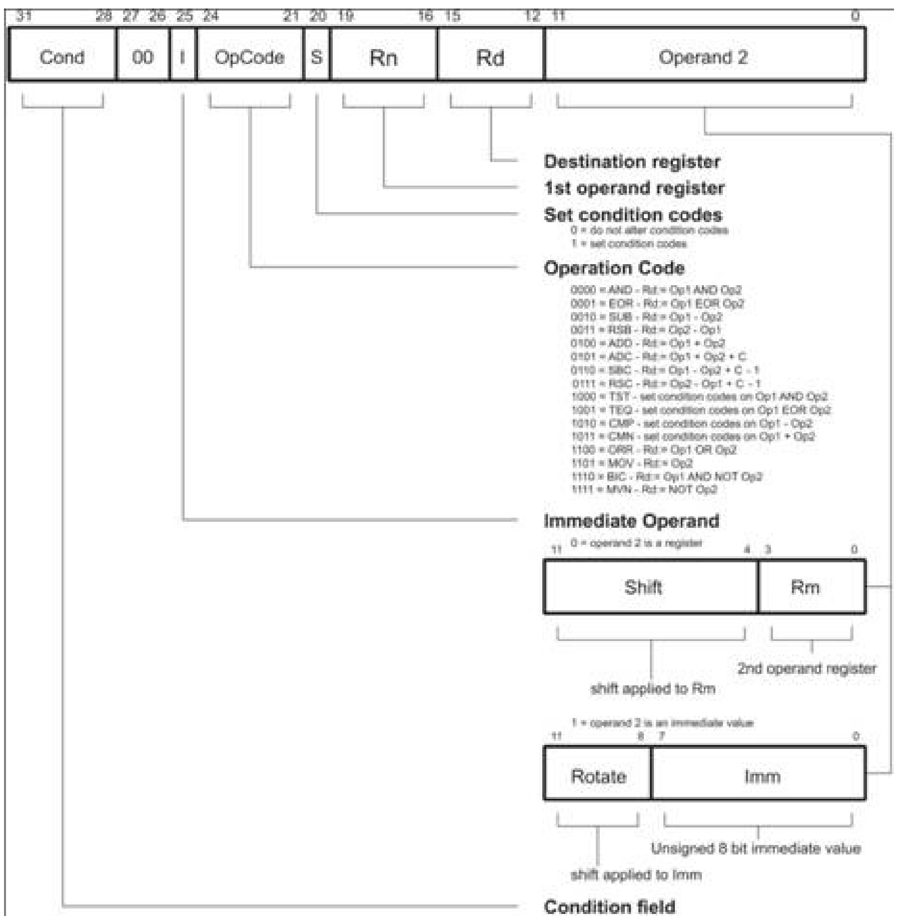

31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

Figure 3- 4: Data Process Instructions

In all ARM instructions bits 28–31 are put aside for condition field which is covered in Chapter 4.

Bits 26 and 27 are 0 in process instructions showing that the instruction is a process instruction and the opcode is represented by bits 24–21. Using 4 bits 16 different instructions are provided as shown in Figure 3-4.

The S bit (bit 20) shows if the flags should be updated. When we add an S (e.g. MOVS) the bit will be set which shows that the flags should be updated by the CPU.

Bits 12–15 contain the destination register. Using 4 bits we can select registers R0–R15.

Bits 16–19 represent the first operand register.

Bit 25 (I) shows the type of second operand. The bit is 1 when the second operand is an immediate value. Whenever the second operand is a register this bit is zero.

Immediate values: In the case that I is 1, bits 0–7, contain an immediate value which can be a number between 0–0xFF.

Second register: In the case that I is 0, bits 0–11 represent the second operand register, together with the amount of shift/rotate and type of shift/rotate. As mentioned earlier the shift amount can be provided either by a register or an immediate value. Whenever the I bit is cleared, bit 4 shows the way shift amount is provided. See Figure 3-5.

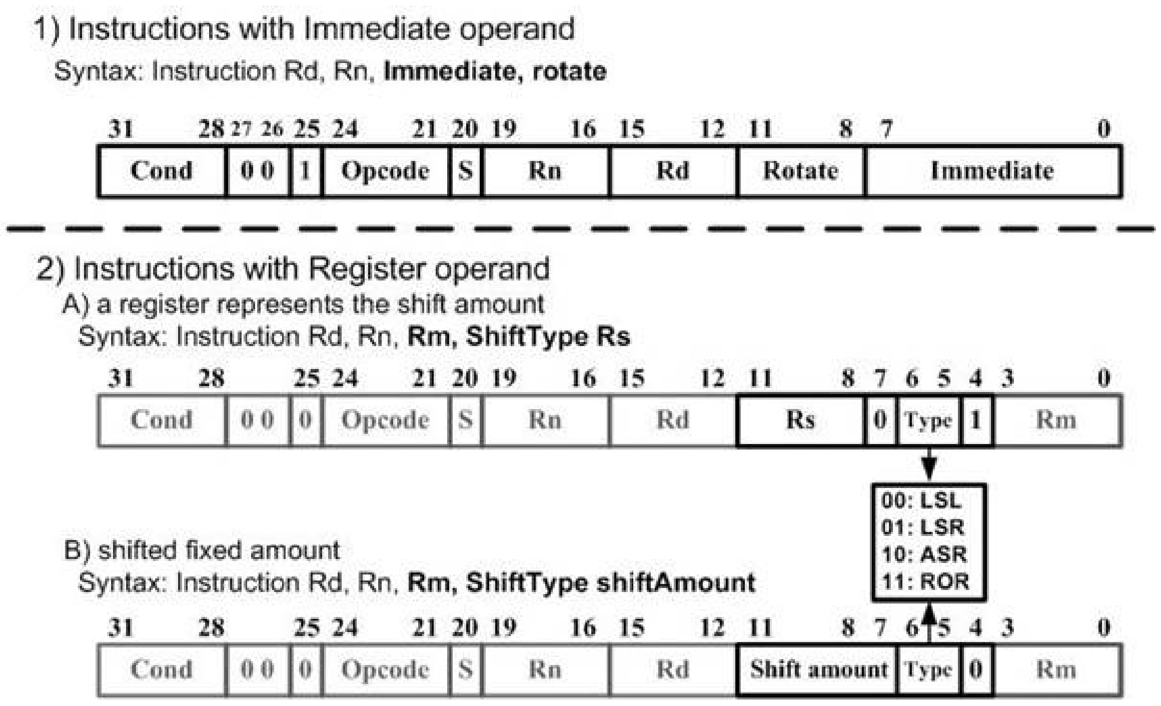


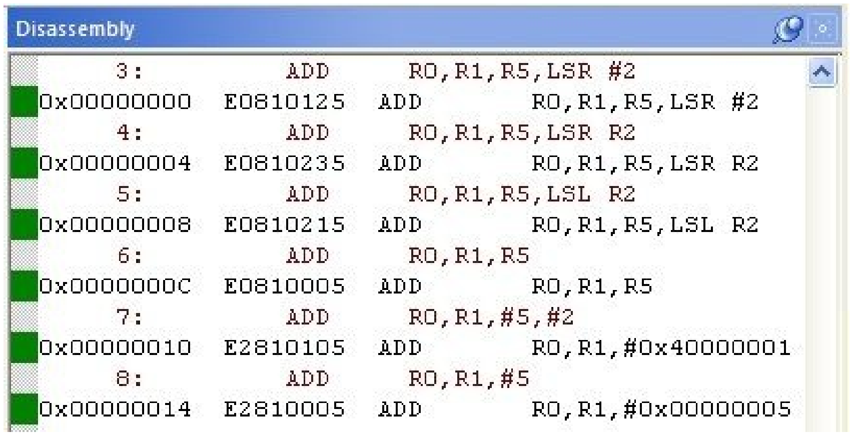
Figure 3- 5: Data Process Instructions

Example 3-26 should help to clarify this.

Example 3-26

Using the Keil IDE, assemble the following program and compare the instruction with the process instruction format.

```
    AREA EX3_26,CODE,READONLY
    ENTRY
    ADD      R0,R1,R5,LSR #2
    ADD      R0,R1,R5,LSR R2
    ADD      R0,R1,R5,LSL R2
    ADD      R0,R1,R5
    ADD      R0,R1,#5,#2
    ADD      R0,R1,#5
H1       B          H1
    END
```



```
      3:         ADD     R0,R1,R5,LSR #2
0x00000000  E0810125  ADD       R0,R1,R5,LSR #2
      4:         ADD     R0,R1,R5,LSR R2
0x00000004  E0810235  ADD       R0,R1,R5,LSR R2
      5:         ADD     R0,R1,R5,LSL R2
0x00000008  E0810215  ADD       R0,R1,R5,LSL R2
      6:         ADD     R0,R1,R5
0x0000000C  E0810005  ADD       R0,R1,R5
      7:         ADD     R0,R1,#5,#2
0x00000010  E2810105  ADD       R0,R1,#0x40000001
      8:         ADD     R0,R1,#5
0x00000014  E2810005  ADD       R0,R1,#0x00000005
```

Solution:

Condition		I	Opcode	S	Rn	Rd	Shift amount	Shift Type	Arg. Type	Rm
1 1 1 0	0 0	0	0 1 0 0	0	0 0 0 1	0 0 0 0	0 0 0 1 0	0 1	0	0 1 0 1
			ADD		R1	R0	2 bits	LSR		R5

In "ADD R0,R1,R5,LSR #2" the second operand is a register. Therefore, the I bit is cleared. Since the shift amount is an immediate value, the bit 4 is cleared, as well. The shift type is set to 01 representing LSR.

Condition		I	Opcode	S	Rn	Rd	Rs		Shift Type	Arg. Type	Rm
1 1 1 0	0 0	0	0 1 0 0	0	0 0 0 1	0 0 0 0	0 0 1 0	0	0 1	1	0 1 0 1
			ADD		R1	R0	R2		LSR		R5

In "ADD R0,R1,R5,LSR R2" the second operand is a register. Therefore, the I bit is cleared. As a register provides the shift amount, the bit 4 is set.

Condition		I	Opcode	S	Rn	Rd	Rs		Shift Type	Arg. Type	Rm
1 1 1 0	0 0	0	0 1 0 0	0	0 0 0 1	0 0 0 0	0 0 1 0	0	0 0	1	0 1 0 1
			ADD		R1	R0	R2		LSL		R5

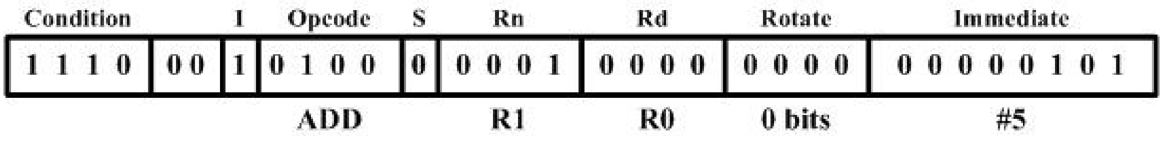
The "ADD R0,R1,R5,LSL R2" instruction is the same as "ADD R0,R1,R5,LSR R2" except that the shift type is set to 00 to represent LSL.

Condition		I	Opcode	S	Rn	Rd	Shift amount	Shift Type	Arg. Type	Rm
1 1 1 0	0 0	0	0 1 0 0	0	0 0 0 1	0 0 0 0	0 0 0 0 0	0 0	0	0 1 0 1
			ADD		R1	R0	0 bits	LSL		R5

The machine code represents in fact the instruction "ADD R0,R1,R5,LSL #0". But, if we shift a number 0 bits to left, the number remains unchanged. As a result, it represents "ADD R0,R1,R5".

Condition		I	Opcode	S	Rn	Rd	Rotate	Immediate
1 1 1 0	0 0	1	0 1 0 0	0	0 0 0 1	0 0 0 0	0 0 0 1	0 0 0 0 0 1 0 1
			ADD		R1	R0	2 bits	#5

In “ADD R0,R1,#5,#2” the second operand is immediate. Therefore, the I bit is set. In immediate operands the value is shifted twice the rotate field. That is why the rotate field is 1.



In “ADD R0,R1,#5”, the immediate value is rotated 0 bits. Therefore, the immediate value remains unchanged.

Loading fixed values into Registers

In this section, we examined loading fixed values into registers. However, as we have seen so far, it is not possible to create every constant value by using the rotate option of the instructions.

Another way is to use combination of these instructions such as MVN, MOV, ADD, AND, OR, and so on to load any value into a register. For example the following program loads R1 with 0x87654321:

```
MOV     R1,#0x21              ;R1 = 0x00000021
ORR      R1,R1,#0x43,#24
;R1 = 0x00000021 ORed 0x00004300 = 0x00004321
ORR      R1, R1,#0x65,#16
;R1 = 0x00004321 ORed 0x00650000 = 0x00654321
ORR      R1, R1,#0x87,#8
;R1 = 0x00654321 ORed 0x87000000 = 0x87654321
```

But it is very tedious and time consuming to come up with such combination of instructions to get the result we need. For this reason the ARM assembler has pseudo-instruction such as LDR. The LDR is not a real instruction like MOV and ADD since you will not find the opcode for it in the ARM manual. When the ARM assembler assembles a program with the LDR it replaces the LDR with a group of instructions to do the job of loading a fixed value in the most efficient way possible. As we have seen in previous chapters, the LDR uses = sign for the immediate value instead of # used by the MOV instruction. Indeed this is the way we distinguish between them. Examine the following codes to see the differences in syntaxes:

```
MOV     R1,#0x55              ;R1 = 0x55
LDR      R2,=0x5555          ;R2 = 0x5555
```

It needs to be emphasized that we must use the MOV instruction for loading fixed value of less than (or equal to) 0xFF and use the LDR pseudo-instruction only for loading values larger than 0xFF. This is due to the fact that MOV is a real instruction and takes less memory space when it is compiled. See Chapter 6 for more on LDR and ADR pseudo-instructions.

Review Questions

1. Find the contents of R3 after executing the following code:

```
MOV    R0,#0x04
MOV    R3,R0,LSR #2
```

2. Find the contents of R4 after executing the following code:

```
LDR     R1,=0xA0F2
MOV    R2,#0x3
MOV    R4,R1,LSR R2
```

3. Find the contents of R3 after executing the following code:

```
LDR     R1,=0xA0F2
MOV    R2,#0x3
MOV    R3,R1,LSL R2
```

4. Find the contents of R5 after executing the following code:

```
SUBS   R0,R0,R0
MOV    R0,#0xAA
MOV    R5,R0,ROR #4
```

5. Find the contents of R0 after executing the following code:

```
LDR     R2,=0xA0F2
MOV    R1,0x4
MOV    R0,R2,RRX R1
```

6. Give the result in R1 for the following:

```
MVN    R1,#0x01, #2
```

7. Give the result in R2 for the following:

```
MVN    R2,#0x02, #28
```

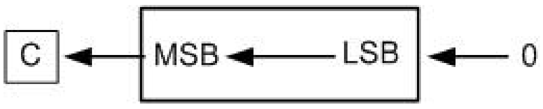
Section 3.4: Shift and Rotate Instructions in ARM Cortex (Case Study)

ARM Cortex has new instructions specifically for shift and rotate. In this section, we discuss these instructions. For full instruction set see Appendix A. For the purpose of comparison we have copied this section from Appendix A.

LSL Logical Shift Left

```
    LSL     Rd, Rm, Rn
```

Function: As each bit of Rm register is shifted left, the MSB is removed and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSL does not updates the flags.



Example 1:

```
    LDR     R2,=0x00000010
    LSL     R0,R2,#8  ;R0=R2 is shifted left 8 times
                      ;now, R0= 0x00001000, flags not updated
```

Example 2:

```
    LDR     R0,=0x00000018
    MOV     R1, #12
    LSL     R2,R0,R1   ;R2=R0 is shifted left R1 number of times
              ;now, R2= 0x000018000, flags not updated
```

Example 3:

```
    LDR     R0,=0x0000FF18
    MOV     R1, #16
    LSL     R2,R0,R1  ;R2=R0 is shifted left R1 number of times
      ;now, R2= 0xFF180000, flags not updated
```

LSLS Logical Shift Left (update the flags)

```
    LSLS    Rd, Rm, Rn
```

Function: As each bit of Rm register is shifted left, the MSB is copied to C flag and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSLS updates the flags.

Example 1:

```
    LDR     R2,=0x00000010
    LSLS    R0,R2,#8          ;R0=R2 is shifted left 8 times
```

```
                                      ;now, R0= 0x00001000, C=0, N=0, Z=0
```

Example 2:

```
LDR      R0,=0x00000018
MOV     R1, #12
LSLS     R2,R0,R1  ;R2=R0 is shifted left R1 number of times
                    ;now, R2= 0x000018000, C=0, N=0, Z=0
```

Example 3:

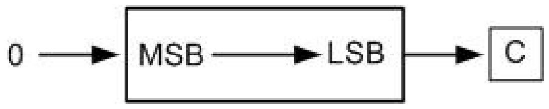
```
LDR      R0,=0x000FFF18
MOV     R1, #16
LSLS     R2,R0,R1  ;R2=R0 is shifted left R1 number of times
          ;now, R2= 0xFF180000, C=1, Z=0, N=0
```

The logical shift left is used for shifting unsigned numbers. LSLS essentially multiplies Rm by a power of 2 after each bit is shifted.

LSR   Logical Shift Right

```
LSR       Rd, Rm, Rn
```

Function: As each bit of Rm register is shifted right, the LSB is removed and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSR does not update the flags.



Example 1:

```
LDR      R2,=0x00001000
LSR       R0,R2,#8   ;R0=R2 is shifted right 8 times
    ;now, R0= 0x00000010, C=0
```

Example 2:

```
LDR      R0,=0x000018000
MOV     R1, #12
LSR       R2,R0,R1  ;R2=R0 is shifted right R1 number of times
                    ;now, R2= 0x00000018, C=0
```

Example 3:

```
LDR      R0,=0x7F180000
MOV     R1, #16
```

```
    LSR      R2,R0,R1  ;R2=R0 is shifted right R1 number of times
                       ;now, R2=0x00007F18, C=0
```

The logical shift right is used for shifting unsigned numbers. LSR essentially divides Rm by a power of 2 after each bit is shifted.

### LSRS		Logical Shift Right (update the flags)

```
    LSRS     Rd, Rm, Rn
```

Function: As each bit of Rm register is shifted right, the LSB is copied to C flag and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSRS updates the flags.

Example 1:

```
    LDR      R2,=0x00001FFF
    LSRS     R0,R2,#8   ;R0=R2 is shifted right 8 times
                ;now, R0= 0x0000001F, C=1, N=0, Z=0
```

Example 2:

```
    LDR      R0,=0x00000018
    MOV     R1, #12
    LSRS     R2,R0,R1    ;R2=R0 is shifted right R1 number of times
        ;now, R2= 0x000000000, C=0, N=0, Z=1,
```

Example 3:

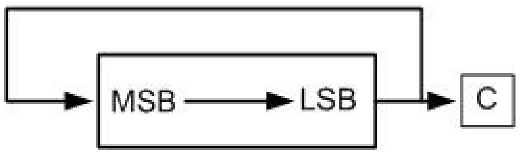
```
    LDR      R0,=0x000FFF18
    MOV     R1, #16
    LSRS     R2,R0,R1    ;R2=R0 is shifted right R1 number of times
        ;now, R2= 0x0000000F, C=1, Z=0,N=0
```

The logical shift right is used for shifting unsigned numbers. LSRS essentially divides Rm by a power of 2 after each bit is shifted.

### ROR		Rotate Right

```
    ROR     Rd,Rm,Rn         ;Rd=rotate Rm right Rn bit positions
```

Function: As each bit of Rm register is shifted from left to right, they exit from the end (LSB) and entered from left end (MSB). The number of bits to be rotated right is given by Rn and the result is placed in Rd register. The ROR does not update the flags.



Example 1:

```
LDR     R2,=0x00000010
ROR     R0,R2,#8          ;R0=R2 is rotated right 8 times
                            ;now, R0 = 0x10000000, C=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV     R1, #12
ROR     R2,R0,R1   ;R2=R0 is rotated right R1 number of times
                    ;now, R2 = 0x01800000, C=0
```

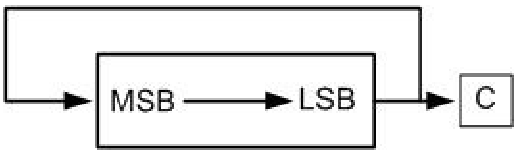
Example 3:

```
LDR     R0,=0x0000FF18
MOV     R1, #16
ROR     R2,R0,R1   ;R2=R0 is rotated right R1 number of times
            ;now, R2 = 0xFF180000, C=0
```

RORS   **Rotate Right (update the flags)**

```
RORS    Rd,Rm,Rn        ;Rd=rotate Rm right Rn bit positions
```

Function: As each bit of Rm register shifts from left to right, they exit from the right end (LSB) and enter from the left end (MSB). In addition as each bit exits the LSB, a copy of it is given to C flag. The number of bits to be rotated right is given by Rn and the result is placed in Rd register. The RORS updates the flags.



Example 1:

```
LDR     R2,=0x00000010
RORS    R0,R2,#8   ;R0=R2 is rotated right 8 times
                    ;now, R0= 0x01000000, C=0, N=0, Z=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV     R1, #12
RORS    R2,R0,R1   ;R2=R0 is rotated right R1 number of times
                    ;now, R2= 0x01800000, C=0, N=0, Z=0
```

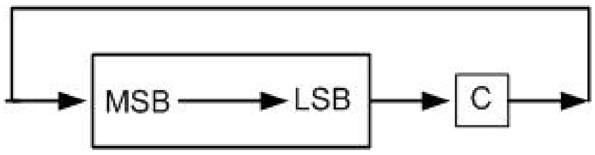
Example 3:

```
LDR     R0,=0x0000FF18
MOV     R1, #16
RORS    R2,R0,R1   ;R2=R0 is rotated right R1 number of times
                   ;now, R2= 0xFF180000, C=1, N=0, Z=0
```

RRX **Rotate Right with extend**

```
RRX     Rd,Rm            ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRX does not update the flags.



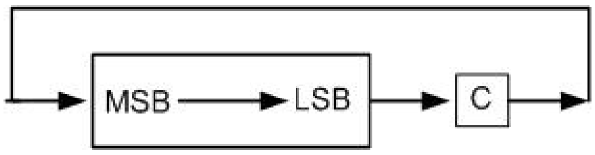
Example:

```
LDR     R2,=0x00000002
RRX     R0,R2     ;R0=R2 is shifted right one bit
;now, R0=0x00000001
```

RRXS **Rotate Right with extend (update the flags)**

```
RRXS    Rd,Rm   ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRXS updates the flags.



Example 1:

```
LDR     R2,=0x00000002
RRXS    R0,R2     ;R0=R2 is shifted right one bit
;now, R0=0x00000001
```

Review Questions

1. True or false. ARM Cortex has its own instruction for rotate and shift.
2. Find the contents of R3 after executing the following code:

```
MOV     R1,#0x08
ROR     R2,R1,#2
```

3. Find the contents of R4 after executing the following code:

```
MOV     R2,#0x3
LSL        R5,R3,#2
```

Section 3.5: BCD and ASCII Conversion

This section covers binary, BCD, and ASCII conversions with some examples.

BCD number system

BCD stands for binary coded decimal. BCD is needed because we use the digits 0 to 9 for numbers in everyday life. BCD system is widely used in real-time clock (RTC) of the embedded systems. Binary representation of 0 to 9 is called BCD. In computer literature one encounters two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD.

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3-9: BCD Codes

Unpacked BCD

In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0. For example, "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively. In the case of unpacked BCD it takes 1 byte of memory location or a register of 8 bits to hold the number.

Packed BCD

In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits. For example, "0101 1001" is packed BCD for 59. It takes only 1 byte of memory to store the packed BCD operands. This is one reason to use packed BCD since it is twice as efficient in storing data.

ASCII numbers

In ASCII keyboards, when key "0" is pressed, "011 0000" (0x30) is provided to the computer. In the same way, 0x31 (011 0001) is provided for key "1", and so on, as shown in the following list:

Key	ASCII	Binary(hex)	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	45	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

It must be noted that although ASCII is standard in the United States (and many other countries), BCD numbers have universal application. Now since the keyboard, printers, and monitors are all in ASCII, how does data get converted from ASCII to BCD, and vice versa? These are the subjects covered next.

ASCII to unpacked BCD conversion

To convert ASCII data to unpacked BCD, the programmer must get rid of the tagged "011" in the upper 4 bits of the ASCII. To do that, each ASCII number is ANDed with "0000 1111" (0x0F).

ASCII to packed BCD conversion

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of the 3) and then combined to make packed BCD. For example, for 2 and 7 the keyboard gives 0x32 and 0x37, respectively. The goal is to produce 0x27 or "0010 0111", which is called packed BCD, as discussed earlier. This process is illustrated in detail below.

Key	ASCII	Unpacked BCD	Packed BCD
2	32	00000010	
7	37	00000111	00100111 (0x27)

```
    MOV    R1,#0x37          ;R1 = 0x37
```

```
MOV     R2,#0x32          ;R2 = 0x32
AND     R1,R1,#0x0F       ;mask 3 to get unpacked BCD
AND     R2,R2,#0x0F       ;mask 3 to get unpacked BCD
MOV     R3,R2,LSL #4      ;shift R2 4 bits to left to get R3 = 0x20
ORR     R4,R3,R1          ;OR them to get packed BCD, R4 = 0x27
```

Packed BCD to ASCII conversion

For data to be displayed on the monitor or be printed by the printer, it must be in ASCII format. Conversion from packed BCD to ASCII is discussed next. To convert packed BCD to ASCII, it must first be converted to unpacked and then the unpacked BCD is tagged with 011 0000 (0x30). The following shows the process of converting from packed BCD to ASCII.

Packed BCD	Unpacked BCD	ASCII
0x29	0x02 & 0x09	0x32 & 0x39
0010 1001	0000 0010 & 0000 1001	011 0010 & 011 1001

```
        MOV     R0,#0x29
AND     R1,R0,#0x0F       ;mask upper four bits
ORR     R1,R1,#0x30       ;combine with 30 to get ASCII
MOV     R2,R0,LSR #04     ;shift right 4 bits to get unpacked BCD
ORR     R2,R2,#0x30       ;combine with 30 to get ASCII
```

Review Questions

1. For the following decimal numbers, give the packed BCD and unpacked BCD representations in binary

   (a) 15 (b) 99

2. For the following packed BCD numbers, give the decimal and unpacked BCD representations.

   (a) 0x41 (b) 0x09

3. Repeat question 2 for ASCII.

Problems

Section 3.1: Arithmetic Instructions

1. Find C and Z flags for each of the following. Also indicate the result of the addition and where the result is saved.

(a)
```
MOV     R1,#0x3F
MOV     R2,#0x45
ADDS    R3,R1,R2
```

(b)
```
LDR     R0,=0x95999999
LDR     R1,=0x94FFFF58
ADDS    R1,R1,R0
```

(c)
```
LDR     R0,=0xFFFFFFFF
ADDS    R0,R0,#1
```

(d)
```
LDR     R2,=0x00000001
LDR     R1,=0xFFFFFFFF
ADDS    R0,R1,R2
ADCS    R0,R0,#0
```

(e)
```
LDR     R0,=0xFFFFFFFE
ADDS    R0,R0,#2
ADC     R1,R0,#0x0
```

2. State the three steps involved in a SUB and show the steps for the following data.

(a) 0x23 – 0x12 (b) 0x43 – 0x51 (c) 0x99 – 0x99

Section 3.2: Logic Instructions

3. Assume that the following registers contain these hex contents: R0 = 0xF000, R1 = 0x3456, and R2 = 0xE390. Perform the following operations. Indicate the result and the register where it is stored.

Note: the operations are independent of each other.

(a) AND R3,R2,R0
(b) ORR R3,R2,R1
(c) EOR R0,R0,#0x76
(d) AND R3,R2,R2
(e) EOR R0,R0,R0
(f) ORR R3,R0,R2
(g) AND R3,R0,#0xFF
(h) ORR R3,R0,#0x99
(i) EOR R3,R1,R0
(j) EOR R3,R1,R1

4. Give the value in R2 after the following code is executed:

```
MOV     R0,#0xF0
MOV     R1,#0x55
BIC      R2,R1,R0
```

5. Give the value in R2 after the following code is executed:

```
LDR      R1,=0x55555555
MVN     R0,#0
EOR      R2,R1,R0
```

Section 3.3: Rotate and Barrel Shifter

6. Assuming C = 0, what is the value of R1 after the following?

```
MOV     R1,#0x25
MOVS  R1,R1,ROR #4
```

7. Assuming C = 0, what are the values of R0 and C after the following?

```
LDR   R0,=0x3FA2
MOV   R2,#8
MOVS  R0,R0,ROR R2
```

8. Assuming C = 0 what is the value of R2 and C after the following?

```
MOV   R2,#0x55
            MOVS  R2,R2,RRX
```

9. Assuming C = 0 what is the value of R1 after the following?

```
MOV   R1,#0xFF
MOV   R3,#5
MOVS   R1,R1, ROR R3
```

10. Give the register value for each of the following instructions after it is executed.

a) MOV R1,#0x88,#4	b) MOV R0,#0x22,#22
c) MOV R2,#0x77,#8	d) MOV R4,#0x5F,#28
e) MOV R6,#0x88,#22	f) MOV R5,#0x8F,#16
g) MOV R7,#0xF0,#20	h) MOV R1,#0x33,#28

11. Give the register value for each of the following instructions after it is executed.

a) MVN R2,#0x1	b) MVN R2,#0xAA,#20

c) MVN R1,0x55,#4

d) MVN R0,#0x66,#28

e) MVN R1,#0x80,#24

f) MVN R6,#0x10,#20

g) MVN R7,#0xF0,#24

h) MVN R4,#0x99,#4

12. Find the contents of registers and C flag after executing each of the following codes:

a)
```
MOV R0,#0x04
MOVS R1,R0,LSR #2
MOVS R3,R0,LSR R1
```

b)
```
LDR R1,=0xA0F2
MOV R2,#0x3
MOVS R3,R1,LSL R2
```

c)
```
LDR R1,=0xB085
MOV R2,#3
MOVS R4,R1,LSR R2
```

13. Find the contents of registers and C flag after executing each of the following codes:

a)
```
SUBS R2,R2,R2
MOV R0,#0xAA
MOVS R1,R0,ROR #4
```

b)
```
MOV R2,#0xAA,#4
MOV R0,#1
MOVS R1,R2,ROR R0
```

c)
```
LDR    R1,=0x1234
MOV  R2,#0x010,#2
MOVS R1,R0,ROR R2
```

d)
```
MOV   R0,#0xAA
MOVS R1,R0,RRX
```

14. Using MOV instruction, show how you rotate left the fixed value of 0x33 total of a) 4, b) 8, and c) 12 times. Also give the value in the register after the rotation.

Section 3.5: BCD and ASCII Conversion

15. Write a program to convert 0x76 from packed BCD number to ASCII. Place the ASCII codes into R1 and R2.

16. For 3 and 2 the keyboard gives 0x33 and 0x32, respectively. Write a program to convert 0x33 and 0x32 to packed BCD and store the result in R2.

Answers to Review Questions

Section 3.1: Arithmetic Instructions

1. The ADDS instruction updates the flag bits while ADD does not do that.
2. Rd = Rn + Op2 + C
3. 0x4F + 0xB1 = 0x100, since it is a byte addition and result is less than 32-bit the C = 0 and Z = 0.
4. 0x4F + 0xFFFFFFB1 = 00000000, since it is a word addition and result is greater than 32-bit, the C = 1 and Z = 1.
5.

0x43	0100 0011		00000000000000000000000001000011
<u>–0x05</u>	<u>0000 0101</u>	2's complement =	<u>+11111111111111111111111111111011</u>
0x3E			1 00000000000000000000000000111110

C = 1; therefore, the result is positive

6. R2 = R2 – R3 – C + 1 = 0x95 – 0x4F – 1 + 1 = 0x46
7. R2
8. R2 = 1

Section 3.2: Logic Instructions

1. (a) 0x4202 (b) 0xCFFF (c) 0x8DFD
2. The operand will remain unchanged; all zeros
3. All ones
4. All zeros
5. ORR R7,R7,#0x10 ;R7 = R7 ORed 0001 0000
6. AND R5,R5,#0x8 ;R5 = R5 ORed 0000 1000

Section 3.3: Rotate and Barrel Shifter Operation

1. R3 = 1
2. R4 = 0x0000141E
3. R3 = 0x00050790
4. R5 = 0xA000000A
5. R0 = 0x00005079
6. 0xBFFFFFFF
7. 0xFFFFFFDF

Section 3.4: Shift and Rotate Instructions in ARM Cortex

1. True
2. 0x01
3. 0x0C

Section 3.5: BCD and ASCII Conversion

1. (a) 15 = 0001 0101 packed BCD = 0000 0001 0000 0101 unpacked BCD

   (b) 99 = 1001 1001 packed BCD = 0000 1001 0000 1001 unpacked BCD

2. (a) 0x41 = 0000 0100 0000 0001 unpacked BCD = 41 in decimal

(b) 0x09 = 0000 0000 0000 1001 unpacked BCD = 9 in decimal

3. (a) 0x34,0x31

   (b) 0x30,0x39

Chapter 4: Branch, Call, and Looping in ARM

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. (e.g. when a function is called, execution of a loop is repeated, or an instruction executes conditionally) There are many instructions in ARM to achieve this. This chapter covers the control transfer instructions available in ARM Assembly language. In Section 4.1, we discuss instructions used for looping, as well as instructions for conditional and unconditional branches (jumps). In the second section, we examine the instructions associated with calling subroutine. In Section 4.3, instruction pipelining of the ARM is examined. Instruction timing and time delay subroutines are also discussed in Section 4.3. In Section 4.4, we examine the conditional execution of the ARM instructions which is a unique feature of ARM.

Section 4.1: Looping and Branch Instructions

In this section we first discuss how to perform a looping action in ARM and then the branch (jump) instructions, both conditional and unconditional.

Looping in ARM

Repeating a sequence of instructions or an operation a certain number of times is called a *loop*. The loop is one of the most widely used programming techniques. In the ARM, there are several ways to repeat an operation many times. One way is to repeat the operation over and over until it is finished, as shown below:

```
MOV     R0,#0           ;R0 = 0
MOV     R1,#9           ;R1 = 9
ADD     R0,R0,R1        ;R0 = R0 + R1, add 9 to R0 (Now R0 is 0x09)
ADD     R0,R0,R1        ;R0 = R0 + R1, add 9 to R0 (Now R0 is 0x12)
ADD     R0,R0,R1        ;R0 = R0 + R1, add 9 to R0 (Now R0 is 0x1B)
ADD     R0,R0,R1        ;R0 = R0 + R1, add 9 to R0 (Now R0 is 0x24)
ADD     R0,R0,R1        ;R0 = 0x2D
ADD     R0,R0,R1        ;R0 = 0x36
```

In the above program, we add 0x9 to R0 six times. That makes 6 × 9 = 54 = 0x36. One problem with the above program is that too much code space would be needed to increase the number of repetitions to 50 or 1000. A much better way is to use a loop. Next, we describe the method to do a loop in ARM.

Using instruction BNE for looping

The BNE (branch if not equal) instruction uses the zero flag in the status register. The BNE instruction is used as follows:

```
BACK    .........          ;start of the loop
    .........          ;body of the loop
    .........          ;body of the loop
    SUBS    Rn,Rn,#1       ;Rn = Rn - 1, set the flag Z = 1 if Rn = 0
    BNE     BACK           ;branch if Z = 0
```

In the last two instructions, the Rn (e.g. R2 or R3) is decremented; if it is not zero, it branches (jumps) back to the target address referred to by the label. Prior to the start of the loop, the Rn is loaded with the counter value for the number of repetitions. Notice that the BNE instruction refers to the Z flag of the status register affected by the previous instruction, SUBS. This is shown in Example 4-1.

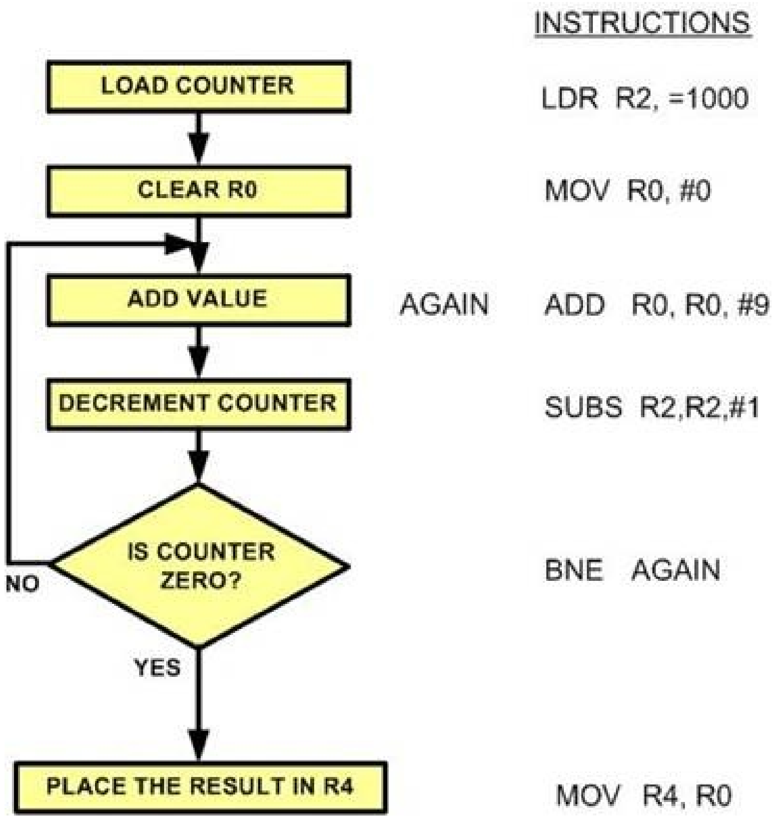
Example 4-1

Write a program to (a) clear R0, (b) add 9 to R0 a thousand times, then (c) place the sum in R4.

Use the zero flag and BNE instruction.

Solution:

```
    ;– this program adds value 9 to the R0 a 1000 times –
    AREA     EXAMPLE4_1, CODE, READONLY
    ENTRY
    LDR       R2,=1000             ;R2 = 1000 (decimal) for counter
    MOV     R0,#0                  ;R0 = 0 (sum)
AGAIN ADD       R0,R0,#9             ;R0 = R0 + 9 (add 09 to R1, R1 = sum)
    SUBS     R2,R2,#1             ;R2 = R2 - 1 and set the flags. Decrement counter
    BNE       AGAIN                 ;repeat until COUNT = 0 (when Z = 1)
    MOV     R4,R0                  ;store the sum in R4
HERE      B           HERE                    ;stay here
    END
```



In the program in Example 4-1, register R2 is used as a counter. The counter is first set to 1000. In each iteration, the SUBS instruction decrements the R2 and sets the flag bits accordingly. If R2 is not zero (Z = 0), it jumps to the target address associated with the label “AGAIN”. This looping action continues until R2 becomes zero. After R2 becomes zero (Z = 1), it falls through the loop and executes the instruction immediately below it, in this case “MOV R4,R0”.

It must be emphasized again that we must use SUBS instead of SUB since the SUB instruction will not change (update) the flags. Since we are monitoring the Z flag for the loop counter we must use SUBS instruction for the decrementing the counter. As we mentioned in Chapter 3, many of the ARM instructions have the option of affecting the flags. In these instructions the default is not to affect the flags. Therefore to make them to effect the flag we must add letter S to the instruction. That means SUBS and ADDS instructions are different from SUB and ADD, as far as the flags are concerned. As another example see Example 4-2.

Example 4-2

Write a program to place value 0x55 into 100 bytes of RAM locations.

Solution:

```
    AREA     EXAMPLE4_2, CODE, READONLY
    ENTRY
RAM_ADDR EQU          0x40000000       ;change the address for your ARM

     MOV    R2,#25                 ;counter (25 times 4 = 100 byte block size)
     LDR      R1,=RAM_ADDR              ;R1 = RAM Address
     LDR      R0,=0x55555555             ;R0 = 0x55555555

OVER     STR       R0,[R1]               ;send it to RAM
     ADD      R1,R1,#4            ;R1 = R1 + 4 to increment pointer
     SUBS    R2,R2,#1            ;R2 = R2 – 1 for dec. counter
     BNE      OVER                 ;keep doing it
HERE      B          HERE
     END
```

Looping a trillion times with loop inside a loop

As shown in Example 4-3, the maximum count is 2^{32}-1. What happens if we want to repeat an action more times than that? To do that, we use a loop inside a loop, which is called a nested loop. In a nested loop, we use two registers to hold the count. See Example 4-3.

Example 4-3

Explain what is the maximum number of times that the loop in Example 4-1 can be repeated? Now, write a program to (a) load the R0 register with the value 0x55, and (b) complement it 16,000,000,000 (16 billion) times.

Solution:

Because Rx is a 32-bit register, it can hold a maximum of 0xFFFFFFFF (2^{32} – 1 decimal); therefore, the loop can be repeated a maximum of 2^{32} – 1 times. This example shows how to create a nesting loop to go beyond 4 billion times. Because 16,000,000,000 is larger than 0xFFFFFFFF (the maximum capacity of any R0–R12 registers), we use two registers to hold the count. The following code shows how to use R2 and R1 as a register for counters in a nesting loop.

```
    AREA    EXAMPLE4_3,   CODE, READONLY
    ENTRY
    MOV    R0,#0x55            ;R0 = 0x55
    MOV    R2,#16              ;load 16 into R2 (outer loop count)
L1         LDR      R1,=1000000000          ;R1 = 1,000,000,000 (inner loop count)
L2         EOR      R0,R0,#0xFF      ;complement R0 (R0 = R0 Ex-OR 0xFF)
    SUBS    R1,R1,#1           ;R1 = R1 – 1, dec. R1 (inner loop)
    BNE      L2                    ;repeat it until R1 = 0
    SUBS    R2,R2,#1           ;R2 = R2 – 1, dec. R2 (outer loop)
    BNE      L1                    ;repeat it until R2 = 0
HERE      B          HERE                ;stay here
    END
```

In this program, R1 is used to keep the inner loop count. In the instruction “BNE L2”, whenever R1 becomes 0 it falls through and “SUBS R2,R2,#1” is executed. The next instructions force the CPU to load the inner count with 1,000,000,000 if R2 is not zero, and the inner loop starts again. This process will continue until R2 becomes zero and the outer loop is finished. If you use the Keil IDE to verify the operation of the above program use smaller values for counter to go through the iterations. See Figure 4-1.

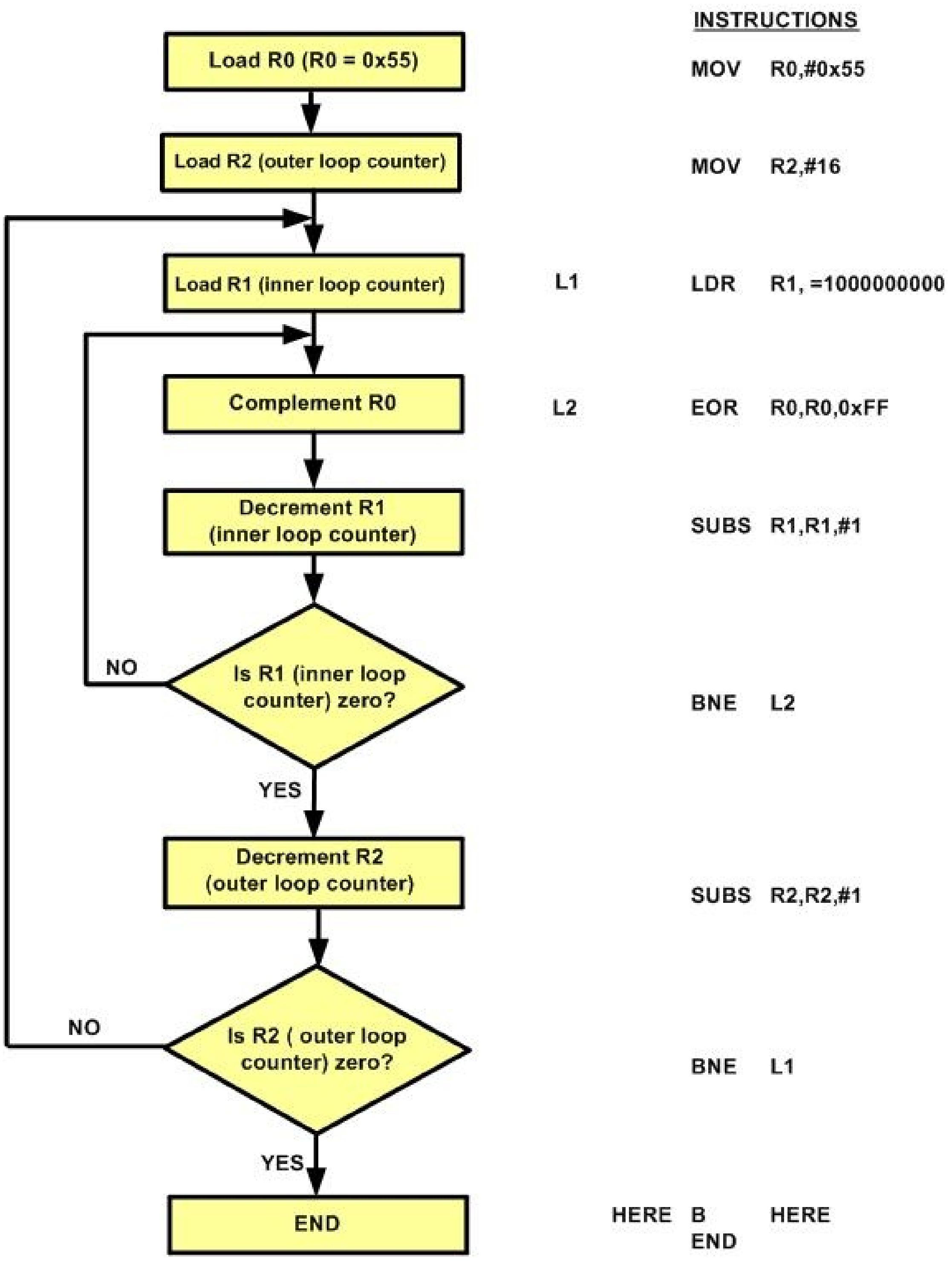

INSTRUCTIONS
Load R0 (R0 = 0x55)
MOV R0,#0x55
Load R2 (outer loop counter)
MOV R2,#16
Load R1 (inner loop counter)
L1
LDR R1, =1000000000
Complement R0
L2
EOR R0,R0,0xFF
Decrement R1 (inner loop counter)
SUBS R1,R1,#1
NO
Is R1 (inner loop counter) zero?
BNE L2
YES
Decrement R2 (outer loop counter)
SUBS R2,R2,#1
NO
Is R2 (outer loop counter) zero?
BNE L1
YES
END
HERE B HERE
END

Figure 4- 1: Flowchart for Example 4-3

Other conditional Branches

As we mentioned in Chapter 3, C and Z flags reflect the result of calculation on unsigned numbers. Table 4-1 lists available conditional branches for unsigned numbers that use C and Z flags. More details of each instruction are provided in Appendix A. In Table 4-1 notice that the instructions, such as BEQ (Branch if Z = 1) and BCS (Branch if carry set, C = 1), jump only if a certain condition is met. Next, we examine some conditional branch instructions with examples. The other conditional branch instructions associated with the signed numbers are discussed in Chapter 5 when arithmetic operations for signed numbers are discussed.

Instruction		Action
BCS/BHS	branch if carry set/branch if higher or same	Branch if C = 1
BCC/BLO	branch if carry clear/branch lower	Branch if C = 0
BEQ	branch if equal	Branch if Z = 1
BNE	branch if not equal	Branch if Z = 0
BLS	branch if less or same	Branch if Z = 1 or C = 0
BHI	branch if higher	Branch if Z = 0 and C = 1

Table 4-1: ARM Conditional Branch Instructions for Unsigned Data

BCC (branch if carry clear, branch if C = 0)

In this instruction, the carry flag bit in program status registers (CPSR) is used to make the decision whether to branch. In executing "BCC label", the processor looks at the carry flag to see if it is cleared (C = 0). If it is, the CPU starts to fetch and execute instructions from the address of the label. If C = 1, it will not jump but will execute the next instruction below BCC. See Example 4-4.

Example 4-4

Examine the following code and give the result in registers R0, R1, and R2.

```
    MOV     R1,#0               ;clear high word (R1 = 0)
    MOV     R0,#0               ;clear low word (R0 = 0)
    LDR     R2,=0x99999999          ;R2 = 0x99999999
    ADDS    R0,R0,R2            ;R0 = R0 + R2 and set the flags
    BCC     L1                   ;if C = 0, jump to L1 and add next number
    ADDS    R1,R1,#1            ;ELSE, increment ( R1 = R1 + 1)
L1      ADDS    R0,R0,R2            ;R0 = R0 + R2 and set the flags
    BCC     L2                   ;if C = 0, add next number
    ADDS    R1,R1,#1            ;if C = 1, increment
L2      ADDS    R0,R2                ;R0 = R0 + R2 and set the flags
    BCC     L3                   ;if C = 0, add next number
    ADDS    R1,R1,#1            ;C = 1, increment
L3          ADDS    R0,R2               ;R0 = R0 + R2 and set the flags
    BCC     L4                  ;if C = 0, add next number
    ADDS    R1,R1,#1            ;if C = 1, and set the flags
L4
```

Solution:

This program adds 0x99999999 together four times.

	R1 (high byte)	R0 (low byte)
At first	0	0
Just before L1	0	0x99999999
Just before L2	1	0x33333332
Just before L3	1	0xCCCCCCCB
Just before L4	2	0x66666664

Here is the loop version of the above program that runs 10 times.

```
    AREA    EXAMPLE4_4,CODE, READONLY
    ENTRY
    MOV     R1,#0               ;clear high word (R1 = 0)
    MOV     R0,#0               ;clear low word (R0 = 0)
    LDR     R2,=0x99999999           ;R2 = 0x99999999
    MOV     R3,#10              ;counter
L1          ADDS    R0,R2               ;R0 = R0 + R2 and set the flags
    BCC     NEXT                ;if C = 0, add next number
    ADD     R1,R1,#1            ;if C = 1, increment the upper word
NEXT    SUBS    R3,R3,#1            ;R3 = R3 - 1 and set the flags
    ;(Decrement counter)
    BNE     L1                  ;next round if Z = 0
```

```
HERE    B       HERE            ;stay here
   END
```

Note that there is also a “BCS label” instruction. In the BCS instruction, if C = 1 it jumps to the target address. We will give more examples of these instructions in the context of applications.

Comparison of unsigned numbers

```
   CMP    Rn,Op2          ;compare Rn with Op2 and set the flags
```

The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged. There is no destination register and the second source operands can be a register or an immediate value not larger than 0xFF. It must be emphasized that “CMP Rn,Op2” instruction is really a subtract operation. Op2 is subtracted from Rn (Rn – Op2) and the result is discarded and flags are set accordingly. The contents of Rn and Op2 remain unchanged after the execution of CMP instruction. Although all the C, S, Z, and V flags reflect the result of the comparison, only C and Z are used for unsigned numbers, as outlined in Table 4-2.

Instruction	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

Table 4- 2: Flag Settings for Compare (CMP Rn, Op2) of Unsigned Data

Look at the following case:

```
   LDR    R1,=0x35F       ;R1 = 0x35F
   LDR    R2,=0xCCC       ;R2 = 0xCCC
   CMP    R1,R2           ;compare 0x35F with 0xCCC
   BCC    OVER            ;branch if C = 0
   MOV    R1,#0           ;if C = 1, then clear R1
OVER  ADD    R2,R2,#1     ;R2 = R2 + 1 = 0xCCC + 1 = 0xCCD
```

Figure 4-2 shows the diagram and the C language version of the code.

Pseudo code:
R1 = 0x35F
R2 = 0xCCC
	In C:
	R1 = 0x35F;
	R2 = 0xCCC;	

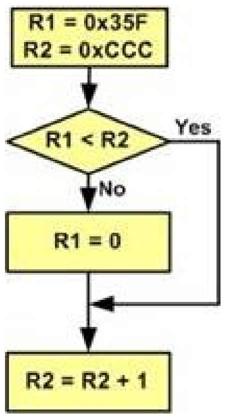
IF (R1 >= R2) THEN
R1 = 0
ENDIF
 	
R2 = R2 + 1
	if (R1 >= R2)
	{
	R1 = 0;
	}
	
	R2 = R2 + 1;
		 R1 = 0x35F
		R2 = 0xCCC
		R1 < R2
		Yes
		No
		R1 = 0
		R2 = R2 + 1

Figure 4- 2: Flowchart of if Instruction

In the above program, R1 is less than the R2 (0x35F < 0xCCC); therefore, C = 0 and BCC (branch if carry clear) will go to target OVER. In contrast, look at the following:

```
    LDR     R1,=0xFFF
    LDR     R2,=0x888
    CMP     R1,R2           ;compare 0xFFF with 0x888
    BCC     NEXT
    ADD     R1,R1,#0x40
NEXT ADD    R1,R1,#0x25
```

In the above, R1 is greater than R2 (0xFFF > 0x888), which sets C = 1, making “BCC NEXT” fall through so that “ADD R1,R1,0x40” is executed.

Again, it must be emphasized that in CMP instructions, the operands are unaffected regardless of the result of the comparison. Only the flags are affected. This is despite the fact that CMP uses the SUB operation to set or reset the flags. It also may be noted that, unlike other arithmetic and logic instructions, there is no need to put S in the CMP instruction to update the flags. In other words, the CMP instruction automatically updates the flags.

Program 4-1 uses the CMP instruction to search for the highest byte in a series of 5 data bytes. To search for the highest value the instruction “CMP R1,R3” works as follows, where R1 is the contents of the memory location brought into R1 register by the [R2] pointer.

a) If R1 < R3, then C = 0 and R3 becomes the basis of the new comparison.

b) If R1 ≥ R3, then C = 1 and R1 is the larger of the two values and remains the basis of comparison.

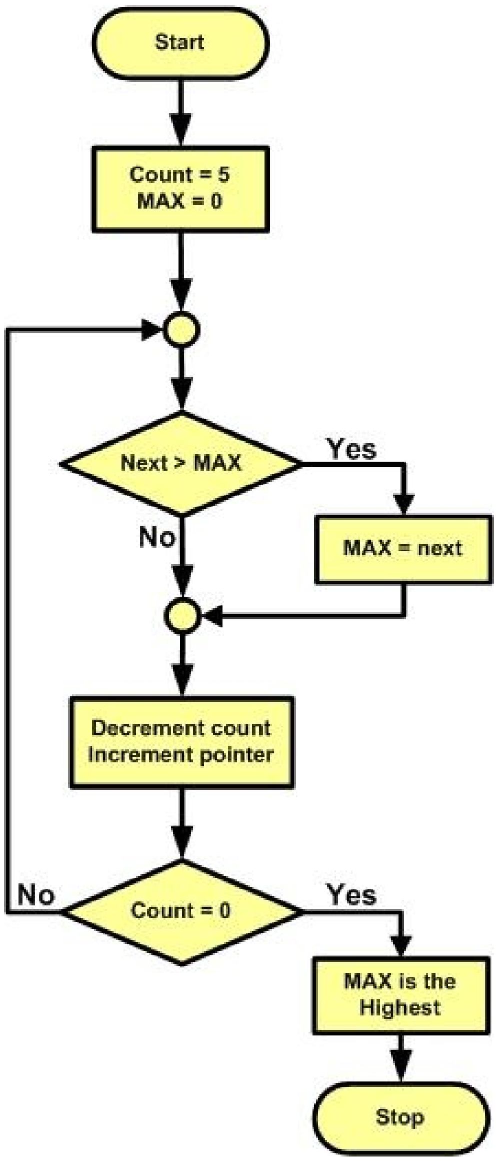
Program 4-1

Assume that there is a class of five people with the following grades: 69, 87, 96, 45, and 75. Find the highest grade.

```
;searching for highest value
COUNT RN          R0          ;COUNT is the new name of R0
MAX      RN          R1          ;MAX is the new name of R1
    ;(MAX has the highest value)
POINTER              RN          R2          ;POINTER is the new name of R2
NEXT     RN          R3          ;NEXT is the new name of R3

                AREA   PROG_4_1D, DATA, READONLY
MYDATA              DCD       69,87,96,45,75
               AREA   PROG_4_1, CODE, READONLY
    ENTRY
    MOV     COUNT,#5          ;COUNT = 5
    MOV     MAX,#0                              ;MAX = 0
    LDR        POINTER,=MYDATA
    ;POINTER = MYDATA ( address of first data )
AGAIN  LDR        NEXT,[POINTER]
    ;load contents of POINTER location to NEXT
    CMP      MAX,NEXT
    ;compare MAX and NEXT
    BHS        CTNU
    ;if MAX > NEXT branch to CTNU
    MOV     MAX,NEXT          ;MAX = NEXT
CTNU    ADD       POINTER,POINTER,#4
    ;POINTER=POINTER+4 to point to the next
    SUBS     COUNT,COUNT,#1          ;decrement counter
    BNE        AGAIN  ;branch AGAIN if counter is not zero

HERE      B             HERE
    END
```

Program 4-1 searches through five data items to find the highest value. The program has a variable called “MAX” that holds the highest grade found so far. One by one, the grades are compared to Highest. If any of them is higher, that value is placed in MAX.

This continues until all data items are checked. A REPEAT-UNTIL structure was chosen in the program design. Figure 4-3 shows the flowchart for Program 4-1. This design could be used to code the program in many different languages.



Pseudo code:

```
Count = 5
Highest = 0

REPEAT
  IF (Next > MAX)
  THEN
  MAX = Next
  ENDIF
  Increment pointer
  Decrement Count
UNTIL Count = 0

; now MAX is the Highest
```

In C:

```
//In Keil, long is 32-bit wide

unsigned long myData[5]= {69,87,96,45,75};
unsigned long count = 5;
unsigned long max = 0;
unsigned long next;
unsigned long *pointer = myData;

do
{
  next = *pointer;

  if (max < next)
  max = next;

  pointer ++;
  count —;
}while(count != 0);
```

Figure 4- 3: Flowchart and Pseudocode for Program 4-1

Using CMP followed by conditional branches we can make any comparison on unsigned numbers, as shown in Table 4-3. Although BCS (branch carry set) and BCC (branch carry clear) check the carry flag and can be used after a compare instruction, it is recommended that BHS (branch higher or same) and BLO (branch below) be used for two reasons. One reason is that assemblers will unassemble BCS as BLO, and BCC as BHS, which may be confusing to beginner programmers. Another reason is that "branch higher" and "branch below" are easier to understand than "branch carry set" and "branch carry clear," since it is more immediately apparent that one number is larger than another, than whether a carry would be generated if the two numbers were subtracted.

Instruction		Action
BCS/BHS	branch if carry set/branch if higher or same	Branch if Rn ≥ Op2
BCC/BLO	branch if carry clear/branch lower	Branch if Rn < Op2
BEQ	branch if equal	Branch if Rn = Op2
BNE	branch if not equal	Branch if Rn ≠ Op2
BLS	branch if less or same	Branch if Rn ≤ Op2
BHI	branch if higher	Branch if Rn > Op2

Table 4- 3: ARM Conditional Branch Instructions for Unsigned Data

Division of unsigned numbers in ARM

The older ARM family members do not have an instruction for division of unsigned numbers since it took too many gates to implement it. However, many of the ARM Cortex chips implement the divide instruction. In ARMs with no divide instructions we can use SUB instruction to perform the division. Program 4-2 shows an example of unsigned division using subtract operation. In the program the numerator is placed in a register and the denominator is subtracted from it repeatedly. The quotient is the number of times we subtracted and the remainder is in the register upon completion. See Figure 4-4.

Program 4-2: Division by Repeated Subtractions

```
AREA   PROG_4_2, CODE, READONLY      ;Division by subtractions
ENTRY
```

```
   LDR        R0,=2012              ;R0 = 2012 (numerator )
   ;it will contain remainder
   MOV      R1,#10                  ;R1 = 10 ( denominator )
   MOV      R2,#0                   ;R2 = 0 ( quotient )
L1         CMP      R0,R1                     ;Compare R0 with R1 to see if less than 10
   BLO        FINISH                ;if R0 < R1 jump to finish
   SUB        R0,R0,R1              ;R0 = R0 - R1 (division by subtraction)
   ADD        R2,R2,#1              ;R2 = R2 + 1 (quotient is incremented)
   B          L1                    ;goto L1 (B is discussed in the next section)
FINISH B            FINISH
```

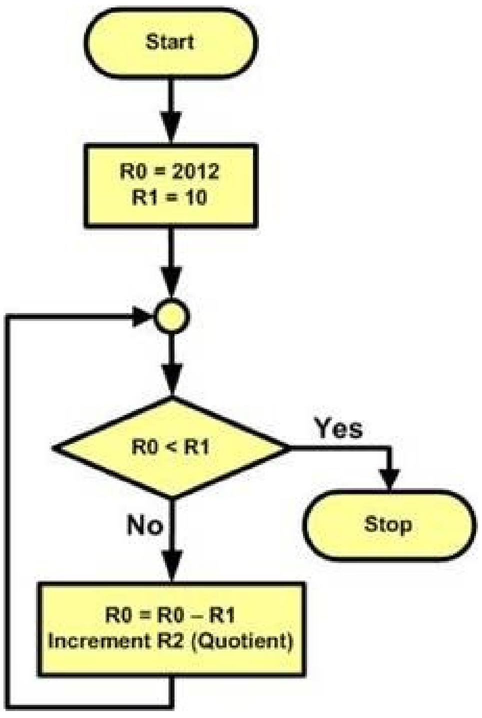
 Start
R0 = 2012
R1 = 10
R0 < R1
Yes
No
Stop
R0 = R0 – R1
Increment R2 (Quotient)
	Pseudo code:
	
	NUM = 2012
	DENOM = 10
	
	WHILE NUM >= DENOM
	  Subtract DENOM from NUM
	  Increment QUOTIENT
	END WHILE
	
	In C:
	
	R0 = 2012;
	R1 = 10;
	
	while (R0 >= R1)
	{
	  R0 = R0 - R1;
	  R2 = R2 + 1;
	}

Figure 4- 4: Flowchart and Pseudo-code for Program 4-2

TST (Test)

```
    TST       Rn,Op2              ;Rn AND with Op2 and flag bits are updated
```

The TST instruction is used to test the contents of register to see if any bit is set to HIGH. After the operands are ANDed together the flags are updated. After the TST instruction if result is zero, then Z flag is raised and one can use BEQ (branch equal) to make decision. Look at the following example:

```
    MOV     R0,#0x04            ;R0=00000100 in binary
    LDR       R1,=myport      ;port address
OVER    LDRB      R2,[R1]              ;load R2 from myport
    TST       R2,R0                    ;is bit 2 HIGH?
    BEQ       OVER                     ;keep checking
```

In TST, like other logic or arithmetic data processing instructions, the Op2 can be an immediate value of less than 0xFF. Look at the following example:

```
    LDR       R1,=myport      ;port address
OVER    LDRB      R2,[R1]              ;load R2 from myport
    TST       R2,#0x04            ;is bit 2 HIGH?
    BEQ       OVER                     ;keep checking
```

See Example 4-5.

Example 4-5

Assume address location 0x200000 is assigned to an input port address and connected to 8 DIP switches. Write a simple short program to check the PORT and whenever both pins 4 or 6 are LOW, R4 register is incremented.

Solution:

```
MYPORT            EQU      0x200000
    MOV     R0,#2_01010000            ;R0=0x50 (01010000 in binary)
    LDR       R1,=MYPORT     ;R1 = port address
OVER    LDRB      R2,[R1]              ;get a byte from PORT and place it in R2
    TST       R2,R0                    ;are bits 4 and 6 LOW?
    BNE       OVER                     ;keep checking
    ADD       R4,R4,#1
```

TEQ (test equal)

```
    TEQ       Rn,Op2              ;Rn EX-ORed with Op2 and flag bits are set
```

The TEQ instruction is used to test to see if the contents of two registers are equal. After the source operands are Ex-ORed together the flag bits are set according to the result. After the TEQ instruction if result is 0, then Z flag is raised and one can use BEQ (branch zero) to make decision. Recall that if we Exclusive-OR a value with itself, the result is zero. Look at the following example for checking the temperature of 100 on a given port:

```
TEMP    EQU      100
    MOV     R0,#TEMP          ;R0 = Temp
    LDR       R1,=myport       ;port address
OVER     LDRB      R2,[R1]              ;load R2 from myport
    TEQ       R2,R0                   ;is it 100?
    BNE       OVER                    ;keep checking
```

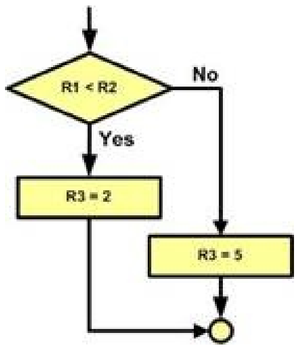
Unconditional branch (jump) instruction

The unconditional branch is a jump in which control is transferred unconditionally to the target location. In the ARM there are two unconditional branches: B (branch) and BX (branch and exchange). This is discussed next.

B (Branch)

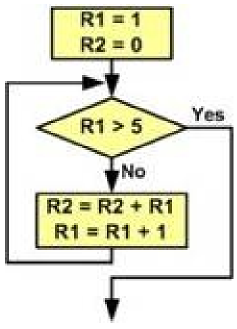
B (branch) is an unconditional jump that can go to any memory location in the 32M byte address space of the ARM. Another syntax for B instruction is BAL (branch always).

B has different usages like implementing if/else, while, and for instructions. In the following code you see an example of implementing the if/else instruction:

CMP R1,R2
BHS L1
MOV R3,#2
B OVER
L1 MOV R3,#5
OVER
		//in C
		if(R1 < R2)
		{
		R3 = 2;
		}
		else
		{
		R3 = 5;
		}

In the above code, R3 is initialized with 2 when R1 is lower than R2. Otherwise, it is initialized with 5.

As an example of implementing the while instruction see the following program. It calculates the sum of numbers between 1 and 5:

```
  MOV   R1,#1
  MOV   R2,#0
L1  CMP   R1,#5
  BHI   L2
  ADD   R2,R2,R1
  ADD   R1,R1,#1
  B     L1
L2  MOV   R3,#5
```
	 R1 = 1
	R2 = 0
	R1 > 5
	Yes
	No
	R2 = R2 + R1
	R1 = R1 + 1
		//in C
		unsigned long R1 = 1;
		unsigned long R2 = 0;
		while (R1 <= 5)
		{
		  R2 = R2 + R1;
		  R1 = R1 + 1;
		}

The *for* instruction can be implemented the same way as the *while* instruction. For example, the above assembly program can be considered as a *for* loop.

In cases where there is no operating system or monitor program, we use the Branch to itself in order to keep the microcontroller busy. A simple way of doing that is shown below:

```
HERE    B        HERE    ;stay here
```

Another syntax for the B instruction is BAL (branch always) as shown below:

```
HERE    BAL      HERE    ;stay here
```

Since ARM instruction is 32-bit, 8 bits are used for the opcode, and the other 24 bits represent the address of the target location. The 24-bit target address is shifted left twice and that allows a jump to –32M to +32M bytes of memory locations from the address of current instruction. Next, we explain the reason for this.

All branches are short branches (jumps)

It must be noted that all branch instructions (conditional and unconditional) are short jumps, meaning the address of the target must be within 32M bytes of the program counter (PC). That means the short jumps cannot cover the entire address space of 4G bytes (0x00000000 to 0xFFFFFFFF).

Calculating the short branch address

All conditional branches such as BCC, BEQ, and BNE are short branches. This is due to the fact that the ARM instructions are 32-bit and some of the 32-bit must be used for opcode and therefore it cannot cover the entire 4G bytes address space of ARM. In the branch instruction the opcode is 8 bits and the relative address is 24 bits. See Figure 4-5. The target address is relative to the value of the program counter. If the relative address is positive, the jump is forward. If the relative address is negative, then the jump is backwards. The relative address can cover memory space of –32Mbytes to +32Mbytes from current location of program counter. The reason for 32MB is the fact that 24 bits are shifted left twice (multiplied by 4) by the ARM CPU automatically. That gives us 26 bits. Now, since one bit is used for positive or negative sign, we have only 25 bits for magnitude. The 25 bits magnitude gives us 32M bytes (2^{25} = 32M) in each direction. That is –32Mbytes if it is backward and +32MB if it is forward jump. Therefore, to calculate the target address, the relative address is shifted left twice and added to the address of the next instruction to be fetched [target address = relative address shifted left twice + address of the next instruction to be fetched]. It must be noted that the next instruction to be fetched is two instructions below the current branch instruction. The reason is the instruction right below the branch is already in the pipeline. See Figure 4-5.

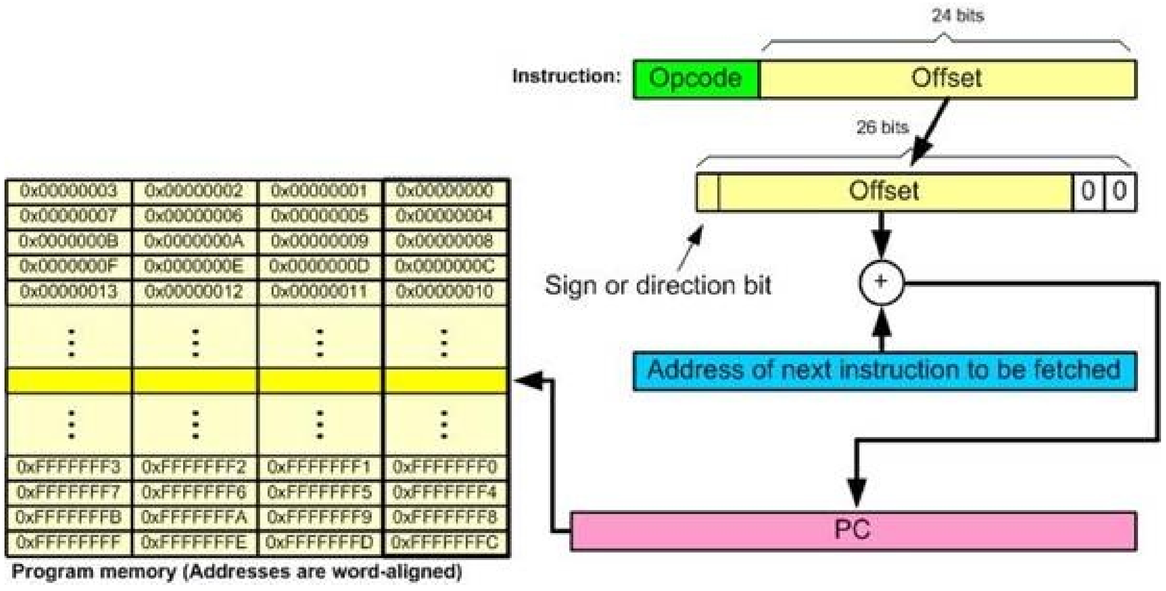


Figure 4- 5: B (Branch) Instruction

You might ask why we add the relative address to the address of two instructions below the current instruction. (why don't we add the relative address to the address of the instruction right below the current instruction as it is in other CPUs). This is due to the pipeline issues as we will see in next section and Chapter 7.

Now, to calculate the target address of the branch we add address of the next instruction to be fetched to the offset shifted left twice (multiplied by 4). The reason for shifting left twice is to make sure it is word aligned. Given the fact that all the ARM instructions are 4-byte (word) long and are word aligned, with 2^{25} = 32M bytes address space for the forward and backward jumps the target address of jump (branch) can be up to 8M instructions (32MBytes / 4byes = 8M instructions) from the current instruction in each direction. Although this does not cover the entire 1G instructions (4GBytes /4byes = 1G instructions) of ARM memory space, it is more than adequate for many applications. See Example 4-6.

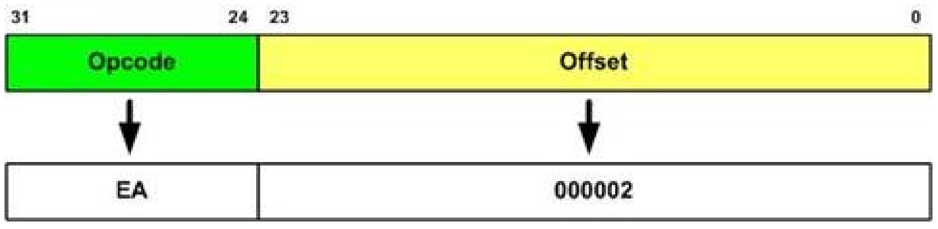
Example 4-6

In ARM7, the next instruction to be fetched is 2 instructions below the current executing instruction. Using the following list file verify the jump forward address calculation.

```
LINE   ADDRESS     Machine      Mnemonic      Operand
1      00000000                         AREA    EXAMPLE_4_6, CODE, READONLY
2      00000000                         ENTRY
3      00000000    E3A01015              MOV     R1, #0x15        ;R1 = 0x15
4      00000004    EA000002              B       THERE
5      00000008    E3A01025              MOV     R1, #0x25        ;R1 = 0x25
6      0000000C    E3A02035              MOV     R2, #0x35        ;R2 = 0x35
7      00000010    E3A03045              MOV     R3, #0x45        ;R3 = 0x45
8      00000014    E3A04055       THERE  MOV     R4, #0x55        ;R4 = 0x55
9      00000018    EAFFFFFE              HERE    B        HERE
10                                      END
```

Solution:

First notice that the B instruction in line 4 jumps forward. To calculate the target address, the relative address (offset) is shifted left twice and added to the PC of the next instruction to be fetched. The position of the next instruction to be fetched is 2 instructions below the current instruction. Recall that each instruction of ARM takes 4 bytes. So the next instruction to be fetched is 2 × 4 bytes = 8 bytes below the current instruction address (00000004). So the address of the next instruction to be fetched is 0000004 + 8 = 000000C (the position of MOV instruction in line 6). In line 4 the instruction “B THERE” has the machine code of EA000002. To distinguish the operand and opcode parts, we should compare the machine code with the B instruction format. In the following, you see the format of the B instruction. In this example the machine code is EA000002. If we compare it with the B format, we see that, the operand is 000002 and the opcode is EA. The 000002 is the offset, relative to the address of the next instruction. Recall that to calculate the target address, the relative address (offset) is shifted left twice and added to the current value of the PC (Program Counter). Shifting the offset (000002) left twice results in 000008 and then adding it to the address of the next instruction to be fetched (0000000C) we have 000008 + 0000000C = 00000014 which is exactly the address of THERE label. All the jump instructions, whose mnemonics begin with B, have the same instruction format, and the opcode changes from instruction to instruction. So, we can calculate the short branch address for any of them, as we just did in this example.



It must also be noted that for the backward branch the relative value is negative (2’s complement). That is shown in Example 4-7.

Example 4-7

Verify the calculation of backward jumps for the listing of Example 4-1, shown below.

```
LINE    ADDRESS        Machine        Mnemonic         Operand
5       00000000       E3A02FFA               LDR      R2,=1000 ;R2 = 1000
6       00000004       E3A00000               MOV      R0,#0     ;R0 = 0, sum
7       00000008       E2800009       AGAIN   ADD      R0,R0,#9 ;R0 = R0 + 9
8       0000000C       E2522001               SUBS     R2,R2,#1 ;R2 = R2 - 1
9       00000010       1AFFFFFC                        BNE      AGAIN    ;repeat
10      00000014       E1A04000               MOV      R4,R0     ;store the sum in R4
11      00000018       EAFFFFFE       HERE    B        HERE      ;stay here
12      0000001C                              END
```

Solution:

In the program list, “BNE AGAIN” in line 9 has machine code 1AFFFFFC. To specify the operand and opcode, we compare the instruction with the branch instruction format, which you saw in the previous example. The opcode is 1A and the operand (relative offset address) is FFFFFC. The FFFFFC gives us –4, which means the displacement is (–4 × 4 = –16 = **–0x10**).

The branch is located in address 0x0010. The address of the next instruction to be fetched is two instructions ahead of current branch instruction, and each instruction is 4-byte wide. Therefore, address of the next instruction to be fetched = 0x0010 + (2 × 4) = **0x0018**

When the relative address of –0x10 is added to 00000018, we have –0x0010 + 0x0018 = 0x08

Notice that 00000008 is the address of the label AGAIN.

FFFFFC is a negative number and that means it will branch backward. For further discussion of the addition of negative numbers, see Chapter 5.

Branching beyond 32M byte limit

To branch beyond the address space of 32M bytes, we use BX (branch and exchange) instruction. The “BX Rn” instruction uses register Rn to hold target address.

Since Rn can be any of the R0–R14 registers and they are 32-bit registers, the “BX Rn” instruction can land anywhere in the 4G bytes address space of the ARM. In the instruction “BX R2” the R2 is loaded into the program counter (R15) and CPU starts to fetch instructions from the target address pointed to by R15, the program counter. See Figure 4-6. Since the instructions are word aligned, we must make sure that the lower two bits of the Rn are 0s.

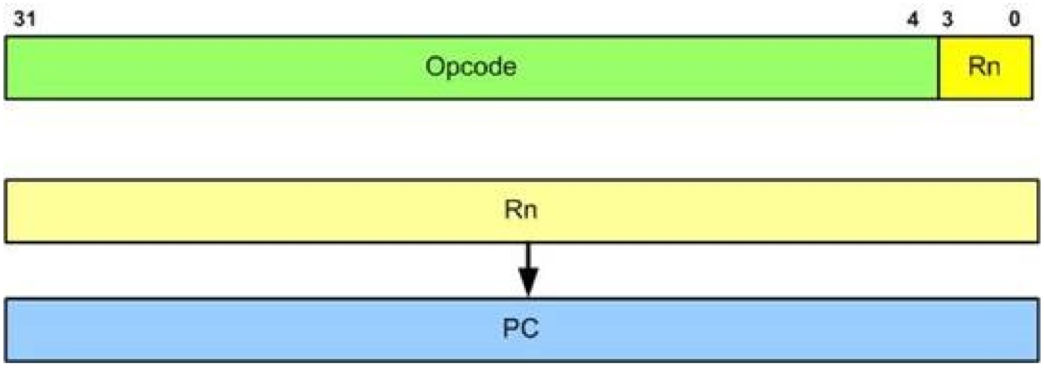


Figure 4- 6: BX (Branch and exchange) Instruction Target Address

The BX instruction is also used to switch to THUMB version of the ARM CPU. For more information see the ARM manual.

Review Questions

1. The mnemonic BNE stands for _______.
2. True or false. “BNE BACK” makes its decision based on the last instruction affecting the Z flag.
3. “BNE HERE” is a ___ -byte instruction.
4. In “BEQ NEXT”, which flag bit is checked to see if it is high?
5. B(ranch) is a(n) ___ -byte instruction.
6. Compare B and BX instructions.

Section 4.2: Calling Subroutine with BL

Another control transfer instruction is the BL (branch and link) instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the ARM there is only one instruction for call and that is BL (branch and link). To use BL instruction for call, we must leave the R14 register unused since that is where the ARM CPU stores the address of the next instruction where it returns to resume executing the program.

BL (Branch and Link) instruction and calling subroutine

In the 32-bit instruction BL, 8 bits are used for the opcode and the other 24 bits, k23–k0, are used for the address of the target subroutine, just like in the Branch instruction. Therefore, BL can be used to call subroutines located anywhere within the 32M address space of the ARM, as shown in Figure 4-7.

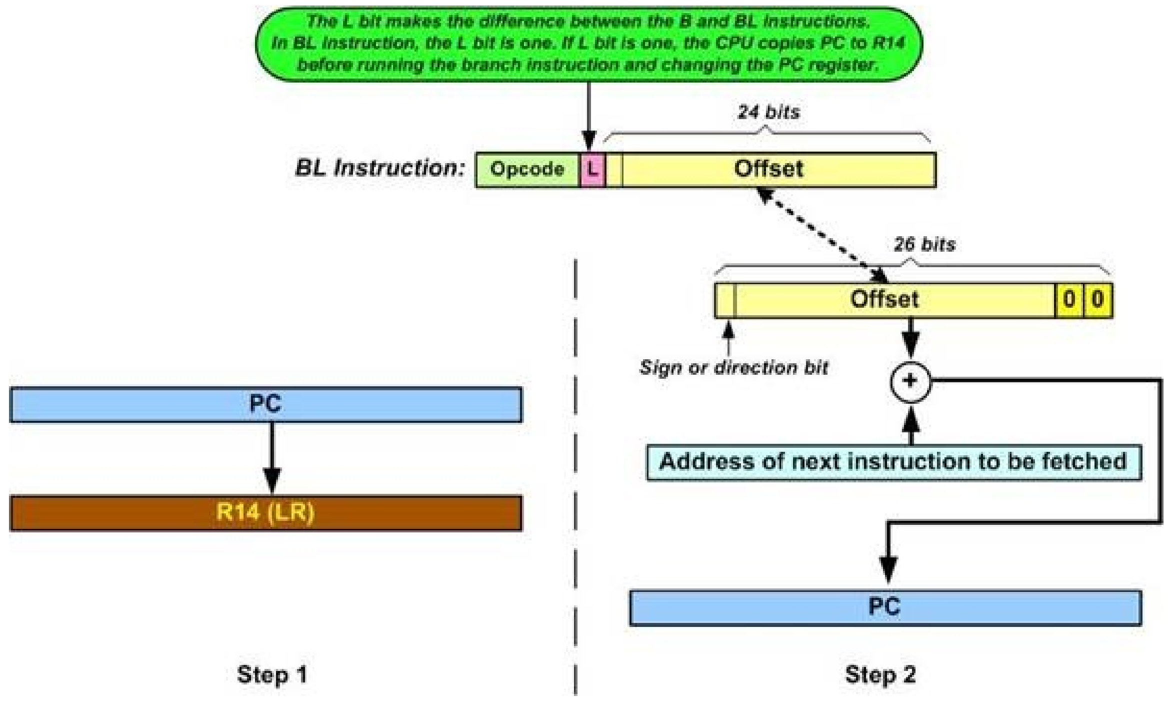

The L bit makes the difference between the B and BL instructions.
In BL instruction, the L bit is one. If L bit is one, the CPU copies PC to R14 before running the branch instruction and changing the PC register.
24 bits
BL Instruction:
Opcode
L
Offset
26 bits
Offset
0
0
Sign or direction bit
+
Address of next instruction to be fetched
PC
PC
R14 (LR)
Step 1
Step 2

Figure 4- 7: BL (Branch and Link) Instruction

To make sure that the ARM knows where to come back to after execution of the called subroutine, the ARM automatically saves in the link register (LR), the R14, the address of the instruction immediately below the BL. When a subroutine is called by the BL instruction, control is transferred to that subroutine, and the processor saves the PC (program counter) in the R14 register and begins to fetch instructions from the new location. After finishing execution of the subroutine, we must use “BX LR“ instruction to transfer control back to the caller. Every subroutine needs “BX LR” as the last instruction for return address.

BL instruction and the role of linker register

When a subroutine is called using BL instruction, first the processor saves the address of the instruction just below the BL instruction on the R14 register (LR, linker register), and then control is transferred to that subroutine. This is how the CPU knows where to resume when it returns from the called subroutine.

The linker register and returning from subroutine

There is no RETurn instruction in ARM7. Therefore at the end of the subroutine we must copy the linker register, R14, to the program counter (R15). When the BL instruction is executed the address of the instruction below the BL instruction is placed into R14 register, so, when the execution of the function finishes, the address of the instruction below the BL must be reloaded back into the PC, and so the CPU can resume the execution of instructions below the BL instruction.

To understand the role of the R14 register in BL instruction and the return, examine the Examples 4-8. The following points should be noted for the Example 4-8:

1. Notice the DELAY subroutine. Upon executing the first “BL DELAY”, the address of the instruction right below it, “MOV R0,#0xAA”, is saved onto the R14 register, and the ARM starts to execute instructions at DELAY subroutine.
2. In the DELAY subroutine, first the counter R3 is set to 5 (R3 = 5); therefore, the inner loop is repeated 5 times. When R3 becomes 0, control falls to the “BX LR” instruction, which restores the address into the program counter and returns to main program to resume executing the instructions after the BL.

Example 4-8

Write a program to toggle all the bits of address 0x40000000 by sending to it the values 0x55 and 0xAA continuously. Put a time delay between each issuing of data to address location.

Solution:

```
    AREA     EXAMPLE4_8, CODE, READONLY
    ENTRY
RAM_ADDR      EQU      0x40000000       ;change the address for your ARM
    LDR       R1,=RAM_ADDR              ;R1 = RAM address
AGAIN  MOV     R0,#0x55             ;R0 = 0x55
    STRB      R0,[R1]               ;send it to RAM
    BL        DELAY                 ;call delay (R14 = PC of next instruction)
    MOV      R0,#0xAA              ;R0 = 0xAA
    STRB      R0,[R1]               ;send it to RAM
    BL        DELAY                 ;call delay
    B         AGAIN                 ;keep doing it
    ;————————DELAY SUBROUTINE
```

```
DELAY   LDR       R3,=5                  ;R3 =5, modify this value for different size delay
L1          SUBS     R3,R3,#1            ;R3 = R3 - 1
    BNE       L1
    BX          LR                       ;return to caller
    ;————–end of DELAY subroutine
    END                     ;notice the place for END directive
```

Use Keil IDE simulator for ARM to simulate the above program and examine the registers and memory location 0x40000000. You might have to change the address 0x40000000 to some other value depending on the RAM address of the ARM chip you use.

In above program, in place of “BX LR” for return, we could have used “BX R14”, “MOV R15,R14”, or “MOV PC, LR” instructions. All of them do the same thing; but it is recommended to use the “BX LR” instruction.

The amount of time delay in Example 4-8 depends on the frequency of the ARM chip. How to calculate the time will be explained in the last section of this chapter.

Main Program and Calling Subroutines

In real world projects we divide the programs into small subroutines (also called functions) and the subroutines are called from the main program. Figure 4-8 shows the format.

```
    ;MAIN program calling subroutines
    AREA     PogramName, CODE, READONLY
    ENTRY
MAIN    BL          SUBR_1                          ;Call Subroutine 1
    BL          SUBR_2                          ;Call Subroutine 1
    BL          SUBR_3                          ;Call Subroutine 1
HERE      BAL        HERE                      ;stay here. BAL is the same as B
    ;—-end of MAIN

    ;————–SUBROUTINE 1
SUBR_1                          ….
                ….
    BX          LR          ;return to main
    ;—     end of subroutine 1
    ;————————SUBROUTINE 2
SUBR_2              ….
    ….
    BX         LR         ;return to main
    ;—     end of subroutine 2

    ;————————SUBROUTINE 3
SUBR_3              ….
    ….
    BX         LR         ;return to main
    ;—     end of subroutine 3
    END                   ;notice the END of file
```

Figure 4- 8: ARM Assembly Main Program That Calls Subroutines

Program 4-3 shows an example of the main program calling subroutine.

Program 4-3

```
    ;This program fills a block of memory with a fixed value and
    ;then transfers (copies) the block to new area of memory
    AREA     PROGRAM4_3, CODE, READONLY
    ENTRY
RAM1_ADDR     EQU       0x40000000        ;Change the address for your ARM
RAM2_ADDR     EQU       0x40000100        ;Change the address for your ARM
    BL           FILL                        ;call block fill subroutine
    BL           COPY                       ;call block transfer subroutine
HERE      BAL        HERE      ;BAL(branch always) is the same as B
    ;———-BLOCK FILL SUBROUTINE
FILL        LDR        R1,=RAM1_ADDR            ;R1 = RAM Address pointer
    MOV      R0,#10                                ;counter
    LDR        R2,=0x55555555
L1            STR        R2,[R1]                              ;send it to RAM
    ADD       R1,R1,#4                             ;R1 = R1 + 4 to increment pointer
    SUBS     R0,R0,#1                             ;R0 = R0 - 1 for dec counter
    BNE        L1                                        ;keep doing it
    BX      LR                              ;return to caller
    ;————BLOCK COPY SUBROUTINE
COPY  LDR   R1,=RAM1_ADDR        ;R1 = RAM Address pointer (source)
    LDR     R2,=RAM2_ADDR        ;R2 = RAM Address pointer  (dest.)
    MOV     R0,#10               ;counter
L2        LDR   R3,[R1]          ;get from RAM1
    STR     R3,[R2]              ;send it to RAM2
    ADD     R1,R1,#4             ;R1 = R1 + 4 to increment pointer for RAM1
    ADD     R2,R2,#4             ;R2 = R2 + 4 to increment pointer for RAM2
    SUBS    R0,R0,#1             ;R0 = R0 – 1 for decrementing counter
    BNE     L2                   ;keep doing it
    BX      LR                   ;return to caller
    ;——-
    END                          ;notice the place of END directive
```

There are cases in which we need to call a subroutine within a call (nested call). To do that we must use stack since the ARM CPU can support only one call at a time since there is only one linker register (R14).

Review Questions

1. The mnemonic BL stands for ________.
2. True or false. “BL DELAY” saves the address of the instruction below BL in LR register.
3. “BL DELAY” is a ___ -byte instruction.
4. LR is an ___ -bit register.
5. LR is the same as ______register.
6. Explain the difference between B and BL instructions.

Section 4.3: ARM Time Delay and Instruction Pipeline

In this section we discuss how to generate various time delays and calculate time delays for the ARM. We will also discuss instruction pipelining and its impact on execution time.

Delay calculation for the ARM

In creating a time delay using Assembly language instructions, one must be mindful of two factors that can affect the accuracy of the delay:

1. **The crystal frequency:** The frequency of the crystal oscillator connected to the CPU is one factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.
2. **The ARM design:** Since the 1970s, both the field of IC technology and the architectural design of microprocessors have seen great advancements. Due to the limitations of IC technology and limited CPU design experience for many years, the instruction cycle duration was longer. Advances in both IC technology and CPU design in the 1980s and 1990s have made the single instruction cycle a common feature of many microprocessors. Indeed, one way to increase performance without losing code compatibility with the older generation of a given family is to reduce the number of instruction cycles it takes to execute an instruction. One might wonder how microprocessors such as ARM are able to execute an instruction in one cycle. There are three ways to do that: (a) Use Harvard architecture to get the maximum amount of code and data into the CPU, (b) use RISC architecture features such as fixed-size instructions, and finally (c) use pipelining to overlap fetching and execution of instructions. We have examined the Harvard and RISC architectures in Chapter 2. Next, we give a brief discussion of pipelining. Chapter 7 covers the ARM pipeline in much more detail.

Pipelining

In early microprocessors such as the 8085, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, decode, and then execute it, and then fetch the next instruction, decode and execute it, and so on as shown in Figure 4-9. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time. That is an instruction is being fetched while the previous instruction is being executed.

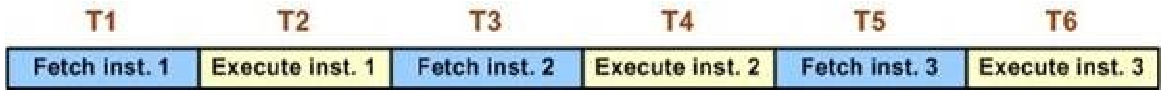


Figure 4- 9: Non-pipeline execution

We can use a pipeline to speed up execution of instructions. In pipelining, the process of executing instructions is split into small steps that are all executed in parallel. In this way, the execution of many instructions is overlapped. One limitation of pipelining is that the speed of execution is limited to the slowest stage of the pipeline. Compare this to making pizza. You can split the process of making pizza into many stages, such as flattening the dough, putting on the toppings, and baking, but the process is limited to the slowest stage, baking, no matter how fast the rest of the stages are performed. What happens if we use two or three ovens for baking pizzas to speed up the process? This may work for making pizza but not for executing programs, because in the execution of instructions we must make sure that the sequence of instructions is kept intact and that there is no out-of-step execution.

ARM multistage execution pipeline

As shown in Figure 4-10, in the ARM, each instruction is executed in 3 stages: Fetch, Decode, and Execute.

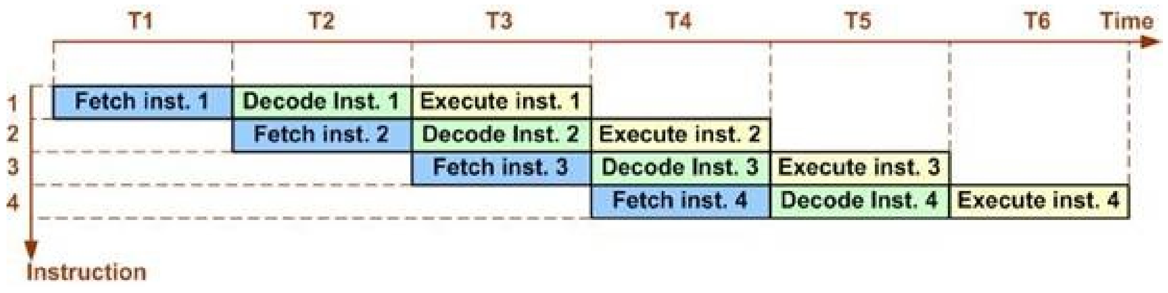


Figure 4- 10: Pipeline in ARM

In step 1, the opcode is fetched. In step 2, the opcode is decoded. In step 3, the instruction is executed and result is written into the destination register. This 3-stage pipeline was used in the original ARM. The newer version of ARM may have more stages of pipeline. See Chapter 7 and your ARM manual.

Instruction cycle time for the ARM

It takes a certain amount of time for the CPU to execute an instruction. This amount of time is referred to as *machine cycles*. Thanks to the RISC architecture, ARM executes most instructions in one machine cycle. In the ARM family, the length of the machine cycle depends on the frequency of the oscillator connected to the ARM system. The crystal oscillator, along with on-chip circuitry, provide the clock source for the ARM CPU. In the ARM, one machine cycle consists of one oscillator period, which means that with each oscillator clock, one machine cycle passes. Therefore, to calculate the machine cycle for the ARM, we take the inverse of the crystal frequency, as shown in Example 4-9.

Example 4-9

The following shows the crystal frequency for four different ARM-based systems. Find the period of the instruction cycle in each case.

(a) 80 MHz (b) 160 MHz (c) 100 MHz (d) 50 MHz

Solution:

(a) instruction cycle is 1/80 MHz = 0.0125 ms (microsecond) = 12.5 ns (nanosecond)

(b) instruction cycle = 1/160 MHz = 0.00625 ms = 6.25 ns

(c) instruction cycle = 1/100 MHz = 0.01 ms = 10 ns

(d) instruction cycle = 1/50 MHz = 0.02 ms = 20 ns

Branch penalty

The overlapping of fetch and execution of the instruction is widely used in today's microprocessors such as ARM. For the concept of pipelining to work, we need a buffer or queue in which an instruction is prefetched and ready to be executed. In some circumstances, the CPU must flush out the queue. For example, when a branch instruction is executed, the CPU starts to fetch codes from the new memory location and the code in the queue that was fetched previously is discarded. In this case, the execution unit must wait until the fetch unit fetches the new instruction. This is called a branch penalty. The penalty is an extra instruction cycle to fetch the instruction from the target location instead of executing the instruction right below the branch. Remember that the instruction below the branch has already been fetched and is next in line to be decoded when the CPU branches to a different address. This means that while the vast majority of ARM instructions take only one machine cycle, some instructions take three machine cycles. These are Branch, BL (call), and all the conditional branch instructions such as BNE, BLO, and so on. The conditional branch instruction can take only one machine cycle if it does not jump. For example, the BNE will jump if Z = 0 and that takes three machine cycles. If Z = 1, then it falls through and it takes only one machine cycle. See Examples 4-10 and 4-11.

Example 4-10

For an ARM system of 100 MHz, find how long it takes to execute each of the following instructions:

(a) MOV	(b) SUB	(c) B
(d) ADD	(e) NOP	(f) BHI
(g) BLO	(h) BNE	(i) EQU

Solution:

The machine cycle for a system of 100 MHz is 10 ns, as shown in Example 4-9.
Therefore, we have:

Instruction	*Instruction cycles*	*Time to execute*
(a) MOV	1	1 × 10 ns = 10 ns

(b) SUB	1	1 × 10 ns = 10 ns
(c) B	3	3 × 10 ns = 30 ns
(d) ADD	1	1 × 10 ns = 10 ns
(e) NOP	1	1 × 10 ns = 10 ns

For the following, due to branch penalty, 3 clock cycles if taken and 1 if it falls through:

(f) BHI	3/1	3 × 10 ns = 30 ns
(g) BLO	3/1	3 × 10 ns = 30 ns
(h) BNE	3/1	3 × 10 ns = 30 ns
(i) EQU	0	(directives do not produce machine instructions)

Notice that ARM chip does not have the NOP instruction. The ARM Assembler replaces it with some other 1-cycle instruction. On your ARM Assembler, you might get a warning that NOP does not exist, but it still compiles the program. It does not give an error.

Delay calculation for ARM

A delay subroutine consists of two parts: (1) setting a counter, and (2) a loop. Most of the time delay is performed by the body of the loop, as shown in Example 4-11.

Example 4-11

Find the size of the delay of the code snippet below if the crystal frequency is 100 MHz:

```
DELAY   MOV     R0,#255
AGAIN  NOP
    NOP
    SUBS    R0,R0,#1
    BNE      AGAIN
    MOV     PC,LR                ;return
```

Solution:

We have the following machine cycles for each instruction of the DELAY subroutine:

Instruction   Machine Cycle

```
DELAY   MOV     R0,#255              ;            1
AGAIN  NOP                            ;            1
    NOP                             ;            1
    SUBS     R0,R0,#1           ;            1
    BNE       AGAIN              ;            3/1
    MOV     PC,LR                 ;            1
```

Therefore, we have a time delay of [1 + ((1+1+1+3) × 255) + 1] × 10 ns = 15,320 ns.

Notice that BNE takes three instruction cycles if it jumps back, and takes only one cycle when falling through the loop. That means the above number should be 153.0 ns. Because the last time, when R0 is zero, the BNE takes only one cycle because it falls through the loop

---

Very often we calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

In Example 4-11, the largest value the R0 register can take is 2^{32} = 4G. One way to increase the delay is to use NOP instructions in the loop. NOP, which stands for "no operation," simply wastes time, but takes 4 bytes of program memory and that is too heavy a price to pay for just one instruction cycle. A better way is to use a nested loop.

Loop inside a loop delay

Another way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*. See Example 4-12.

Example 4-12

In a given ARM trainer an I/O port is connected to 8 LEDs. The following program toggles the LEDs by sending to it 0x55 and 0xAA values continuously. Calculate the time delay for toggling of LEDs. Assume the system clock frequency of 100 MHz.

Solution:

```
    AREA     Example4_12, CODE, READONLY
    ENTRY
PORT_ADDR       EQU       0x40000000        ;change the address for your ARM
    LDR        R1,=PORT_ADDR             ;R1 = port address
AGAIN  MOV     R0,#0x55             ;R0 = 0x55
```

```
    STRB     R0,[R1]            ;send it to LEDs
    BL       DELAY              ;call delay
    MOV      R0,#0xAA           ;R0 = 0xAA
    STRB     R0,[R1]            ;send it to LEDs
    BL       DELAY              ;call delay
    BAL      AGAIN              ;keep doing it forever (BAL is the same as B)

    ;————————DELAY SUBROUTINE
DELAY
    MOV      R3,#100  ;R3 = 100,modify this value for different size delay
L1       LDR      R4,=250000        ;R4 = 250,000 (inner loop count)
L2       SUBS     R4,R4,#1          ;1 clock
    BNE      L2                 ;3 clock
    SUBS     R3,R3,#1           ;R3 = R3 - 1
    BNE      L1
    MOV      PC,LR              ;return to caller
    END
```

Ignoring the delay associated with the outer loop, we have the following time delay:

[(1 + 3) × 250,000 × 100] × 10 ns = 1 second since 1/100 MHz = 10 ns.

Examine the working frequency for your ARM trainer, change the above address 0x40000000 to your ARM trainer port address and verify the time delay using oscilloscope.

From these discussions we conclude that the use of instructions in generating time delay is not the most reliable method. To get more accurate time delay Timers are used. All ARM microcontrollers come with on-chip Timers. We can use Keil uVision's simulator to verify delay time and number of cycles used. Meanwhile, to get an accurate time delay for a given ARM microcontroller, we must use an oscilloscope to measure the exact time delay.

Review Questions

1. True or false. In the ARM, the machine cycle lasts 1 clock period of the crystal frequency.

2. The minimum number of machine cycles needed to execute an ARM instruction is ______.
3. Find the machine cycle for a crystal frequency of 66 MHz.
4. Assuming a crystal frequency of 100 MHz, find the time delay associated with the loop section of the following DELAY subroutine:

```
DELAY       LDR     R2,=50000000
HERE        NOP
        NOP
        NOP
        NOP
        NOP
        SUBS    R2,R2,#1
        BNE     HERE
        MOV     PC,LR
```

5. Find the machine cycle for an ARM if the crystal frequency is 50 MHz.
6. True or false. In the ARM, the instruction fetching and execution are done at the same time.
7. True or false. B and BL will always take 2 machine cycles.
8. True or false. The BNE instruction will always take 3 machine cycles.

Section 4.4: Conditional Execution

Every microprocessor has the conditional branch (jump) instruction based on the status of flag bits such as Z and C. Instructions such as BEQ (branch equal, Z = 1) or BNC (branch if no carry, C = 0) are common in all CPUs. The ARM CPU has a unique feature that we do not see in other microprocessors. In ARM, the concept of conditional execution is implemented for all instruction and not just for branch which makes you able to decide to run or ignore each single instruction depending on the status of flag bits. In other words, not only the branch instruction but all of the ARM instructions can be conditional. As we discussed in Chapters 2 and 3, the ADD, SUB, and other arithmetic instruction do not affect the flag bits in CPSR (current program status register) register unless they have letter S in the syntax. The default is not to update the flags. We override the default by having letter S in the instruction. The same thing is true about conditional field of each instruction. If we do not add a condition after an instruction, it will be executed unconditionally because the default is not to check the flags and execute unconditionally. If we want an instruction to be executed only when a condition is met, we put the condition syntax right after the instruction.

This feature of ARM allows the execution of an instruction conditionally based on the status of Z, C, V and N flags. To do that, the ARM instructions have set aside the most significant 4 bits of the instruction field for the conditions. See Figure 4-11. The 4 bits gives us 16 possible conditions. Table 4-4 shows the list of all the 16 possible conditions.

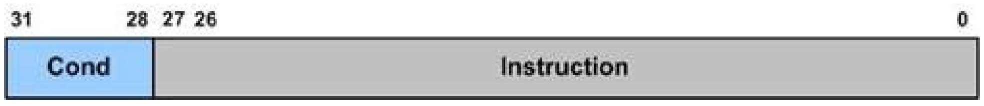


Figure 4- 11: Condition Field in ARM Instructions

Bits	Mnemonic Extension	Meaning	Flag
0000	EQ	Equal	Z = 1
0001	NE	Not equal	Z = 0
0010	CS/HS	Carry Set/Higher or Same	C = 1
0011	CC/LO	Carry Clear/Lower	C = 0
0100	MI	Minus/Negative	N = 1
0101	PL	Plus	N = 0
0110	VS	V Set (Overflow)	V = 1
0111	VC	V Clear (No Overflow)	V = 0
1000	HI	Higher	C = 1 and Z = 0
1001	HS	Lower or Same	C = 1 and Z = 1

1010	GE	Greater than or Equal	N = V
1011	LT	Less than	N ≠ V
1100	GT	Greater than	Z = 0 and N = V
1101	LE	Less than or Equal	Z = 0 or N ≠ V
1110	AL	Always (unconditional)	
1111	–	Not Valid	

Table 4- 4: ARM Condition codes for the Opcode bits [31-28]

Note!

By default, all the instructions are executed unconditionally. As a result the AL (Always) suffix has no effect on the instruction. For example, BAL (Branch Always) is exactly the same as B (Branch). The same is true for all instructions.

To make an instruction conditional, simply we put the condition syntax from Table 4-4 in front of it. See the following examples:

```
MOV     R1,#10  ;R1 = 10
MOV     R2,#12  ;R2 = 12
CMP     R2,R1     ;compare 12 with 10, Z = 0 because they are not equal
MOVEQ               R4,#20  ;this line is not executed because
; the condition EQ is not met
```

The following code adds 10 to R1 if it is not zero:

```
CMP     R1,#0                  ;compare R1 with 0
ADDNE R1,R1,#10           ;this line is executed if Z = 0
                               ;(if in the last CMP operands were not equal)
```

Note that we can add both S and condition to syntax of an instruction. It is common to put S after the condition. See the following examples:

```
ADDNES              R1,R1,#10
;this line is executed and set the flags if Z = 0
```

One advantage of using conditional execution is it saves time in the execution of an instruction by avoiding branch penalty. As we discussed earlier, each instruction goes through three pipeline stages of fetch, decode, and execution. When a branch instruction such as BCC (branch if C cleared) enters the pipeline, the instruction below is also fetched into the CPU not knowing whether the C flag is high or low. If the C = 0, the branch will jump to new location resulting in emptying the instruction pipeline queue which was working on the instruction below the BCC instruction. This penalty can be avoided by using conditional execution feature of the ARM. We can make the execution of many of the ARM instructions conditional by simply adding a mnemonic extension to the end of it. For example, we can use ADDCS instead of ADD. The ADDCS means add if C=1. (ADDCS: Add if Carry Set). By the same token the ADDEQ means add only if Z=1.

Example 4-13 shows two versions of the Example 4-4 we covered earlier. In the new version, we use the conditional execution instructions. Simulate and compare both versions to see how the conditional instructions are executed.

Example 4-13

The following program adds 0x99999999 together 10 times. Compare the code syntax to see how the conditional execution of the code is used.

Code 1: (Example 4-4 without conditional execution)

```
    AREA    EXAMPLE4_4, CODE, READONLY
    ENTRY
    MOV     R1,#0               ;clear high word (R1 = 0)
    MOV     R0,#0               ;clear low word (R0 = 0)
    LDR     R2,=0x99999999      ;R2 = 0x99999999
    MOV     R3,#10              ;counter
L1      ADDS    R0,R0,R2        ;R0 = R0 + R2 and update the flags
    BCC     NEXT                ;if C = 0, go to next number
    ADD     R1,R1,#1            ;if C = 1, increment the upper word
NEXT    SUBS    R3,R3,#1        ;R3 = R3 - 1 and update the flags
    BNE     L1                  ;next round if z = 0
HERE    B       HERE            ;stay here
    END
```

Code 2: (Example 4-4 with conditional execution)

```
    AREA    EXAMPLE4_13,  CODE, READONLY
    ENTRY
    MOV     R1,#0               ;clear high word (R1 = 0)
    MOV     R0,#0               ;clear low word (R0 = 0)
    LDR     R2,=0x99999999      ;R2 = 0x99999999
    MOV     R3,#10              ;counter
```

```
L1          ADDS     R0,R0,R2            ;R0 = R0 + R2 and update the flags
    ADDCS  R1,R1,#1            ;if C set (C = 1),increment the upper word
NEXT     SUBS     R3,R3,#1            ;R3 = R3 - 1 and update the flags
    BNE       L1                       ;next round if z = 0
HERE     B            HERE                  ;stay here
    END
```

See also Examples 4-14 and 4-15.

Example 4-14

Rewrite the main part of Program 4-1 using conditional execution of ARM instructions

Solution:

```
    MOV     COUNT,#5          ;COUNT = 5
    MOV     MAX,#0                          ;MAX = 0
    LDR       POINTER,=MYDATA
    ;POINTER = MYDATA (address of first data)
AGAIN
    LDR       NEXT,[POINTER]
    ;load contents of POINTER location to NEXT
    CMP      MAX,NEXT         ;compare MAX and NEXT
    MOVLO              MAX,NEXT
    ;if MAX is lower than NEXT then MAX=NEXT
    ADD       POINTER,POINTER,#4
    ;POINTER = POINTER  +  4 to point to next
    SUBS     COUNT,COUNT,#1         ;decrement counter
    BNE       AGAIN                 ;branch AGAIN if counter is not zero
```

Example 4-15

Using rotation, write a program that counts the number of 1s in R0.

Solution:

```
    AREA    EXAMPLE4_15, CODE, READONLY
    ENTRY
    LDR      R0,=0x34F37D36
    MOV     R1, #0                  ;number of 1s
    MOV     R2, #32 ;counter
BEGIN   MOV     R0,R0,RRX          ;Rotate right with carry the R0 register
    ADDCS  R1,R1,#1             ;if C = 1 then increment R1
    SUBS     R2,R2,#1             ;decrement counter
    BNE       BEGIN                  ;if counter is not equal to zero branch BEGIN
    END
```

Review Questions

1. True or false. All the ARM instructions have the conditional execution feature.
2. How many bits of the ARM instruction are set aside for the condition codes?
3. The ADDAL stands for…………….. .
4. True or false. MOVVC is a valid instruction in ARM
5. True or false. SUBEQS is a valid instruction in ARM

Problems

Section 4.1: Looping and Branch Instructions

1. In the ARM, looping action using a single register is limited to _______ iterations.
2. If a conditional branch is not taken, what is the next instruction to be executed?
3. In calculating the target address for a branch, a displacement is added to the contents of register _______.
4. The mnemonic BNE stands for __________.
5. What is the advantage of using BX over B?
6. True or false. The target of a BNE can be anywhere in the 4G word address space.
7. True or false. All ARM branch instructions can branch to anywhere in the 4G byte address space.
8. Dissect the B instruction, indicating how many bits are used for the operand and the opcode, and indicate how far it can branch.
9. True or false. All conditional branches are 2-byte instructions.
10. Show code for a nested loop to perform an action 10,000,000,000 times.
11. Show code for a nested loop to perform an action 200,000,000,000 times.
12. Find the number of times the following loop is performed:

```
MOV  R0,#0x55
MOV  R2,#40
L1          LDR  R1,=10000000
L2          EOR  R0,R0,#0xFF
SUB  R1,R1,#1
BNE  L2
SUB  R2,R2,#1
BNE  L1
```

13. Indicate the status of Z and C after CMP is executed in each of the following cases.

(a)
```
MOV     R0,#50
MOV     R1,#40
CMP     R0,R1
```

(b)
```
MOV     R1,#0xFF
MOV     R2,#0x6F
CMP     R1,R2
```

(c)
```
MOV     R2,#34
MOV     R3,#88
CMP     R2,R3
```

(d)

```
SUB  R1,R1,R1
MOV     R2,#0
CMP     R1,R2
```

(e)

```
EOR      R2,R2,R2
MOV     R3,#0xFF
CMP     R2,R3
```

(f)

```
EOR      R0,R0,R0
EOR      R1,R1,R1
CMP     R0,R1
```

(g)

```
MOV R4,#0x78
MOV     R2,#0x40
CMP     R4,R2
```

(h)

```
MOV     R0,#0xAA
AND      R0,R0,#0x55
CMP     R0,#0
```

14. Rewrite Program 4-1 to find the lowest grade in that class.
15. The target address of a BNE is backward if the relative address portion of opcode is ________ (negative, positive).
16. The target address of a BNE is forward if the relative address portion of opcode is ________ (negative, positive).

Section 4.2: Calling Subroutine with BL

17. BL is a(n) ___-byte instruction.
18. In ARM, which register is the linker register?
19. True or false. The BL target address can be anywhere in the 4G byte address space.
20. Describe how we can return from a subroutine in ARM.
21. In ARM, which address is saved when BL instruction is executed.

Section 4.3: ARM Time Delay and Instruction Pipeline

22. Find the oscillator frequency if the machine cycle = 1.25 ns.
23. Find the machine cycle if the crystal frequency is 200 MHz.
24. Find the machine cycle if the crystal frequency is 100 MHz.
25. Find the machine cycle if the crystal frequency is 160 MHz.
26. Find the time delay for the delay subroutine shown below if the system has an ARM with a frequency of 80 MHz:

```
MOV     R8,#200
BACK   LDR       R1,=400000000
```

```
 HERE    NOP
   SUBS     R1,R1,#1
   BNE       HERE
        SUBS     R8,R8,#1
   BNE       BACK
```

27. Find the time delay for the delay subroutine shown below if the system has an ARM with a frequency of 50 MHz:

```
   MOV     R2,#100
 BACK   LDR       R0,=50000000
 HERE    NOP
   NOP
   SUBS     R0,R0,#1
   BNE       HERE
        SUBS     R2,R2,#1
   BNE       BACK
```

28. Find the time delay for the delay subroutine shown below if the system has a ARM with a frequency of 40 MHz:

```
   MOV     R1,#200
 BACK   LDR       R0,#20000000
 HERE    NOP
   NOP
   NOP
   SUBS     R0,R0,#1
                BNE       HERE
   SUBS     R1,R1,#1
   BNE       BACK
```

29. Find the time delay for the delay subroutine shown below if the system has an ARM with a frequency of 100 MHz:

```
  MOV     R8,#500
BACK   LDR       R1,=20000
HERE    NOP
  NOP
  NOP
  SUBS     R1,R1,#1
  BNE       HERE
  SUBS     R8,R8,#1
  BNE       BACK
```

30. The ARM chip does not have the NOP instruction. Examine the list file generated by your ARM assembler to see what instruction is used in place of the NOP.

Section 4.4: Conditional Execution

31. Which bits of the ARM instruction are set aside for condition execution?
32. True or false. Only ADD and MOV instructions have conditional execution feature.
33. True or false. In ARM, the conditional execution is default.
34. Which flag bit is examined before the MOVEQ instruction is executed?
35. State the difference between the ADDEQ and ADDNE instructions.
36. State the difference between the BAL and B instructions .
37. State the difference between the SUBCC and SUBCS instructions.
38. State the difference between the ANDEQ and ANDNE instructions.
39. True or false. The decision to execute the SUBCC is based on the status of Z flag.
40. True or false. The decision to execute the ADDEQ is based on the status of Z flag.

Answers to Review Questions

1. Branch if not Equal
2. True
3. 4
4. Z flag of CPSR (status register)
5. 4
6. The B can only branch to an address location within 32 MB address space, while the BX can go anywhere in the 4 GB address space of ARM.

Section 4.2: Calling Subroutine with BL

1. Branch and Link
2. True
3. 4
4. 32
5. R14
6. In both of them the target address is relative to the value of the program counter and the relative address can cover memory space of –32MB to +32MB from current location of program counter. The BL instruction saves the address of the next instruction in the LR register before jumping, while the B instruction just jumps without saving anything.

Section 4.3: ARM Time Delay and Instruction Pipeline

1. True
2. 1
3. MC = 1/66 MHz = 0.015 ms = 15 ns
4. [50,000,000 × (1 + 1 + 1 + 1 + 1 + 1 + 3)] × 10 ns = 4.5 seconds
5. Machine Cycle = 1 / 50 MHz = 0.02 ms = 20 ns
6. True
7. False
8. False. It takes 3 cycles, only if it branches to the target address.

Section 4.4: Conditional Execution

1. True
2. 4 bits
3. ADD always regardless of the status flag.

4. True
5. True

Chapter 5: Signed Numbers and IEEE 754 Floating Point

This chapter deals with signed number instructions and operations. In Section 5.1, we focus on the concept of signed numbers in software engineering. Signed number arithmetic operations and instructions are explained along with examples in Section 5.2. The IEEE 754 floating point will be explained in Section 5.3.

Section 5.1: Signed Numbers Concept

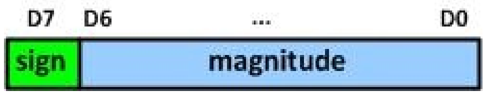
All data items used so far have been unsigned numbers, meaning that the entire 8-bit, 16-bit or 32-bit operand was used for the magnitude. Many applications require signed data. In this section the concept of signed numbers is discussed.

Concept of signed numbers in computers

In everyday life, numbers are used that could be positive or negative. For example, a temperature of 5 degrees below zero can be represented as -5, and 20 degrees above zero as +20. Computers must be able to accommodate such numbers. To do that, computer scientists have devised the following arrangement for the representation of signed positive and negative numbers: The most significant bit (MSB) is set aside for the sign (+ or -) and the rest of the bits are used for the magnitude. The sign is represented by 0 for positive (+) numbers and 1 for negative (-) numbers. Signed byte and word representations are discussed below.

Signed byte operands

In signed byte operands, D7 (MSB) is the sign and D0 to D6 are set aside for the magnitude of the number. If D7 = 0, the operand is positive, and if D7 = 1, it is negative.



The range of positive numbers that can be represented by the above format is 0 to +127.

Dec.	Binary
0	0000 0000
+1	0000 0001
…	…. ….
+5	0000 0101
…	…. ….
+127	0111 1111

Dec.	Binary
-0	1000 0000
-1	1000 0001
…	…. ….
-5	1000 0101
…	…. ….
-127	1111 1111

Negative numbers using 2’s complement

To save ALU circuitry, we use 2’s complement method to implement the negative numbers. For negative numbers, D7 is 1 but the non-sign (magnitude) portion is represented in 2’s complement. Although the assembler does the conversion, it is still important to understand how the conversion works. To convert to negative number representation (2’s complement), follow these steps:

1. Write the magnitude of the number in 8-bit binary (no sign).
2. Invert each bit.
3. Add 1 to it.

Dec.	Binary
0	0000 0000
+1	0000 0001
…	…. ….
+5	0000 0101
…	…. ….
+127	0111 1111

Dec.	Binary
0	0000 0000
-1	1111 1111
…	…. ….
-5	1111 1011
…	…. ….
-127	1000 0001
-128	1000 0000

Examples 5-1, 5-2, and 5-3 demonstrate these three steps.

Example 5-1

Show how the computer would represent -5.

Solution:

1. 0000 0101	5 in 8-bit binary
2. 1111 1010	invert each bit
3. 1111 1011	add 1 (0xFB)

This is the signed number representation in 2's complement for -5.

Example 5-2

Show -34 hex as it is represented internally.

Solution:

1. 0011 0100
2. 1100 1011
3. 1100 1100 (which is 0xCC)

Example 5-3

Show the representation for -128_{10}.

Solution:

1. 1000 0000
2. 0111 1111
3. 1000 0000 Notice that this is not negative zero (–0).

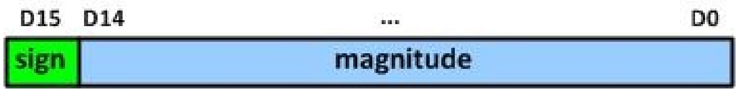
From the examples above it is clear that the range of byte-sized negative numbers is -1 to -128. The following lists byte-sized signed number ranges:

Decimal	Binary	Hex
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
…	…. …	..
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
	0000 0001	01
+2	0000 0010	02
…	…. ….	..

+127	0111 1111	7F

Halfword-sized signed numbers

In ARM CPU a half-word is 16 bits in length. Setting aside the MSB (D15) for the sign leaves a total of 15 bits (D14–D0) for the magnitude. This gives a range of –32,768 (–2^{15}) to +32,767 (2^{15}–1).



If a number is larger than 16-bit, it must be treated as a 32-bit word operand. The following shows the range of signed half-word operands. To convert a negative number to its half-word operand representation, the steps discussed in negative byte operands are used.

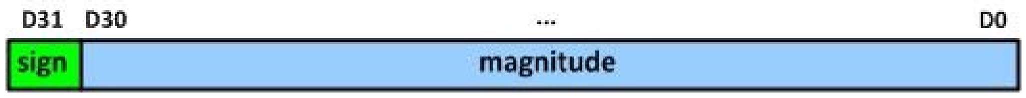
Decimal	Binary	Hex
-32,768	1000 0000 0000 0000	8000
-32,767	1000 0000 0000 0001	8001
-32,766	1000 0000 0000 0010	8002
…	…. …	..
-2	1111 1111 1111 1110	FFFE
-1	1111 1111 1111 1111	FFFF
0	0000 0000 0000 0000	0000
+1	0000 0000 0000 0001	0001
+2	0000 0000 0000 0010	0002
…	…. ….	…
+32,766	0111 1111 1111 1110	7FFE
+32,767	0111 1111 1111 1111	7FFF

Using Microsoft Windows calculator for signed numbers

All Microsoft Windows operating systems come with a handy calculator. Use it to verify the signed number operations in this section.

Word-sized signed numbers

In ARM CPUs, a word is 32 bits in length. Setting aside the MSB (D31) for the sign leaves a total of 31 bits (D30–D0) for the magnitude. This gives a range of -(2^{31}) to +(2^{31}-1).



To convert a negative number to its word operand representation, the three steps discussed in negative byte operands are used. See Example 5-4.

Example 5-4

Show how the computer would represent -5 for (a) 8-bit, (b) 16-bit, and (c) 32-bit data sizes.

Solution:

(a) 8-bit

1. 0000 0101 5 in 8-bit binary
2. 1111 1010 invert each bit
3. 1111 1011 add 1 (0xFB)

(b) 16-bit

1. 0000 0000 0000 0101 5 in 16-bit binary
2. 1111 1111 1111 1010 invert each bit
3. 1111 1111 1111 1011 add 1 (0xFFFB)

(c) 32-bit

1. 0000 0000 0000 0000 0000 0000 0000 0101 5 in 32-bit binary
2. 1111 1111 1111 1111 1111 1111 1111 1010 invert each bit
3. 1111 1111 1111 1111 1111 1111 1111 1011 add 1 (0xFFFFFFFB)

Use the Windows calculator to verify these examples.

If a number is larger than 32-bit, it must be treated as a 64-bit doubleword operand and be processed chunk by chunk the same way as unsigned numbers. The following shows the range of signed word operands.

Decimal	Binary	Hex

-2,147,483,648	10000000000000000000000000000000	80000000
-2,147,483,647	10000000000000000000000000000001	80000001
-2,147,483,646	10000000000000000000000000000010	80000002
…	…	…
-2	11111111111111111111111111111110	FFFFFFFE
-1	11111111111111111111111111111111	FFFFFFFF
0	00000000000000000000000000000000	00000000
+1	00000000000000000000000000000001	00000001
+2	00000000000000000000000000000010	00000002
…	…	…
+2,147,483,646	01111111111111111111111111111110	7FFFFFFE
+2,147,483,647	01111111111111111111111111111111	7FFFFFFF

Table 5-1 shows a summary of signed data ranges.

Data Size	Bits	2^n	Decimal	Hexadecimal
Byte	8	-2^7 to $+2^7-1$	-128 to +127	0x80–0x7F
Half-word	16	-2^{15} to $+2^{15}-1$	-32,768 to +32,767	0x8000–0x7FFF
Word	32	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647	0x80000000–0x7FFFFFFF

Table 5-1: Signed Data Range Summary

Review Questions

1. In an 8-bit operand, bit ______ is used for the sign bit, whereas in a 16-bit operand, bit ______ is used for the sign bit. Repeat for 32-bit signed data.
2. Compute the byte-sized 2’s complement of 0x16.
3. The range of byte-sized signed operands is -_______ to +______. The range of half word-sized signed operands is -___________ to +___________.
4. The range of word-sized signed operands is -___________ to +___________.
5. Compute the 2’s complement of 0x500000.

Section 5.2: Signed Number Instructions and Operations

In this section we examine issues associated with signed number arithmetic operations. We will also discuss the ARM instructions for signed numbers and how to use them. It must be noted that in ARM the N flag bit in CSPR is the sign bit. N=0 is for positive and N=1 for negative numbers.

Overflow problem in signed number operations

When using signed numbers, a serious problem arises that must be dealt with. This is the overflow problem. The CPU indicates the existence of the problem by raising the V (oVerflow) flag, but it is up to the programmer to take care of it. The CPU understands only 0s and 1s and ignores the human convention of positive and negative numbers. Now what is an overflow? If the result of an operation on signed numbers is too large for the register, an overflow occurs and the programmer must be notified. Look at Example 5-5.

Example 5-5

Look at the following case for 8-bit data size:

```
 + 96     0110 0000
 + 70    +0100 0110
 ----    ----------
+166      1010 0110
```

According to the CPU, this is -90, which is wrong. (V = 1, N = 1, C = 0)

In the example above, +96 is added to +70 and the result according to the CPU is -90. Why? The reason is that the result was more than 8 bits could handle. The 8-bit registers can only contain up to +127. The designers of the CPU created the overflow flag specifically for the purpose of informing the programmer that the result of the signed number operation is erroneous.

When the overflow flag is set in 8-bit operations

In 8-bit signed number operations, V is set to 1 if either of the following two conditions occurs:

1. There is a carry from D6 to D7 but no carry out of D7 (C = 0).
2. There is a carry from D7 out (C = 1) but no carry from D6 to D7.

In other words, the overflow flag is set to 1 if there is a carry from D6 to D7 or from D7 out, but not both. This means that if there is a carry both from D6 to D7 and from D7 out, V = 0. In Example 5-5, since there is only a carry from D6 to D7 and no carry from D7 out, V = 1. Examples 5-6, 5-7, and 5-8 give further illustrations of the overflow flag in signed number arithmetic.

Example 5-6

Examine the following case:

```
   - 128     1000 0000
 +    -2    +1111 1110
   - 130     0111 1110   V=1, N=0 (positive), C=1
```

According to the CPU, the result is +126, which is wrong. The error is indicated by the fact that V = 1.

Example 5-7

Observe the results of the following:

```
;assume R3=+7 (R3=0x07)
;assume R2=+18 (R2=0x12)
ADD R2,R2,R3    ;(R2=0x19=+25, correct)
```

```
    +7     0000 0111
 + +18    +0001 0010
   +25     0001 1001
```

V = 0, C = 0, and N = 0 (positive).

Example 5-8

Observe the results of the following:

```
   -2    1111 1110
 + -5    1111 1011
   -7    1111 1001
```

V = 0, C = 0, and N = 1 (negative); the result is correct since V = 0.

Overflow flag in 16-bit operations

In a 16-bit operation, V is set to 1 in either of two cases:

1. There is a carry from D14 to D15 but no carry out of D15 (C = 0).
2. There is a carry from D15 out (C = 1) but no carry from D14 to D15.

Again the overflow flag is low (not set) if there is a carry from both D14 to D15 and from D15 out. The V is set to 1 only when there is a carry from D14 to D15 or from D15 out, but not from both. See Examples 5-9 and 5-10.

Example 5-9

Observe the results in the following 16-bit hex numbers:

```
   +6E2F   0110 1110 0010 1111
 + +13D4   0001 0011 1101 0100
                                = –0x7DFD = –32,253 incorrect!
   +8203   1000 0010 0000 0011  V = 1, C = 0, N = 1
```

Example 5-10

Observe the results in the following 16-bit hex numbers:

```
   +542F     0101 0100 0010 1111
 + +12E0    +0001 0010 1110 0000
                                   = +0x670F = +26,383 (correct answer);
   +670F     0110 0111 0000 1111
                                   V = 0, C = 0, N = 0
```

Overflow flag in 32-bit operations

In a 32-bit operation, V is set to 1 in either of two cases:

1. There is a carry from D30 to D31 but no carry out of D31 (C = 0).
2. There is a carry from D31 out (C = 1) but no carry from D30 to D31.

Again the overflow flag is low (not set) if there is a carry from both D30 to D31 and from D31 out. The V is set to 1 only when there is a carry from D30 to D31 or from D31 out, but not from both. See Examples 5-11 and 5-12.

Example 5-11

Observe the results in the following 32-bit hex numbers:

+6E2F356F   0110 1110 0010 1111 0011 0101 0110 1111

+ +13D49530   +0001 0011 1101 0100 1001 0101 0011 0000

+8203CA9F   1000 0010 0000 0011 1100 1010 1001 1111   = –0x7DFC3561

incorrect! V = 1, C = 0, N = 1

Example 5-12

Observe the results in the following 32-bit hex numbers:

+542F356F   0101 0100 0010 1111 0011 0101 0110 1111

+ +12E09530   0001 0010 1110 0000 1001 0101 0011 0000

+670FCA9F   0110 0111 0000 1111 1100 1010 1001 1111   = +670FCA9F

correct answer; V = 0, C = 0, N = 0

Sign extension and avoiding erroneous results in signed number operations

To avoid the problems associated with signed number operations, one can sign-extend the operand. Sign extension copies the sign bit (D7) of the lower byte of a register into the upper bits of the 32-bit register, or copies the sign bit of a 16-bit register into 32-bit. The LDRSB (load register signed byte) instruction loads into the register a byte from memory and sign extends the D7 to the entire 32-bit register. The LDRSH (load register signed half-word) instruction loads into the register a half-word from memory and sign extends the D15 to the entire 32-bit register. They work as follows:

LDRSB loads into the destination register a byte from memory and sign extends (copy D7, the sign flag) to all 32 bits. This is illustrated in Figure 5-1.

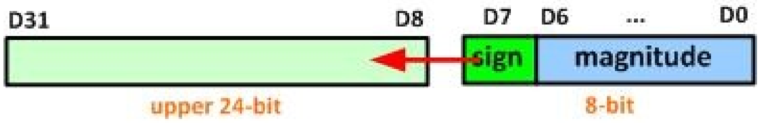


Figure 5-1: Sign Extending a Byte

Look at the following example:

```
;assume location 0x80000 has +96 = 0110 0000 and R1=0x80000
LDRSB   R0,[R1] ;now R0 =  00000000000000000000000001100000
;assume location 0x80000 contains -2 = 1111 1110 and R2=0x80000
LDRSB   R4,[R2]  ;now R4 = 11111111111111111111111111111110
```

As can be seen in the above examples, LDRSB does not alter the lower 8 bits. The sign of 8 bits is copied to the rest of the 32-bit register.

LDRSH loads the destination register with a 16-bit signed number and sign-extends to the 32-bit register. It copies D15 of Rd to all bits of the Rd register. This is used for signed half-word operand and is illustrated in Figure 5-2.

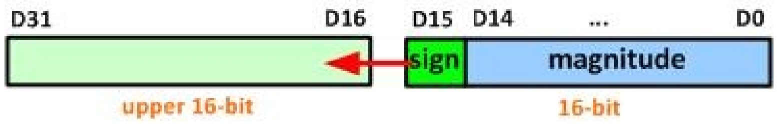


Figure 5- 2: Sign Extending a Half-word

Look at the following example:

```
;assume 0x80000 contains +260 = 0000 0001 0000 0100 and R1=0x80000
LDRSH   R0,[R1]  ;R0=0000 0000 0000 0000 0000 0001 0000 0100
```

Another example:

```
;assume location 0x20000 has -327660=0x8002 and R2=0x20000
LDRSH   R1,[R2]              ;R1=FFFF8002
```

As can be seen in the above examples, LDRSH does not alter the lower 16 bits. The sign of 16 bits is copied to the rest of the 32-bit register. How can these instructions help correct the overflow error? To answer that question, Example 5-13 shows Example 5-5 rewritten to correct the overflow problem.

Example 5-13

Write a program for Example 5-5 to handle the overflow problem.

Solution:

```
AREA     EXAMPLE5_13,CODE,READONLY
    ENTRY
    LDR       R1,=DATA1
    LDR       R2,=DATA2
    LDR       R3,=RESULT
    LDRSB   R4, [R1]              ;R4 = +96
    LDRSB   R5, [R2]              ;R5 = +70
    ADD      R4,R4,R5           ;R4 = R4 + R5 = 96 + 70 = +166
    STR      R4,[R3]            ;Store +166 in location RESULT
HL       B        HL

DATA1 DCB     +96
DATA2 DCB     +70

    AREA    VARIABLES,DATA,READWRITE    ;The following is stored in RAM
RESULT DCW     0
    END
```

The following is an analysis of the values in Example 5-13. Each is sign-extended and then added as follows:

Sign	**Binary numbers**	**Decimal**
0	000 0000 0000 0000 0000 0000 0110 0000	+96 after sign ext.
0	000 0000 0000 0000 0000 0000 0100 0110	+70 after sign ext.
0	000 0000 0000 0000 0000 0000 1010 0110	+166

As a rule, if the possibility of overflow exists, all byte-sized signed numbers should be sign-extended into a word, and similarly, all halfword-sized signed operands should be sign-extended to a word before they are processed. This is shown in Program 5-1. Program 5-1 finds total sum of a group of signed number data.

Program 5-1

```
    ; This program calculates the sum of signed numbers

    AREA    PROG5_1,CODE,READONLY
    ENTRY
    LDR       R0,=SIGN_DAT
    MOV     R3,#9
    MOV     R2,#0
LOOP    LDRSB   R1, [R0]
    ;Load into R1 and sign extend it.
```

```
   ADD      R2,R2,R1            ;R2 = R2 + R1
   ADD      R0,R0,#1            ;point to next
   SUBS     R3,R3,#1            ;decrement counter
   BNE      LOOP
   LDR      R0,=SUM
   STR      R2,[R0]             ;Store R2 in location SUM
HERE     B          HERE

SIGN_DAT DCB             +13,-10,+19,+14,-18,-9,+12,-19,+16

   AREA     VARIABLES,DATA,READWRITE
SUM      DCD      0
   END
```

Signed number multiplication

Signed number multiplication is similar in its operation to the unsigned multiplication described in Chapter 3. The only difference between them is that the operands in signed number operations can be positive or negative; therefore, the result must indicate the sign. In ARM we have SMULL (signed multiply long) but no SMUL (sign multiply). Table 5-2 summarizes signed number multiplication; it is similar to Table 3-3 in Chapter 3. See Examples 5-14 and 5-15.

Multiplication	Operand 1	Operand 2	Result
word×word	Rm	Rs	RdHi= upper 32-bit,RdLo=lower 32-bit
Note: *Using SMULL (signed multiply long) for word × words multiplication provides the 64-bit result in RdLo and RdHi register. This is used for 32-bit × 32-bit numbers in which result can go beyond 0xFFFFFFFF.*			

Table 5-2: Signed Multiplication (SMULL RdLo,RdHi,Rm,Rs) Summary

Example 5-14

Observe the results of the following multiplication of signed numbers:

```
   LDR      R1,=-3500           ;R1 = -3500 (0xFFFFF254)
   LDR      R0,=-100            ;R0 = -100 (0xFFFFFF9C)
   SMULL  R2,R3,R0,R1
```

Solution:

-3500 × -100 = 350,000 = 55730 in hex. After executing the above program R2 and R3 will contain 0x55730 and 00000000, respectively.

Example 5-15

The following program is similar to Example 5-14. But, instead of SMULL, the UMULL instruction is used. Observe the results of the following multiplication:

```
    LDR      R1,=-3500          ;R1 = -3500 (0xFFFFF254)
    MOV    R0,#-100           ;R0 = -100 (0xFFFFFF9C)
    UMULL  R2,R3,R0,R1
```

Solution:

0xFFFFF254 × 0xFFFFFF9C = 0xFFFFF1F000055730. Thus, R2 and R3 will contain 0x00055730 and 0xFFFFF1F0, respectively. As you can see, the results of the programs are completely different. In the previous program the SMULL instruction considers the operands signed numbers and the result of two negative numbers becomes positive. As a result, the sign bit becomes zero, but in this example the operands are considered as unsigned numbers.

Signed number comparison

In Chapter 4 we saw that the CMP instruction affects the Z and C flags; using the flags we compared unsigned numbers. This instruction affects the N and V flags, as well; We can use flags Z, V, and N to compare signed numbers. The Z flag shows if the numbers are equal or not. When the numbers are equal the Z flag is set to one. N and V flags show if the left operand is bigger than the right operand or not. When N and V have the same value, the first operand has a greater value.

In summary, after executing the instruction *CMP Rn, Op2* the flags are changed as follows:

Op2 > Rn	V = N
Op2 = Rn	Z = 1
Op2 < Rn	N ≠ V

Table 5-3 lists the branch instructions which check the Z, V, and N flags. The instructions can be used together with the CMP instruction to compare signed numbers.

Instruction		Action
BEQ	branch equal	Branch if Z = 1
BNE	Branch not equal	Branch if Z = 0
BMI	Branch minus (branch negative)	Branch if N = 1
BPL	Branch plus (branch positive)	Branch if N = 0
BVS	Branch if V set (branch overflow)	Branch if V = 1
BVC	Branch if V clear (branch if no overflow)	Branch if V = 0
BGE	Branch greater than or equal	Branch if N = V
BLT	Branch less than	Branch if N ≠ V
BGT	Branch greater than	Branch if Z = 0 and N = V
BLE	Branch less than or equal	Branch if Z = 1 or N ≠ V

Table 5- 3: ARM Conditional Branch (Jump) Instructions for Signed Data

Program 5-2 finds the lowest number among a list of numbers.

Program 5-2

```
    ;Finding the lowest of signed numbers
    AREA     PROG5_2,CODE,READONLY
    ENTRY
    LDR       R0,=SIGN_DAT
    MOV     R3,#8
    LDRSB   R2,[R0]
    ;bring into R2 the first sign number and sign extend it
    ADD       R0,R0,#1              ;point to next
BEGIN   LDRSB   R1, [R0]                ;R1 = contents of loc. pointed to by R0
    CMP      R1,R2                       ;compare R1 and R2
    BGE       NEXT
    ;branch to NEXT if R1 is greater than or equal to R2
    MOV     R2,R1
    ;R2 = R1 (use the new number for comparison)
NEXT     ADD       R0,R0,#1              ;point to next
```

```
    SUBS    R3,R3,#1            ;decrement counter
    BNE     BEGIN               ;if R3 is not zero branch BEGIN

    LDR     R0,=LOWEST          ;R0 = address of LOWEST
    STR     R2,[R0]             ;store R2 in location SUM
HERE    B       HERE

SIGN_DAT DCB            +13,-10,+19,+14,-18,-9,+12,-19,+16

    AREA    VARIABLES,DATA,READWRITE
LOWEST          DCD     0
    END
    ;Notice that we use the first sign number as basis for
    ;comparison and keep comparing it with other numbers.
    ;Anytime any of the number  is smaller we use it as basis
    ;for comparison.
```

CMN instruction

```
  CMN     Rn,Op2
```

In ARM we have two compare instructions: CMP and CMN. While the CMP instruction sets the flags by subtracting the source operand from the destination, the CMN sets the flags by adding source to destination. As a result CMN compares the destination operand with the negative of the source operand:

destination > (-1 × source)	V = N
destination = (-1 × source)	Z = 1
destination < (-1 × source)	N = negation of V

When the source operand is an immediate value, the instructions can be used interchangeably. Example 5-16 is an example of using the CMN instruction.

Example 5-16

Assuming R5 has a positive value, write a program that finds its negative match in an array of data (OUR_DATA).

Solution:

```
    AREA     EXAMPLE5_16,CODE,READONLY
    ENTRY

    MOV     R5,#13
    LDR      R0,=OUR_DATA
    MOV     R3,#9
BEGIN
    LDRSB   R1, [R0]                          ;R1 = contents of loc. pointed to by R0
                                          ;(sign extended)
    CMN      R1,R5                    ;compare R1 and negative of R5
    BEQ       FOUND                   ;branch if R1 is equal to negative of R5

    ADDS     R0,R0,#1              ;increment pointer
    SUBS     R3,R3,#1              ;decrement counter
    BNE       BEGIN                   ;if R3 is not zero branch BEGIN

NOT_FOUND     B          NOT_FOUND
FOUND               B          FOUND

OUR_DATA        DCB       +13,-10,-13,+14,-18,-9,+12,-19,+16
    END
```

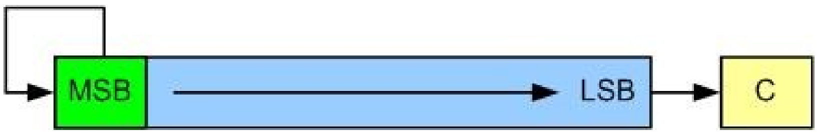
In the above program R5 is initialized with 13. Therefore, it finishes searching when it gets to –13.

Arithmetic shift

As was discussed in Chapter 3, there are two types of shifts: logical and arithmetic. Logical shift, which is used for unsigned numbers, was discussed in Chapter 3. The arithmetic shift is used for signed numbers. It is basically the same as the logical shift, except that the sign bit is copied to the shifted bits.

ASR (arithmetic shift right)

```
    MOV     Rn,Op2, ASR count
```



As the bits of the source are shifted to the right into C, the empty bits are filled with the sign bit. One can use the ASR instruction to divide a signed number by 2, as shown below:

```
MOV     R0,#-10             ;R0 = -10 = 0xFFFFFFF6
MOV     R3,R0,ASR #1       ;R0 is arithmetic shifted right once
;R3 = 0xFFFFFFFB = -5
```

Review Questions

1. Explain the difference between an overflow and a carry.
2. Explain the purpose of the LDRSB and LDRSH instructions. Demonstrate the effect of LDRSB on R0 = 0xF6. Demonstrate the effect of LDRSH on R1 = 0x124C.
3. The instruction for signed multiplication is ________.
4. For each of the following instructions, indicate the flag condition necessary for each branch to occur: (a) BLE (b) BGT

Section 5.3: IEEE 754 Floating-Point Standards

In this section we study the IEEE standard for floating-point numbers.

IEEE floating-point standard

Up to the late 1970s, real numbers (numbers with decimal points) were represented differently in binary form by different computer manufacturers. This made many programs incompatible for different machines. In 1980, an IEEE committee standardized the floating-point data representation of real numbers. This standard, much of which was contributed by Intel based on the 8087 math coprocessor, recognized the need for different degrees of precision by different applications; therefore, it established single precision and double precision. Since almost all software and hardware companies now abide by these standards, each one is explained thoroughly.

IEEE 754 single-precision floating-point numbers

IEEE single-precision floating-point numbers use only 32 bits of data to represent any real number range 2^{128} to 2^{-126}, for both positive and negative numbers. This translates approximately to a range of 1.2×10^{-38} to $3.4 \times 10^{+38}$ in decimal numbers, again for both positive and negative values. In some math coprocessor terminology, these single-precision 32-bit floating-point numbers are referred to as short real. Assignment of the 32 bits in the single-precision format is shown in Figure 5-3.

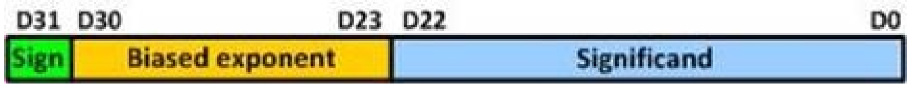


Figure 5- 3: IEEE 754 Single-precision Floating-point Numbers

To make the hardware design of the math processors much easier and less transistor consuming, the exponent part is added to a constant of 0x7F (127 decimal). This is referred to as a biased exponent. Conversion from real to floating point involves the following steps.

1. The real number is converted to its binary form.
2. The binary number is represented in scientific form: 1.xxxx E yyyy
3. Bit 31 is either 0 for positive or 1 for negative.
4. The exponent portion, yyyy, is added to 7F to get the biased exponent, which is placed in bits 23 to 30.
5. The significand, xxxx, is placed in bits 22 to 0.

Examples 5-17, 5-18, and 5-19 demonstrate this process.

Example 5-17

Convert 9.75_{10} to single-precision (short real) floating point.

Solution:

decimal 9.75 = binary 1001.11 = scientific binary 1.00111 E 3

Sign bit 31 is 0 for positive.

Exponent bits 30 to 23 are 1000 0010 (3 + 7F = 82H) after biasing.

Significand bits 22 to 0 are 0011100000000000000000 …00.

Putting it all together gives the following binary form, under which is written the hex form:

```
0100     0001     0001     1100     0000     0000     0000     0000
4          1          1         C          0          0          0          0
```

This can be verified by using an assembler such as Keil.

Example 5-18

Convert 0.078125_{10} to short IEEE floating-point standard real FP (single precision).

Solution:

decimal 0.078125 = binary 0.000101 = scientific binary 1.01 E –4

Sign bit 31 is 0 for positive.

Exponent bits 30–23 are 0111 1011 (–4 + 7F = 7B) after biasing.

Significand bits 22–0 are 01000000….000.

This number will be represented in binary and hex as

```
0011     1101     1010     0000     0000     0000     0000     0000
3          D          A         0          0          0          0          0
```

Example 5-19

Convert –96.27$_{10}$ to single-precision FP format.

Solution:

decimal 96.27 = binary 1100000.01000101000111101 =

scientific binary 1.10000001000101000111101E 6

Sign bit 31 is 1 for negative.

Exponent bits 30–23 are 1000 0101 (6 + 7F = 85H) after biasing,

Fraction bits 22–0 are 10000001000101000111101,

The final form in binary and hex is

1100	0010	1100	0000	1000	1010	0011	1101
C	2	C	0	8	A	3	D

It must be noted that conversion of the decimal portion 0.27 to binary can be continued beyond the point shown above, but because the fraction part of the single precision is limited to 23 bits, this was all that was shown. For that reason, double-precision FP numbers are used in some applications to achieve a higher degree of accuracy.

IEEE 754 double-precision floating-point numbers

Double-precision FP (called *long real* by Intel) can represent numbers in the range 2.3×10^{-308} to 1.7×10^{308}, both positive and negative. A total of 52 bits (bits 0 to 51) are for the significand, 11 bits (bits 52 to 62) are for the exponent, and finally, bit 63 is for the sign. The conversion process is the same as for single precision in that the real number must first be represented as 1.xxxxxxx E YYYY, then YYYY is added to 3FF to get the biased exponent. See Figure 5-4 and Example 5-20.

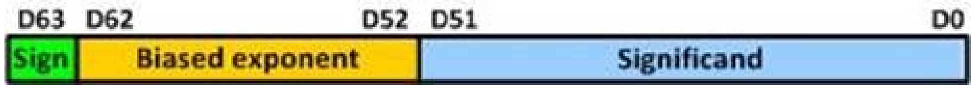


Figure 5- 4: IEEE 754 Double-precision Floating-point Numbers

Example 5-20

Convert 152.1875$_{10}$ to double-precision FP.

Solution:

decimal 152.1875 = binary 10011000.0011 =

scientific binary 1.00110000011 E 7

Bit 63 is 0 for positive.

Exponent bits 62–53 are 10000000110 (7 + 3FF = 406) after biasing.

Fraction bits 52–0 are 00110000011000…..000.

0100	0000	0110	0011	0000	0110	0000	0000	0000	…	0000
4	0	6	3	0	6	0	0	0	…	0

This example will be verified by an assembler in the next section.

Math coprocessor in ARM

Using a general-purpose microprocessor such as the ARM to perform mathematical functions such as floating point calculation, log, sine, and others is very time consuming, not only for the CPU but also for programmers writing such programs. In the absence of a math coprocessor, programmers must write subroutines using ARM instructions for mathematical functions. Although some of these subroutines are already written, no matter how good the subroutine, its CPU run time will still be quite long. To accelerate complicated mathematical and floating point calculation, math or floating point coprocessors are used. A simpler variation of math coprocessor is called floating point unit (FPU) and some ARM chips come with on-chip FPU. In Keil there is a library named fplib to do floating point calculation.

Review Questions

1. Single-precision IEEE FP standard uses __________ bits to represent data.
2. Double-precision IEEE FP standard uses _________ bits to represent data.
3. To get the biased exponent portion of IEEE single-precision floating-point data we add _________.
4. To get the biased exponent portion of IEEE double-precision floating-point data we add _________.
5. True or false. In the absence of a math processor, the general-purpose processor must perform all math calculations.
6. True or false. All of the ARM chips come with the FPU.

Problems:

Section 5.1: Signed Numbers Concept

1. Show how the 32-bit computers would represent the following numbers and verify each with a calculator.

(a) -23	(b) +12	(c) -0x28
(d) +0x6F	(e) -128	(f) +127
(g) +365	(h) -32,767	

2. Show how the 32-bit computers would represent the following numbers and verify each with a calculator.

(a) -230	(b) +1200	(c) - 0x28F
(d) +0x6FF		

Section 5.2: Signed Number Instructions and Operations

3. Find the overflow flag for each case and verify the result using an ARM IDE. Do byte-sized calculation on them.

(a) (+15) + (-12)	(b) (-123) + (-127)	(c) (+0x25) + (+34)
(d) (-127) + (+127)	(e) (+100) + (-100)	

4. Sign-extend the following and write simple programs in using ARM IDE to verify them.

(a) -122	(b)-0x999	(c) +0x17
(d) +127	(e) -129	

5. Modify Program 5-2 to find the highest temperature. Verify your program.

Section 5.3: IEEE 754 Floating-Point Standards

6. What is the disadvantage of using a general-purpose processor to perform math operations?
7. Show the bit assignment of the IEEE single-precision standard.
8. Convert (by hand calculation) each of the following real numbers to IEEE single-precision standard.

   (a) 15.575 (b) 89.125 (c) –1022.543 (d) –0.00075

9. Show the bit assignment of the IEEE double-precision standard.

10. In single-precision FP (floating point), the biased exponent is calculated by adding ________ to the _________ portion of a scientific binary number.

11. In double-precision FP, the biased exponent is calculated by adding ________ to the _________ portion of a scientific binary number.

12. Convert the following to double-precision FP.

    (a) 12.9375 (b) 98.8125

Answers to Review Questions

Section 5.1

1. D7, D15, and D31 for 32-bit signed data.
2. 0x16 = 0001 0110; its 2's complement is: 1110 1010
3. –128 to +127; –32,768 to +32,767 (decimal)
4. -2,147,483,648 to +2,147,483,647
5. 0x500000 = 0101 0000 0000 0000 0000 0000;

   Its 2's complement is: 1011 0000 0000 0000 0000 0000

Section 5.2

1. C flag is raised when there is a carry out of the result, but V flag is raised when there is a carry to the sign bit and no carry out of the sign bit or when there is no carry to the sign bit and there is a carry out of the sign bit. C flag is used to indicate overflow in unsigned arithmetic operations while V flag is involved in signed operations.
2. The LDRSB instruction sign extends the sign bit of a byte into a word; the LDRSH instruction sign extends the sign bit of a half-word into a word.

   In 0xF6 the sign bit is 1; thus, it is sign-extended into 0xFFFFFFF6

   0x124C sign-extended into R1 would be R0 = 0x0000124C.
3. SMULL

   (a) BLE will jump if V is the inverse of N, or if Z = 1.

   (b) BGT will jump if V equals N, and if Z = 0.

Section 5.3

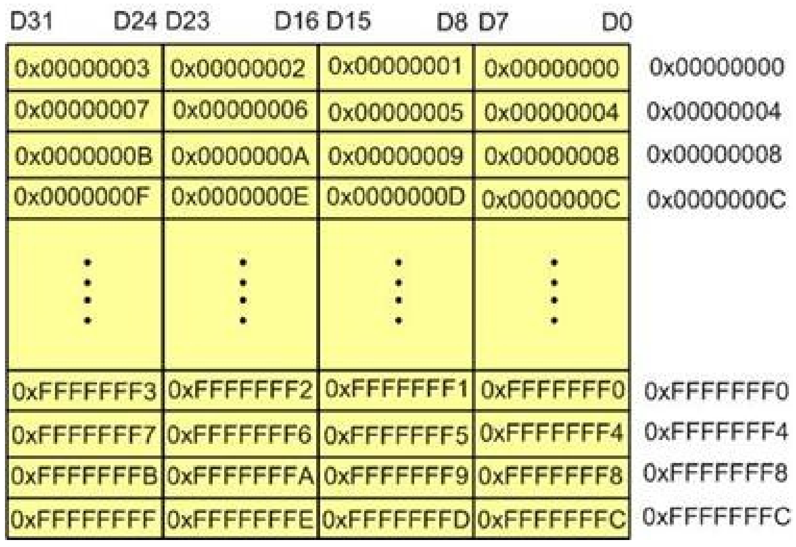
1. 32
2. 64
3. 0x7F
4. 0x3FF
5. True
6. False

Chapter 6: ARM Memory Map, Memory Access, and Stack

This chapter discusses the issue of memory access and the stack. Section 6.1 is dedicated to ARM memory map and memory access. We will also explain the concepts of align, non-align, little endian, and big endian data access. In Section 6.2, we examine the use of the stack in ARM. We discuss the bit-addressable (bit-band) SRAM and peripherals in Section 6.3. Advanced indexed addressing mode is explained in Section 6.4. In Section 6.5, we describe the PC relative addressing mode and its use in implementing ADR and LDR.

Section 6.1: ARM Memory Map and Memory Access

The ARM CPU uses 32-bit addresses to access memory and peripherals. This gives us a maximum of 4 GB (gigabytes) of memory space. This 4GB of directly accessible memory space has addresses 0x00000000 to 0xFFFFFFFF, meaning each byte is assigned a unique address (ARM is a byte-addressable CPU). See Figure 6-1.



D31 D24	D23 D16	D15 D8	D7 D0	
0x00000003	0x00000002	0x00000001	0x00000000	0x00000000
0x00000007	0x00000006	0x00000005	0x00000004	0x00000004
0x0000000B	0x0000000A	0x00000009	0x00000008	0x00000008
0x0000000F	0x0000000E	0x0000000D	0x0000000C	0x0000000C
⋮	⋮	⋮	⋮	
0xFFFFFFF3	0xFFFFFFF2	0xFFFFFFF1	0xFFFFFFF0	0xFFFFFFF0
0xFFFFFFF7	0xFFFFFFF6	0xFFFFFFF5	0xFFFFFFF4	0xFFFFFFF4
0xFFFFFFFB	0xFFFFFFFA	0xFFFFFFF9	0xFFFFFFF8	0xFFFFFFF8
0xFFFFFFFF	0xFFFFFFFE	0xFFFFFFFD	0xFFFFFFFC	0xFFFFFFFC

Figure 6-1: Memory Byte Addressing in ARM

The 4GB of memory space is divided into three regions: code, data, and peripheral devices. See Table 6-1.

Address range	Name	Description
0x00000000-0x1FFFFFFF	Code	ROM or Flash memory
0x20000000-0x3FFFFFFF	SRAM	SRAM region used for on-chip RAM
0x40000000-0x5FFFFFFF	Peripheral	On-chip peripheral address space
0x60000000-0x9FFFFFFF	RAM	Memory, cache support
0xA0000000-0xDFFFFFFF	Device	Shared and non-shared device space
0xE0000000-0xFFFFFFFF	System	PPB and vendor system peripherals

Table 6- 1: Memory Space Allocation in ARM Cortex

In other words, the ARM uses the memory mapped I/O. For the ARM microcontrollers, generally the Flash ROM is used for program code, SRAM for scratch pad data, and memory-mapped I/O ports for peripherals. While there is an absolute standard for the ARM instructions that all licensees of ARM must follow, there is no standard for exact locations and types of memory and peripherals. Therefore the licensees can implement the memory and peripherals as they choose. For this reason the amount and the address locations of memory used by Flash ROM, SRAM, and I/O peripherals varies among the family members and chip manufacturers. The ARM manufacturer datasheet should give you the details of the memory map for both on-chip and off-chip memory and peripherals. Make sure to examine the memory map of a given ARM chip before you start to program it. From Table 6-1, notice that some of the memory addresses are set aside for the external (off-chip) memory and peripherals. At the time of this writing, no ARM manufacturer has populated the entire 4 GB of memory space with on-chip ROM, RAM, and I/O peripherals.

ARM-based Motherboards

In ARM systems for Microsoft Windows, Unix, and Android operating systems the ARM motherboards use DRAM for the RAM memory, just like the x86 and Pentium PCs. As the ARM CPU is pushed into the laptop, desktop, and tablets PCs, and the high end of embedded systems products such as routers, we will see the use of DRAM as primary memory to store both the operating systems and the applications. In such systems, the Flash memory will be holding the POST (power on self test), BIOS (basic Input/output systems) and boot programs. Just like x86 system, such systems have both on-chip and off-chip high speed SRAM for cache. Currently, there are ARM chips on the market with some on-chip Flash ROM, SRAM, and memory decoding circuitry for connection to external (off-chip) memory. This off-chip memory can be SRAM, Flash, or DRAM. The datasheet for such ARM chips provide the details of memory map for both on-chip and off-chip memories. Next, we examine the ARM buses and memory access.

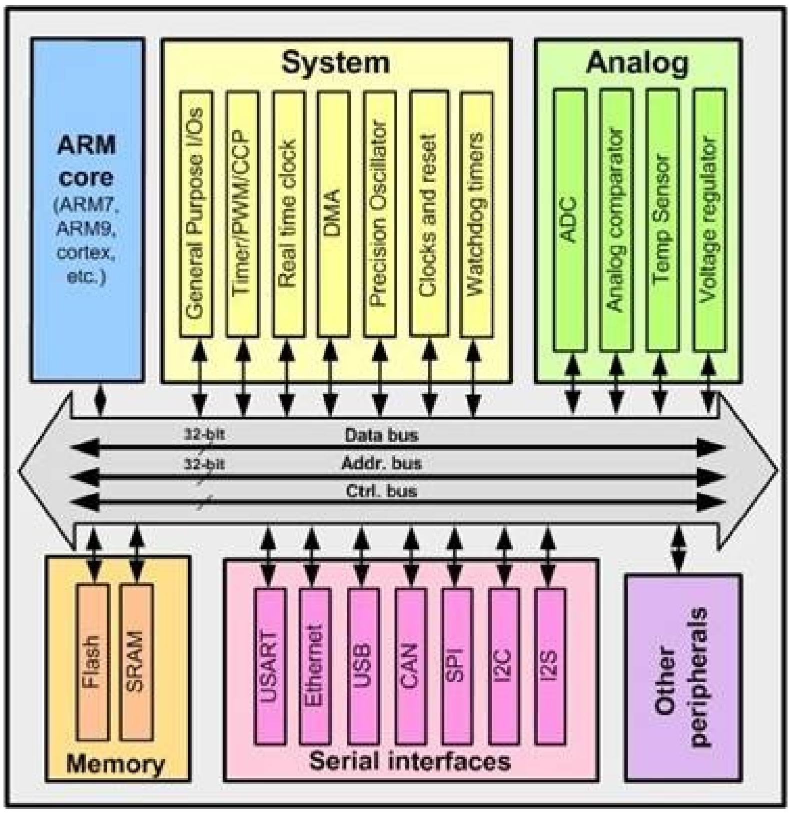


Figure 6-2: Memory Connection Block Diagram in ARM

D31–D0 Data bus

The 32-bit data bus of the ARM provides the 32-bit data path to the on-chip and off-chip memory and peripherals. They are grouped into 8-bit data chunks, D0–D7, D8–D15, D16–D23, and D24–D31.

A31–A0

These signals provide the 32-bit address path to the on-chip and off-chip memory and peripherals. Since the ARM supports data access of byte (8 bits), half word (16 bits), and word (32 bits), the buses must be able to access any of the 4 banks of memory connected to the 32-bit data bus. The A0 and A1 are used to select one of the 4 bytes of the D31-D0 data bus. See Figure 6-3.

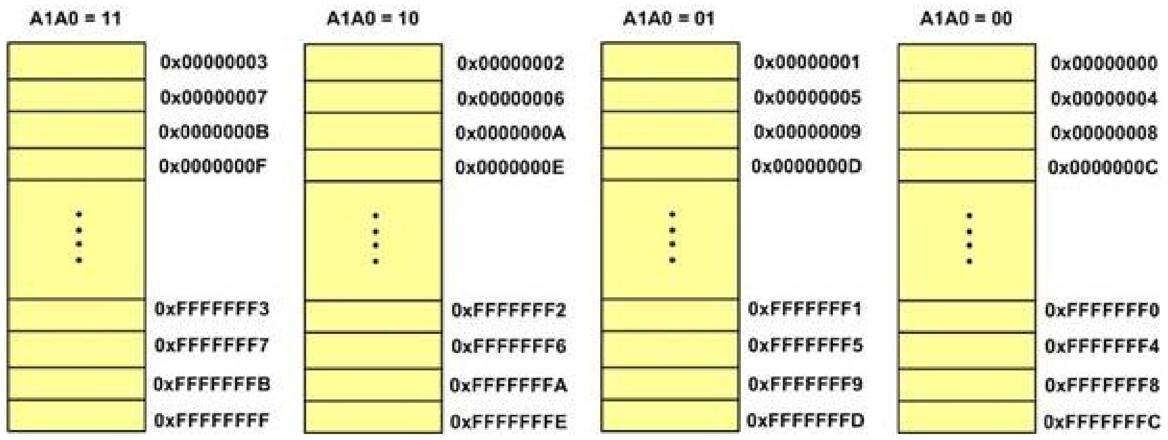


Figure 6-3: Memory Block Diagram in ARM

AHB and APB buses

The ARM CPU is connected to the on-chip memory via an AHB (advanced high-performance bus). The AHB is used not only for connection to on-chip ROM and RAM, it is also used for connection to some of the high speed I/Os (input/output) such as GPIO (general purpose I/O). ARM chip also has the APB (advanced peripherals bus) bus dedicated for communication with the on-chip peripherals such as timers, ADC, serial COM, SPI, I2C, and other peripheral ports. While we need the 32-bit data bus between CPU and the memory (RAM and ROM), many slower peripherals are 8 or 16 bits and there is no need for entire fast 32-bit data bus pathway. For this reason, ARM uses the AHB-to-APB bridge to access the slower on-chip devices such as peripherals. Also since peripherals do not need a high speed bus, a bridge between AHB and APB allows going from the higher speed bus of AHB to lower speed bus of peripherals. The AHB bus allows a single-cycle access. See Figure 6-4 for AHB-to-APB bridge.

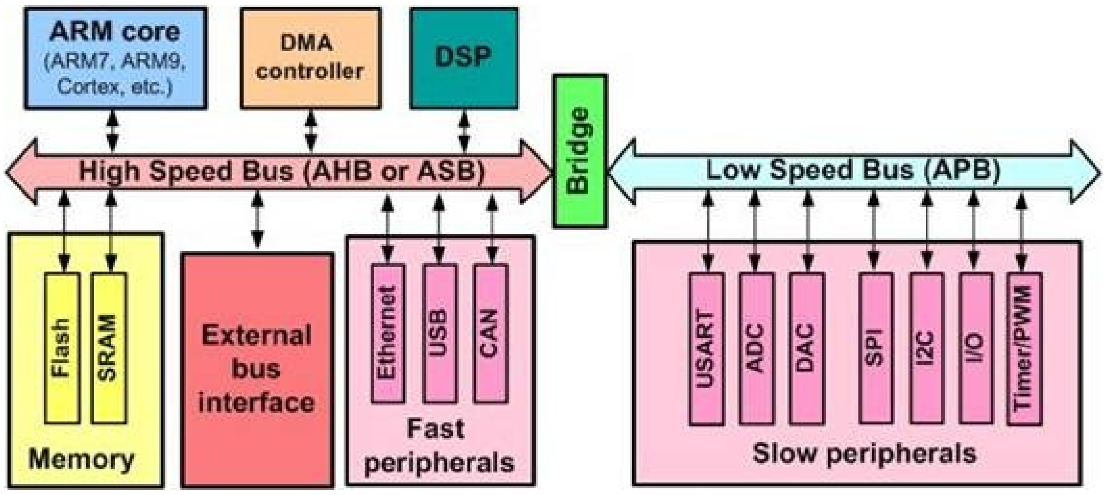


Figure 6-4: AHB and APB in ARM

Bus cycle time

To access a device such as memory or I/O, the CPU provides a fixed amount of time called a bus cycle time. During this bus cycle time, the read or write operation of memory or I/O must be completed. The bus cycle time used for accessing memory is often referred to as MC (memory cycle) time. The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called memory read cycle time. While for on-chip memory the cycle time can be 1 clock, in the off-chip memory the cycle time is often 2 clocks. If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time. This extra time is called a wait state (WS). In the 1980s, the clock speed for memory cycle time was the same as the CPU's clock speed. For example, in the 20 MHz processors, the buses were working at the same speed of 20 MHz. This resulted in 2 × 50 ns = 100 ns for the memory cycle time (1/20 MHz = 50 ns). See Example 6-1.

Example 6-1

Calculate the memory cycle time of a 50-MHz bus system with

(a) 0 WS,

(b) 1 WS, and

(c) 2 WS.

Assume that the bus cycle time for off-chip memory access is 2 clocks.

Solution:

1/50 MHz = 20 ns is the bus clock period. Since the bus cycle time of zero wait states is 2 clocks, we have:

Memory cycle time with 0 WS	2 × 20 = 40 ns
Memory cycle time with 1 WS	40 + 20 = 60 ns

Memory cycle time with 2 WS	$40 + 2 \times 20 = 80$ ns

It is preferred that all bus activities be completed with 0 WS. However, if the read and write operations cannot be completed with 0 WS, we request an extension of the bus cycle time. This extension is in the form of an integer number of WS. That is, we can have 1, 2, 3, and so on WS, but not 1.25 WS.

When the CPU's speed was under 100 MHz, the bus speed was comparable to the CPU speed. In the 1990s the CPU speed exploded to 1 GHz (gigahertz) while the bus speed maxed out at around 200 MHz. The gap between the CPU speed and the bus speed is one of the biggest problems in the design of high-performance systems. To avoid the use of too many wait states in interfacing memory to CPU, cache memory and other high-speed DRAMs are used. These are discussed in Chapters 9 and 10.

Bus bandwidth

The rate of data transfer is generally called bus bandwidth. In other words, bus bandwidth is a measure of how fast buses transfer information between the CPU and memory or peripherals. The wider the data bus, the higher the bus bandwidth. However, the advantage of the wider external data bus comes at the cost of increasing the die size for system on-chip (SOC) or the printed circuit board size for off-chip memory. Now you might ask why we should care how fast buses transfer information between the CPU and outside, as long as the CPU is working as fast as it can. The problem is that the CPU cannot process information that it does not have. In other words, the speed of the CPU must be matched with the higher bus bandwidth; otherwise, there is no use for a fast CPU. This is like driving a Porsche or Ferrari in first gear; it is a terrible under usage of CPU power. Bus bandwidth is measured in MB (megabytes) per second and is calculated as follows:

bus bandwidth = (1/bus cycle time) × bus width in bytes

In the above formula, bus cycle time can be for both memory and I/O since the ARM uses the memory mapped I/O. Example 6-2 clarifies the concept of bus bandwidth. As can be seen from Example 6-2, there are two ways to increase the bus bandwidth: Either use a wider data bus or shorten the bus cycle time (or do both). That is exactly what many processors have done. Again, it must be noted that although the processor's speed can go to 1 GHz or higher, the bus speed for off-chip memory is limited to around 200 MHz. The reason for this is that the signals become too noisy for the circuit board if they are above 100 MHz.

Example 6-2

Calculate memory bus bandwidth for the following CPU if the bus speed is 100 MHz.

(a) ARM Thumb with 0 WS and 1 WS (16-bit data bus)

(b) ARM with 0 WS and 1 WS (32-bit data bus)

Assume that the bus cycle time for off-chip memory access is 2 clocks.

Solution:

The memory cycle time for both is 2 clocks, with zero wait states. With the 100 MHz bus speed we have a bus clock of 1/100 MHz = 10 ns.

(a) Bus bandwidth = (1/(2 × 10 ns)) × 2 bytes = 100M bytes/second (MB/s)

With 1 wait state, the memory cycle becomes 3 clock cycles

3 × 10 = 30 ns and the memory bus bandwidth is = (1/30 ns) × 2 bytes = 66.6 MB/s

(b) Bus bandwidth = (1/(2 × 10 ns)) × 4 bytes = 200 MB/s

With 1 wait state, the memory cycle becomes 3 clock cycles

3 × 10 = 30 ns and the memory bus bandwidth is = (1/30 ns) × 4 bytes = 126.6 MB/s

From the above it can be seen that the two factors influencing bus bandwidth are:

1. The read/write cycle time of the CPU
2. The width of the data bus

Code memory region

The 4 GB of ARM memory space is organized as 1G × 32 bits since the ARM instructions are 32-bit. The internal data bus of the ARM is 32-bit, allowing the transfer of one instruction into the CPU every clock cycle. This is one of the benefits of the RISC fixed instruction size. The fetching of an instruction in every clock cycle can work only if the code is word aligned, meaning each instruction is placed at an address location ending with 0, 4, 8, or C. Example 6-3 shows the placement of code in ARM memory. Notice that the code addresses go up by 4 since the ARM instructions are fixed at 4 bytes each. While compilers ensure that codes are word aligned, it is job of the programmer to make sure the data in SRAM is word aligned too. We will examine this important topic soon.

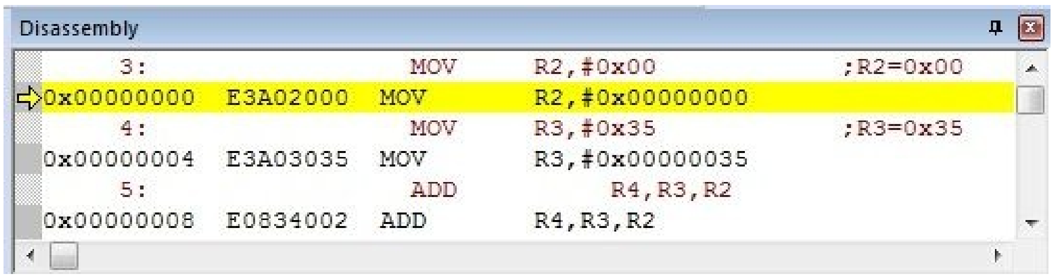
Example 6-3

Compile and debug the following code in Keil and see the placement of instructions in memory locations.

```
    AREA     ARMex, CODE, READONLY
    ENTRY
    MOV     R2,#0x00          ;R2=0x00
    MOV     R3,#0x35          ;R3=0x35
    ADD      R4,R3,R2
    END      ;Mark end of file
```

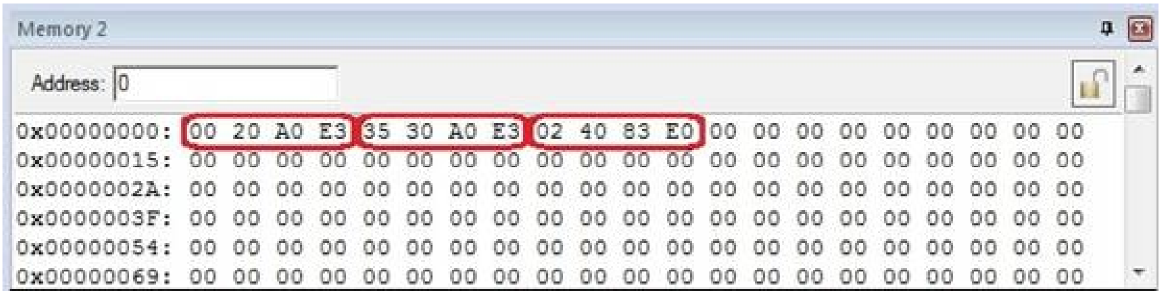
Solution

As you can see in the figure, the first MOV instruction starts from location 0x00000000, the second MOV instruction starts from location 0x00000004 and the ADD instruction starts from location 0x00000008.



```
        3:              MOV       R2,#0x00           ;R2=0x00
0x00000000  E3A02000  MOV       R2,#0x00000000
        4:              MOV       R3,#0x35           ;R3=0x35
0x00000004  E3A03035  MOV       R3,#0x00000035
        5:              ADD          R4,R3,R2
0x00000008  E0834002  ADD       R4,R3,R2
```

The following image displays the first locations of memory. The code of the first MOV instruction is located in the first word (four bytes) of memory which is word aligned. The same rule applies for the other instructions. Note that the code of MOV R2,0 is E3 A0 20 00 but 00 20 A0 E3 is stored in the memory. We will discuss the reason in this chapter when we focus on the concept of big endian and little endian.



```
Address: 0
0x00000000: 00 20 A0 E3 35 30 A0 E3 02 40 83 E0 00 00 00 00 00 00 00 00 00
0x00000015: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000002A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000003F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000054: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000069: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

SRAM memory region

A section of the memory space is used by SRAM. The SRAM can be on-chip or off-chip (external). The same way that every ARM chip has some on-chip Flash ROM for code, a portion of the memory region is used by the on-chip SRAM. This on-chip SRAM is used by the CPU for scratch pad to store parameters. It is also used by the CPU for the purpose of the stack. We examine the stack usage by the ARM in the next section. In using the SRAM memory for storing parameters, we must be careful when loading or storing data in the SRAM lest we use unaligned data access. Next, we discuss this important issue.

Data misalignment in SRAM

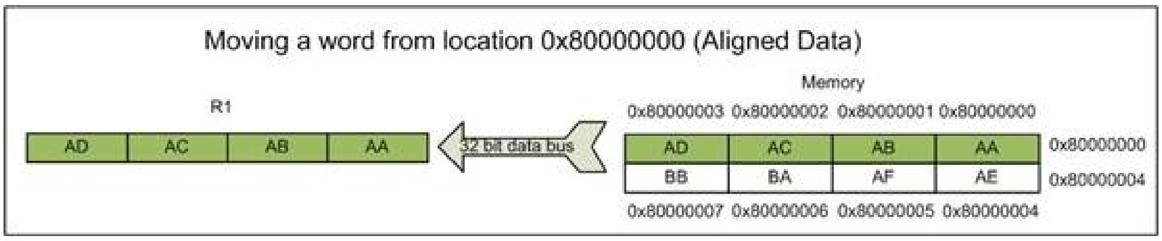
The case of misaligned data has a major effect on the ARM bus performance. If the data is aligned, for every memory read cycle, the ARM brings in 4 bytes of information (data or code) using the D31–D0 data bus. Such data alignment is referred to as word alignment. To make data word aligned, the least significant digits of the hex addresses must be 0, 4, 8, or C (in hex).

While the compilers make sure that program codes (instructions) are always aligned (Example 6-3), it is the placement of data in SRAM by the programmer that can be nonaligned and therefore subject to memory access penalty. In other words, the single cycle access of memory is also used by ARM to bring into registers 4 bytes of data every clock cycle assuming that the data is aligned. To make sure that data are also aligned we use the align directive. The use of align directive for RAM data makes sure that each word is located at an address location ending with address of 0, 4, 8, or C. If our data is word size (using DCDU directive) then the use of align directive at the start of the data section guaranties all the data placements will be word aligned. When a word size data is defined using the DCD directive, the assembler aligns it to be word aligned.

Accessing non-aligned data

As we have stated many times before, ARM defines 32-bit data as a word. The address of a word can start at any address location. For example, in the instruction “LDR R1,[R0]” if R0 = 0x20000004, the address of the word being fetched into R1 starts at an aligned address. In the case of “LDR R1,[R0]” if R0 = 0x20000001 the address starts at a non-aligned address. In systems with a 32-bit data bus, accessing a word from a non-aligned addressed location can be slower. This issue is important and applies to all 32-bit processors.

In the 8-bit system, accessing a word (4 bytes) is treated like accessing four consecutive bytes regardless of the address location. Since accessing a byte takes one memory cycle, accessing 4 bytes will take 4 memory cycles. In the 32-bit system, accessing a word with an aligned address takes one memory cycle. That is because each byte is carried on its own data path of D0–D7, D8–D15, D16–D23, and D24–D31 in the same memory cycle. However, accessing a word with a non-aligned address requires two memory cycles. For example, see how accessing the word in the instruction “LDR R1, [R0]” works as shown in Figure 6-5. As a case of aligned data, assume that R0 = 0x80000000. In this instruction, 4 bytes contents of memory locations 0x80000000 through 0x80000003 are being fetched in one cycle. In only one cycle, the ARM CPU accesses locations 0x80000000 through 0x80000003 and puts it in R1.


Moving a word from location 0x80000000 (Aligned Data)
R1
AD
AC
AB
AA
32 bit data bus
Memory
0x80000003 0x80000002 0x80000001 0x80000000
AD AC AB AA
BB BA AF AE
0x80000000
0x80000004
0x80000007 0x80000006 0x80000005 0x80000004

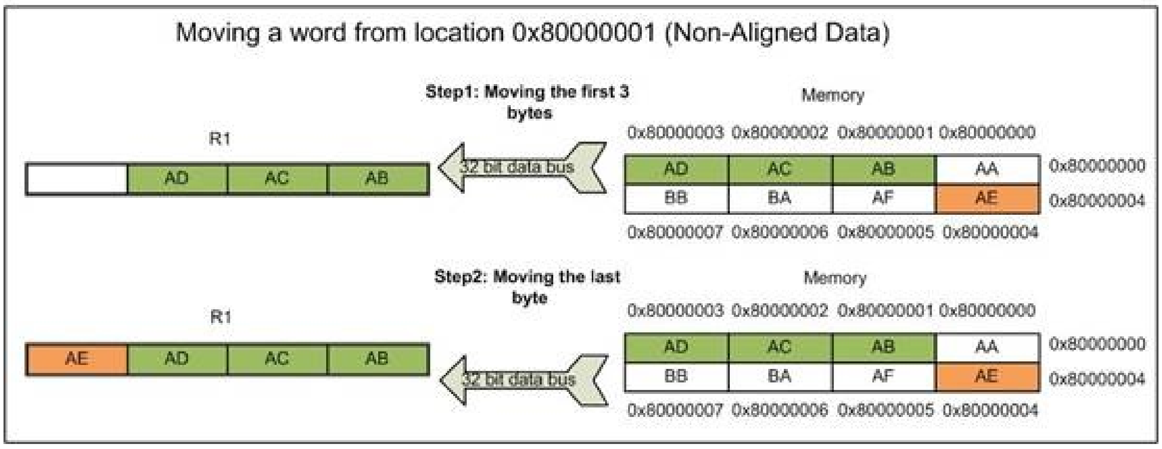

Moving a word from location 0x80000001 (Non-Aligned Data)
Step1: Moving the first 3 bytes
R1
AD
AC
AB
32 bit data bus
Memory
0x80000003 0x80000002 0x80000001 0x80000000
AD AC AB AA
BB BA AF AE
0x80000000
0x80000004
0x80000007 0x80000006 0x80000005 0x80000004
Step2: Moving the last byte
R1
AE
AD
AC
AB
32 bit data bus
Memory
0x80000003 0x80000002 0x80000001 0x80000000
AD AC AB AA
BB BA AF AE
0x80000000
0x80000004
0x80000007 0x80000006 0x80000005 0x80000004

Figure 6-5: Memory Access for Aligned and Non-aligned Data

Now assuming that R0 = 0x80000001 in this instruction, 8 bytes contents of memory locations 0x80000000 through 0x80000007 are being fetched in two consecutive cycles but only 4 bytes of it are used. In the first cycle, the ARM CPU accesses locations 0x80000000 through 0x80000003 and puts them in R1 only the desired three bytes of locations 0x800000001 through 0x80000003. In the second cycle, the contents of memory locations 0x8000004 through 0x80000007 are accessed and only the desired byte of 0x80000004 is put into R1. See Example 6-4.

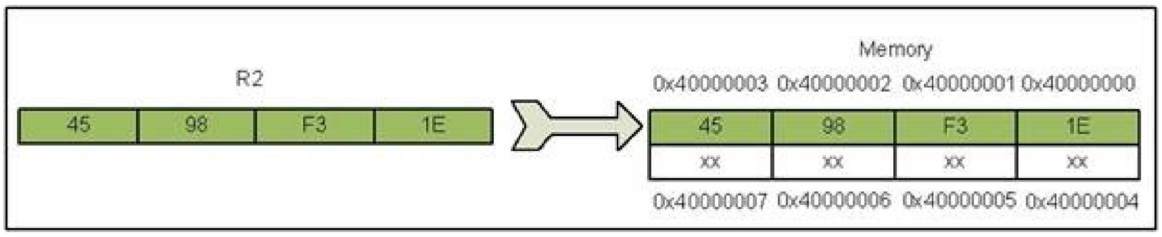
Example 6-4

Show the data transfer of the following cases and indicate the number of memory cycle times it takes for data transfer. Assume that R2 = 0x4598F31E.

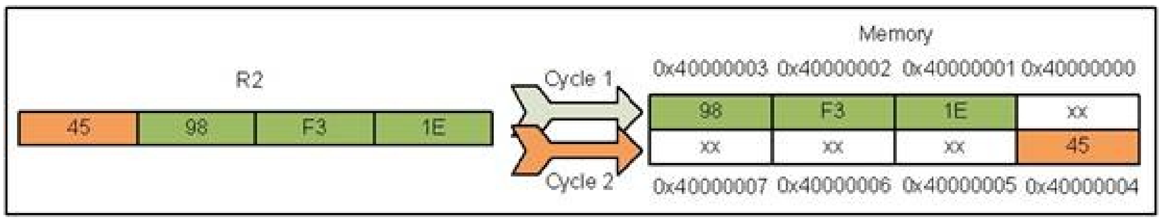
```
LDR     R1,=0x40000000          ;R1=0x40000000
LDR     R2,=0x4598F31E          ;R2=0x4598F31E
STR     R2,[R1]             ;Store R2 to location 0x40000000
ADD     R1,R1,#1            ;R1 = R1 + 1 = 0x40000001
STR     R2,[R1]             ;Store R2 to location 0x40000001
ADD     R1,R1,#1            ;R1 = R1 + 1 = 0x40000002
STR     R2,[R1]             ;Store R2 to location 0x40000002
ADD     R1,R1,#1            ;R1 = R1 + 1 = 0x40000003
STR     R2,[R1]             ;Store R2 to location 0x40000003
```

Solution:

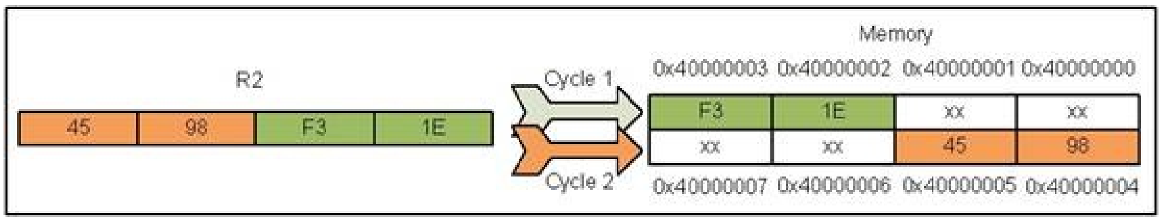
For the first STR R2,[R1] instruction, the entire 32 bits of R2 is stored into locations with addresses of 0x40000000, 0x40000001, 0x40000002, and 0x40000003, The 4-byte content of register R2 is stored into memory locations with starting address of 0x40000000 via the 32-bit data bus of D31–D0. This address is word aligned since address of the least significant digit is 0. Therefore, it takes only one memory cycle to transfer the 32-bit data.


R2
45
98
F3
1E
Memory
0x40000003 0x40000002 0x40000001 0x40000000
45
98
F3
1E
xx
xx
xx
xx
0x40000007 0x40000006 0x40000005 0x40000004

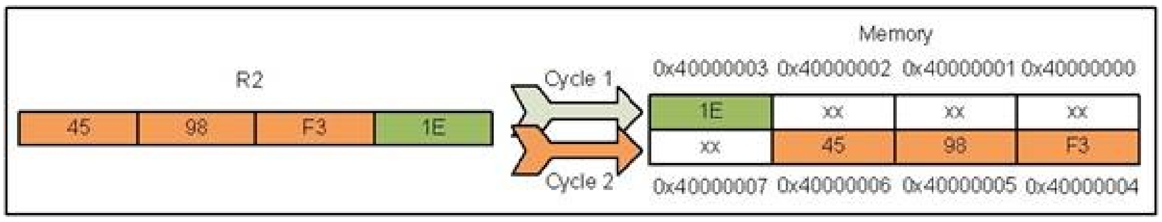
For the second STR R2,[R1] instruction, in the first memory cycle, the lower 24 bits of R2 is stored into locations 0x40000001, 0x40000002, and 0x40000003. In the second memory cycle, the upper 8 bits of R2 is stored into the 0x40000004 location.


R2
45
98
F3
1E
Cycle 1
Cycle 2
Memory
0x40000003 0x40000002 0x40000001 0x40000000
98
F3
1E
xx
xx
xx
xx
45
0x40000007 0x40000006 0x40000005 0x40000004

For the third STR R2,[R1] instruction, in the first memory cycle, the lower 16 bits of R2 is stored into locations 0x40000002 and 0x40000003. In the second memory cycle, the upper 16 bits of R2 is stored into locations 0x40000004 and 0x40000005.


R2
45
98
F3
1E
Cycle 1
Cycle 2
Memory
0x40000003 0x40000002 0x40000001 0x40000000
F3
1E
xx
xx
xx
xx
45
98
0x40000007 0x40000006 0x40000005 0x40000004

For the fourth STR R2,[R1] instruction, in the first memory cycle, the lower 8 bits of to R2 is stored into locations 0x40000003. In the second memory cycle, the upper 24 bits of R2 is stored into the locations 0x40000004, 0x40000005, and 0x40000006.


R2
45
98
F3
1E
Cycle 1
Cycle 2
Memory
0x40000003 0x40000002 0x40000001 0x40000000
1E
xx
xx
xx
xx
45
98
F3
0x40000007 0x40000006 0x40000005 0x40000004

The lesson to be learned from this is to try not to put any words on a non-aligned address location in a 32-bit system. Indeed this is so important that directive ALIGN is specifically designed for this purpose. Next, we discuss the issue of aligned data.

Using LDR instruction with DCD and ALIGN directives

The DCD and DCDU directives are used for 32-bit (word) data. The DCD directive ensures 32-bit data types are aligned, in contrast to DCDU which does not. DCD is used as follows:

```
VALUE1          DCD     0x99775533
```

This ensures that VALUE1, a word-sized operand, is located in a word aligned address location. Therefore, an instruction accessing it will take only a single memory cycle. Since performance of the CPU depends on how fast it can fetch the data we must ensure that any memory access reading 32-bit data is done in a single clock cycle. This means we must make sure all 32-bit data are word aligned. This is so important that ARM has an interrupt (exception) dedicated to misaligned data, meaning any time it accesses misaligned data, it lets us know that there is a problem. The one-time use of ALIGN directive at the beginning of data area using DCDU makes the data aligned for that group of data.

Using LDRH with DCW and ALIGN directives

The problem of misaligned data is also an issue when the data size is in half-words (16-bit). In many cases using DCWU, we must use the ALIGN directive multiple times in the data area of a given program to ensure they are aligned. This is in contrast to the DCW directive which ensures data type to be half-word aligned. This is especially the case when we use the LDRH instruction. See Example 6-5. Aligned data is also an issue for the Thumb version of the ARM.

Example 6-5

Show the data transfer of the following LDRH instructions and indicate the number of memory cycle times it takes for data transfer.

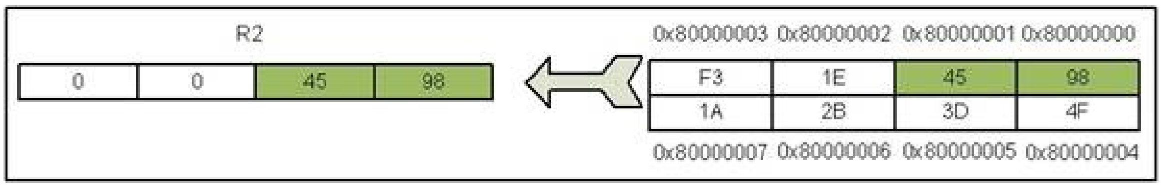
```
LDR     R1,=0x80000000          ;R1=0x80000000
LDR     R3,=0xF31E4598          ;R3=0xF31E4598
LDR     R4,=0x1A2B3D4F          ;R4=0x1A2B3D4F
STR     R3,[R1]
;(STR R3,[R1])stores R3 to location 0x80000000
STR     R4,[R1,#4]
;(STR R4,[R1+4]) stores R4 to location 0x80000004
LDRH    R2, [R1]
;loads two bytes from location 0x80000000 to R2
LDRH    R2, [R1,#1]
;loads two bytes from location 0x80000001 to R2
LDRH    R2, [R1,#2]
;loads two bytes from location 0x80000002 to R2
```

LDRH R2, [R1,#3]

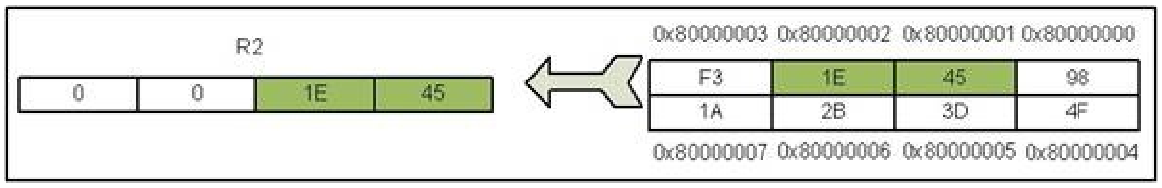
;loads two bytes from location 0x80000003 to R2

Solution:

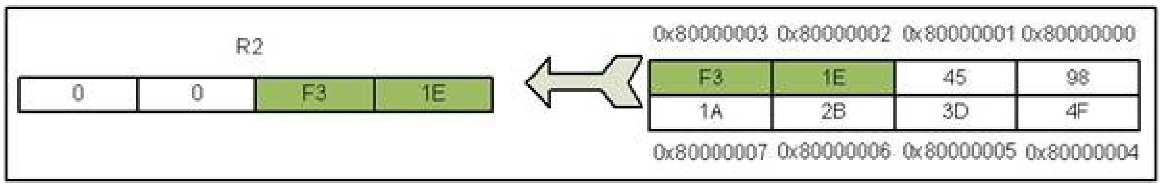
In the LDRH R2,[R1] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000000 and 0x80000001 are used to get the 16 bits to R2. This address is halfword aligned since the least significant digit is 0. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00004598


R2
0
0
45
98
0x80000003 0x80000002 0x80000001 0x80000000
F3
1E
45
98
1A
2B
3D
4F
0x80000007 0x80000006 0x80000005 0x80000004

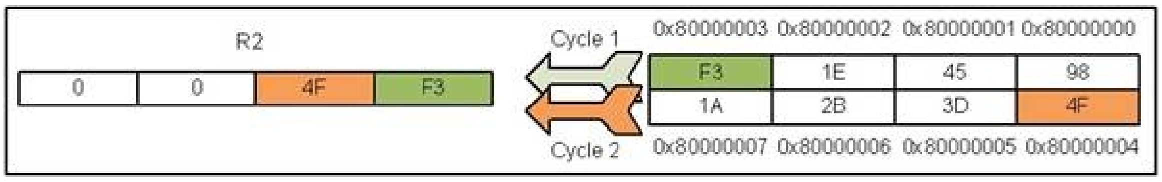
For the LDRH R2,[R1,#1], instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000001 and 0x80000002 are used to get the 16 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00001E45.


R2
0
0
1E
45
0x80000003 0x80000002 0x80000001 0x80000000
F3
1E
45
98
1A
2B
3D
4F
0x80000007 0x80000006 0x80000005 0x80000004

For the LDRH R2,[R1,#2], instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000002 and 0x80000003 are used to get the 16 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x0000F31E.


R2
0
0
F3
1E
0x80000003 0x80000002 0x80000001 0x80000000
F3
1E
45
98
1A
2B
3D
4F
0x80000007 0x80000006 0x80000005 0x80000004

For the LDRH R2,[R1,#3] instruction, in the first memory cycle, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000003 is used to get the lower 8 bits to R2. In the second memory cycle, the address locations 0x80000004, 0x80000005, 0x80000006, and 0x80000007 are accessed where only the 0x80000004 location is used to get the upper 8 bits to R2. Now, R2=0x00004FF3.


R2
0
0
4F
F3
Cycle 1
Cycle 2
0x80000003 0x80000002 0x80000001 0x80000000
F3
1E
45
98
1A
2B
3D
4F
0x80000007 0x80000006 0x80000005 0x80000004

Using LDRB with DCB and ALIGN directives

The problem of misaligned data does not exist when the data size is bytes. In cases such as using the string of ASCII characters with the DCB directive, accessing a byte takes the same amount of time (one memory cycle) as an aligned word (4 bytes), regardless of the address location of the data. The only problem with the LDRB is it brings into the CPU only a single byte of data in each memory cycle instead of 4 bytes if LDR is. See Example 6-6.

Example 6-6

Show the data transfer of the following LDRB instructions and indicate the number of memory cycle times it takes for data transfer.

```
    LDR      R1,=0x80000000          ;R1=0x80000000
    LDR      R3,=0xF31E4598          ;R3=0xF31E4598
    LDR      R4,=0x1A2B3D4F          ;R4=0x1A2B3D4F
    STR      R3,[R1]
    ;Store R3 to location 0x80000000
    STR      R4,[R1,#4]
    ;(STR R4,[R1+4]) Store R4 to location 0x80000004
            LDRB     R2,[R1]
    ;load one byte from location 0x80000000 to R2
            LDRB     R2,[R1,#1]
    ;(LDRB R2,[R1+1]) load one byte from location 0x80000001
            LDRB     R2,[R1,#2]
    ;(LDRB R2,[R1+2]) load one byte from location 0x80000002
            LDRB     R2,[R1,#3]
    ;(LDRB R2,[R1+3]) load one byte from location 0x80000003
```

Solution:

In the LDRB R2,[R1] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000000 is used to get the 8 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00000098.

In the LDRB R2,[R1,#1] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000001 is used to get the 8 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x00000045.

In the LDRB R2,[R1,#2] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000002 is used to get the 8 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x0000001E.

In the LDRB R2,[R1,#3] instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000003 is used to get the 8 bits to R2. Therefore, it takes only one memory cycle to transfer the data. Now, R2=0x000000F3.

Peripheral region

Table 6-1 showed a section of memory is set aside for peripherals. The type of peripherals and memory address locations used is unique to a vendor. The ARM manufacturers provide the details of memory map for the peripherals. If you examine the manufacturer data sheet you will see that they use aligned addresses to make sure there is no clock penalty for accessing them.

Little Endian vs. Big Endian war

In storing data, the ARM follows the little endian convention. The little endian places the least significant byte (little end of the data) in the low address and the big endian is the opposite. The origin of the terms *big endian* and *little endian* is from a Gulliver's Travels story about how an egg should be opened: from the big end or the little end. ARM supports both little and big endian. In ARM little endian is the default. Some ARM chip manufacturers provide an option for changing it to big endian. See Example 6-7 to understand little endian and big endian data storage.

Example 6-7

Show how data is placed after execution of the following code using

a) little endian and

b) big endian.

```
LDR     R2,=0x7698E39F          ;R2=0x7698E39F
LDR     R1,=0x80000000
STR     R2,[R1]
```

Solution:

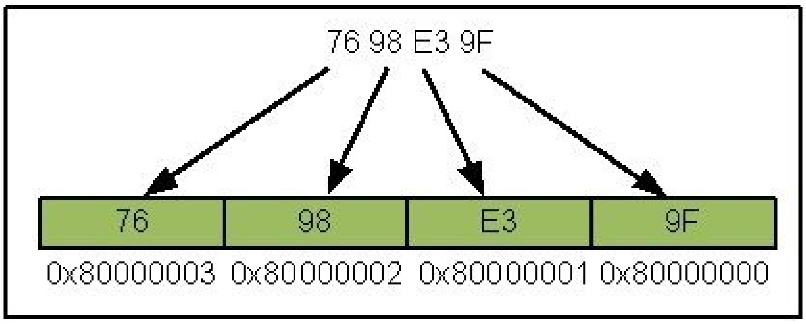
a) For little endian we have:

Location 80000000 = (9F)

Location 80000001 = (E3)

Location 80000002 = (98)

Location 80000003 = (76)



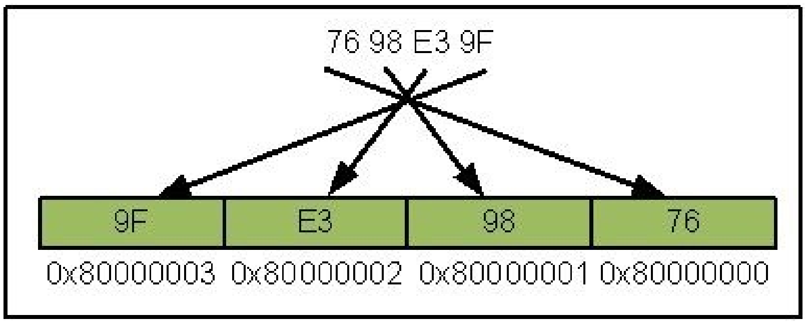
b) For big endian we have:

Location 80000000 = (76)

Location 80000001 = (98)

Location 80000002 = (E3)

Location 80000003 = (9F)



In Example 6-7, notice how the least significant byte (the little end of the data) 0x9F goes to the low address 0x80000000, and the most significant byte of the data 0x76 goes to the high address 0x80000003. This means that the little end of the data goes in first, hence the name little endian. In the ARM with big endian option enabled, data is stored the opposite way: The big end (most significant byte) goes into the low address first, and for this reason it is called big endian. Many of recent RISC processors allow selection of mode, big endian or little endian.

Harvard Architecture and ARM

In recent years many ARM manufacturers are using the Harvard architecture for ARM CPUs. Old ARM architectures up to ARM7 use Von Neumann architecture. The Harvard architecture feeds the CPU with both code and data at the same time via two sets of buses, one for code and one for data. This increases the processing power of the CPU since it can bring in more information.

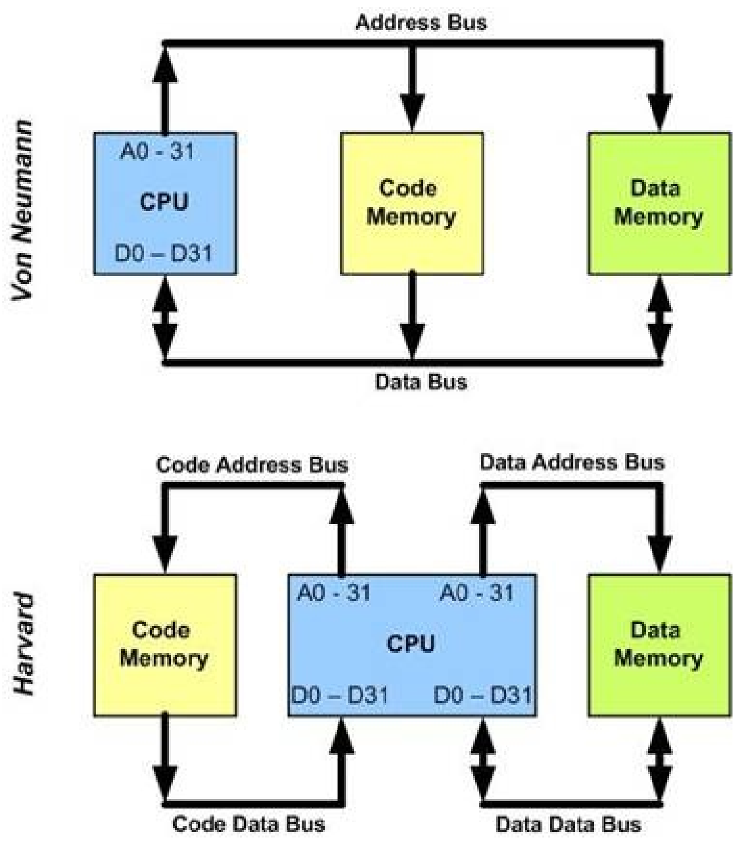


Figure 6-6: Von Neumann vs. Harvard Architecture

Review Questions

1. In ARM, all the instructions are ___bytes?
2. Who makes sure that instruction are aligned on word boundary?
3. In ARM, the _____ endian is the default.
4. A 66 MHz system has a memory cycle time of _____ ns if it is used with a zero wait state.
5. To interface a 100 MHz processor to a 50 ns access time ROM, how many wait states are needed?
6. True or false. ARM uses big endian format when is powered up.

Section 6.2: Stack and Stack Usage in ARM

The stack is a section of RAM used by the CPU to store information temporarily. This information could be data or an address or CPU registers when calling a subroutine. Stack is also widely used when executing an interrupt service routine (ISR). The CPU needs this storage area because there are only a limited number of registers.

How stacks are accessed

If the stack is a section of RAM, there must be a register inside the CPU to point to it. In the ARM CPU the register used to access the stack is R13.

The storing of CPU information such as the registers on the stack is called a PUSH, and loading the contents of the stack back into a CPU register is called a POP. In other words, a register is pushed onto the stack to save it and popped off the stack to retrieve it. The following describes each process.

Pushing onto the stack

The stack pointer (SP) points to the top of the stack (TOS). In the ARM register R13 is designated as stack pointer. As we push (store) data onto the stack, the data are saved in SRAM (where the SP points to) and SP must be decremented (or incremented) to point to the next location. In the ARM we have a choice of either incrementing the SP or decrementing it. Notice that this is different from many other microprocessors, notably x86 processors, in which the SP is decremented automatically when data is pushed onto the stack. Again it must be emphasized that while in traditional CPUs such as x86, the stack pointer is decremented automatically by the CPU itself, in the ARM CPU we must actually code the instruction for stack pointer decrementation (or incrementation) for that matter. Therefore, to push a register onto stack we use the STR and SUB instructions as shown in the following code:

```
STR     Rr,[R13]        ;Rr can be any registers (R0-R12)
SUB     R13,R13,#4      ;decrement stack pointer
```

For example, to store the value of R1 we can write the following instructions:

```
STR     R1,[R13]        ;store R1 onto the stack,
SUB     R13,R13,#4      ;and decrement SP
```

Popping from the stack

Popping (loading) the contents of the stack back into a given register is the opposite process of pushing. When the POP is executed, the SP is incremented (or decremented) and the top location of the stack is copied (loaded) back to the register. That means the stack is LIFO (Last-In-First-Out) memory.

To retrieve data from stack we can use the LDR instruction.

```
ADD     R13,R13,#4      ;increment stack pointer
LDR     Rr,[R13]        ;Rr can any of the registers (R0-R13)
```

For example, the following instructions pop from the top of stack and copy to R1:

```
ADD     R13,R13,#4       ;increment SP
LDR     R1, [R13]        ;load (POP) the top of stack to R1
```

Initializing the stack pointer in ARM

When the ARM is powered up, the R13 (SP) register contains value 0. Therefore, we must initialize the SP at the beginning of the program so that it points to somewhere in the internal SRAM. In ARM, we can make the stack to grow from a higher memory location to a lower memory location. In this case when we push (store) onto the stack the SP is decremented. We can also make the stack to grow from a lower memory location to a higher memory location, therefore when we push (store) onto the stack, the SP is incremented. It is common to initialize the SP to the uppermost RAM memory region, which means as we push data onto the stack, the stack pointer must be decremented.

Different ARMs have different amounts of RAM. In some ARM assembler Stack_Top represents the address of top of the stack. So, if we want to initialize the SP, we can simply load Stack_Top into the SP. Notice that SP (R13) is a 32-bit register. So, we can use any 32-bit address of SRAM as a stack section.

Example 6-8 shows how to initialize the SP and use the store and load instructions for PUSH and POP operations.

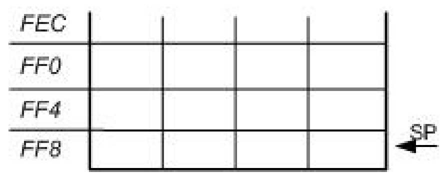
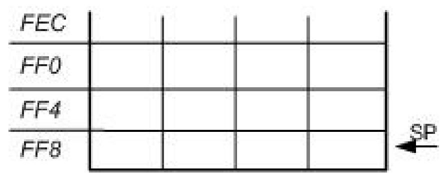
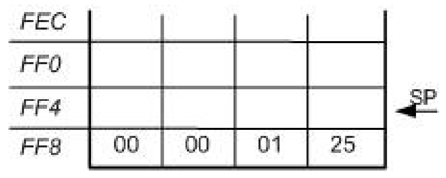
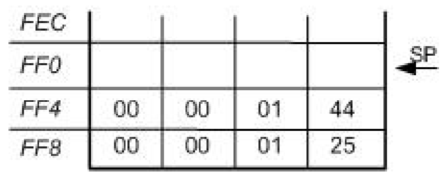
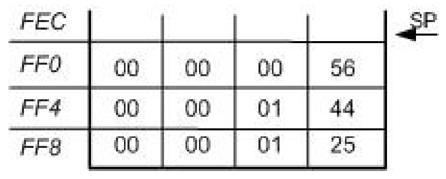
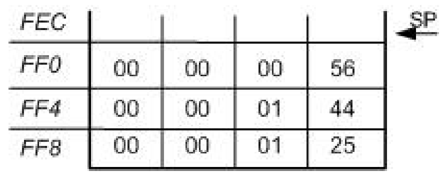
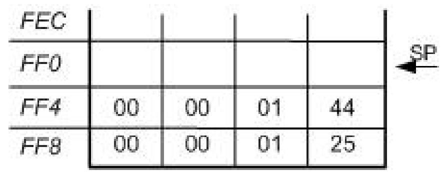
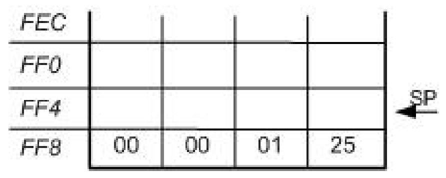
Example 6-8

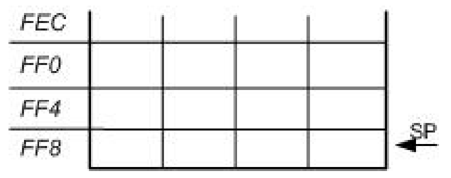
The following ARM program places some data into registers and calls a subroutine that uses the same registers. It shows how to use the stack. Examine the stack, stack pointer, and the registers used after the execution of each instruction.

```
    ;initialize the SP to point
    ;to the last location of RAM (Stack_Top)
    ;Assume Stack_Top = 0xFF8
    LDR     R13,=Stack_Top          ;load SP
    LDR     R0,=0x125        ;R0 = 0x125
    LDR     R1,=0x144        ;R1 = 0x144
    MOV     R2,#0x56         ;R2 = 0x56
    BL      MY_SUB           ;call a subroutine
    ADD     R3,R0,R1
    ;R3 = R0 + R1 = 0x125 + 0x144 = 0x269
    ADD     R3,R3,R2
    ;R3 = R3 + R2 = 0x269 + 0x56 = 0x2BF
HERE    B       HERE             ;stay here
```

```
    ;__________
MY_SUB
    ;_______-
    ;save R0, R1, and R2 on stack
    ;before they are used by a loop
    STR      R0,[R13]          ;save R0 on stack
    SUB      R13,R13,#4
    ;R13 = R13 - 4, to decrement the stack pointer
    STR      R1,[R13]          ;save R1 on stack
    SUB      R13,R13,#4
    ;R13 = R13 - 4, to decrement the stack pointer
    STR      R2,[R13]          ;save R2 on stack
    SUB      R13,R13,#4
    ;R13 = R13 - 4, to decrement the stack pointer
    ;——R0,R1, and R2 are changed
    MOV     R0,#0              ;R0 = 0
    MOV     R1,#0              ;R1 = 0
    MOV     R2,#0              ;R2 = 0
    ;_______________
    ;restore the original registers contents from stack
    ADD      R13,R13,#4
    ;R13 = R13 + 4 to increment the stack pointer
    LDR      R2,[R13]          ;restore R2 from stack
    ADD      R13,R13,#4
    ;R13 = R13 + 4 to increment the stack pointer
    LDR      R1,[R13]          ;restore R1 from stack
    ADD      R13,R13,#4
    ;R13 = R13 + 4 to increment the stack pointer
    LDR      R0,[R13]          ;restore R0 from stack

    BX       LR                 ;return to caller
```

Solution:

After the execution of	Contents of some the registers (in Hex)				Stack
	R0	R1	R2	SP (R13)	
LDR R13,=0xFF8	0	0	0	FF8	
LDR R0,=0x125 LDR R1,=0x144 LDR R2,=0x56	125	144	56	FF8	
STR R0,[R13] SUB R13,R13,#4	125	144	56	FF4	
STR R1,[R13] SUB R13,R13,#4	125	144	56	FF0	
STR R2,[R13] SUB R13,R13,#4	125	144	56	FEC	
MOV R0,#0 MOV R1,#0 MOV R2,#0	0	0	0	FEC	
ADD R13,R13,#4 LDR R2,[R13]	0	0	56	FF0	
ADD R13,R13,#4 LDR R1,[R13]	0	144	56	FF4	

ADD R13,R13,#4 LDR R0,[R13]	125	144	56	FF8	

The stack limit and nested calls in ARM

As mentioned earlier, we can define the stack anywhere in the read/write memory. So, in the ARM the stack can be as big as its RAM. In ARM, the stack is used for calls and interrupts. We must remember that upon calling a subroutine from the main program using the BL instruction, R14, the linker register, keeps track of where the CPU should return after completing the subroutine. Now, if we have another call inside the subroutine using the BL instruction, then it is our job to store the original R14 on the stack. Failure to do that will end up crashing the program. For this reason, we must be very careful when manipulating the stack contents.

Using LDM and STM instructions for the stack

Another way to push register contents onto the stack is to use STM (store multiple) and LDM (load multiple) instructions. In many interrupt and multitasking applications we need to save the contents of multiple registers on the stack and restore them back. Pushing the registers onto the stack one register at a time is time consuming. For this reason ARM has the STM and LDM instructions. The STM and LDM allow to store (push) and load (pop) multiple registers with a single instruction. Next, we explain each instruction and how it is used.

STM

Below shows the syntax for the STM. Notice we can specify the number of registers to be stored and destination RAM address (pointer) to which the data is pushed onto. Examine the following instructions:

```
STM     R11,{R0-R3}
;Store R0 through R3 onto memory pointed to by R11
STM     R8,{R0-R7}
;Store R0 through R7 onto memory pointed to by R8
STM     R7,{R0,R3,R5}
;Store R0, R3, R5 onto memory pointed to by R7
STM     R11,{R0-R10}
;Store R0 through R10 onto memory pointed to by R11
```

LDM

Below shows the syntax for the LDM. Notice we can specify the number of registers to be loaded and destination RAM address (pointer) from which the data is popped from. Examine the following instructions:

```
LDM R11,{R0-R3}
;Load R0 through R3 from memory pointed to by R11
LDM R8,{R0-R7}
;Load R0 through R7 from memory pointed to by R8
LDM R7,{R0,R3,R5}
;Load R0,R3,R5 from memory pointed to by R7
LDM R11,{R0-R10}
;Load R0 through R10 from memory pointed to by R11
```

Example 6-9 shows how we can use STM and LDM to simplify a code and prevent unwanted errors.

Example 6-9

Modify the Example 6-8 using the LDM and STM instructions.

Solution:

```
    ;initialize the SP to point to
    ;the last location of RAM (Stack_Top)
    ;Assume Stack_Top = 0xFF8
    LDR     R13,=Stack_Top          ;load SP
    LDR     R0,=0x125         ;R0 = 0x125
    LDR     R1,=0x144         ;R1 = 0x144
    MOV    R2,#0x56          ;R2 = 0x56
    BL      MY_SUB            ;call a subroutine
    ADD     R3,R0,R1          ;R3 = R0 + R1
    ;  = 0x125 + 0x144 = 0x269
    ADD     R3,R3,R2          ;R3 = R3 + R2
                                ;  = 0x269 + 0x56 = 0x2BF
HERE    B       HERE            ;stay here
    ;__________
MY_SUB
```

```
;—————
;save R0,R1, and R2 on stack
;before they are used by a loop
STM      R13,{R0-R2}       ;save R0,R1,R2 on stack
SUB      R13,R13,#12
;——R0,R1, and R2 are changed
MOV     R0,#0              ;R0=0
MOV     R1,#0              ;R1=0
MOV     R2,#0              ;R2=0
;——
;restore the original registers contents from stack
ADD      R13,R13,#12
LDM      R13,{R0-R2}       ;restore R0,R1, and R2 from stack
BX       LR                ;return to caller
;——-
```

We can also use the PUSH and POP pseudo-instructions to do the same thing.

Copying a block of data with LDM and STM

To copy a block of data, we bring into the CPU's register a word of data from memory and then copy it from the register to a location in RAM. In that case we copy one word (4 bytes) at a time. So to copy 16 words we have to set a counter to 16 for a loop iteration. We can use the LDM and STM to do the same thing with much less coding. Using LDM and STM instructions to copy a block of data, we need a register for the source address and another one for the destination address. Program 6-1 uses R11 and R12 for source and destination addresses, respectively. The registers R0–R9 are used as temporary place for data before they are copied to the destination. That gives us 10 words (40 bytes) transfer at a time. Notice that in the STM and LDM instructions, registers are transferred to or from memory in the order of the lowest to highest, so R0 will always be transferred to or from a location lower than the location of R1 in the memory and so on. That is why we should not be worrying about the order of registers in the stack when we use STM and LDM and it is not needed to PUSH and POP registers in opposite order as is common in normal POP and PUSH instructions.

Program 6-1: Copying a Block of Memory

This program copies a block of 10 words (40 bytes) memory from source to destination. The registers R11 and R12 are used for source and destination addresses.

```
AREA     PROG6_1, CODE, READONLY
ENTRY
LDM      R11,{R0-R9}   ;Load R0 thru R9 from memory pointed to by R11
STM      R12,{R0-R9}   ;Store R0 thru R9 to memory pointed to by R12
END
```

Review Questions

1. The ______ register is the default stack pointer.
2. How deep is the size of the stack in the ARM?
3. Write a program that pushes R5, R6, R7, and R8 into the stack.
4. Write a program that pops R5, R6, R7, and R8 from the stack.
5. What does the following program do?

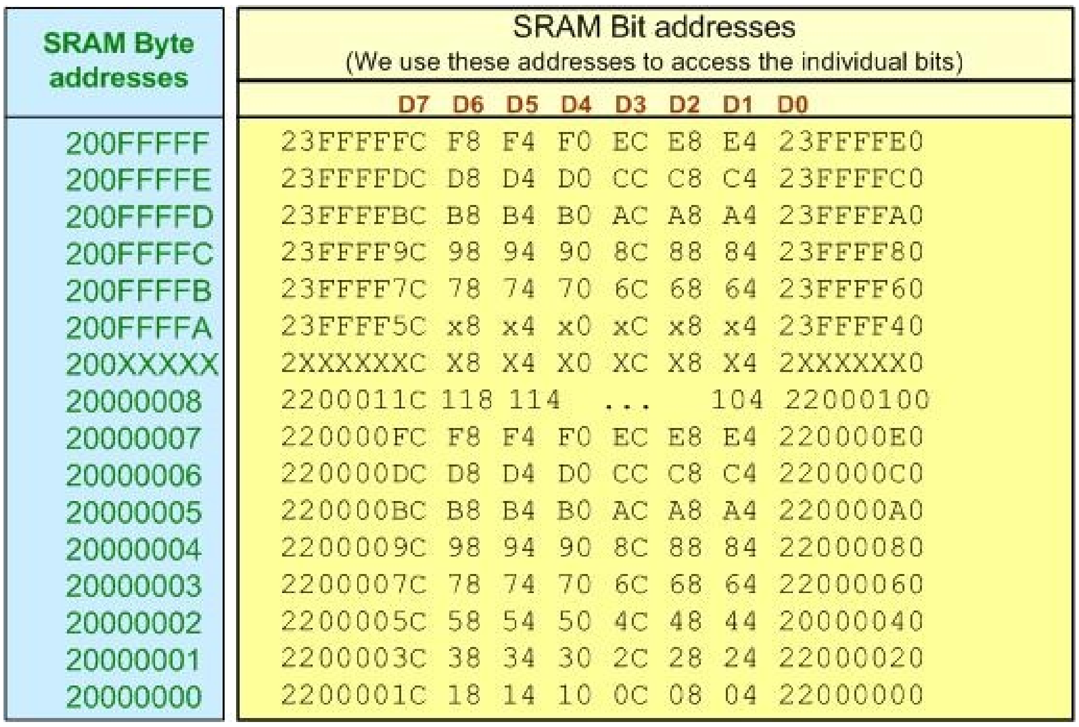
```
LDR      R5,=0x40000000
LDM      R5,{R1,R4}
LDR      R5,=0x50000000
STM      R5,{R1,R4}
```

Section 6.3: ARM Bit-Addressable Memory Region

Many microprocessors allow programs to access memory and I/O ports in byte size increments only. In other words, if you need to check a single bit of an I/O port, you must read the entire byte first and then manipulate the whole byte with some logic instructions to get hold of the desired bit. This is not the case with ARM microprocessors. Indeed, one of the most important features of the CPU is the ability to access the RAM and I/O ports in bits instead of bytes. This is a very important and powerful feature of the ARM used widely in the embedded system design and applications. In this section we show address assignment of bits of I/O and RAM, in addition to ways of programming them.

Bit-addressable (bit-band) SRAM

Of the 4GB memory space of the ARM, a number of bytes are bit-addressable. The ARM literature refers to this region as the bit-band region. Since the ARM instruction is 32-bit and the CPU executes code one instruction at a time, the code (Flash ROM) space is always word size and word aligned, as we have seen throughout the chapters. That means the bit-addressable regions must be located in SRAM and I/O peripherals regions. The bit-addressable RAM and I/O locations vary among the family members and vendors. The ARM generic manual defines the location addresses of bit-band as 0x20000000 to 0x200FFFFF for SRAM and 0x40000000 to 0x400FFFFF for peripherals. Notice they are located at the lowest 1 MB address space of SRAM and peripherals. See Table 6-1. It must be also noted that the bit-band (bit-addressable) regions are the only region that can be accessed in both bit and byte/halfword/word formats while the other area of memory must be accessed in byte/halfword/word size. In other words, the memory space region outside of the bit-band must always be accessed in byte/halfword/word formats only. For the ARM Cortex-M, the bit-band SRAM has addresses of 0x20000000 to 0x200FFFFF. This 1M bytes bit-addressable region is given alias addresses of 0x22000000 to 0x23FFFFFF. Therefore, the bit addresses 0x22000000 to 0x2200001F are for the first byte of SRAM location 0x20000000, and 0x22000020 to 0x2200003F are the bit addresses of the second byte of SRAM location 0x20000001, and so on. See Figure 6-7.



SRAM Byte addresses	SRAM Bit addresses (We use these addresses to access the individual bits)							
	D7	D6	D5	D4	D3	D2	D1	D0
200FFFFF	23FFFFFC	F8	F4	F0	EC	E8	E4	23FFFFE0
200FFFFE	23FFFFDC	D8	D4	D0	CC	C8	C4	23FFFFC0
200FFFFD	23FFFFBC	B8	B4	B0	AC	A8	A4	23FFFFA0
200FFFFC	23FFFF9C	98	94	90	8C	88	84	23FFFF80
200FFFFB	23FFFF7C	78	74	70	6C	68	64	23FFFF60
200FFFFA	23FFFF5C	x8	x4	x0	xC	x8	x4	23FFFF40
200XXXXX	2XXXXXXC	X8	X4	X0	XC	X8	X4	2XXXXXX0
20000008	2200011C	118	114	...			104	22000100
20000007	220000FC	F8	F4	F0	EC	E8	E4	220000E0
20000006	220000DC	D8	D4	D0	CC	C8	C4	220000C0
20000005	220000BC	B8	B4	B0	AC	A8	A4	220000A0
20000004	2200009C	98	94	90	8C	88	84	22000080
20000003	2200007C	78	74	70	6C	68	64	22000060
20000002	2200005C	58	54	50	4C	48	44	20000040
20000001	2200003C	38	34	30	2C	28	24	22000020
20000000	2200001C	18	14	10	0C	08	04	22000000

Figure 6-7: SRAM bit-addressable region and their alias addresses

Since each byte of SRAM has 8 bits we need an address for each bit. This means we need at least 8M address locations to access 8M bits, one address for each bit. However, to make the addresses word-aligned the ARM provides 4-byte alias address for each bit. For example, 0x22000000 to 0x2200001F is assigned to a single byte location of 0x20000000. That means we have 0x22000000 to 0x23FFFFFF (total of 32M locations, as alias addresses) for 1M bytes of address.

Bit map for SRAM

From Figure 6-7 once again notice the following facts:

1. The bit address 0x22000000 is assigned to D0 of SRAM location 0x20000000.
2. The bit address 0x22000004 is assigned to D1 of SRAM location 0x20000000.
3. The bit address 0x22000008 is assigned to D2 of SRAM location of 0x20000000.
4. The bit address 0x2200000C is assigned to D3 of SRAM location of 0x20000000.
5. The bit address 0x22000010 is assigned to D4 of SRAM location 0x20000000.
6. The bit address 0x22000014 is assigned to D5 of RAM location 0x20000000.
7. The bit address 0x22000018 is assigned to D6 of SRAM location 0x20000000.
8. The bit address 0x2200001C is assigned to D7 of SRAM location 0x20000000.

Notice that SRAM locations 0x20000000 – 0x200FFFFF are both byte-addressable and bit-addressable. The only difference is when we access it in byte (or halfword or word) we use addresses 0x20000000 to 0x200FFFFF, but when they are accessed in bit, they are accessed via their alias addresses of 0x22000000 to 0x23FFFFFF. The reason it is called aliases is because it is same physical location but accessed by two different addresses. It is like a same person but different names (aliases) See Examples 6-10 through 6-12.

Example 6-10

The generic ARM chip has the following address assignments. Calculate the space and the amount of memory given to each region.

(a) Address range of 0x20000000–200FFFFF for SRAM bit-addressable region

(b) Address range of 0x22000000–23FFFFFF for alias addresses of bit-addressable SRAM

Solution:

(a) 200FFFFF – 20000000 = FFFFF bytes. Converting FFFFF to decimal, we get 1,048,575 + 1 = 1,048,576, which is equal to 1M bytes.

(b) 23FFFFFF – 22000000 = 1FFFFFF bytes. Converting 1FFFFFF to decimal, we get 33,554,431 + 1 = 33,554,432, which is equal to 32M bytes.

Example 6-11

Write a program to set HIGH the D6 of the SRAM location 0x20000001 using a) byte address and b) the bit alias address.

Solution:

a)

```
LDR     R1,=0x20000001          ;load the address of the byte
LDRB    R2,[R1]                 ;get the byte
ORR     R2,R2,#2_01000000       ;make D6 bit high
;(binary representation in Keil for 0b01000000)
STRB    R2,[R1]                 ;write it back
```

b) From Figure 6-7 we have address 0x22000038 as the bit address of D6 of SRAM location 0x20000001.

```
LDR     R1,=0x22000038          ;load the alias address of the bit
MOV    R2,#1                    ;R2 = 1
STRB    R2,[R1]           ;Write one to D6
```

Example 6-12

Write a program to set LOW the D0 bit of the SRAM location 0x20000005 using a) byte address and b) the bit alias address.

Solution:

a)

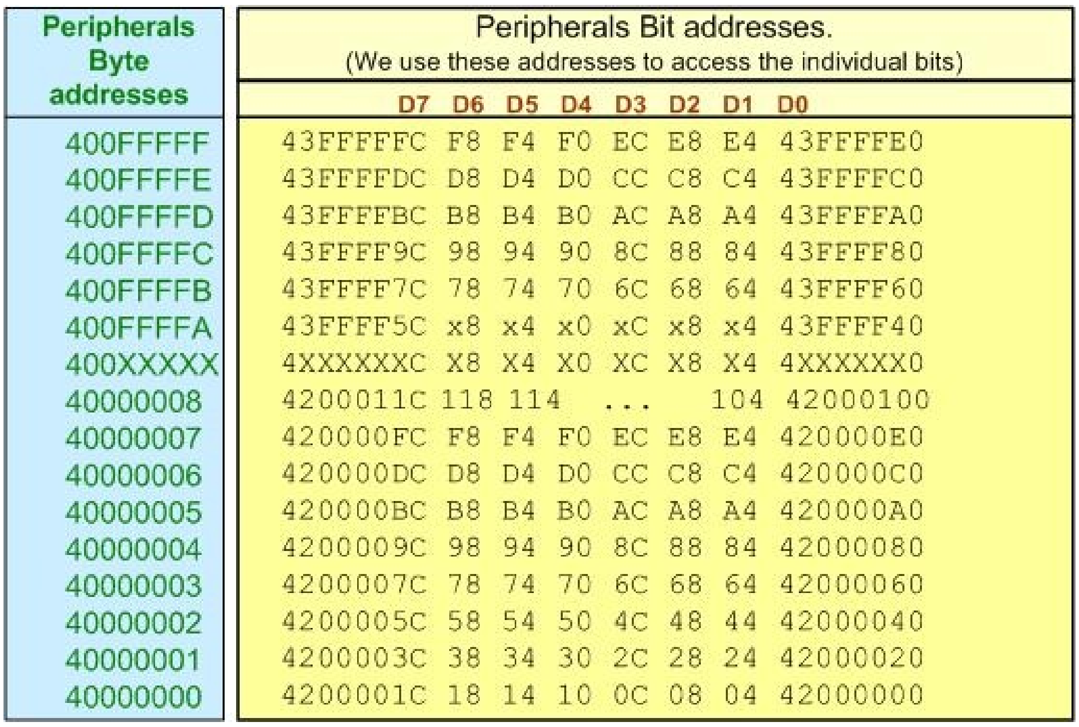
```
LDR     R1,=0x20000005          ;load the address of byte
LDRB     R2,[R1]           ;get the byte
AND     R2,R2,#2_11111110      ;make D0 bit low
STRB    R2,[R1]           ;write it back
```

b) From Figure 6-7 we have address 0x220000A0 as the bit address of D0 of SRAM location 0x20000005.

```
LDR R2,=0x220000A0     ;load the alias address of the bit
MOV R0,0                ;R0 = 0
STRB R0,[R2]            ;write zero to D0
```

Peripheral I/O port bit-addressable region

The general purpose I/O (GPIO) and peripherals such as ADC, DAC, RTC, and serial COM port are widely used in the embedded system design. In many ARM-based trainer boards we see the connection of LEDs, switches, and LCD to the GPIO pins of the ARM chip. In such trainers the vendor provides the details of I/O port and peripheral connections to the ARM chip in addition to their address map. As we discussed earlier, the ARM has set aside 1M bytes of address space to be used bit-band (bit-addressable) I/O and peripherals. The address space assigned to bit-band peripherals and GPIO is 0x40000000 to 0x400FFFFF with address aliases of 0x42000000 to 0x43FFFFFF. Examine your trainer board data sheet for the bit-band addresses implemented on the ARM chip.



Peripherals Byte addresses	Peripherals Bit addresses. (We use these addresses to access the individual bits)							
	D7	D6	D5	D4	D3	D2	D1	D0
400FFFFF	43FFFFFC	F8	F4	F0	EC	E8	E4	43FFFFE0
400FFFFE	43FFFFDC	D8	D4	D0	CC	C8	C4	43FFFFC0
400FFFFD	43FFFFBC	B8	B4	B0	AC	A8	A4	43FFFFA0
400FFFFC	43FFFF9C	98	94	90	8C	88	84	43FFFF80
400FFFFB	43FFFF7C	78	74	70	6C	68	64	43FFFF60
400FFFFA	43FFFF5C	x8	x4	x0	xC	x8	x4	43FFFF40
400XXXXX	4XXXXXXC	X8	X4	X0	XC	X8	X4	4XXXXXX0
40000008	4200011C	118	114		...		104	42000100
40000007	420000FC	F8	F4	F0	EC	E8	E4	420000E0
40000006	420000DC	D8	D4	D0	CC	C8	C4	420000C0
40000005	420000BC	B8	B4	B0	AC	A8	A4	420000A0
40000004	4200009C	98	94	90	8C	88	84	42000080
40000003	4200007C	78	74	70	6C	68	64	42000060
40000002	4200005C	58	54	50	4C	48	44	42000040
40000001	4200003C	38	34	30	2C	28	24	42000020
40000000	4200001C	18	14	10	0C	08	04	42000000

Figure 6-8: Peripherals bit-addressable region and their alias addresses.

Bit map for I/O peripherals

From Figure 6-8 once again notice the following facts.

1. The bit address 0x42000000 is assigned to D0 of peripherals location 0x40000000.
2. The bit address 0x42000004 is assigned to D1 of peripherals location 0x40000000.
3. The bit address 0x42000008 is assigned to D2 of peripherals location of 0x40000000.
4. The bit address 0x4200000C is assigned to D3 of peripherals location of 0x40000000.
5. The bit address 0x42000010 is assigned to D4 of peripherals location 0x40000000.
6. The bit address 0x42000014 is assigned to D5 of peripherals location 0x40000000.
7. The bit address 0x42000018 is assigned to D6 of peripherals location 0x40000000.
8. The bit address 0x4200001C is assigned to D7 of peripherals location 0x40000000.

When accessing a peripheral port in a single-bit manner, we must use the address aliases of 0x42000000 – 0x43FFFFFF. Indeed the bit-addressable peripherals are widely used in embedded system design.

Review Questions

1. True or false. All bytes of SRAM in ARM are bit-addressable.
2. True or false. All bits of the I/O peripherals in ARM are bit-addressable.
3. True or false. All ROM locations of the ARM are bit-addressable.
4. Of the 4G bytes of memory in the ARM, how many bytes are bit-addressable? List them.
5. How would you check to see whether bit D0 of location 0x20000002 is high or low?
6. Find out to which byte each of the following bits belongs. Give the address of the RAM byte in hex.

(a) 0x23000030 (b) 0x23000040 (c) 0x23000048

(d) 0x4200003C (e) 0x43FFFFFC

Section 6.4: Advanced Indexed Addressing Mode

In previous chapters we showed how to use a simple indexed addressing mode in STR, LDR, LDM and STM. The advanced addressing modes bring very important advantages in the ARM and will be discussed in this section.

Indexed addressing mode

In the indexed addressing mode, a register is used as a pointer to the data location. The ARM provides three indexed addressing modes. These modes are: preindex, preindex with write back, and post index. Table 6-2 summarizes these modes. Each of these indexed addressing mode can be used with offset of fixed value or offset of a shifted register. See Table 6-3. In this section we will discuss each mode in detail.

Indexed Addressing Mode	Syntax	Pointing Location in Memory	Rm Value After Execution
Preindex	LDR Rd,[Rm,#k]	Rm+#k	Rm
Preindex with WB*	LDR Rd,[Rm,#k]!	Rm+#k	Rm + #k
Postindex	LDR Rd,[Rm],#k	Rm	Rm + #k
WB means Writeback* * Rd and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095*			

Table 6-2: Indexed Addressing in ARM

Offset	Syntax	Pointing Location
Fixed value	LDR Rd,[Rm,#k]	Rm+#k
Shifted register	LDR Rd,[Rm,Rn, <shift>]	Rm+(Rn shifted <shift>)
** Rn and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095* *** <shift> is any of shifts studied in Chapter3 like LSL#2*		

Table 6-3: Offset of Fixed Value vs. Offset of Shifted Register

Preindexed addressing mode with fixed offset

In this addressing mode, a register and a positive or negative immediate value are used as a pointer to the data location. The value of register does not change after instruction is executed. This addressing mode can be used with STR, STRB, STRH, LDR, LDRB, and LDRH. See Example 6-13.

Example 6-13

Write a program to store contents of R5 to the SRAM location 0x10000000 to 0x10000000F using preindexed addressing mode with fixed offset.

Solution:

```
    LDR     R5,=0x55667788
    LDR     R1,=0x10000000
    ;load the address of first location
    STR     R5,[R1] ;store R5 to location 0x10000000
    STR     R5,[R1,#4]
    ;store R5 to location 0x10000000 + 4 (0x10000004)
    STR     R5,[R1,#8]
    ;store R5 to location 0x10000000 + 8 (0x10000008)
    STR     R5,[R1,#0x0C]
    ;store R5 to location 0x10000000 + 0x0C ( 0x1000000C)
```

Notice that after running this code the content of R1 is still 0x10000000

It is a common practice to use a register to point to the first location of the memory space and access the different locations using proper offsets. For example see the following program:

```
    ADR     R0,OUR_DATA             ;point to OUR_DATA
    LDRB    R2,[R0,#1]              ;load R2 with BETA
    …
OUR_DATA
    ALFA    DCB     0x30
    BETA    DCB     0x21
```

Preindexed addressing mode with write-back and fixed offset

This addressing mode is like preindexed addressing mode with fixed offset except that the calculated pointer is written back to the pointing register. We put ! after the instruction to tell the assembler to enable writeback in the instruction. See Example 6-14.

Example 6-14

Rewrite Example 6-13 using preindexed addressing mode with writeback and fixed offset.

Solution:

```
    LDR     R1,=0x10000000          ;load the address of first location
    STR     R5,[R1]            ;store R5 to location 0x10000000
    STR     R5,[R1,#4]!
    ;store R5 to location 0x10000000 + 4 (0x10000004)
    ;writeback makes R1 = 0x10000004
    STR     R5,[R1,#4]!
    ;store R5 to location 0x10000004 + 4 (0x10000008)
            ;writeback makes R1 = 0x10000008
    STR     R5,[R1,#4]!
    ;store R5 to location 0x10000008 + 4 (0x1000000C)
            ;writeback makes R1 = 0x1000000C
```

Notice that after running this code the content of R1 is 0x1000000C

Postindexed addressing mode with fixed offset

This addressing mode is like preindexed addressing mode with writeback and fixed offset except that the instruction is executed on the location that Rn is pointing to regardless of offset value. After running the instruction, the new value of the pointer is calculated and written back to the pointing register. Examine the following instructions:

```
    STR     R1,[R2],#4       ;store R1 onto memory pointed to by
                               ; R2 and then writeback R2 + 4 to R2
    LDRB    R5,[R3],#1       ;load a byte from memory pointed to
                               ; by R3 and then writeback R3 + 1 to R3
```

Notice that writeback is by default enabled in postindexed addressing and there is no need to put ! after instructions because postindexing without writeback is useless as the index is neither used in the instruction nor written back to the pointing register. See Example 6-15.

Example 6-15

Rewrite Example 6-13 using postindexed addressing mode with fixed offset.

Solution:

```
    LDR     R1,=0x10000000          ;load the address of first location
    STR     R5,[R1],#4       ;store R5 to location 0x10000000 and writeback
```

```
                                  ;0x10000000 + 4 (0x10000004) to R1
   STR        R5,[R1],#4        ;store R5 to location 0x10000004 and writeback
                                  ;0x10000004 + 4 (0x10000008) to R1
   STR        R5,[R1],#4        ;store R5 to location 0x10000008 and writeback
                                  ;0x10000008 + 4 (0x1000000C) to R1
   STR        R5,[R1],#4        ;store R5 to location 0x1000000C and writeback
                                  ;0x1000000C + 4 (0x10000010) to R1
```

Notice that after running this code the content of R1 is 0x10000010.

Preindexed address mode with offset of a shifted register

This advanced addressing mode is a very important feature in the ARM. We start describing this mode from simple case with no shift (shift = 0) and then we will focus on more complex formats.

Simple format of preindexed address mode with offset register

The following is the simple syntax for LDR and STR.

```
   LDR       Rd,[Rm,Rn]       ;Rd is loaded from location Rm + Rn of memory
   STR       Rs,[Rm,Rn]       ;Rs is stored to location Rm + Rn of memory
```

This addressing mode is widely used in implementing of object oriented programs when we want to make a dynamic array of variables. Example 6-16 shows how we use this addressing mode in accessing different locations of an array defined in memory.

Example 6-16

Examine the value of R5 and R6 after the execution of the following program.

```
POINTER          RN        R2
ARRAY1           RN        R1
   AREA    EXAMPLE_6_16, CODE, READONLY
   ENTRY
   LDR       ARRAY1, = MYDATA

   LDRB      R4,[ARRAY1]
   ;load POINTER location of ARRAY1 to R4 (R4= 0x45)
```

```
    MOV     POINTER,#1       ;POINTER = 1 to point to location 1 of array
    LDRB      R5,[ARRAY1,POINTER]
    ;Load POINTER location of ARRAY1 to R5
    ;(R5 = 0x24)
    MOV     POINTER,#2       ;POINTER = 2 to point to location 2 of array
    LDRB      R6,[ARRAY1,POINTER]
    ;load POINTER location of ARRAY1 to R6
    ;(R6 = 0x18)
HERE      B           HERE
MYDATA            DCB       0x45,0x24,0x18,0x63
    END
```

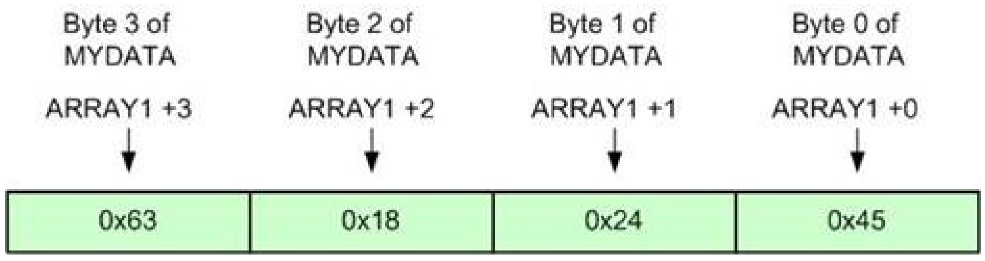
Solution:

After running the LDRB R4,[ARRAY1] instruction, location 0 of MYDATA is loaded to R4. Now R4=0x45.

Next, after running the LDRB R5,[ARRAY1,POINTER] instruction, location 1 of MYDATA is loaded to R5. Now R5 = 0x24.

Next, after running the LDRB R6,[ARRAY1,POINTER] instruction, location 2 of MYDATA is loaded to R6. So the content of R6 = 0x18.

Notice that purposely we used DCB and LDRB in this example so each location of MYDATA takes one byte.



In Example 6-16, we purposely used DCB and LDRB so each location of MYDATA takes one byte. If we define an array using DCD, then we will not be able to use LDRB R5,[ARRAY1,POINTER] to load the POINTER location of ARRAY. See Example 6-17 for clarification.

Example 6-17

In Example 6-16, change MYDATA DCB 0x45,0x24,0x18,0x63 to MYDATA DCD 0x45,0x2489ACF5 and examine the value of R5 and R6 after the execution of the following program.

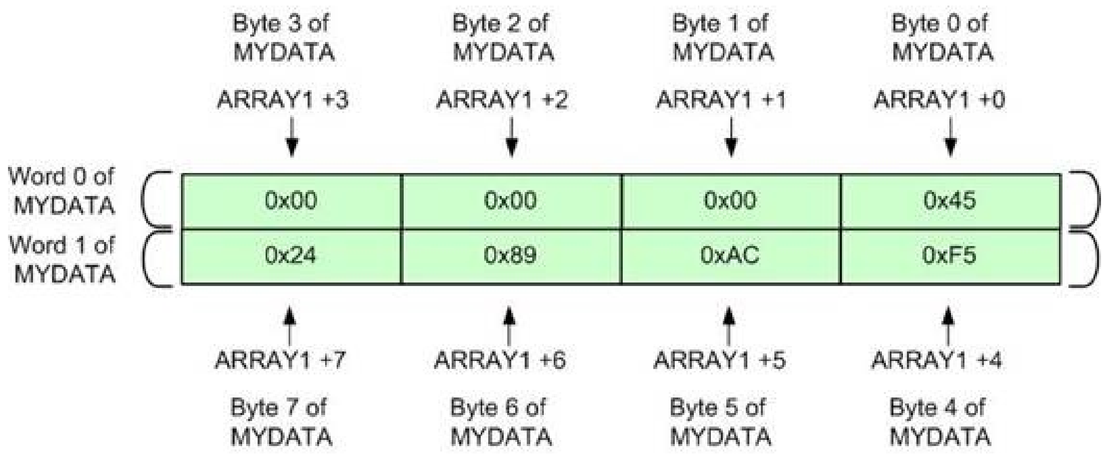
```
POINTER          RN       R2
ARRAY1           RN       R1
            AREA   EXAMPLE_6_17, CODE, READONLY
   ENTRY
   LDR      ARRAY1, = MYDATA

   LDRB     R4,[ARRAY1]      ;load POINTER location of
   ;ARRAY1 to R4 (R4= 0x45)
   MOV     POINTER,#1       ;POINTER = 1 to point to
   ;location 1 of array
   LDRB     R5,[ARRAY1,POINTER]
   ; Load POINTER location of ARRAY1 to R5
   MOV     POINTER,#2       ;POINTER = 2 to point to
   ;location 2 of array
   LDRB     R6,[ARRAY1,POINTER]
   ; load POINTER location of ARRAY1 to R6
HERE     B          HERE
MYDATA           DCD      0x45,0x2489ACF5
   END
```

Solution:

After running the LDRB R4,[ARRAY1] instruction, location 0 of MYDATA is loaded to R4. Now R4=0x45.

Next, after running the LDRB R5,[ARRAY1,POINTER] instruction, location 1 of MYDATA is loaded to R5. Now R5 = 0x00.

Next, after running the LDRB R6,[ARRAY1,POINTER] instruction, location 2 of MYDATA is loaded to R6. So the content of R6 = 0x00.



To access locations of a word size array we have to multiply the pointer by four. Similarly, to access locations of a half-word size array we have to multiply the pointer by two. For example, we can correct program of Example 6-17 by replacing

```
MOV     POINTER,#2       ;POINTER = 2 to point to location 2 of array
```

instruction with following instructions:

```
MOV     POINTER,#2       ;POINTER = 2
MOV     POINTER,POINTER,LSL#2 ;POINTER is shifted left two bits (×4)
;to point to word 2 of the array
```

Notice that by shifting left a value two bits, we multiply it by four. Next we will see how we can use indexed addressing with shifted registers to combine multiplication with the LDR and STR instructions.

General format of preindexed address mode with offset register

The general format of indexed addressing with shifted register for LDR and STR is as follows:

```
LDR Rd,[Rm,Rn,<shift>]           ;(Shifted Rn) + Rm is used as the pointer
STR Rd,[Rm,Rn,<shift>]           ;(Shifted Rn) + Rm is used as the pointer
```

In the above instructions <shift> can be any of shift instructions studied in Chapter 3 such as LSL, LSR, ASR and ROR. Examine the following instructions:

```
LDR     R1,[R2,R3,LSL#2]         ;R2 +(R3 × 4) is used as the pointer
                         ;content of location R2+(R3×4) is loaded to R1
STR     R1,[R2,R3,LSL#1]         ;R2 +(R3 × 2) is used as the pointer
                                 ;R1 is stored to location R2 + (R3×2)
STRB    R1,[R2,R3,LSL#2]         ;R2 +(R3 × 4) is used as the pointer
                                 ;first byte of R1 is stored to location R2+(R3×4)
LDR     R1,[R2,R3,LSR#2]         ;R2 +(R3 / 4) is used as the pointer
```

```
                                        ;content of location R2+(R3 / 4) is
                              ;loaded to R1
```

From the above codes we can see that indexed addressing with shifted register is mostly used to multiply the pointer by a power of two and that is why it is also called indexed addressing with scaled register. Examine Example 6-18 to see how we can use scaled register indexing to access an array of words.

Notice that scaled register indexing is not supported for halfword load and store instructions.

Example 6-18

Examine the value of R5 and R6 after the execution of the following program.

```
POINTER         RN       R2
ARRAY1          RN       R1
   AREA   EXAMPLE_6_18, CODE, READONLY
   ENTRY
   LDR      ARRAY1, = MYDATA
   MOV     POINTER,#0                    ; POINTER = 0
   LDR      R4,[ARRAY1,POINTER,LSL#2]     ;

   MOV     POINTER,#1                    ; POINTER = 1
   LDR      R5,[ARRAY1,POINTER,LSL#2]     ;

   MOV     POINTER,#2                    ; POINTER = 2
   LDR      R6,[ARRAY1,POINTER,LSL#2]     ;

HERE    B         HERE
MYDATA          DCD     0x45,0x2489ACf5,0x2489AC23
   END
```

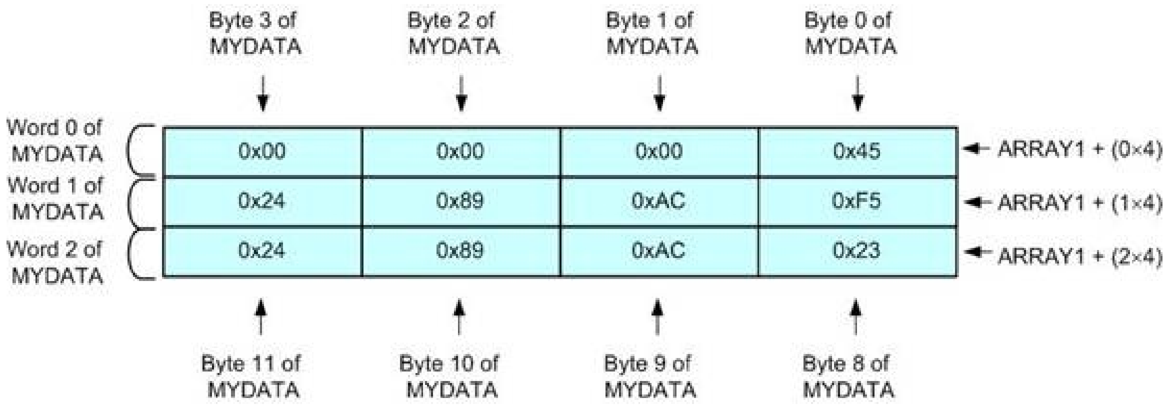
Solution:

After running the LDR R4,[ARRAY1,POINTER,LSL#2]] instruction, location 0 of

MYDATA is loaded to R4. Now R4=0x45.

Next, after running the LDR R5,[ARRAY1,POINTER,LSL#2] instruction, location 1 of MYDATA is loaded to R5. Now R5 = 0x2489ACF5.

Next, after running the LDR R6,[ARRAY1,POINTER,LSL#2] instruction, location 2 of MYDATA is loaded to R6. So the content of R6 = 0x2489AC23.



Writeback sign (!) in preindexed load and store with scaled register

We can force all scaled register load and store instructions to writeback the calculated pointer to the pointing register by putting ! after each load and store instructions. Examine the following instructions:

```
LDR     R1,[R2,R3,LSL#2]!
;R2 +(R3 ×4) is used as the pointer
                    ;content of location R2+(R3×4) is loaded to R1
          ;R2 = R2 +(R3 × 4) (R2 is updated.)
STR     R1,[R2,R3,LSL#1]!
;R2 +(R3 ×2) is used as the pointer
                    ;R1 is stored to location R2+(R3 × 2)
          ;R2 = R2 +(R3 × 2) (R2 is updated.)
```

Scaled register postindex

The following instructions are some examples of scaled register postindex in load and store instructions:

```
STR     R1,[R2],R3,LSL #2
;store R1 at location R2 of memory and write back
                    ;R2 + (R3 × 4) to R2.
LDR     R1,[R2],R3,LSL #2
;load location R2 of memory to R1 and write back
                    ;R2 + (R3 × 4) to R2.
```

Look-up table

One use of array is for implementing look-up tables. The look-up table is a widely used concept in microcontroller programming. It allows access to elements of a frequently used table with minimum operations. As an example, assume that for a certain application we need 4 + x^2 values in the range of 0 to 9. To do so, the look-up table is stored in program memory space and accessed as an array. This is shown in Examples 6-19 through 6-21.

Example 6-19

Write a program to get the x value from R9 and send $x^2 + 2x + 3$ to R10. Assume R9 has the x value of 0–9. Use a look-up table instead of a multiply instruction.

Solution:

```
     AREA LOOKUP_EXAMP6_19,READONLY,CODE
     ENTRY
     ADR      R2,LOOKUP        ;point to LOOKUP
     LDRB     R10,[R2,R9]      ;R10 = location R9 of lookup array
HERE     B        HERE               ;stay here forever

LOOKUP DCB     3, 6, 11, 18, 27, 38, 51, 66, 83, 102
    END
```

Example 6-20

Write a program to get the x value from R9 and send factorial of x to R10. Assume R9 has the x value of 0–10. Use a look-up table instead of a multiply instruction.

Solution:

```
    AREA LOOKUP_EXAMP6_20,READONLY,CODE
    ENTRY
    MOV   R9,#5
    ADR   R2,LOOKUP            ;point to LOOKUP
    LDR   R10,[R2,R9,LSL #2]  ;R10 = location R9 of lookup array
```

```
HERE    B   HERE                    ;stay here forever

LOOKUP DCD    1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800
    END
```

Example 6-21

Write a program that calculates 10 to the power of R2 and stores the result in R3. Assume R2 has the x value of 0–6. Use a look-up table instead of a multiply instruction.

Solution:

```
     AREA LOOKUP_EXAMP_6_21,READONLY,CODE
     ENTRY
     ADR     R1,LOOKUP       ;point to LOOKUP
     LDR     R3,[R1,R2,LSL #2]          ;R3 = location R2 of lookup array
HERE    B         HERE                  ;stay here forever

LOOKUP DCD    1, 10, 100, 1000, 10000, 100000, 1000000
    END
```

Writeback options of STM and LDM

The STM and LDM instructions allow you to store and load multiple registers with a single instruction. We can also specify the action to be taken for the pointer. The action can be increment or decrement before or after the pop is done. This is shown in Table 6-4.

Option	Description
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

Table 6-4: Options for LDM and STM instructions

IA stands for Increment After and adds four (the size of register in bytes) to the pointer after load or storing each register.

IB stands for Increment Before and adds four (the size of register in bytes) to the pointer before load or storing each register.

DA stands for Decrement After and subtracts four (the size of register in bytes) from the pointer after load or storing each register.

DB stands for Decrement Before and subtracts four (the size of register in bytes) from the pointer before load or storing each register.

For further clarification assume that R1 = 0x100. Figure 6-9 shows the memory after running STM R1!,{R2,R3} with each of IA, IB, DA and DB options.

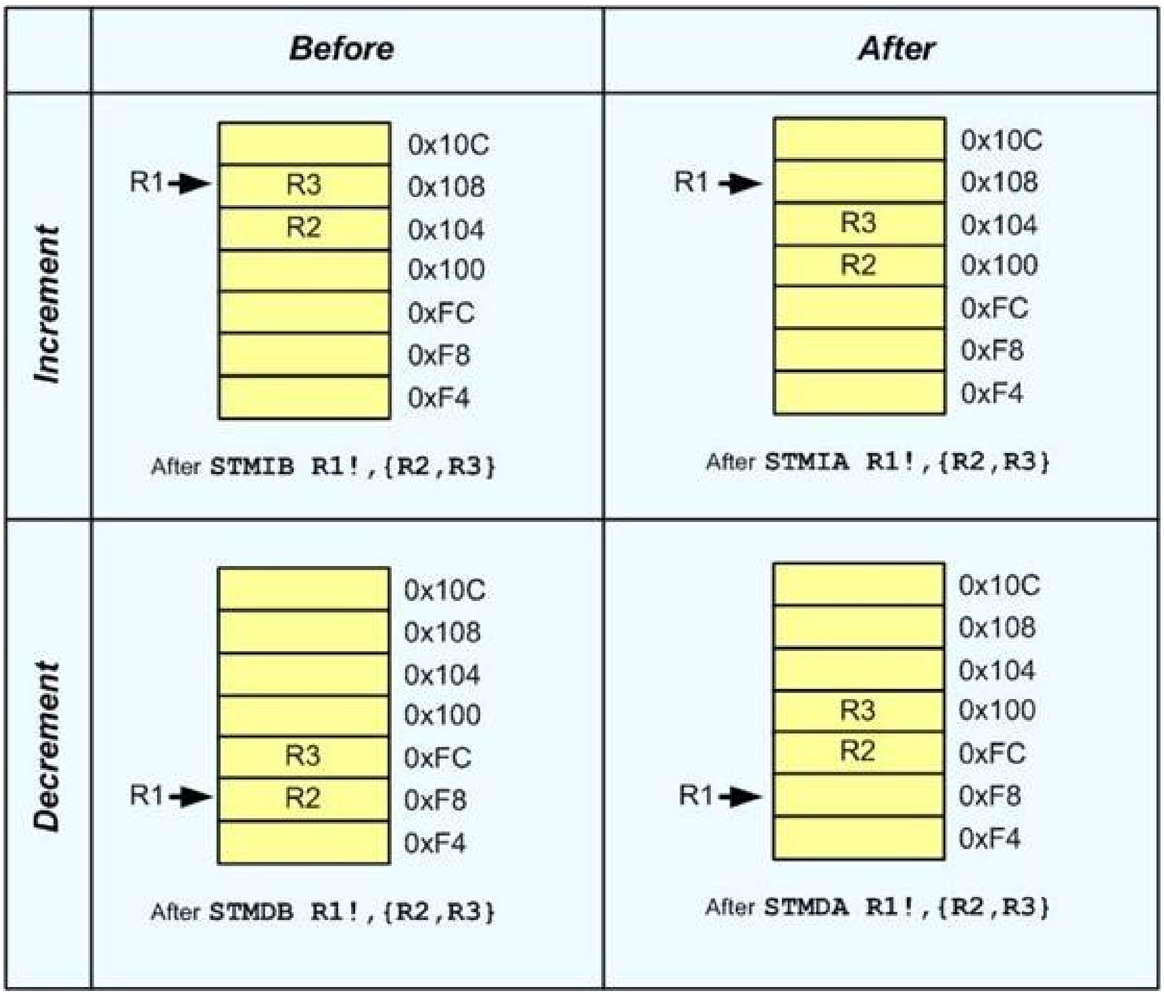
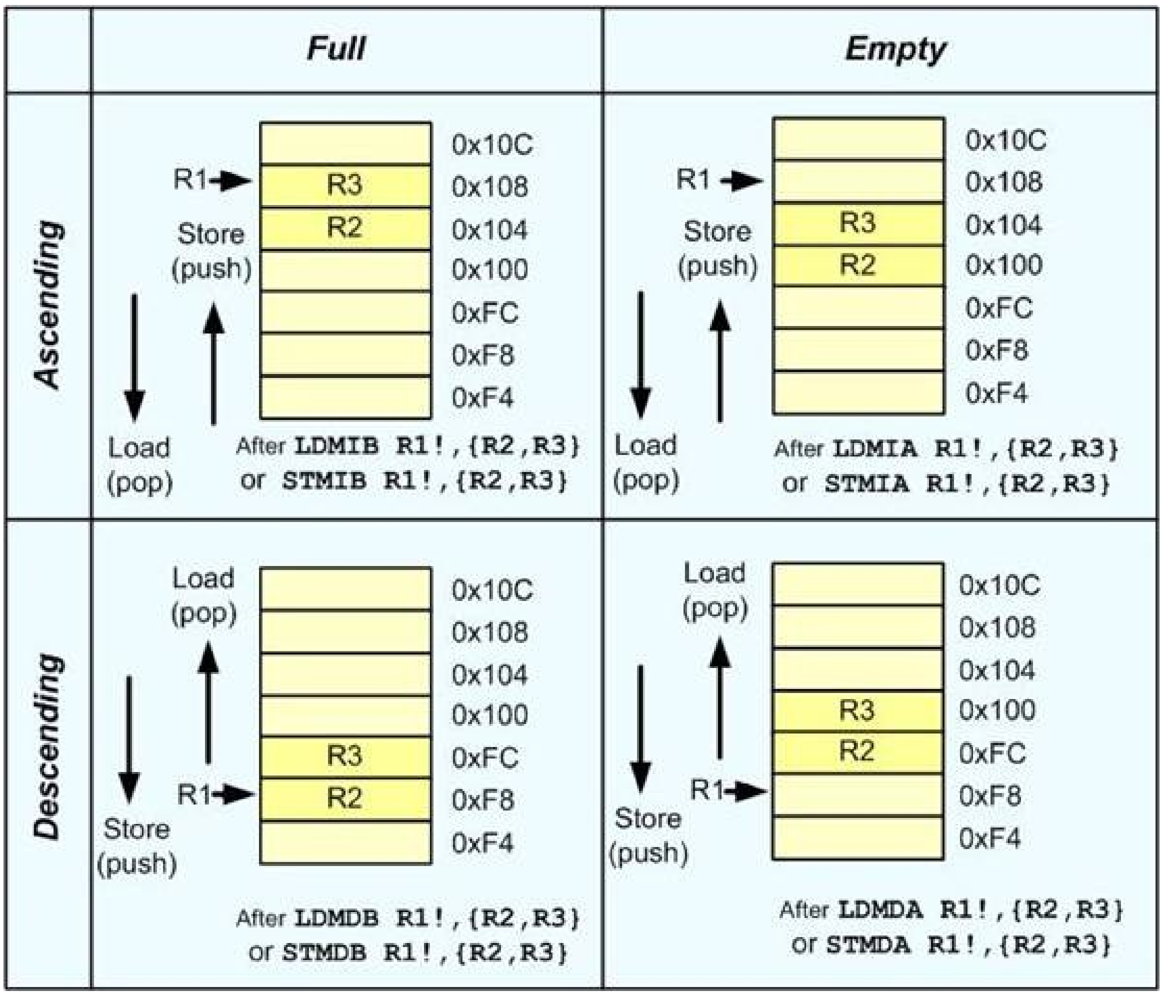


Figure 6- 9: Four Options of STM and LDM in ARM

Notice that generally we have four stack structure, either it is ascending or descending. The stack is called ascending when it is incremented after each store (PUSH) instruction and decremented after each load (POP) instruction. It is called descending when it is decremented after each store (PUSH) instruction and incremented after each load (POP) instruction. The stack pointer can point to the last filled location; in this case the stack is called Full Stack. The stack pointer can point to the next available location, as well; which is called an Empty Stack. See Figure 6-10 for more clarification.



Full
Empty
Ascending
Descending
R1
R3
R2
Store (push)
Load (pop)
0x10C
0x108
0x104
0x100
0xFC
0xF8
0xF4
After LDMIB R1!,{R2,R3} or STMIB R1!,{R2,R3}
After LDMIA R1!,{R2,R3} or STMIA R1!,{R2,R3}
After LDMDB R1!,{R2,R3} or STMDB R1!,{R2,R3}
After LDMDA R1!,{R2,R3} or STMDA R1!,{R2,R3}

Figure 6-10: Four General Stack Structure

To implement a Full Ascending stack we have to use STMIB because the stack pointer should increment on store instruction and it should be incremented before storing each register because it should point to full location. On the other hand we have to use LDMDA for pop instruction because the stack pointer should decrement on load instruction and it should be decremented before loading each full location because it was pointing to an empty location before decrementing. Table 6-5 lists appropriate load and store instruction for each stack structure. It is difficult and prone to error to remember which load and store should be used with each stack structure. To solve this problem, each of load and store options has alternate name which is easy to remember when it is used for stack operation. The last two columns of Table 6-5 list the alternate names.

Stack Structure	Load	Store	Load (alternate Names)	Store (alternate Names)
Full Ascending	LDMDA	STMIB	LDMFA	STMFA
Full Descending	LDMIA	STMDB	LDMFD	STMFD
Empty Ascending	LDMDB	STMIA	LDMEA	STMEA
Empty Descending	LDMIB	STMDA	LDMED	STMED

Table 6- 5: Options for LDM and STM instructions

Program 6-2 uses STM and LDM to simplify program of Example 6-8.

Program 6-2: Using STMFA and LDMFA for Stack (Repeat of Example 6-8)

```
   ;Using Full Ascending Load and Store for stack
   AREA     PROG6_2, CODE, READONLY
   ENTRY
   LDR       R13,=Stack_Top              ;load SP
   LDR       R0,=0x125          ;R0 = 0x125
   LDR       R1,=0x144          ;R1 = 0x144
   MOV     R2,#0x56             ;R2 = 0x56
   BL          MY_SUB             ;call a subroutine
   ADD       R3,R0,R1             ;R3 = R0 + R1 = 0x125 + 0x144 = 0x269
   ADD       R3,R3,R2             ;R3 = R3 + R2 = 0x269 + 0x56 = 0x2BF
HERE      B            HERE                     ;stay here
   ;—————
MY_SUB
   ;——save R0, R1, and R2 on stack before they are used by a loop
   STMFA  R13,{R0-R2}       ;save R0,R1,R2 on stack using Full Ascending
   ;——R0,R1, and R2 are changed
   MOV     R0,#0                  ;R0=0
   MOV     R1,#0                  ;R1=0
   MOV     R2,#0                  ;R2=0
   ;——restore the original registers contents from stack
   LDMFA R13,{R0-R2}
   ;restore R0,R1,and R2 from stack using F. Ascending
   BX          LR                         ;return to caller
   END
```

It must be noted that ARM Cortex has PUSH and POP instructions. See Appendix A for more information.

Review Questions

1. True or false. In a full stack the pointer points to the last stored location.

2. True or false. In a descending stack the pointer increments after each push.
3. Write an instruction that pushes LR (R14) into an empty descending stack.
4. Write an instruction that pops R10 from a full ascending stack.

Section 6.5: ADR, LDR, and PC Relative Addressing

In indexed addressing modes, any registers including the PC (R15) register can be used as the pointer register. For example, the following instruction reads the contents of memory location PC+4:

```
    LDR     R0,[PC,#4]
```

In this way, the data which has a known distance from the current executing line can be accessed. As discussed in Chapter 4, the PC register points 8 bytes (2 instructions) ahead of executing instruction. As a result, “LDR R0,[PC,#4]” accesses a memory location whose address is 4+8 bytes ahead of the current instruction. Generally speaking, the address of the memory location which is being accessed using “LDR R0,[PC,offset]” can be found using this formula: the address of current instruction + 8 + offset.

For instance, if “LDR R0,[PC,#4]” is located in address 0x10 the effective address is: 0x10 + 8 + 4 = 0x1C.

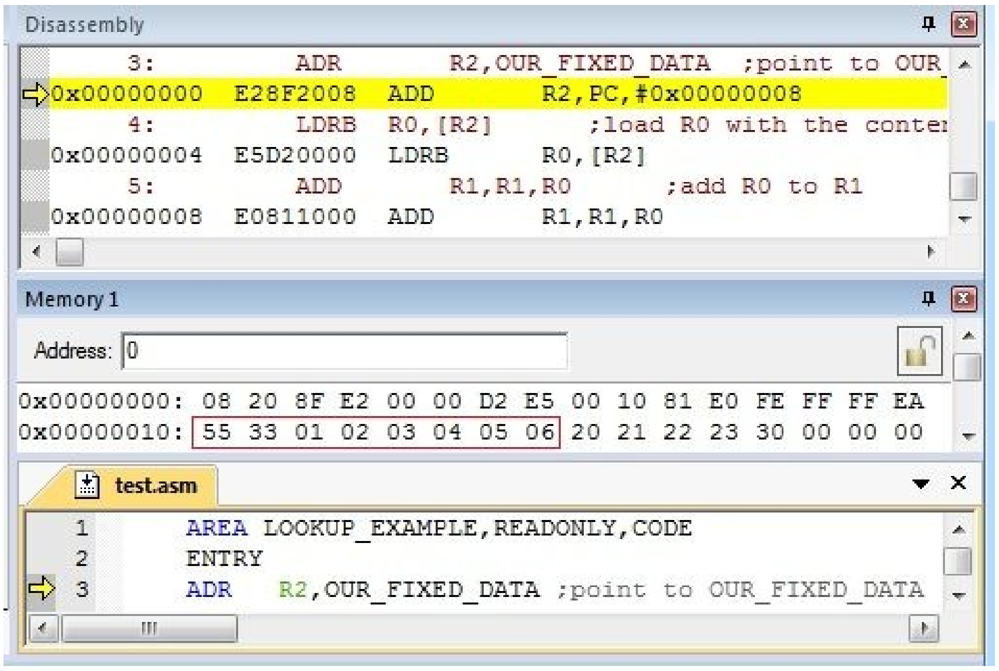
This addressing mode can be considered as a subset of indexed addressing modes. In ARM application notes, the program relative addressing mode is considered as a separate addressing mode.

Implementing the ADR Directive (LDR Rn,=value)

The ADR directive uses the PC relative addressing mode to load registers. For example see the following program:

```
    AREA    LOOKUP_EXAMPLE,READONLY,CODE
    ENTRY
    ADR     R2,OUR_FIXED_DATA   ;point to OUR_FIXED_DATA
    LDRB    R0,[R2]             ;load R0 with the contents
    ;of memory pointed to by R2
    ADD     R1,R1,R0            ;add R0 to R1
HERE    B       HERE                ;stay here forever
OUR_FIXED_DATA
    DCB     0x55,0x33,1,2,3,4,5,6
    DCD     0x23222120,0x30
    DCW     0x4540,0x50
    END
```

See Figure 6-11. At compile time, the ADR is replaced with “ADD R2,PC,#0x08”. Since the instruction is in address 0x00, the instruction accesses location 0 + 8 + 0x08 = 0x10. As shown in the Figure, 0x10 is the address of OUR_FIXED_DATA.



```
Disassembly
       3:          ADR       R2,OUR_FIXED_DATA  ;point to OUR
0x00000000  E28F2008  ADD        R2,PC,#0x00000008
       4:          LDRB  R0,[R2]       ;load R0 with the conter
0x00000004  E5D20000  LDRB       R0,[R2]
       5:          ADD       R1,R1,R0      ;add R0 to R1
0x00000008  E0811000  ADD        R1,R1,R0

Memory 1
Address: 0
0x00000000: 08 20 8F E2 00 00 D2 E5 00 10 81 E0 FE FF FF EA
0x00000010: 55 33 01 02 03 04 05 06 20 21 22 23 30 00 00 00

test.asm
        AREA LOOKUP_EXAMPLE,READONLY,CODE
        ENTRY
        ADR   R2,OUR_FIXED_DATA ;point to OUR_FIXED_DATA
```

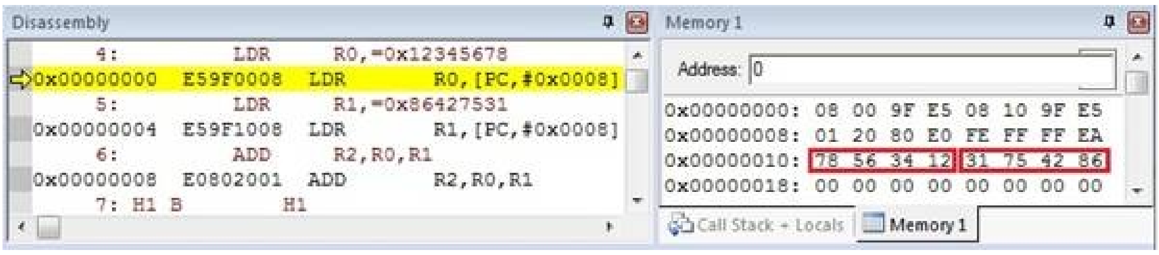
Figure 6- 11: Memory Dump for ADR Instruction

Implementing the LDR Directive

To implement the LDR directive, assembler stores the value as a fixed data in program memory and then accesses it using the LDR instruction and the PC relative addressing mode. Figure 6-12 shows the implementation of Program 6-3. For example, 0x12345678 is stored in memory locations 0x10–0x13, and the LDR directive is replaced with LDR R0,[PC,#0x08]. Since PC=0, the LDR R0, =0x1234567 is located at address 0x000000. Now we have 0+8+8=16=0x10.

Program 6-3: LDR Directive

```
    AREA    EXAMPLE,READONLY,CODE
    ENTRY
    LDR     R0,=0x12345678
    LDR     R1,=0x86427531
    ADD     R2,R0,R1
H1      B       H1
    END
```



```
Disassembly
       4:          LDR       R0,=0x12345678
0x00000000  E59F0008  LDR        R0,[PC,#0x0008]
       5:          LDR       R1,=0x86427531
0x00000004  E59F1008  LDR        R1,[PC,#0x0008]
       6:          ADD       R2,R0,R1
0x00000008  E0802001  ADD        R2,R0,R1
       7: H1 B          H1

Memory 1
Address: 0
0x00000000: 08 00 9F E5 08 10 9F E5
0x00000008: 01 20 80 E0 FE FF FF EA
0x00000010: 78 56 34 12 31 75 42 86
0x00000018: 00 00 00 00 00 00 00 00
```

Figure 6- 12: Memory Dump for LDR Instruction

The same way for the LDR R1,=0x86427531 is located in ROM address 0x00000004. Therefore we have 4+8+8=20=0x14, which is the address of the data 0x86427531.

Review Questions

1. Which register is used as the pointer in PC relative addressing mode?
2. Which directive is more optimized ADR or LDR? Why?

Problems

Section 6.1: ARM Memory Map and Memory Access

1. What is the bus bandwidth unit?
2. Give the variables that affect the bus bandwidth.
3. True or false. One way to increase the bus bandwidth is to widen the data bus.
4. True or false. An increase in the number of address bus pins results in a higher bus bandwidth for the system.
5. Calculate the memory bus bandwidth for the following systems.

   (a) ARM of 100 MHz bus speed and 0 WS

   (b) ARM of 80 MHz bus speed and 1 WS
6. Indicate which of the following addresses is word aligned.

(a) 0x1200004A	(b) 0x52000068	(c) 0x66000082
(d) 0x23FFFF86	(e) 0x23FFFFF0	(f) 0x4200004F
(g) 0x18000014	(h) 0x43FFFFF3	(i) 0x44FFFF05

7. Show how data is placed after execution of the following code using (a) little endian and (b) big endian.

```
LDR     R2,=0xFA98E322
LDR     R1,=0x20000100
STR     [R1],R2
```

8. True or false. In ARM, instructions are always word aligned.
9. True or false. In a word aligned address the lower digit of the address is 0, 4, 8, or C.
10. Show how many memory cycles does it take to fetch the following data into register

```
LDR     R1,=0x20000004
        LDRD    [R1],R2
```

11. Show how many memory cycles does it take to fetch the following data into register

```
        LDR     R1,=0x20000102
        LDRD    [R1],R2
```

12. Show how many memory cycles does it take to fetch the following data into register

```
        LDR     R1,=0x20000103
        LDRD    [R1],R2
```

13. Show how many memory cycles does it take to fetch the following data into register

```
LDR     R1,=0x20000006
        LDRH    [R1],R2
```

14. Show how many memory cycles does it take to fetch the following data into register

```
        LDR     R1,=0x20000C10
        LDRB     [R1],R2
```

Section 6.2: Stack and Stack Usage in ARM

15. True or false. In ARM the R13 is designated as stack pointer.
16. When BL is executed, how many locations of the stack are used?
17. When B is executed, how many locations of the stack are used?
18. In ARM, stack pointer is __________ register.
19. Describe how the action associated with the return operation is performed in ARM.
20. Give the size of the stack in ARM.
21. In ARM, which address is saved when BL instruction is executed.

Section 6.3: ARM Bit-Addressable Memory Region

22. Which memory regions of ARM are bit-addressable?
23. Give the bit-addressable SRAM region address for generic ARM.
24. What bit addresses are assigned to byte address of 0x20000004?
25. What bit addresses are assigned to byte address of 0x20000010?
26. What bit addresses are assigned to byte address of 0x200FFFFF?
27. What bit addresses are assigned to byte address of 0x20000020?
28. What bit addresses are assigned to byte address of 0x40000008?
29. What bit addresses are assigned to byte address of 0x4000000C?
30. What bit addresses are assigned to byte address of 0x40000020?
31. The following are bit addresses. Indicate where each one belongs.

(a) 0x2200004C	(b) 0x22000068	(c) 0x22000080
(d) 0x23FFFF80	(e) 0x23FFFF00	(f) 0x4200004C

(g) 0x42000014 (h) 0x43FFFFF0 (i) 0x43FFFF00

32. Of the 4G bytes of memory locations in the ARM, how many of them are also assigned a bit address as well? Indicate which bytes those are.
33. True or false. The bit-addressable region cannot be access in byte.
34. True or false. The bit-addressable region cannot be access in word.
35. Write a program to see whether the D7 bit of RAM location 0x20000020 is high. If so, send a 1 to D1 of RAM location 0x20000000.
36. Write a program to see whether the D7 bit of I/O location 0x40000000 is low. If so, send a 0 to the D0 of location 0x400FFFFF.
37. Write a program to set high all the bits of RAM locations 0x2000000 using the following methods:

    (a) byte addresses (b) bit addresses
38. Write a program to see whether the SRAM location 0x20000000 is divisible by 8.
39. Explain how the LDM instruction works.
40. Explain how the STM instruction works.
41. Explain the difference between LDM and LDR instructions.
42. Explain how the difference between STM and STR instructions.
43. Explain the LDMIA operation and its impact on the SP.
44. Explain the LDMIB operation and its impact on the SP.
45. Explain the STMIA operation and its impact on the SP.
46. Explain the STMIB operation and its impact on the SP.

Section 6.4: Advanced Indexed Addressing Mode

47. True or false. Writeback is by default enabled in preindexed addressing mode.
48. Indicate the addressing mode in each of the following instructions

    (a) LDR R1,[R5],R2,LSL #2 (b) STR R2,[R1,R0]

    (c) STR R2,[R1,R0, LSL #2]! (d) STR R9,[R1],R0
49. What is an ascending stack?
50. What is the difference between an empty and a full stack?
51. Write an instruction that stores R0 in a full descending stack.
52. Write an instruction that loads R9 from an empty descending stack.

Section 6.5: ADR, LDR, and PC Relative Addressing

53. Assuming that the instruction “LDR R2,[PC,#8] is located in address 0x300,

calculate the address of the memory location which is accessed.

54. Using PC relative addressing mode, write an LDR instruction that accesses a memory location which is 0x20 bytes ahead of itself.

Answers to Review Questions

Section 6.1

1. 4 bytes
2. Compilers ensure that codes are word aligned.
3. little endian
4. 1/66 MHz = 15.15 ns is the bus clock period. Since the bus cycle time of zero wait states is 2 clocks, we have 2 × 15.15 = 30.3 ns
5. 1/100 MHz = 10 ns is the bus clock period. 50 ns - 10 ns = 40 ns. The Number of WS is 40 ns / 10 ns = 4.
6. False

Section 6.2

1. R13
2. The stack can be as big as its RAM
3. STM R13, {R5-R8}

   SUB R13, R13, #16
4. ADD R13, R13, #16

   LDM R13, {R5-R8}
5. It copies the contents of locations 0x40000000–0x4000000F into locations 0x50000000–0x5000000F using the LDM and STM instructions.

Section 6.3

1. False
2. False
3. False
4. 2MBytes; locations 0x20000000 to 0x200FFFFF of SRAM and 0x40000000 to 0x400FFFFF of GPIO
5.

```
    LDR     R0,=0x22000040
    LDR     R1,[R0]
    CMP     R1,#0
    BNE     L1
    …
L1:
```

6.

(a) 0x23000030 - 0x22000000 = 0x1000030; 0x1000030 / 0x20 = 0x80001; thus it is in location 0x20000000 + 0x80001 = 0x20080001

(0x1000030 % 32) / 4 = (48 % 32) / 4 = 16 / 4 = 4; it is D4 of 0x20080001.

(b) 0x1000040 / 0x20 = 0x80002; it is in location 0x20080002

(0x1000040 % 0x20) / 4 = 0; it is D0 of location 0x20080002

(c) 0x1000048 / 0x20 = 80002; it is in location 0x20080002

(0x1000048 % 0x20) / 4 = 2; it is D2 of location 0x20080002

(d) 0x4200003C - 0x42000000 = 0x03C; 0x03C / 0x20 = 0x01

(0x3C % 0x20) / 4 = 0x1C / 4 = 7; it is D7 of 0x40000001

(e) 0x43FFFFFC – 0x42000000 = 0x1FFFFFC; 0x1FFFFFC / 0x20 = 0xFFFFF

(0x1FFFFFC % 0x20) / 4 = 0x1C / 4 = 7; D7 of 0x400FFFFF

Section 6.4

1. True
2. False
3. STMED SP!,{LR}
4. LDMFA SP!,{R10}

Section 6.5

1. PC (R15)
2. ADR, To implement the LDR directive the value is stored in memory; as a result, it uses more memory while the ADR uses no memory.

Chapter 7: ARM Pipeline and CPU Evolution

This chapter will look at pipeline evolution in ARM while examining other CPU enhancements. In Section 7.1 the ARM's pipelines are studied. Section 7.2 explores various processors enhancements.

Section 7.1: ARM Pipeline Evolution

There are many ways available to processor designers to increase the processing power of the CPU. Here we list some of them that are used in ARM.

1. Increase the clock frequency of the chip. One drawback of this method is that the higher the frequency, the more the power dissipation and the more difficult and expensive the design of the microprocessor and motherboard.

2. Increase the number of data buses to bring more information (code and data) into the CPU to be processed. For example, Von Neumann architecture in ARM7 has been replaced by Harvard architecture in newer versions. See Chapter 6.

3. Change the internal architecture of the CPU to overlap the execution of more instructions. This requires a lot of transistors. There are two trends for this option, superpipeline and superscalar. In superpipelining, the process of fetching and executing instructions is split into many small steps and all are done in parallel. In this way the execution of many instructions is overlapped. The number of instructions being processed at a given time depends on the number of pipeline stages, commonly termed the pipeline depth. Some designers use as many as 8 stages of pipelining. One limitation of superpipelining is that the speed of the execution is limited to the slowest stage of the pipeline. Compare this to making pizza. You can split the process of making pizza into many stages, such as flattening the dough, putting on the toppings, and baking, but the process is limited to the slowest stage, baking, no matter how fast the rest of the stages are performed. What happens if we use two or three ovens for baking pizzas to speed up the process? This may work for making pizza but not for executing programs, since in the execution of instructions we must make sure that the sequence of instructions is kept intact and that there is no out-of-step execution. The difficulties associated with a stalled pipeline (a slowdown in one stage of the pipeline, which prevents the remaining stages from advancing) has made CPU designers abandon superpipelining in favor of superscaling. In superscaling, the entire execution unit has been doubled and each unit has 5 pipeline stages. Therefore, in superscalar, there is more than one execution unit and each has many stages, rather than one execution unit with 8 stages as in the case of a superpipelined processor. In some superscalar processors, there are two execution units each with 4 pipeline stages instead of a single execution unit with 8 pipeline stages as superpipelining proponents would have it. In other words, in superscaling we have two (or even three) execution units and as the instructions are fetched they are issued to the various execution units. Using the analogy of pizza, superscalar is like doubling or tripling the entire crew flattening the dough, putting toppings on, and baking. Of course, you will need a lot more people involved in the process and you have to have more ovens, but at the same time you are doubling or tripling the pizza output. In cases of recent microprocessor architecture, a vast majority of designers have chosen superscaling over superpipelining. This requires numerous transistors to duplicate several execution units, just like needing more people in our pizza-making analogy. Fortunately, advances in IC design have allowed designers access to hundreds of million transistors to throw around for the

implementation of powerful superscaling. There are some problems with superscaling, such as data dependency issues, which can be solved by the compiler.

4. Combining more than one core in a single processor is another way of improving the speed of high end processers. Cortex-A series of ARM supports up to four cores in combination with each other.

Next, we will examine the issue of pipelining. See Figure 7-1.

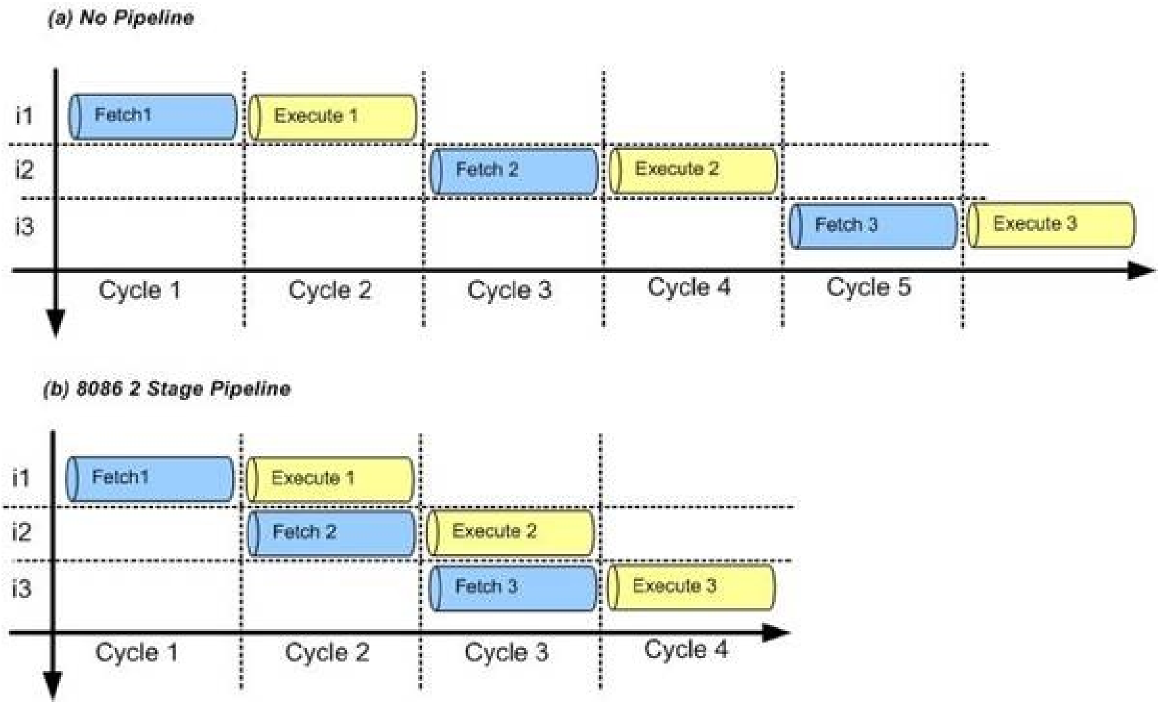


Figure 7-1: Non-Pipelined Instruction Execution vs. 2-stage Pipeline (8086)

More about pipelining

In the early CPUs there was no pipelining. At any given moment, it either fetched or it executed. It could not do both at the same time. In the non-pipelined CPU, while the buses were fetching the instructions (opcodes) and data, the CPU was sitting idle, and in the same way, when the CPU was executing instructions, buses were sitting idle. However, in the early pipelined CPU such as 8086 the fetch and execute were performed in parallel by two sections inside the CPU called the BIU (bus interface unit) and EU (execution unit).

For the concept of pipelining to work, we need a buffer or queue in which an instruction is prefetched and ready to be executed. In some circumstances, the CPU must flush out the queue. For example, when a branch (B, BNE, BCS, and so on) or call instruction is executed, the CPU starts to fetch codes from the new memory location and the code in the queue that was fetched previously is discarded. In this case, the execution unit must wait until the fetch unit fetches the new instruction. This is called a branch penalty. The penalty is an extra instruction cycle to fetch the instruction from the target location instead of executing the instruction right below the branch. Remember that the instruction below the branch has already been fetched and is next in line to be executed when the CPU branches to a different address. Note that newer CPUs have more stages in their pipeline. For example a 3 stage pipeline may divide the code execution to Fetch, Decode and Execute stages. When the number of stages in a pipeline increases, more stages should be flushed out when a branch instruction is executed. Examine Example 7-1 to see how branch penalty slows down the execution of a code. Next, you will see how branch prediction solves the problem.

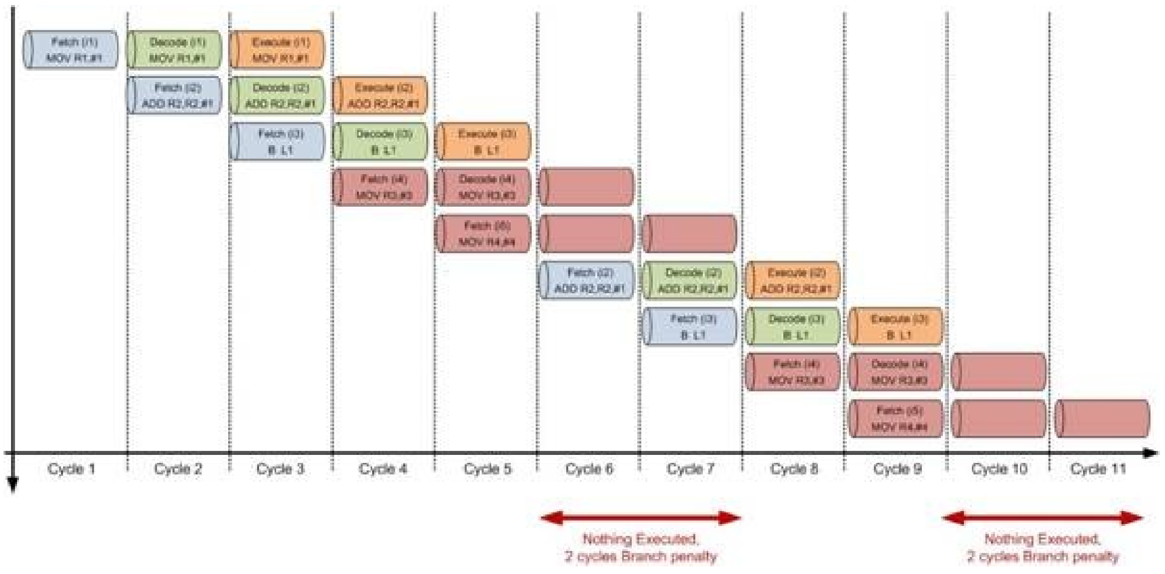
Example 7-1

How many cycles does it take for a 3 stage pipelined CPU to run 3 iteration of the following code?

```
    MOV     R1,#0
L1          ADD     R2,R2,#1
    B           L1
    MOV     R3,#3
    MOV     R4,#4
```

Solution:

For the first instruction (MOV R1,#0), it takes 3 cycles to pass through the stages of pipeline and be executed. After the third cycle, one instruction is executed in each cycle. When the Branch instruction is executed in cycle 5, the CPU flushes the pipeline because the fetch and decode instruction in cycles 4 and 5 are not needed. It causes two clock cycles branch penalty. The same scenario happens each time the CPU executes a branch. As we can see in the figure, It takes 13 cycles to run 7 instructions.



Branch prediction

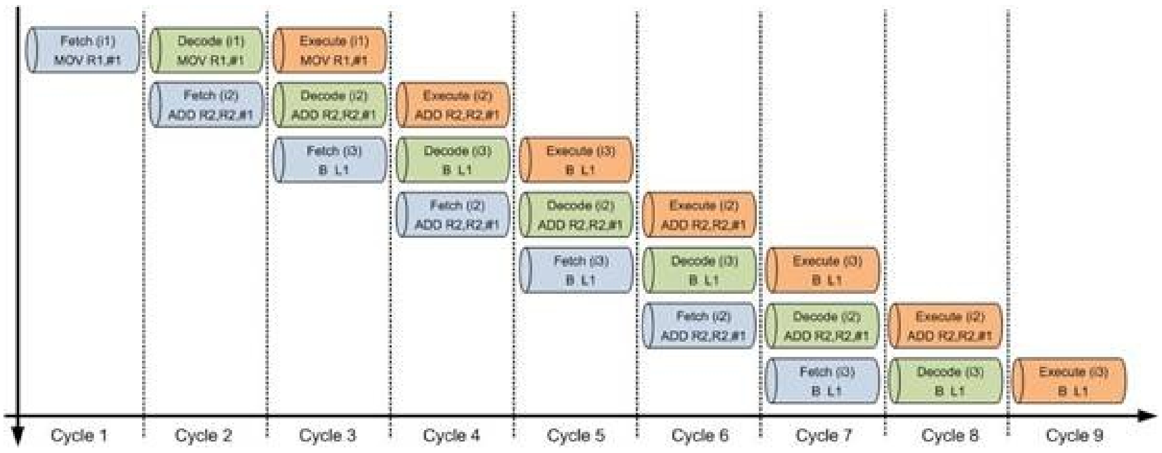
Branch prediction is another new feature of the new CPUs. The penalty for branching is very high for a high-performance pipelined microprocessor such as the ARM. For example, in the case of the BNE (branch if not equal) instruction, if it branches, the pipeline must be flushed and refilled with instructions from the target location. This takes time. In contrast, the instruction immediately below the BNE is already in the pipeline and is advancing without delay. Some processors have the capability to predict and prefetch code from both possible locations and have them advanced through the pipeline without waiting (stalling) for the outcome of the zero flag. The ability to predict branches and avoid the branch penalty can result in a substantial reduction in the clock count for a given program. Some CPUs have branch prediction, but with greater capability. When it encounters branch instructions (such as BNE), it creates a list of them in what is called the branch target buffer (BTB). The BTB predicts the target of the branch and starts executing from there. When the branch is executed, the result is compared with what the prediction section of the CPU said it would do. If they match, the branch is retired. If not, all instructions behind the branch are removed from the pool and the correct branch target address is provided to the BTB. From there the BTB refills the pipeline with instructions from the new target address. See Example 7-2.

Example 7-2

Show how many cycles does it take for a 3-stage pipelined CPU to run 3 iteration of the code in Example 7-1? Assume that the branch prediction unit has predicted all branches.

Solution:

It takes 9 cycles to run 7 instructions. In cycles 4 to 8 instructions are predicted by branch prediction unit.



Note that stores are never performed speculatively since there is no transparent way to undo them. Stores are also never re-ordered among themselves. A store is dispatched only when both the address and the data are available and there are no older stores awaiting dispatch.

3-stage pipeline in ARM7

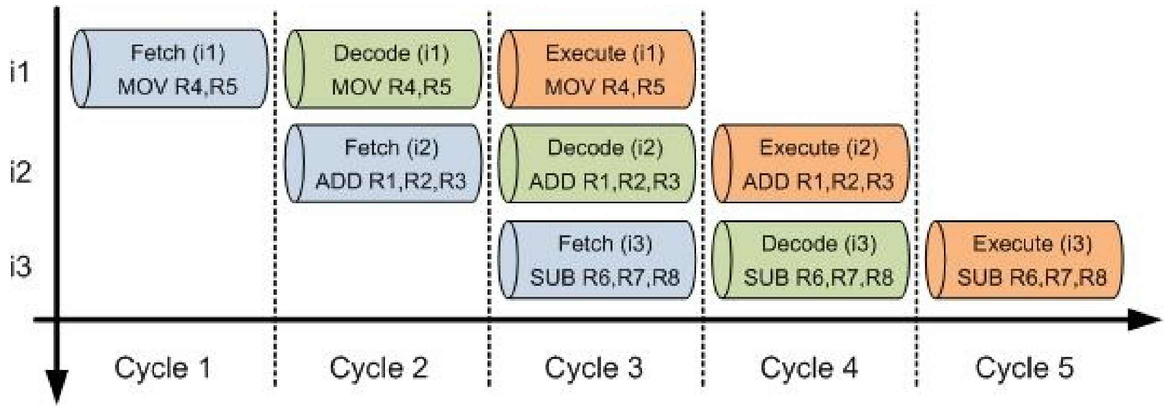
Since the introduction of the 8086 microprocessor in 1978, processor designers have come to rely more and more on the concept of pipelining to increase the processing power of the CPU. ARM7 used the concept of pipelining with three stages of fetch, decode, and execute. See Example 7-3.

Example 7-3

Show how the following code is executed in ARM7.

```
MOV    R4,R5
ADD     R1,R2,R3
SUB     R6,R7,R8
```

Solution:



5-stage pipeline in ARM9

As we mentioned earlier the ARM7 has a 3-stage pipeline. As shown in Figure 7-2, the ARM9 has extended the pipeline to 5 stages. They are:

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write

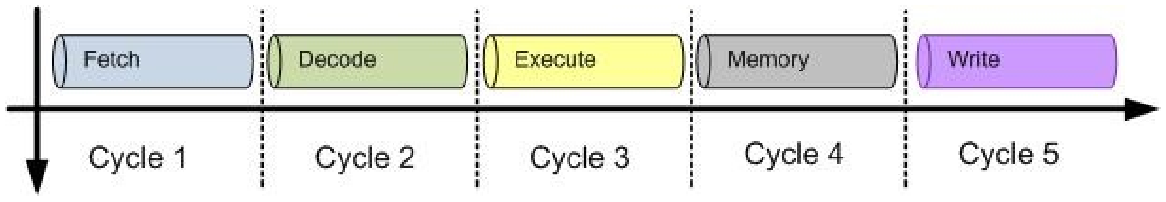


Figure 7- 2: 5-Stage Pipeline in ARM9

Fetch: In the Fetch stage the instructions are fetched from memory and placed in the queue and wait to be decoded. In this stage the Program Counter (PC) is also incremented by 4 since the ARM instructions are 4-bytes.

Decode: In the decode stage the instruction is decoded and the register file is also accessed to get everything ready for the Execute stage.

Execute: In this stage, any effective address calculation and sign extending of a byte or a half-word are done. In the instructions such as Load and Store this stage gets everything ready for the next stage of memory access. In instructions such as "ADD R1,R2,R3" in which all the resources needed for the execution of the instruction are ready before it comes to this stage, the registers are added and it goes directly to the write-back stage to write the result to the register file.

Memory: For instructions such as Load and Store in which external memory accesses are needed, the memory stage fetches the data from the external memory and has the data inside the CPU ready for the next stage of the write-back. If an instruction does not need to access memory, this stage is bypassed and the result is forwarded to the last stage of write-back. This means it becomes a 4-stage pipeline.

Write: Also called write-back is the stage in which the instruction is completed by writing the result to the register file and retiring the instruction. As we just stated, if an instruction does not need to access memory, write-back is the stage right after the execute stage, meaning for many instructions we really have 4-stage pipeline.

3-stage vs. 5-stage pipeline

In the 3-stage pipeline of ARM7, the execution, the memory access, and the writing the result to register file are all performed by the Execute stage. In the 5-stage pipeline, the CPU decouples the memory access and execute stage. With the two new stages of memory and write-back, the ARM9 increases the processing power of the CPU by allowing the CPU to work concurrently on 5 instructions instead of 3 instructions at a given time. This is a major enhancement of the ARM9 over ARM7.

Review Questions

1. What is superpipelining?
2. What is the limitation of superpipeline?
3. What is superscaling?
4. True or false. The 5-stage pipeline has better performance than the 3-stage pipeline.
5. Give the names of the 5-stage pipeline in ARM9.

Section 7.2: Other CPU Enhancements

There are many other ways available to microprocessor designers to increase the processing power of the CPU. Next, we examine some of them.

Superscalar CPUs

Another unique feature of the many of new CPUs is its superscalar architecture. A large number of transistors are used to put more than one execution unit inside the CPU. As the instructions are fetched, they are issued to these execution units. Figure 7-3 shows the concept of superscalar.

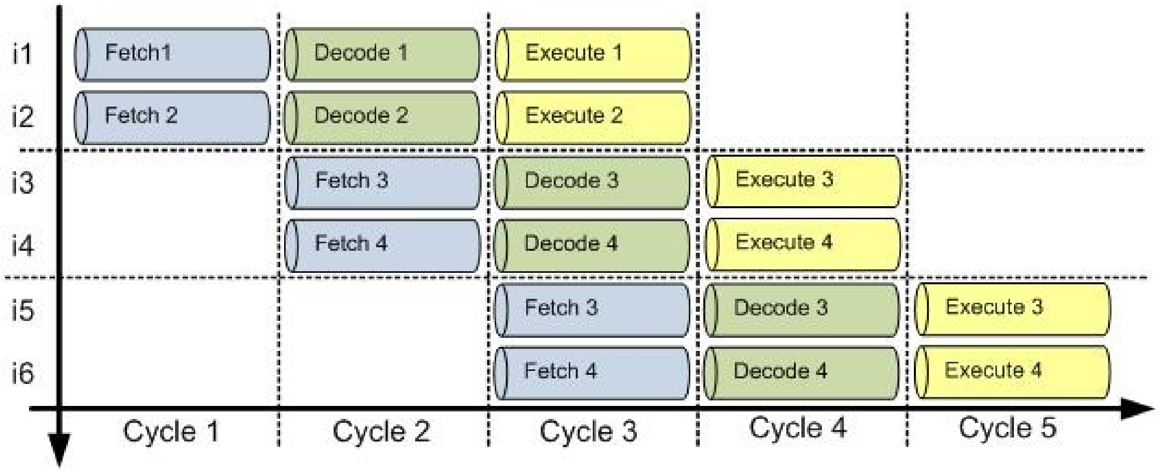


Figure 7- 3: Superscalar CPUs

Issuing two instructions at the same time to different execution units can work only if the execution of one does not depend on the other one, in other words, if there is no data dependency. As an example, look at the following instructions.

```
ADD    R1,R2,R3
SUB    R4,R1,R5
AND    R6,R7,R8
MOV   R9,R10
```

In the above code, the ADD and SUB instructions cannot be issued to two execution units since R1, the destination of the first instruction, is used immediately by the second instruction. This is called *read-after-write dependency* since the SUB instruction wants to read the R1 contents, but it must wait until after the ADD is finished writing it into R1. The problem is that ADD will not write into R1 until the last stage of the pipeline, and by then it is too late for the pipeline of the SUB instruction. This prevents the SUB instruction from advancing in the pipeline, therefore causing the pipeline to be stalled until the ADD finishes writing and then the SUB instruction can advance through the pipeline. This kind of data dependency raises the clock count for the SUB instruction. What if the instructions are rescheduled? We will discuss out of order execution next in this chapter.

```
ADD    R1,R2,R3
AND    R6,R7,R8
```

```
SUB     R4,R1,R5
MOV     R9,R10
```

If they are rescheduled as shown above, each can be issued to separate execution units, allowing parallel execution of both instructions by two different units of the CPU. Since the clock count for each instruction is one, having two execution units leads to executing two instructions by pairing them together, thereby using only one clock count for two instructions. In the case of the above program, if it is run on the CPU with superscalar it will take only 2 clocks instead of 4, assuming that two instructions are paired together. This reordering of instructions to take advantage of the two internal execution units of the CPU can be done by compiler or CPU itself and is called *instruction scheduling*. Currently, some compilers are being equipped to do instruction scheduling to remove dependencies. The process of issuing two instructions to the two execution units is commonly referred to as *instruction pairing*.

Superpipelined and superscalar

Some microprocessors use a 10-stage pipeline for the CPU. In contrast to the 5-pipestage, although each pipe stage of the 10-pipestage performs less work, there are more stages. This means that in such processors, more instructions can be worked on and finished at a time. These CPUs with their 10- or 12-stage pipeline are referred to as *superpipelined*. Since they also have multiple execution units capable of working in parallel, they are also superscalar. Another advantage of the superpipelined concept is that it can achieve a higher clock rate (frequency) with the given transistor technology. They also use what is called out-of-order execution to increase the performance of the CPU. This is explained next.

Decoupling and out-of-order execution

In CPU architecture, when one of the pipeline stages is stalled, the prior stages of fetch and decode are also stalled. In other words, the fetch stage stops fetching instructions if the execution stage is stalled, due, for example, to a delay in memory access. This dependency of fetch and execution has to be resolved in order to increase CPU performance. That is exactly what many designers have done with the CPU and is called *decoupling* the fetch and execution phases of the instructions. In these processors, instructions are fetched from memory and placed into a pool called the *instruction pool*. See Figure 7-4.

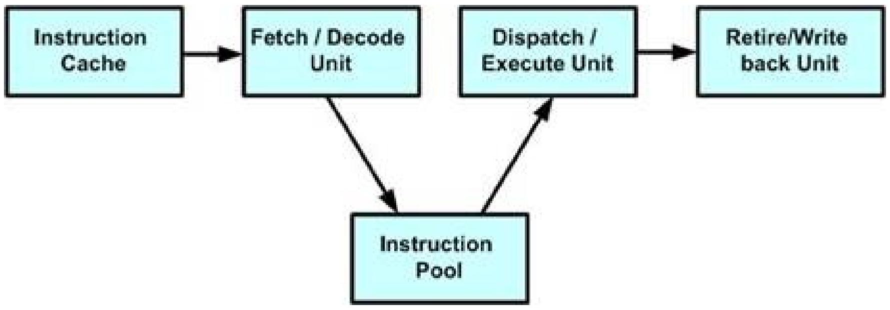


Figure 7- 4: CPU Instruction Execution

This fetch/decode of the instructions is done in the same order as the program was coded by the programmer (or compiler). However, when they are placed in the instruction pool they can be executed in any order as long as the data needed is available. In other words, if there is no dependency, the instructions are executed out of order, not in the same order as the programmer coded them. Such a speculative execution can go 20–30 instructions deep into the program. It is the job of the retire unit to provide the results to the programmer's (visible) registers (e.g., R0, R1) according to the order in which the instructions were coded. Again, it is important to note that the instructions are fetched in the same order that they were coded, but executed out of order if there is no dependency, and ultimately retired in the same order as they were coded. This out-of-order execution can boost performance in many cases. Look at Example 7-4.

Example 7-4

The following ARM code (a) sets the pointer for three different arrays, and the counter value, (b) gets each element of ARRAY_1, adds a fixed value of 100 to it, and stores the result in ARRAY_2, and (c) complements the element and stores it in ARRAY_3. Analyze the execution of the code in light of the out-of-order execution and branch prediction capabilities of an ARM CPU.

```
(i1)               LDR     R1,=ARRAY_1   ;load pointer
(i2)               LDR     R2,=ARRAY_2   ;load pointer
(i3)               LDR     R3,=ARRAY_3   ;load pointer
(i4)               MOV     R4,#COUNT     ;load the counter
(i5)        AGAIN  LDR     R5,[R1]       ;load the element
(i6)               ADD     R5,R5,#100    ;add the fix value
(i7)               ADD     R1,R1,#4      ;update the pointer
(i8)               STR     R5,[R2]       ;store the result
(i9)               ADD     R2,R2,#4      ;update the pointer
(i10)              MVN     R5,R5         ;complement the result
(i11)              STR     R5,[R3]       ;and store it
(i12)              ADD     R3,R3,#4      ;update the pointer
(i13)              SUBS    R4,R4,#4      ;
(i14)              BNE     AGAIN         ;stay in the loop
(i15)
```

Solution:

The fetch/decode unit fetches and puts instructions into the pool. Since there is no dependency for instructions i1 through i5, they are dispatched, executed, and retired except for i5. Notice that the pointer values of i1 to i4 are immediate values; therefore, they are embedded into the instruction when the fetch/decode unit gets them. Now i5 is a memory fetch that can take many clocks, depending on whether the needed data is located in cache or main memory. Meanwhile i6, i8, i10, and i11 must wait until the data is available. However i7, i9, and i12 can be executed out of order. More importantly, the BNE instruction is predicted to go to the target address of AGAIN and i5, i6, … are dispatched once more for the next iteration. This time the memory fetch will take very few clocks since in the previous data fetch, the CPU read some bytes of data into the cache. This process will go on until the last round of looping where R4 becomes zero and falls through. At this time, due to misprediction, all the instructions belonging to instructions i5, i6, i7, … (start of the loop) are removed and the whole pipeline restarts with instructions belonging to i15, i16, and so on.

Due to the fact that memory fetches (due to cache misses) can take many clock cycles and result in underutilization of the CPU, out-of-order execution is a way of finding something to do for the CPU. Simply put, the idea of out-of-order execution is to look deep into the stream of instructions and find the ones that can be executed ahead of others, provided that resources are available. Again, it is important to note that these processors will not immediately provide the results of out-of-order executions to programmer-visible registers such as R0, R1, and so on, since it must maintain the original order of the code. Instead, the results of out-of-order executions are stored in the pool and wait to be retired in the same order as they were coded. Therefore, programmer-visible registers are updated in the same sequence as expected by the programmer.

Register renaming

There are some cases in which instructions are not really dependent on each other but there is a kind of implicit dependency called *register dependency*. See Example 7-5.

Example 7-5

For the following code, indicate the instructions that can be executed in parallel or out of order.

```
(i1) LDR  R4,[R2]          ;load R4 from memory pointed to by R2
(i2) ADD  R3,R4,R7         ;R4+R7–>R3
(i3) ADD  R6,R8,R10        ;R8+R10–>R6
(i4) SUB  R5,R1,R9         ;R1-R9–>R5
(i5) ADD  R6,R12,#1        ;R12+1–->R6
```

Solution:

Instruction i2 cannot be executed until the data is brought in from memory (either cache or main memory DRAM). Therefore, i2 is dependent on i1 and must wait until the R4 register has the data. However, instructions i3 and i4 can be executed out of order and parallel with each other since there is no dependency among them. Notice that i5 is not really dependant on i3 because i5 does not use any of data generated by i3. But i5 and i3 cannot be executed out of order or in parallel because R6 is modified by both of i3 and i5. This kind of dependency is called register dependency and is solved by a method called register renaming.

In the following code none of the instructions can be executed in parallel because of using R1 in all instructions:

```
MOV    R1,#5
ADD     R3,R1,#2
MOV    R1,#6
ADD     R4,R1,#2
```

If you examine the above code carefully you will see that the first two lines of code are independent from the second two lines of code and we can remove the implicit dependency by changing R1 to another register such as R2 in the last two lines of code:

```
MOV    R1,#5
ADD     R3,R1,#2
MOV    R2,#6
ADD     R4,R2,#2
```

Renaming the registers before issuing the instructions to execution unit is done in many of new advanced CPU and it is called register renaming.

Putting them all together in an ARM CPU

In Figure 7-5 you can see a top-level diagram of the ARM Cortex A9 processor. It has most of the parts discussed in this chapter.

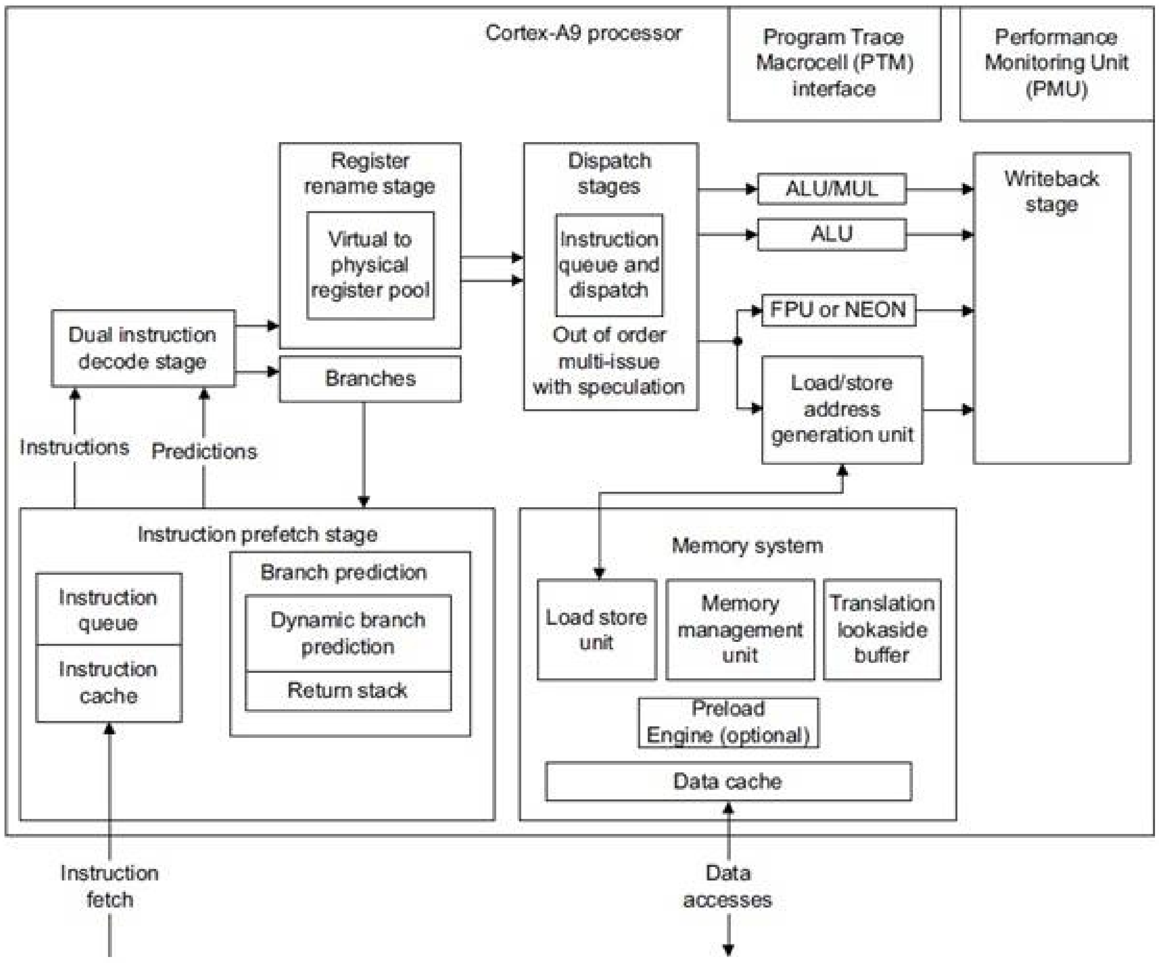

Cortex-A9 processor
Program Trace Macrocell (PTM) interface
Performance Monitoring Unit (PMU)
Register rename stage
Virtual to physical register pool
Dispatch stages
Instruction queue and dispatch
Out of order multi-issue with speculation
ALU/MUL
ALU
FPU or NEON
Load/store address generation unit
Writeback stage
Dual instruction decode stage
Branches
Instructions
Predictions
Instruction prefetch stage
Instruction queue
Instruction cache
Branch prediction
Dynamic branch prediction
Return stack
Memory system
Load store unit
Memory management unit
Translation lookaside buffer
Preload Engine (optional)
Data cache
Instruction fetch
Data accesses

Figure 7- 5: Top-level diagram of the ARM Cortex A9 processor

Bus frequency vs. internal frequency in CPU

Frequently you may see an advertisement for a 1-GHz or 2-GHz CPUs. It is important to note that the stated frequency is the internal frequency of the CPU and not the bus frequency. This is due to the fact that designing a 1-GHz motherboard is very difficult and expensive. Such a design requires a very fast logic family and memory in addition to a massive simulation to avoid crosstalk and signal radiation. The bus frequency for such systems is currently less than 1 GHz.

Review Questions

1. True or false. The ARM instruction set is in triadic form.
2. True or false. The branch prediction task is performed by circuitry inside the CPU.
3. Why are some CPUs called a superscalar processor?
4. Instruction scheduling is done by ________.
5. Out of order execution is arranged by ________.

Problems

Section 7.1: ARM Pipeline Evolution

1. The ARM7 uses a pipeline of _______ stages.
2. Give the names of the pipeline stages in the ARM7
3. The ARM9 uses a pipeline of _______ stages.
4. Give the names of the pipeline stages in the ARM9

Section 7.2: Other CPU Enhancements

5. The number of pipeline stages in a superpipeline system is ________ (less, more) than in a superscalar system.
6. Which has one or more execution units, superpipeline or superscalar?
7. Which part of on-chip cache in the ARM is write protected, data or code?
8. What is instruction pairing, and when can it happen?
9. What is data dependency, and how is it avoided?
10. True or false. Instructions are fetched according to the order in which they were written.
11. True or false. Instructions are executed according to the order in which they were written.
12. True or false. Instructions are retired according to the order in which they were written.
13. The visible registers R0, R1, and so on, are updated by which unit of the CPU?
14. True or false. Among the instructions, STRs (store) are never executed out of order.

Answers to Review Questions

Section 7.1

1. In superpipelining, the process of fetching and executing instructions is split into many small steps and all are done in parallel. In this way the execution of many instructions is overlapped.
2. The speed of the execution is limited to the slowest stage of the pipeline.
3. In superscaling, the entire execution unit has been doubled
4. True
5. Fetch, decode, execute, memory, write back

Section 7.2

1. True
2. True
3. Since it has two execution units (pipelines) capable of executing two instructions with one clock
4. circuitry inside the CPU and the compiler
5. the CPU

Appendix A: ARM Cortex-M3 Instruction Description

Section A.1: List of ARM Cortex-M3 Instructions

ADC	Add with Carry
ADCS	Add with Carry (and update the flags)
ADD	ADD
ADDS	ADD and update the flags
ADR	Load PC-Relative Address
AND	Logical AND
ANDS	Logical AND (update flags)
ASR	Arithmetic Shift right
ASRS	Arithmetic Shift right (update the flags)
B	Branch (unconditional jump)
Bxx	Branch Conditional
BFC	Bit Field Clear
BFI	Bit Field Insert
BIC	Bit Clear
BICS	Bit Clear (update flags)
BKPT	Breakpoint
BL	Branch with Link (this is Call instruction)
BLX	Branch Indirect with Link
BX	Branch Indirect (BX LR is used for Return)
CBNZ	Compare and Branch on Non-Zero
CBZ	Compare and Branch on Zero
CDP	Coprocessor Data processing
CLREX	Clear Exclusive
CLZ	Count Leading Zero
CMN	Compare Negative
CMP	Compare
CPSID	Change processor ID and Disable Interrupt
CPSIE	Change Processor State and Enable Interrupt
DMB	Data Memory Barrier
DSB	Data Synchronization Barrier

EOR	Exclusive OR
EORS	Exclusive OR and update the flags
ISB	Instruction Synchronization Barrier
IT	If-Then Condition Block
LDC	Load Coprocessor
LDM	Load Multiple registers
LDMDB	Load Multiple registers and Decrement Before each access
LDMEA	Load Multiple registers from Empty Ascending
LDMFD	Load Multiple registers Full Descending
LDMIA	Load Multiple registers and Increment after each Access
LDR	Load Register
LDR Rx,=Value	Load Register with 32-bit value
LDRB	Load Register Byte
LDRBT	Load Register Byte with Translation
LDREX, LDREXB, LDREXH	Load Register Exclusive
LDRH	Load Register Halfword
LDRSB	Load Register signed Byte
LDRSH	Load Register Signed Halfword
LDRT	Load Register with Translation
LSL	Logical Shift Left
LSLS	Logical Shift Left (update the flags)
LSR	Logical Shift Right
LSRS	Logical Shift Right (update the flags)
MCR	Move to Coprocessor from ARM Register
MLA	Multiply Accumulate
MLS	Multiply and Subtract
MOV	Move (ARM7)
MOV	Move (ARM Cortex)
MOVS	Move (and update flags)
MOVT	Move Top
MOVW	Move 16-bit constant

MRC	Move to ARM Register from Coprocessor
MRS	Move to general Register from Special register
MSR	Move to Special register from general Register
MUL	Unsigned Multiplication
MVN	Move Negative
MVNS	Move Negative and update the flags
NOP	No Operation
ORN	Logical OR Not
ORNS	OR Not and update flags
ORR	Logical OR
ORRS	Logical OR and update the flags
POP	POP register from Stack
PUSH	PUSH register onto stack
RBIT	Reverse Bits
REV	Reverse byte order in a word
RV16	Reverse byte order in 16-bit
REVSH	Reverse byte order in bottom halfword and sign extend
ROR	Rotate Right
RORS	Rotate Right (update the flags)
RRX	Rotate Right with extend
RRXS	Rotate Right with extend (update the flags)
RSB	Reverse Subtract
RSBS	Reverse Subtract and update the flags
SBC	Subtract with Carry (Borrow)
SBCS	Subtract with Carry (Borrow) and update the flags
SBFX	Sign Bit Field extract
SDIV	Signed Divide
SEV	Send Event
SMLAL	Signed Multiply Accumulate Long
SMULL	Signed Multiply Long
SSAT	Sign Saturate

STM	Store Multiple
STMDB	Store Multiple register and Decrement Before
STMEA	Store Multiple register Empty Ascending
STMIA	Store Multiple register Empty Ascending
STMFD	Store Multiple register Full Descending
STR	Store Register
STRB	Store Register Byte
STRBT	Store Register Byte with Translation
STRD	Store Register Double (two words)
STREX, STREXB, STREXH	Store Register Exclusive
STRH	Store Register Halfword
STRT	Store Register
SUB	Subtract
SUBS	Subtract
SVC	supervisor Call (Software Interrupt)
SXTB	Sign Extend byte
SXTH	Sign Extend Halfword
TBB	Table Branch Byte
TBH	Table Branch halfword
TEQ	Test Equivalence
TST	Test
UBFX	Unsigned Bit filed extract
UDIV	Unsigned Divide
UMLAL	Unsigned Multiply with Accumulate
UMULL	Unsigned Multiply Long
USAT	Unsigned Saturate
UXBT	Zero extend a byte
UXTH	Zero extend halfword
WFE	Wait for event
WFI	Wait for interrupt

Section A.2: ARM Cortex-M3 Instruction Description

ADC   Add with Carry

Flags: Unaffected.

Format: ADC Rd,Rn,Op2   ;Rd = Rn + Op2 + C

Function: If C = 1 prior to this instruction, then after execution of this instruction, Op2 is added to Rn plus 1 and the result is placed in Rd. If C = 0, Op2 is added to Rn plus 0. Used widely in multiword additions. After the execution the flags are not updated. The ADCS instruction updates the flags.

Example 1:

```
LDR     R0,=0xFFFFFFFB          ;R0=0xFFFFFFFB
LDR     R1,=0xFFFFFFFF          ;R1=0xFFFFFFFF
MOV     R2,#3                   ;R2=3
MOV     R3,#4                   ;R3=4
ADDS    R4,R0,R1                ;R4=R0+R1, C=1
ADC     R5,R2,R3                ;R5=R2+R3+C=R2+R3+1
```

ADCS   Add with Carry (and update the flags)

Flags: Affected: N, Z, V, C.

Format: ADC Rd,Rn,Op2   ;Rd = Rn + Op2 + C

Function: If C=1 prior to this instruction, then after execution of this instruction, Op2 is added to Rn plus 1 and the result is placed in Rd. If C = 0, Op2 is added to Rn plus 0. Used widely in multiword additions. Notice the S indicates the flags will be updated.

Example 1:

```
LDR     R0,=0xFFFFFFFB          ;R0=0xFFFFFFFB
LDR     R1,=0xFFFFFFFF          ;R1=0xFFFFFFFF
MOV     R2,#3                   ;R2=3
MOV     R3,#4                   ;R3=4
ADDS    R4,R0,R1                ;R4=R0+R1, C=1
ADCS    R5,R2,R3                ;R5=R2+R3+C=3+4+1, Z=0,N=0,C=0,
```

ADD   ADD

Flags: Unaffected

Format: ADD Rd,Rn,Op2   ;Rd = Rn + Op2

Function: Adds source operands together and places the result in destination. This will not update the flags. To update the flags we must use ADDS.

Example 1:

```
LDR    R0,=0xFFFFFFFF        ;R0=0xFFFFFFFB
MOV    R1,#0x5               ;R1=0x5
ADD    R2,R0,R1
;R2=R0+R1=0xFFFFFFFB+0x5=00000000
;flags unchanged
```

Example 2:

```
LDR    R0,=0xFFFFFFFF        ;R0=0xFFFFFFFF
ADD    R2,R0,#0xF1
;R2=R0+0xF1=R1=0xFFFFFFFF+0xF1=000000F0
;flags unchanged
```

ADDS ADD and update the flags

Flags: Affected: N, Z, V, C.

Format: ADDS Rd,Rn,Op2 ;Rd=Rn+Op2 and update the flags

Function: Add source operands together and places the result in destination and update the flags. Next, we examine the cases of signed and unsigned numbers.

Unsigned addition:

In addition of unsigned numbers, the status of C, Z, N, and V may change, but only C and Z are of any use to programmers. The most important of these flags is C. It becomes 1 when there is carry from D31 out in a 32-bit (D0–D31) operation.

Example 1:

```
MOV    R1,#0x45        ;R1=0x45
ADDS   R1,R1,#0x4F     ;R1=0x94 (0x45+0x4F=0x94),C=0,Z=0
```

Example 2:

```
MOV    R2,#0xFE        ;R2=0xFE
MOV    R3,#0x75        ;R3=0x75
ADDS   R4,R2,R3        ;R4=0xFE+0x75=0x73,C=0,Z=0
```

Example 3:

```
LDR    R0,=0xFFFFFFFF        ;R0=0xFFFFFFFB
MOV    R1,#0x01        ;R1=0x1
```

```
ADDS     R2,R0,R1
;R2=R0+R1=0xFFFFFFFF+0x1= 00000000, C=1, Z=1
```

Example 4:

```
LDR       R0,=0xFFFF126F              ;R0=0xFFFF126F
LDR       R1,=0xFFFF46D4              ;R1=0xFFFF46D4
ADDS     R2,R0,R1
;R1=0xFFFF126F + 0xFFFF46D5=0xFFFF5943, C=1, Z=0
```

Signed addition:

In addition of signed numbers, the status of V, Z, and N must be noted. Special attention should be given to the overflow flag (V) since this indicates if there is an error in the result of the addition. There are two rules for setting V in signed number operation. The overflow flag is set to 1:

1. If there is a carry from D30 to D31 and no carry from D31 out in a 32-bit operation
2. If there is a carry from D31 out and no carry from D30 to D31 in a 32-bit operation

Notice that if there is a carry both from D31 out and from D30 to D31, then V = 0 in 32-bit operations.

Example 5:

```
MOV     R1,#+8                ;R1=0x00000008
MOV     R2,#+4                ;R2=0x00000004
ADDS     R3,R1,R2              ;R3=0x0000000C N=0,V=0,C=0
```

Notice N = D31 = 0 since the result is positive and V = 0 since there is neither a carry from D30 to D31 nor any carry beyond D31. Since V = 0, the result is correct [(+8) + (+4) = (+12)].

Example 6:

```
LDR       R1,=0x+42FFFFFF           ;R1=0x42FFFFFF, (a positive number)
LDR       R2,=0x+45FFFFFF           ;R2=0x45FFFFFF  (a positive number)
ADDS     R3,R2,R1                    ;R3=0x88FFFFFE, C=0, N=1, Z=0, and V=1
```

In Example 6, the correct result should be +, but the result is –. The V = 1 is an indication of this error. Notice that N =D31 = 1 since the result is negative; V = 1 since there is a carry from D30 to D31and C = 0.

Example 7:

```
LDR       R0,=-12               ;R0=0xFFFFFFF4
LDR       R1,=+17               ;R1=0x00000011
```

```
    ADDS    R2,R1,R2          ;R2=00000005 (which is +5 and correct)
    ;N=0,Z=0,V=0, and C=1
```

Notice that V = 0 since there is a carry from D30 to D31 and a carry from D31 out.

Example 8:

```
    LDR     R0,=-30           ;R0=0xFFFFFFE2
    LDR     R1,=+14           ;R2=0x0000000E
    ADDS    R2,R1,R0          ;R2=FFFFFFF0 (which is -16 and correct)
    ;N=1,Z=0,V=0, and C=0
```

V = 0 since there is no carry from D31 out nor any carry from D30 to D31.

Example 9:

```
    LDR     R1,=-126          ;R1=0xFFFFFF82
    LDR     R2,=-127          ;R2=0xFFFFFF81
    ADDS    R3,R2,R1          ;R3=0xFFFFFF03
    ;(which is -253 and correct),N=1,Z=0 and V=0, C=1
```

V = 0 since there is carry from D31 out and carry from D30 to D31.

ADR **Load PC-Relative Address**

Flags: Unaffected:

Format: ADR Rd,label ;Rd= address of label

Function: This allows loading into Rd register an address relative to the current PC (program counter). The label target address must be within the -4,095 to +4,096 bytes from the address in PC register. That is no farther than 1024 instructions in either direction of backward or forward.

Example:

```
    ADR     R3,MyMessage
HERE    B        HERE
MyMessage    DCB      “Hello”
```

AND **Logical AND**

Flags: Unaffected

Format: AND Rd,Rn,Op2 ;Rd= Rn ANDed Op2

Function: Performs logical AND on the operands, bit by bit, storing the result in the destination. This will not update the flags. To update the flags we must use ANDS. Notice that C flag is updated during calculation of Op2 when LSR or LSL are used.

Inputs		Output
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Example 1:

```
MOV     R0,#0x39            ;R0=0x39
MOV     R1,#0x0F            ;R1=0x0F
AND      R2,R1,R0            ;R2=09
;39       0011 1001
;0F       0000 1111
;—         ——
;09       0000 1001  Flags unchanged
```

Example 2:

```
MOV     R0,#0x37            ;R0=0x37
AND      R1,R0,#0x0F       ;R1 = R0 ANDed 0x0F = 07
;37       0011 0111
;0F       0000 1111
;—         ——
;07       0000 0111  Flags unchanged
```

ANDS   **Logical AND   (update flags)**

Flags: Affected: N,Z,C

Format: AND Rd,Rn,Op2   ;Rd= Rn ANDed Op2

Function: Performs logical AND on the operands, bit by bit, storing the result in the destination. This will also update the flags. Notice that C flag is updated during calculation of Op2 when LSR or LSL are used.

Inputs		Output
X	Y	X AND Y

0	0	0
0	1	0
1	0	0
1	1	1

Example 1:

```
    MOV     R0,#0x39            ;R0=0x00000039
    MOV     R1,#0x0F            ;R1=0x0000000F
    ANDS    R3,R1,R0            ;R3=0x00000009
                        ;39         0011 1001
                        ;0F         0000 1111
                        ;—          ——
    ;09         0000 1001           Z=0,N=0, C=unchanged
```

Example 2:

```
    MOV     R1,#0x39            ;R1=0x00000039
    MOV     R2,#0xC6            ;R2=0x000000C6
    ANDS    R3,R1,R2            ;R3=00000000
    ;39         0011 1001
    ;C6         1100 0110
    ;—          ——
    ;00         0000 0000           Z=1, N=0
```

Example 3:

```
    LDR       R2,=0xFFFFFF82
    LDR       R3,=0xFFFFFF81
    ANDS    R4,R2,R3            ;R4=0xFFFFFF80, Z=0, N=1
```

Example 4:

```
    LDR       R1,=0x55555555
    LDR       R2,=0xAAAAAAAA
    ANDS    R3,R2,R1            ;R3=00000000 Z=1, N=0
```

ASR **Arithmetic Shift right**

Flags: Unaffected. Except C

Format: ASR Rd, Rm, Rn

Function: As each bit of Rm register is shifted right, the LSB is removed and the empty bits filled with the sign bit (MSB). The number of bits to be shifted right is given by Rn and the result is placed in Rd register. The flags are unchanged. To update the flags use ASRS instruction.

Example 1:

```
LDR     R2,=0xFFFFFF82
ASR     R0,R2,#6          ;R0=R2 is shifted right 6 times
;now, R0 = 0xFFFFFFFE
```

Example 2:

```
LDR     R0,=0x2000FF18
MOV     R1, #12
ASR     R2,R0,R1          ;R2=R0 is shifted right R1 number of times.
;now, R2 = 0x0002000F
```

Example 3:

```
LDR     R0,=0x0000FF18
MOV     R1, #16
ASR     R2,R0,R1          ;R2=R0 is shifted right R1 number of times
;now, R2 = 0x00000000
```

ASR arithmetic shift is used for signed number shifting. ASR essentially divides Rm by a power of 2 for each bit shift.

ASRS **Arithmetic Shift right (update the flags)**

Flags: N,Z, and C.

Format: ASR Rd, Rm, Rn

Function: As each bit of Rm register is shifted right, the LSB is copied to C flag and the empty bits filled with the sign bit (MSB). The number of bits to be shifted right is given by Rn and the result is placed in Rd register. The ASRS updates the flags.

Example 1:

```
LDR     R2,=0xFFFFFF82
ASRS    R0,R2,#6          ;R0=R2 is shifted right 6 times.
;now, R0= 0xFFFFFFFE, C=0, N=1, Z=0
```

Example 2:

```
LDR     R0,=0x2000FF18
MOV     R1, #12
ASRS    R2,R0,R1        ;R2=R0 is shifted right R1 number of times.
;now, R2= 0x0002000F, C=1, N=0, Z=0
```

Example 3:

```
LDR     R0,=0x0000FF18
MOV     R1, #16
ASRS    R2,R0,R1        ;R2=R0 is shifted right R1 number of times.
;now, R2= 0x00000000, C=1, N=0, Z=1
```

ASRS arithmetic shift is used for signed number shifting. ASRS essentially divides Rm by a power of 2 for each bit shift.

B Branch (unconditional jump)

Flags: Unchanged.

Format: B target ;jump to target address

Function: This instruction is used to transfer control unconditionally to a new address. The difference between B and BL is that the BL instruction saves the address of the next instruction to LR (the link register, R14). For ARM7, the target address is calculated by (a) shifting the 24-bit signed (2's comp) offset left two bits, (b) sign-extend the result to 32-bit, and (c) add it to contents of PC (program counter). This means the target address could be within the –32M bytes to +32M bytes of address space from the current program counter. For ARM Cortex M3. the target address must be within –16MB to +16 MB address space from current instruction.

Bxx Branch Conditional

Flags: Unaffected.

Format: Bxx target ;jump to target upon condition

Function: Used to jump to a target address if certain conditions are met. In ARM7, the target address cannot be more than –32MB to +32MB bytes away. For ARM Cortex M3. the target address must be within –16MB to +16 MB address space from current instruction. The conditions are indicated by the flag register. The conditions that determine whether the jump takes place can be categorized into three groups:

1. flag values,
2. the comparison of unsigned numbers, and
3. the comparison of signed numbers.

Each is explained next.

1. "B condition" where the condition refers to flag values. The status of each bit of

the flag register has been decided by execution of instructions prior to the jump. The following “B condition” instructions check if a certain flag bit is raised or not.

Instruction		Condition
BCS	Branch if Carry Set	jump if C=1
BCC	Branch if Carry Clear	jump if C=0
BEQ	Branch if Equal	jump if Z=1
BNE	Branch if Not Equal	jump if Z=0
BMI	Branch if Minus/Negative	jump if N=1
BPL	Branch if Plus/Positive	jump if N=0
BVS	Branch if Overflow	jump if V=1
BVC	Branch if No overflow	jump if V=0

2. “B condition” where the condition refers to the comparison of unsigned numbers. After a compare (CMP Rn,Op2) instruction is executed, C and Z indicate the result of the comparison, as follows:

	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

Since the operands compared are viewed as unsigned numbers, the following “B condition” instructions are used.

Instruction		Condition
BHI	Branch if Higher	jump if C=1 and Z=0
BEQ	Branch if Equal	jump if C=1 and Z=1
BLS	Branch if Lower or same	jump if C=0 or Z=1

In reality, the “CMP Rn, Op2” is a subtract instruction (Rn-Op2). After the subtraction the result is discarded and flags are changed according to the result. Notice in ARM the subtract affects the C flag setting differently from the x86 and other CPUs. See the SUB instruction.

3. "B condition" where the condition refers to the comparison of signed numbers. In the case of the signed number comparison, although the same instruction, "CMP Rn,Op2", is used, the flags used to check the result are as follows:

Rn > Op2	V=N or Z=0
Rn = Op2	Z=1
Rn < Op2	V inverse of N

Consequently, the "B condition" instructions used are different. They are as follows:

Instruction		
BGE	Branch Greater or Equal	jump if N=1 and V=1 or N=0 and V=0 (V=N)
BLT	Branch Less than	jump if N=1 and V=0 or N=0 and V=1 (N not equal to V)
BGT	Branch Greater than	jump if Z=0 and either N=1 and V=1 or N=0 and V=0 (N=V)
BLE	Branch Less or Equal	jump if Z=1 or N=1 and V=0. Or N=0 and V=1 (Z=1 or N not equal to V)
BEQ	Branch if Equal	jump if Z = 1

All "B condition" instructions are short jumps, meaning that the target address cannot be more than -32M bytes backward or +32M bytes forward from the PC of the instruction following the jump. In ARM Cortex M3 it is 16MB in each direction. What happens if a programmer needs to use a "B condition" to go to a target address beyond the -32MB to +32MB range? The solution is to use the "BX condition, Rm" since Rm can be 32-bit address and covers the entire 4GB address space of the ARM. This is shown next.

```
    LDR     R4,=MYTARGET
    ADDS    R1,R2,R3
    BXEQ    R4          ;branch to address held by R4 if Z=1
MYTRGT          SUBS    R7,#4
    NOP
    NOP
```

….

	C	Z	N	V
Rn > Op2	0	0	0	N
Rn = Op2	0	1	0	N
Rn < Op2	1	0	1	Inverse of N

BFC Bit Field Clear

Flags: Unaffected.

Format: BFC Rd, #LSB, #Width

Function: Clears selected bits of Rd. The start location of the Rd bit is indicated by #LSB and must be in the range of 0–31. How many bits should be cleared is indicated by #Width and must be in the range of 1–32.

Example 1:

```
    LDR     R1,=0xFFFFFFFF          ;R1=0xFFFFFFFF
    BFC     R1,#2,#14               ;now R1=0xFFFF0003
```

Example 2:

```
    LDR     R2,=0x999999999         ;R2=0x99999999
    BFC     R2,#8,#24               ;now R2=0x00000099
```

BFI Bit Field Insert

Flags: Unaffected.

Format: BFI Rd, Rn, #LSB, #Width

Function: Selected bits of Rn are copied to Rd. The start location of the Rd bit is indicated by #LSB and must be in the range of 0 – 31. How many bits should be copied is indicated by #Width and must be in the range of 1–32. The start bit location of Rn is always bit 0 (D0).

Example:

```
    LDR     R1,=0xABCDABCD          ;R1=0xABCDABCD
    LDR     R2,=0x12345678          ;R2=0x12345678
    BFI     R1,R2,#4,#8             ;now R1=0xABCDA78D
    …
```

BIC Bit Clear

Flags: Unaffected.

Format: BIC Rd, Rn,Op2 ;Rd=Rn ANDed with NOT of Op2

Function: Selected bits of Rn are cleared and placed in Rd. The Op2 provides the bits selection. If the selected bits in Op2 are high then corresponding bits in Rn are cleared and the result is placed in Rd. If the selected bits in Op2 are low the corresponding bits in Rn are left unchanged and the result is placed in Rd. In reality, the BIC performs the AND operation on the bits of Rn with the complement of the bits in Op2. The BIC will not update the flags. To update the flags we must use BICS.

Inputs		**Output**
X	Y	X AND (NOT Y)
0	0	0
0	1	0
1	0	1
1	1	0

Example:

```
    LDR     R1,=0xFFFFFF00          ;R1=0xFFFFFF00
    LDR     R2,=0x99999999          ;R2=0x9999999
    BIC     R3,R2,R1                ;now R3=0x00000099
```

BICS Bit Clear (update flags)

Flags: N and Z.

Format: BICS Rd, Rn, Op2 ;Rd=Rn ANDed with NOT of Op2

Function: Selected bits of Rn are cleared and placed in Rd. The Op2 provides the selected bits. If the selected bits in Op2 is high then the corresponding bits in Rn is cleared and the result is placed in Rd. If the selected bits in Op2 is low them the corresponding bits in Rn is left unchanged and the result is placed in Rd. In reality, the BICS performs the AND operation on the bits of Rn with the complement of the bits in Op2 and updates the flags.

Inputs		**Output**
X	Y	X AND (NOT Y)
0	0	0
0	1	0

1	0	1
1	1	0

Example 1:

```
LDR     R1,=0xFFFFFF00           ;R1=0xFFFFFF00
LDR     R2,=0x99999999           ;R2=0x9999999
BICS    R3,R2,R1                 ;now R3=0x00000099, N=0, Z=0
```

Example 2:

```
LDR     R0,=0xFFFFFFFF           ;R0=0xFFFFFFFF
LDR     R1,=0x01234567           ;R1=0x01234567
BICS    R2,R1,R0                 ;now R2=0, N=0, Z=1
```

Example 3:

```
MOV     R0,#0                    ;R0=0
LDR     R1=0x99999999            ;R1=0x99999999
BICS    R2,R1,R0                 ;now R2=0x99999999, N=1, Z=0
```

BKPT Breakpoint

Flags: Unaffected.

Format: BKPT #imme_value

Function: used by compiler to insert breakpoint into programs. Upon execution of the BKPT instruction the program enters the Debug mode. See your ARM compiler for more information

BL Branch with Link (this is Call instruction)

Flags: Unchanged.

Format: BL Subroutine_Addr ;transfer control to a subroutine

Function: Transfers control to a subroutine. This instruction saves the address of the instruction after the BL in R14 (link register). At the end of the subroutine the control to the instruction after the BL is achieved by copying the LR (R14) register to PC. In ARM7, the target address cannot be more than –32MB to +32MB bytes away. For ARM Cortex M3. the target address must be within –16MB to +16 MB address space from current instruction.

Example:

```
LDR     R7,=20000000
BL      DELAY   ;Call subroutine MY_DELAY
```

```
    ADD     R3,#4    ;address of this instruction is saved in R14
    …
    …
DELAY   SUBS    R7,#4
    NOP
    NOP
    MOV     PC,R14  ;Return, could have used "BX LR" instruction
```

BLX **Branch Indirect with Link**

Flags: Unaffected.

Format: BLX Rm ;transfer control to a subroutine whose

;address is given by Rm

Function: Transfers control to a subroutine whose address is given by the Rm register. This instruction saves the address of the instruction after the BL in R14 (link register). At the end of the subroutine the control to the instruction after the BL is achieved by copying the LR (R14) register to PC. One can use "BX LR" as return instruction. Notice the difference between this instruction and "BL Target_Addr" instruction. In the "BL Target_Addr" instruction the target address of the subroutine is given right there. However, in the "BLX Rm" instruction, the target address of the subroutine is held by register Rm.

Example:

```
    ADR     R2,DELAY
    BLX     R2        ;Call subroutine pointed to by R2
    ADD     R3,#4    ;address of this instruction is saved in R14
    …
    …
DELAY   SUBS    R1,#4
    NOP
    NOP
    BX      LR        ;return
```

BX **Branch Indirect (BX LR is used for Return)**

Flags: Unchanged.

Format: BX Rm ;BX LR is used for Return from a subroutine

Function: The most widely usage of this instruction is in the form of "BX LR" for the purpose of return instruction at the end of subroutine.

Example:

```
    LDR     R1,=20000000
    BL      DELAY               ;Call subroutine MY_DELAY
    ADD     R3,#4               ;address of this instr. is saved in R14
    …
    …
DELAY   SUBS    R1,#4
    NOP
    NOP
    BX      LR                  ;return to caller
```

CBNZ Compare and Branch on Non-Zero

Flags: Unchanged.

Format: CBNZ Rn, Target

Function: Transfers control to the target location if Rn is not equal to zero. The Rn must be in the range of R0–R7 and target address cannot be farther than 130 bytes away from the instruction. This instruction compares the Rn with zero and jumps only if Rn is not zero. The comparison has no effect on flags. This can be used for loops in which the body of the loop is no more than 20 instructions.

Example 1:

```
    MOV     R1,#10              ;R1=10
L1          NOP
    NOP
    NOP
    SUB     R1,R1,1             ;R1=R1-1
    CBNZ    R1,L1
```

CBZ Compare and Branch on Zero

Flags: Unaffected.

Format: CBZ Rn, Target

Function: Transfers control to the target location if Rn is zero. The Rn must be in the range of R0–R7 and target address cannot be farther than 130 bytes away from the instruction. This instruction compares the Rn with zero and jumps only if Rn is zero. The comparison has no effect on flags. This can be used to test a register value after reading a port.

Example 1:

```
    LDR       R0,=MYPORT_ADR        ;R0 = MYPORT address
HERE     LDR       R2,[R0]              ;read from MYPORT
    CBZ       R2,HERE             ;keep reading MYPORT until it is zero
```

CDP Coprocessor Data processing

See ARM Cortex-M Manual.

CLREX Clear Exclusive

See ARM Cortex-M Manual.

CLZ Count Leading Zero

Flags: Unchanged.

Format: CLZ Rd,Rn

Function: Scans the Rn register contents from most significant bit (D31) toward least significant bit (D0) until it find the first HIGH. The number of binary zero bits before it encounters the first binary HIGH is placed in Rd.

Example:

```
    LDR       R3,=0x01FFFFFF
    CLZ        R1,R3     ;R1=7 since there are 7 zeros before the first binary 1
```

CMN Compare Negative

Flags: Affected: V, N, Z,C.

Format: CMN Rn,Op2 ;sets flags as if "Rn + Op2"

;Notice, the Rn -(-Op2)=Rn+Op2

Function: Compares Rn register value with the negative of Op2 value. This is done by Rn - (negative of Op2) which is Rn - (-Op2) = Rn + Op2. The Rn and Op2 operands are not altered. In other words, the CMN adds the Op2 to Rn (Rn+Op2) and sets the flags accordingly. This is the same as ADDS instruction except the operands are unchanged and the result is discarded. See Bxx instruction for possible cases of comparison.

CMP Compare

Flags: Affected: V, N, Z, C.

Format: CMP Rn,Op2 ;sets flags as if "Rn-Op2"

Function: Compares two operands. The operands are not altered. Performs comparison by subtracting the Op2 operand from the Rn and updates flags as if SUBS were performed. As we can see in SUBS, the CMP perform the operation of Rn + 2's comp of Op2 and sets the flags according to the result. See Bxx instruction for possible cases of comparison.

CPSID Change processor ID and Disable Interrupt

Flags: Unaffected

Format: CPSID iflag ;iflag is i in PRIMASK or f in FAULTMASK

Function: Used for disabling the interrupt flags in PRIMASK or FAULTMASK registers. See ARM Cortex manual.

CPSIE Change Processor State and Enable Interrupt

Flags: Unaffected

Format: CPSIE iflag ;iflag is i in PRIMASK or f in FAULTMASK

Function: Used for enabling the interrupt flags in PRIMSK or FAULTMASK registers. See ARM Cortex manual.

DMB Data Memory Barrier

Flags: Unaffected

Format: DMB

Function: It makes sure that all the explicit memory accesses prior to DMB instruction are completed before the explicit memory accesses after the DMB. See ARM Cortex manual.

DSB Data Synchronization Barrier

Flags: Unaffected

Format: DSB

Function: It makes sure that all the explicit memory accesses prior to DSB instruction are completed before the DSB instruction is executed. See ARM Cortex manual.

EOR Exclusive OR

Flags: Unaffected

Format: EOR Rd,Rn,Op2

Function: Performs logical Ex-OR on the Rn and Op2 operands, bit by bit, storing the result in the Rd. This will not update the flags. Use EORS instruction to updates the flags.

Inputs		Output
X	Y	X EOR Y
0	0	0

0	1	1
1	0	1
1	1	0

Example 1:

```
MOV     R0,#0xAA           ;R0=0xAA
EOR      R2,R0,#0xFF       ;now, R2=0x55
;AA      1010 1010
;FF      1111 1111
;—       ——
;55     0101 0101           flags unchanged
```

Example 2:

```
LDR      R0,=0xAAAAAAAA         ;R0=0xAAAAAAAA
LDR      R1,=0x55555555          ;R1=0x55555555
EOR      R2,R1,R0                  ;R2=0xFFFFFFFF
;AA      1010 1010
;55     0101 0101
;—       ——
;FF      1111 1111           flags unchanged
```

The "EOR Rd,Rx,Rx" can be used to clear Rd.

Example 3:

```
MOV     R1,#0x55
EOR      R2,R1,R1            ;R2=0
;55     0101 0101
;55     0101 0101
;—       ——
;00     0000 0000           flags unchanged
```

To complement the bits of Rn, EX-OR it with 0xFF.

Example 4:

```
LDR      R0,=0xAAAAAAAA         ;R0=0xAAAAAAAA
LDR      R1,=0xFFFFFFFF          ;R1=0xFFFFFFFF
```

```
    EOR     R2,R1,R0          ;R2=0x55555555
    ;AA     1010 1010
    ;FF     1111 1111
    ;—      ——
    ;55     0101 0101         flags unchanged
```

EORS   Exclusive OR and update the flags

Flags: Affected: C, V, N, Z

Format: EORS Rd,Rn,Op2

Function: Performs logical Ex-OR on the Rn and Op2 operands, bit by bit, storing the result in the Rd. After the execution the flags are updated.

Inputs		Output
X	Y	X EOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Example 1:

```
    MOV     R0,#0xAA          ;R0=0xAA
    EORS    R2,R0,#0xFF       ;now, R2=0x55
    ;AA     1010 1010
    ;FF     1111 1111
    ;—      ——
    ;55     0101 0101         N=0, C=0,Z=0,V=0
```

Example 2:

```
    LDR     R0,=0xAAAAAAAA        ;R0=0xAAAAAAAA
    LDR     R1,=0x55555555        ;R1=0x55555555
    EORS    R2,R1,R0              ;R2=0xFFFFFFFF
    ;AA     1010 1010
    ;55     0101 0101
    ;—      ——
    ;FF         1111 1111          N=1, C=0,Z=0,V=0
```

The “EOR Rd,Rx,Rx” can be used to clear Rd.

Example 3:

```
    MOV     R1,#0x55
    EORS    R2,R1,R1            ;R2=0x0
    ;55         0101 0101
    ;55         0101 0101
    ;—          ——
    ;00         0000 0000          N=0, C=0,Z=1,V=0
```

ISB Instruction Synchronization Barrier

Flags: Unaffected.

Format: ISB

Function: It flushes the pipeline to make sure the instructions executed right after the ISB instruction are fetched fresh from the cache or memory.

IT If-Then Condition Block

Flags: Unaffected

Format: See ARM manual

Function: It allows the execution of up to four instruction after the IT to be conditional.

LDC Load Coprocessor

See the ARM Manual

LDM Load Multiple registers

Flags: Unaffected.

Format: LDM Rn, {Rx,Ry,..}

Function: Loads into registers from consecutive memory locations. The starting address of memory location is given by Rn register. The destination registers separated by comma and placed in braces. In the ARM Cortex, the stack is descending meaning that as information is pushed onto stack the stack pointer is decremented. This IA (Increment the address after each Access) is the default for loading (Poping). This instruction is widely used for Poping (loading) multiple words from descending stack into CPU registers.

Example:

```
    ;Assume the following memory locations with the contents:
    ;12000=(46)
    ;12001=(10)
    ;12002=(38)
    ;12003=(82)
    ;12004=(56)
    ;12005=(50)
    ;12006=(58)
    ;12007=(15)
    ;12008=(63)
    ;12009=(60)
    ;1200A=(68)
    ;1200B=(39)
    ;1200C=(79)
    ;1200D=(70)
    ;1200E=(75)
    ;1200F=(92)
    LDR     R7,=0x12000
    LDM     R7,{R0,R2,R4}
    ;now, R0=0x82381046, R2=0x15585056, …
    ;the contents of memory locations 0x12000-0x12003 are
    ;moved to register R0,and the contents of memory
    ;locations 0x12004-0x12007 are moved to register
    ;R2, and so on. Therefore we have R0=0x82381046,
    ;R2=0x15585056, and R4=0x39686063.
```

LDMDB **Load Multiple registers and Decrement Before each access**

Flags: Unaffected.

Format: LDMDB Rn, {Rx,Ry,…}

Function: This is the same as LDMEA (load multiple registers from Empty Ascending) used for cases in which the stack is ascending. See LDMEA instruction.

LDMEA **Load Multiple registers from Empty Ascending**

Flags: Unaffected.

Format: LDMEA Rn, {Rx,Ry,…}

Function: Loads into registers from consecutive memory locations. The starting address of memory location is given by Rn register. The destination registers separated by comma and placed in braces. In the ARM Cortex, the default for stack is descending meaning that as information are pushed onto stack the stack pointer is decremented. The IA (Increment the address after each Access) is the default. If we change the default of descending stack to ascending stack then we have to use the EA (Empty Ascending). The ascending stack means as information are pushed onto stack the stack pointer is incremented. The LDMEA is used for Poping (loading) multiple words from ascending stack into CPU registers.

LDMFD Load Multiple registers Full Descending

Flags: Unaffected.

Format: LDMFD Rn, {Rx,Ry,..}

Function: This is the same as LDM and LDMIA.

LDMIA Load Multiple registers and Increment after each Access

Flags: Unaffected.

Format: LDM Rn, {Rx,Ry,..}

Function: This is the same as the LDM instructions. In the ARM Cortex, the stack is descending meaning that as information are pushed onto stack the stack pointer is decremented. This IA (Increment the address after each Access) is the default. We use this for Poping (loading) multiple words from descending stack into CPU registers.

LDR Load Register

Flags: Unaffected.

Format: LDR Rd,[Rx] ;load into Rd a word from memory

;location pointed to be Rx

Function: Loads into destination register the contents of four memory locations. The [Rx] points to address of memory location. This is widely used to load 32-bit data from memory into Rd register of the ARM since in the “MOV Rd,#immediate_value” the immediate value cannot be larger than 0xFF.

Example:

```
;Assume the following memory locations with the contents:
;12000=(46)
;12001=(10)
;12002=(38)
;12003=(82)
LDR R0,=0x12000
```

```
    LDR R1,[R0]
    ;now, R1=82381046.
```

LDR Rx,=Value   Load Register with 32-bit value

Flags: Unaffected.

Format: LDR Rd,=32_bit_value ;load Rd with 32-bit value

Function: Loads into destination register a 32-bit immediate value. This is widely used to load 32-bit immediate value into Rd register of the ARM since in the "MOV Rd,#immediate_value" the immediate value cannot be larger than 0xFF.

Example:

```
    LDR     R0,=0x1200000           ;R0=0x1200000
    LDR     R1,=0x2FFFF             ;R1=0x2FFFF
    LDR     R0,=0xFFFFFFFF          ;R0=0xFFFFFFFF
    LDR     R1,=200000000           ;R1=200000000
```

LDRB   Load Register Byte

Flags: Unaffected.

Format: LDRB Rd,[Rx] ;load into Rd a byte from memory

;location pointed to be Rx

Function: Loads into destination register the contents of a single memory location indicated by Rx.

Example:

```
    ;Assume the following memory locations with the contents:
    ;12000=(46)
    ;12001=(10)
    ;12002=(38)
    ;12003=(82)
    LDR     R0,=0x12000
    LDRB    R1,[R0]
    ;now, R0=00000046
```

LDRBT Load Register Byte with Translation

Flags: Unaffected.

Format: LDRBT Rd,[Rx]

Function: Loads into Rd register a byte from memory location pointed to by Rx and zero-extends the byte to 32-bit word. That means a zero is copied to all the upper 24 bits of the Rd register. Used for unprivileged memory access.

Example:

```
;Assume the following memory locations with the contents:
;12000=(46)
;12001=(10)
;12002=(38)
;12003=(82)
LDR      R0,=0x12000
LDRBT    R1,[R0] ;now, R1=00000046
```

LDREX, LDREXB, LDREXH   Load Register Exclusive

Function: They are used with STREX, STREXB, and STREXH instructions to perform CPU synchronization operation. See ARM Cortex manual.

LDRH   Load Register Halfword

Flags: Unaffected.

Format: LDRH Rd,[Rx] ;load into Rd a 2-byte from memory ;location pointed to be Rx

Function: Loads into destination register the contents of the two consecutive memory locations (halfword) indicated by Rx.

Example:

```
;Assume the following memory locations with the contents:
;12000=(46)
;12001=(10)
;12002=(38)
;12003=(82)
LDR      R0,=0x12000
LDRH     R1,[R0]
;now, R0=00001046
```

LDRSB   Load Register signed Byte

Flags: Unaffected.

Format: LDRSB Rd,[Rx]

Function: Loads into Rd register a byte from memory location pointed to by Rx and sign-extends the byte to 32-bit word. That means the sign (D7) of the byte is copied to all the upper 24 bits of the Rd register.

Example 1:

```
;Assume the following memory locations with the contents:
;12000=(85)
;12001=(10)
;12002=(38)
;12003=(82)
LDR     R0,=0x12000
LDRB    R1,[R0] ;now  R1=FFFFFF85 because MSB of 85 is 1
```

Example 2:

```
;Assume the following memory locations with the contents:
;12000=(15)
;12001=(20)
;12002=(3F)
;12003=(82)
LDR     R0,=0x12000
LDRB    R1,[R0] ;now, R1=00000015 because MSB of 15 is 0
```

LDRSH Load Register Signed Halfword

Flags: Unaffected.

Format: LDRSH Rd,[Rx]

Function: Loads into Rd register a half-word (2-byte) from memory location pointed to by Rx and sign-extends it to 32-bit word. That means the sign (D15) of the 16-bit operand is copied to all the upper 16 bits of the Rd register.

Example 1:

```
;Assume the following memory locations with the contents:
;12000=(46)
;12001=(F3)
;12002=(38)
;12003=(82)
```

```
LDR     R0,=0x12000
LDRB    R1,[R0] ;now, R0=FFFFF346 because MSB of F3 is 1
```

Example 2:

```
;Assume the following memory locations with the contents:
;12000=(4F)
;12001=(23)
;12002=(18)
;12003=(B2)
LDR     R0,=0x12000
LDRB    R1,[R0] ;now, R1=0000234F because MSB of 23 is 0
```

LDRT   Load Register with Translation

Flags: Unaffected

Format: LDRT Rd,[Rx]

Function: Loads into Rd register a byte from memory location pointed to by Rx and zero-extends the byte to 32-bit word. That means a zero is copied to all the upper 24 bits of the Rd register. Used for unprivileged memory access.

Example:

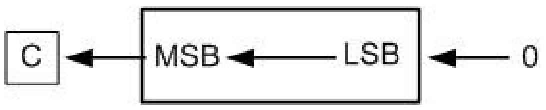
```
;Assume the following memory locations with the contents:
;12000=(46)
;12001=(10)
;12002=(38)
;12003=(82)
LDR     R0,=0x12000
LDRB    R1,[R0] ;now, R1=00000046
```

LSL   Logical Shift Left

Flags: Unaffected.

Format: LSL Rd, Rm, Rn

Function: As each bit of Rm register is shifted left, the MSB is removed and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSL does not updates the flags.



Example 1:

```
LDR     R2,=0x00000010
LSL      R0,R2,#8          ;R0=R2 is shifted left 8 times
;now, R0= 0x00001000, flags not changed
```

Example 2:

```
LDR     R0,=0x00000018
MOV    R1, #12
LSL      R2,R0,R1           ;R2=R0 is shifted left R1 number of times
;now, R2= 0x000018000, flags not changed
```

Example 3:

```
LDR     R0,=0x0000FF18
MOV    R1, #16
LSL      R2,R0,R1           ;R2=R0 is shifted left R1 number of times
;now, R2= 0xFF180000, flags not changed
```

The logical shift left used for unsigned number shifting. LSL essentially multiplies Rm by a power of 2 for each bit shift.

LSLS **Logical Shift Left (update the flags)**

Flags: Affected.

Format: LSLS Rd, Rm, Rn

Function: As each bit of Rm register is shifted left, the MSB is copied to C flag and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSLS updates the flags.

Example 1:

```
LDR     R2,=0x00000010
LSLS    R0,R2,#8          ;R0=R2 is shifted left 8 times
;now, R0= 0x00001000, C=0, N=0, Z=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV    R1, #12
LSLS    R2,R0,R1           ;R2=R0 is shifted left R1 number of times
;now, R2= 0x000018000, C=0, N=0, Z=0
```

Example 3:

```
LDR      R0,=0x000FFF18
MOV     R1, #16
LSLS     R2,R0,R1              ;R2=R0 is shifted left R1 number of times
;now, R2= 0xFF180000, C=1, Z=0, N=0
```

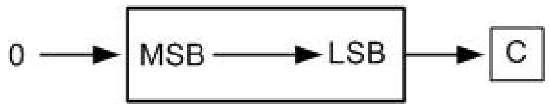
The logical shift left used for unsigned number shifting. LSLS essentially multiplies Rm by a power of 2 for each bit shift.

LSR **Logical Shift Right**

Flags: Unaffected.

Format: LSR Rd, Rm, Rn

Function: As each bit of Rm register is shifted right, the LSB is removed and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSR does not update the flags.



Example 1:

```
LDR      R2,=0x00001000
LSR      R0,R2,#8             ;R0=R2 is shifted right 8 times
;now, R0= 0x00000010, C=0
```

Example 2:

```
LDR      R0,=0x000018000
MOV     R1, #12
LSR      R2,R0,R1              ;R2=R0 is shifted right R1 number of times
;now, R2= 0x00000018, C=0
```

Example 3:

```
LDR      R0,=0x7F180000
MOV     R1, #16
LSR      R2,R0,R1              ;R2=R0 is shifted right R1 number of times
;now, R2=0x00007F18, C=0
```

The logical shift right used for shifting unsigned numbers. LSR essentially divides Rm by a power of 2 for each bit shift.

LSRS **Logical Shift Right (update the flags)**

Flags: Affected.

Format: LSRS Rd, Rm, Rn

Function: As each bit of Rm register is shifted right, the LSB is copied to C flag and the empty bits are filled with zeros. The number of bits to be shifted left is given by Rn and the result is placed in Rd register. The LSRS updates the flags.

Example 1:

```
LDR     R2,=0x00001FFF
LSRS    R0,R2,#8            ;R0=R2 is shifted right 8 times
;now, R0= 0x0000001F, C=1, N=0, Z=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV     R1, #12
LSRS    R2,R0,R1            ;R2=R0 is shifted right R1 number of times
;now, R2= 0x000000000, C=0, N=0, Z=1,
```

Example 3:

```
LDR     R0,=0x000FFF18
MOV     R1, #16
LSRS    R2,R0,R1            ;R2=R0 is shifted right R1 number of times
;Now, R2= 0x0000000F, C=1, Z=0,N=0
```

The logical shift right used for shifting unsigned numbers. LSRS essentially divides Rm by a power of 2 for each bit shift.

MCR Move to Coprocessor from ARM Register

See ARM Manual.

MLA Multiply Accumulate

Flags: Unaffected

Format: MLA Rd,Rs1,Rs2,Rs3 ; Rd= (Rs1 × Rs2) + Rs3

Function: Multiplies an unsigned word held by Rs1 by a unsigned word in Rs2 and the result is added to Rs3 and placed in Rd.

Example:

```
MOV     R0,#0x20            ;R0=0x20
MOV     R1,#0x50            ;R1=0x50
MOV     R2,#0x10            ;R2=0x10
MLA     R4,R0,R1,R2         ;now R4= (0x20 × 0x50)+10= 0xA10
```

MLS Multiply and Subtract

Flags: Unaffected

Format: MLS Rd,Rm,Rs,Rn ; Rd= Rn -(Rs × Rm)

Function: Multiplies an unsigned word held by Rm by a unsigned word in Rs and the result is subtracted from Rn and placed in Rd.

Example:

```
    MOV    R0,#0x20         ;R0=0x20
    MOV    R1,#0x50         ;R1=0x50
    LDR    R2,=0x1000       ;R2=0x1000
    MLS    R4,R0,R1,R2      ;now R4= 0x1000-(0x20×0x50)=0x600
```

MOV Move (ARM7)

Flags: Unaffected.

Format: MOV Rd,#imm_value ; Rd=imm_Value < 0x200

Function: Load the Rd register with an immediate value. The immediate value cannot be larger than 0xFF (0–255). After the execution the flags are not updated. The MOVS instruction updates the flags.

Example 1:

```
    MOV    R0,#0x25         ;R0=0x25
    MOV    R1,#0x5F         ;R1=0x5F
```

To load the ARM register with value larger than 0xFF we must use the "LDR Rd,= 32_bit_data." For example, we can use LDR R2,=0xFFFFFFFF.

Example 2:

```
    LDR    R0,=0x2000000           ;R0=0x2000000
```

MOV Move (ARM Cortex)

Flags: Unaffected.

Format: MOV Rd,#imm_val ;Rd=imm_val (imm_val<0x10000)

Function: Load the Rd register with an immediate value. The immediate value cannot be larger than 0xFFFF (0–65535).

Example:

```
    MOV              R1,#0xF459      ;R1=0xF459
```

To load the ARM register with value larger than 0xFFFF we must use the "LDR Rd,= 32_bit_data." For example we can use LDR R2,=0xFFFFFFF.

MOVS Move (and update flags)

Flags: Affected: C, N, Z

Format: MOV Rd,#immediate_value

Function: Load the Rd register with an immediate value and update the flags.

Example:

```
MOVS   R0,#0x25           ;R0=0x25, N=0,Z=0, and C=0
MOVS   R0,#0x0            ;R0=0x0, N=0,Z=1, and C=0
```

MOVT Move Top

Flags: Unaffected.

Format: MOVT Rd,#imm_value ;imm_value < 0x10000

Function: Loads the upper 16-bit of Rd register with an immediate value. The immediate value cannot be larger than 0xFFFF (0–65535). The lower 16-bit of the Rd register remains unchanged.

Example:

```
LDR     R0,=0x25579934          ;R0=0x25579934
MOVT   R0,#0xAAAA               ;R0=0xAAAA9934
```

MOVW Move 16-bit constant

Flags: Unaffected.

Format: MOVW Rd,#imm_value ;imm_value < 0x10000

Function: Load the Rd register with an immediate value. The immediate value cannot be larger than 0xFFFF (0–65535).

Example:

```
MOVW R1,#0x5555        ;R1=0x5555
```

To load the ARM register with value larger than 0xFFFF we must use the “LDR Rd,= 32_bit_data.” For example we can use LDR R2,=0xFFFFFFF.

MRC Move to ARM Register from Coprocessor

See ARM manual

MRS Move to general Register from Special register

Flags: Unaffected.

Format: MRS Rd,special_reg ;copy special_reg to Rd

Function: Copies the contents of a special function register to a general-purpose register. This instruction along with the MSR is widely used to modify the special function registers such as CONTROL, PRIMASK, and ISPR. This is the only way we can access the special function registers.

Example:

```
MRS     R1,CONTROL     ;R1=CONTROL
AND     R1,#0x00       ;mask the lower 8 bits
MSR     CONTROL,R1
```

MSR **Move to Special register from general Register**

Flags: Unaffected.

Format: MSR special_reg, Rn ;copy special_reg to Rn

Function: Copies the contents of a general-purpose register to special function register. This instruction along with the MRS is widely used to modify the contents of special function registers such as CONTROL, PRIMASK, and ISPR. This is the only way we can access the special function registers.

Example:

```
MRS     R1,CONTROL     ;R1=CONTROL
AND     R1,#0x00       ;mask the lower 8 bits
MSR     CONTROL,R1     ;mask the lower 8 bits of CONTROL reg.
```

MUL **Unsigned Multiplication**

Flags: Affected: N, Z, Unaffected: C, V

Format: MUL Rd,Rn,Rm ;Rd = Rn × Rm

Function: Multiplies a word in register Rn by a word in register Rm and places the result in Rd.

Example 1:

```
MOV    R0,#100        ;R0=100
MOV    R1,#200        ;R1=200
MUL    R3,R0,R1       ;R3 = R0 x R1 = 100 x 200 =20000
```

Example 2:

```
LDR     R0,=10000     ;R0=10000
LDR     R1,=20000     ;R1=20000
MUL    R3,R0,R1       ;R3 = R0 x R1= 10000 x 20000 = 200000000
```

MVN **Move Negative**

Flags: Unaffected.

Format: MVN Rd,Op2 ;Rd = 1’s comp. of Op2

Function: Places in Rd the negation (the 1’s complement) of Op2. Each bit of Op2 is inverted (logical NOT) and placed in Rd while flags remain unchanged.

Example 1:

```
MOV     R0,#0xAA          ;R0=0xAA
MVN     R2,R0             ;now, R2=0xFFFFFF55
```

Example 2:

```
LDR     R0,=0xAAAAAAAA    ;R0=0xAAAAAAAA
MVN     R1,R0             ;R1=0x55555555
```

Example 3:

```
MVN     R0,#0x0F          ;R0=0xFFFFFFF0
```

Example 4:

```
MVN     R2,#0x0           ;R0=0xFFFFFFFF widely used to load Rx with all 1s
```

MVNS **Move Negative and update the flags**

Flags: N and Z are affected

Format: MVNS Rd,Op2 ;Rd = 1’s comp. of Op2 and update flags

Function: Places in Rd the negation (the 1’s complement) of Op2. Each bit of Op2 is inverted (logical NOT) and placed in Rd and N and Z flags are updated.

Example 1:

```
MOV     R0,#0xAA          ;R0=0xAA
MVNS    R2,R0             ;now, R2=0xFFFFFF55, N=1, Z=0
```

Example 2:

```
LDR     R0,=0xAAAAAAAA    ;R0=0xAAAAAAAA
MVNS    R1,R0             ;R1=0x55555555, Z=0 and N=0
```

Example 3:

```
LDR     R0,=0xFFFFFFFF    ;R0=0xFFFFFFFF
MVNS    R1,R0             ;R1=0x00000000, N=0 and Z=1
```

Example 4:

```
MVN     R2,0x0            ;R0=0xFFFFFFFF, N=1, Z=0
```

NOP **No Operation**

Flags: Unaffected.

Format: NOP

Function: Performs no operation. Sometimes used for timing delays to waste clock cycles. Updates PC (program counter) to point to next instruction following NOP. In some ARM CPUs, the pipeline removes the NOP before it reaches the execution stage.

ORN **Logical OR Not**

Flags: Unaffected.

Format: ORN Rd,Rn,Op2 ;Rd = Rn ORed with 1's comp of Op2

Function: Performs the OR operation on the bits of Rn with the complement of the bits in Op2. The ORN will not update the flags. To update the flags we must use ORNS.

Inputs		Output
A	B	A OR (NOT B)
0	0	1
0	1	0
1	0	1
1	1	1

Example 1:

```
LDR    R1,=0xFFFFFF00        ;R1=0xFFFFFF00
LDR    R2,=0x99999999        ;R2=0x9999999
ORN    R3,R2,R1              ;now R3=0x999999FF
```

Example 2:

```
MOV    R1,#0                 ;R1=0
LDR    R0=0xFFFFFFFF         ;R0=0xFFFFFFFF
ORN    R2,R1,R0              ;now, R2=0x0
```

ORNS **OR Not and update flags**

Flags: N and Z are affected

Format: ORNS Rd,Rn,Op2 ;Rd = Rn ORed with 1's comp of Op2

Function: Performs the OR operation on the bits of Rn with the complement of the bits in Op2 and updates the flags. The result is placed in Rd.

Inputs		Output
		A OR (NOT

A	B	B)
0	0	1
0	1	0
1	0	1
1	1	1

Example 1:

```
LDR     R1,=0xFFFFFF00          ;R1=0xFFFFFF00
LDR     R2,=0x99999999          ;R2=0x9999999
ORNS    R3,R2,R1                ;now, R3=0x999999FF, N=1, Z=0
```

Example 2:

```
LDR     R0,=0xFFFFFFFF          ;R0=0xFFFFFFFF
LDR     R1,=0x01234567          ;R1=0x01234567
ORNS    R2,R1,R0                ;now, R2=0x01234567, N=0, Z=0
```

Example 3:

```
MOV     R1,#0                   ;R1=0
LDR     R0=0xFFFFFFFF           ;R0=0xFFFFFFFF
ORNS    R2,R1,R0                ;now, R2=0x0, N=0, Z=1
```

ORR **Logical OR**

Flags: Unaffected

Format: ORR Rd,Rn,Op2 ;Rd= Rn ORed Op2

Function: Performs logical OR on the bits of Rn and Op2, and places the result in Rd. Often used to turn a bit on. ORR will not update the flags.

Example 1:

```
MOV     R0,#0xAA        ;R0=0xAA
ORR     R2,R0,#0x55     ;now, R2=0xFF
```

Example 2:

```
LDR     R0,=0x00010203          ;R0=00010203
LDR     R1,=0x30303030
ORR     R2,R0,R1                ;R2=0x30313233
```

Example 3:

```
LDR    R0,=0x55555555        ;R0=0x55555555
LDR    R1,=0xAAAAAAAA        ;R0=0xAAAAAAAA
ORR    R2,R1,R0              ;R1=0xFFFFFFFF
```

ORRS Logical OR and update the flags

Flags: N and Z are affected.

Format: ORRS Rd,Rn,Op2 ;Rd= Rn ORed Op2 ,update the flags

Function: Performs logical OR on the bits of Rn and Op2, and places the result in Rd. Often used to turn a bit on. ORRS updates the flags.

Inputs		Output
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Example 1:

```
MOV    R0,#0xAA        ;R0=0xAA
ORRS   R2,R0,#0x55     ;now, R2=0xFF, N=0,Z=0
```

Example 2:

```
LDR    R0,=0x00010203        ;R0=00010203
LDR    R1,=0x30303030
ORRS   R2,R0,R1              ;R2=0x30313233 N=0, Z=0
```

Example 3:

```
LDR    R0,=0x55555555        ;R0=0x55555555
LDR    R1,=0xAAAAAAAA        ;R0=0xAAAAAAAA
ORRS   R2,R1,R0              ;R1=0xFFFFFFFF  N=1, Z=0
```

POP POP register from Stack

Flags: Unaffected.

Format: POP {reg_list} ;reg_reg = words off top of stack

Function: Copies the words pointed to by the stack pointer to the registers indicated by the reg_list and increments the SP by 4, 8, 12, 16, … depending on the number of registers in the reg_list.

Example:

```
POP     {R1}              ;POP the top word of stack to R1
POP     {R1,R4,R7}        ;POP the top 3 words of stack to R1,R4,R7
POP     {R2-R6}           ;POP the top 5 words of stack to R2-R6
POP     {R0,R5}           ;POP the top 2 words of stack to R0 and R5
POP     {R0-R7}           ;POP the top 8 words of stack to R0-R7
```

The POP instruction is synonyms for LDMIA.

PUSH **PUSH register onto stack**

Flags: Unaffected.

Format: PUSH {reg_list} ;PUSH reg_list onto stack

Function: Copies the contents of registers stated in reg_list onto the stack and decrements SP by 4, 8, 12, 16, … depending on the number of registers in reg_list.

Example:

```
PUSH    {R1}              ;PUSH the R1 onto top of stack
PUSH    {R1,R4,R7}        ;PUSH R1,R4,R7 onto top of stack
PUSH    {R2-R6}           ;PUSH the R2,R3,R4,R5,R6 onto top of stack
PUSH    {R0,R5}           ;PUSH the R0 and R5 onto top of stack
PUSH    {R0-R7}           ;PUSH the R0 through R7 onto top of stack
```

The PUSH instruction is synonyms for STMDB.

RBIT **Reverse Bits**

Flags: Unaffected.

Format: RBIT Rd,Rn ;Reverse the bit order of Rn and place in Rd

Function: Reverses the bit position order of the 32-bit value in Rn register and place the result in Rd.

Example:

```
MOV     R1,#0x5F
RBIT    R2,R1             ;now, R2=0xF5000000
```

REV **Reverse byte order in a word**

Flags: Unaffected

Format: REV Rd,Rn ;Reverse the byte of Rn and place it in Rd

Function: Reverses the byte position order of the 32-bit value in Rn register and places the result in Rd. This can be used to convert from little endian to big endian or from big endian to little endian.

Example:

```
LDR     R1,=0x12345678
REV     R2,R1              ;now, R2=0x78564312
```

RV16 **Reverse byte order in 16-bit**

Flags: Unaffected

Format: REV16 Rd,Rn ;Reverse the bits if Rn and place it in Rd

Function: Reverses the 16-bit position order of the 32-bit value in Rn register and places the result in Rd. This can be used to convert 16-bit little endian to big endian or from 16-bit big endian to little endian.

Example:

```
LDR     R1,=0x559922FF
RV16    R2,R1              ;now, R2=0x22FF5599
```

REVSH **Reverse byte order in bottom halfword and sign extend**

Flags: Unaffected

Format: REVSH Rd,Rn ;Rd=Reverse the byte and sign extend Rn

Function: Reverses the 16-bit position order of Rn register and after sign extending to 32-bit it is placed in Rd. This can be used to convert a signed 16-bit little endian to 32-bit signed big endian or from signed 16-bit big endian to 32-bit signed little endian.

Example:

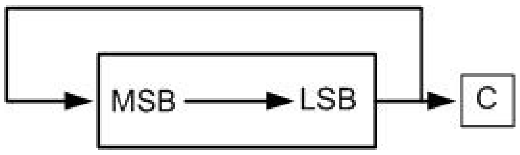
```
LDR     R1,=0x559922FF
REVSH   R2,R1              ;now, R2=0x22FF5599
```

ROR **Rotate Right**

Flags: Unaffected.

Format: ROR Rd,Rm,Rn ;Rd=rotate Rm right Rn bit positions

Function: As each bit of Rm register shifts from left to right, they exit from the right end (LSB) and enter from left end (MSB). The number of bits to be rotated right is given by Rn and the result is placed in Rd register. The ROR does not update the flags.



Example 1:

```
LDR     R2,=0x00000010
ROR     R0,R2,#8        ;R0=R2 is rotated right 8 times
;now, R0 = 0x10000000, C=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV     R1, #12
ROR     R2,R0,R1        ;R2=R0 is rotated right R1 number of times
;now, R2 = 0x01800000, C=0
```

Example 3:

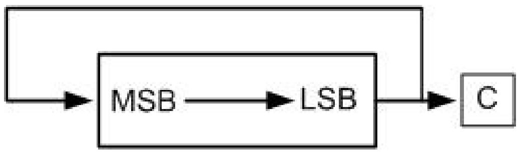
```
LDR     R0,=0x0000FF18
MOV     R1, #16
ROR     R2,R0,R1        ;R2=R0 is rotated right R1 number of times
;Now, R2 = 0xFF180000, C=0
```

RORS **Rotate Right (update the flags)**

Flags: C, N, Z are affected.

Format: RORS Rd,Rm,Rn ;Rd=rotate Rm right Rn bit positions

Function: As each bit of Rm register shifts from left to right, they exit from the right end (LSB) and enters from left end (MSB). In addition as each bit exits the LSB, a copy is of it is given to C flag. The number of bits to be rotated right is given by Rn and the result is placed in Rd register. The RORS updates the flags.



Example 1:

```
LDR     R2,=0x00000010
RORS    R0,R2,#8        ;R0=R2 is rotated right 8 times
;now, R0= 0x01000000, C=0, N=0, Z=0
```

Example 2:

```
LDR     R0,=0x00000018
MOV     R1, #12
RORS    R2,R0,R1        ;R2=R0 is rotated right R1 number of times
```

```
;now, R2= 0x01800000, C=0, N=0, Z=0
```

Example 3:

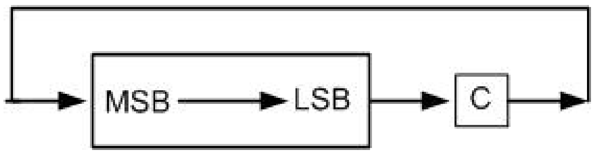
```
LDR       R0,=0x0000FF18
MOV     R1, #16
RORS     R2,R0,R1            ;R2=R0 is rotated right R1 number of times
;now, R2= 0xFF180000, C=1, N=0, Z=0
```

RRX   Rotate Right with extend

Flags: Unaffected.

Format: RRX Rd,Rm   ;Rd=rotate Rm right 1 bit position

Function: Each bit of Rm register is shifted from left to right one bit. The RRX does not update the flags.



Example:

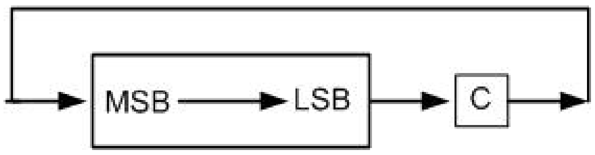
```
LDR       R2,=0x00000002
RRX       R0,R2                  ;R0=R2 is shifted right one bit
;now, R0=0x00000001
```

RRXS   Rotate Right with extend (update the flags)

Flags: Affected.

Format: RRXS Rd,Rm   ;Rd=rotate Rm right 1 bit position

Function: Each bit of Rm register is shifted from left to right one bit. The RRXS updates the flags.



Example 1:

```
LDR       R2,=0x00000002
RRXS      R0,R2     ;R0=R2 is shifted right one bit
;now, R0=0x00000001
```

RSB   Reverse Subtract

Flags: Unaffected

Format: RSB Rd, Rn, Op2 ;Rd = Op2 - Rn

Function: Subtracts the Rn from the Op2 and puts the result in the Rd. The RSB has no effect on flags. The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Takes the 2's complement of the Rn
2. Adds this to the Op2
3. Places the result in Rd

The Op2 and Rn operands remain unchanged by this instruction.

Example:

```
LDR     R0,=0x55555555          ;R0=0x55555555
LDR     R1,=0x99999999          ;R1=0x99999999
RSB     R2,R0,R1          ;R2=R1-R0
;For "RSB R2,R0,R1" we have:
;R2=R1-R0=0x99999999 - 0x55555555 =
;R2=0x99999999 + 2's comp of 0x55555555
;R2=0x99999999 + 0xAAAAAAAB = 0x44444444
;   0x99999999
; - 0x55555555
;   ______
;   0x44444444
```

RSBS **Reverse Subtract and update the flags**

Flags: Affected: V, N, Z, C.

Format: RSBS Rd, Rn, Op2 ;Rd = Op2 - Rn

Function: Subtracts the Rn from the Op2 and puts the result in the Rd. The RSBS updates the flags.

The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Takes the 2's complement of the Rn
2. Adds this to the Op2
3. Places the result in the Rd

The Rn and Op2 operands remain unchanged by this instruction.

Example:

```
LDR     R0,=0x55555555          ;R0=0x55555555
```

```
    LDR       R1,=0x99999999              ;R1=0x99999999
    RSB       R2,R0,R1                     ;R2=R1-R0
    ;For "RSB R2,R0,R1" we have:
    ;R2=R1-R0=0x99999999 - 0x55555555 =
    ;R2=0x99999999  + 2's comp of 0x55555555
    ;R2=0x99999999 + 0xAAAAAAAB = 0x44444444
    ;   0x99999999
    ; - 0x55555555
    ;   ______
    ;   0x44444444    C=0, N=0, Z=0, V=0
```

SBC **Subtract with Carry (Borrow)**

Flags: Unaffected

Format: SBC Rd,Rn,Op2 ;Rd = Rn – Op2 – (1– C)

Function: Subtracts the Op2 operand from the Rn, placing the result in Rd. If C = 0, it subtracts 1 from the result; otherwise, it operates like SUB. The SBC has no effect on flags.This is used widely for multiword (64-bit) subtraction.

Example:

```
    LDR       R0,=0x55555555              ;R0=0x55555555
    LDR       R1,=0x99999999              ;R1=0x99999999
    SUBS      R2,R0,R1              ;R2=R0 - R1
    MOV       R3,#0x09              ;R3=0x09
    SBC       R4,R3,#03             ;R4=R3 - 0x3
    ;For SUBS we have:
    ;R2=R1 - R0 = 0x55555555 - 0x99999999 =
    ;R2=0x55555555 + 2's comp of 0x99999999
    ;R2=0x55555555 + 0x66666667 = 0xBBBBBBBC  C=0
    ;For SBC we have:
    ;R4=R3-0x3=0x09 - 0x3 -(1 - C) = 9 - 3 - 1
    ;R4= 0x9 +2'comp. of -4 = 0x9 + 0xFFFFFFFC = 0x05
    ;   0x0000000955555555
    ; - 0x0000000399999999
```

SBCS **Subtract with Carry (Borrow) and update the flags**

Flags: C,Z,N,V

Format: SBCS Rd,Rn,Op2 ;Rd = Rn – Op2 – (1– C)

Function: Subtracts the Op2 operand from the Rn, placing the result in Rd. If C = 0, it subtracts 1 from the result; otherwise, it executes like SUB. The SBCS updates the flags. This is used widely for multiword (64-bit) subtraction.

Example:

```
LDR     R0,=0x55555555          ;R0=0x55555555
LDR     R1,=0x99999999          ;R1=0x99999999
SUBS    R2,R0,R1            ;R2=R0-R1
MOV     R3,#0x09            ;R3=0x09
SBCS    R4,R3,#03           ;R4=R3-0x3
;For SUBS we have:
;R2=R1-R0=0x55555555 - 0x99999999 =
                        ;R2=0x55555555 + 2’s comp of 0x99999999
;R2=0x55555555 + 0x66666667 = 0xBBBBBBBC  C=0
;For SBCS we have:
;R4=R3-0x3=0x09 - 0x3 -(1 - C) = 9 - 3 - 1
;R4= 0x9 +2’comp. of -4 = 0x9 + 0xFFFFFFFC = 0x05
;   0x0000000955555555
; - 0x0000000399999999
;   ______
;  0x1 0000005BBBBBBBC  C=0, Z=0, V=0, N=0
```

SBFX **Sign Bit Field extract**

Flags: Unaffected

Format: SBFX Rd,Rn,#LSB,#Width

Function: Extracts the bit field from the Rn register and then after sign extending it is placed in Rd. The #LSB indicates which bit and #Width indicates how many bits.

Example 1:

```
LDR     R0,=0x00000543          ;R0=0x00000543
SBFX    R2,R0,#8,#4             ;now, R2=0x00000005
```

Example 2:

```
LDR     R0,=0x00000C43          ;R0=0x00000C43
```

```
    SBFX      R2,R0,#4,#8                  ;now, R2=0xFFFFFFC4
```

SDIV **Signed Divide**

Flags: Unaffected

Format: SDIV Rd,Rn,Rm ;Rd= Rn/Rm

Function: Divides a signed integer word in Rn by another signed integer word in Rm. The quotient result is placed in Rd. If value in Rn register is not divisible by the value in Rm register, the result is rounded to zero and placed in Rd. Divide by zero causes interrupt type 3.

Example:

```
    LDR       R0,=-20000        ;R0=-20000
    LDR       R1,=-1000         ;R1=-1000
    SDIV      R2,R0,R1          ;now, R2 = -2000/-1000= 2
```

SEV **Send Event**

Flags: Affected.

Format: SEV

Function: Sends signal to all the processors in the multiprocessors system. See the ARM Cortex manual.

SMLAL **Signed Multiply Accumulate Long**

Flags: Unaffected

Format: SMLAL Rdlo,Rdhi,Rn,Rm ;Rdhi:Rdlo=(Rm × Rn) + (Rdhi:Rdlo)

Function: Multiplies signed words in Rn and Rm register, adds the 64-bit result to Rdhi:Rdlo register, and saves the final result in Rdhi:Rdlo. The Rdlo (low) and Rdhi(high) are the lower word and higher word of a 64-bit value.

Example 1:

```
    LDR       R0,=0
    LDR       R1,=0x23
    LDR       R2,=-5000
    LDR       R3,=-4000
    SMLAL R0,R1,R2,R3       ;now, R3:R2= (R3:R2)+ (R1 × R0)
    ;= 0x2300000000 + (-5000 × -4000)
    ;= 0x2300000000 + 20000000
    ;= 0x23000000 + 0x1312D00 = 0x2301312D00
    ;=> R0 = 0x1312D00 and R1 = 0x23
```

SMULL **Signed Multiply Long**

Flags: Unaffected

Format: SMULL Rdlo,Rdhi,Rn,Rm ;Rdhi:Rdlo = Rm × Rn

Function: Multiplies signed words in Rn and Rm register, and saves the result in Rdhi:Rdlo. The Rdl (low) and Rdh(high) are the lower word and higher word of a 64-bit value.

Example:

```
LDR     R0,=-20000      ;R0=-20000 (signed 2’s comp)
LDR     R1,=-1000000    ;R0=-100000 (signed 2’s comp)
SMLAL  R2,R3,R0,R1      ;now, R3:R2= R1 × R0 = -20000 × -1000000 =
;20000000000 =0x4A817C800 => R3 = 0x4 and
;R2 = 0xA817C800
```

SSAT **Sign Saturate**

Flags: Unaffected.

Format: SSAT Rd,#n,Rm,shift#

Function: Used for saturation operation. See ARM Cortex manual.

STM **Store Multiple**

Flags: Unaffected.

Format: STM Rn, {Rx,Ry,…}

Function: Stores registers Rx, Ry,… into consecutive memory locations. The starting address of memory location is given by Rn register. The source registers are separated by comma and placed in braces. In the ARM Cortex, the default stack is descending meaning that as information are pushed onto stack the stack pointer is decremented. This IA (Increment the address After each access) is the default. This instruction is widely used for Pushing (storing) multiple registers into ascending stack.

Example:

```
LDR     R7,=0x12000
LDR     R0,=0x82381046          ;R0=0x82381046
LDR     R2,=0x15585056          ;R2=0x15585056
LDR     R4,=0x39686063          ;R4,=0x39686063
STM    R7,{R0,R2,R4}            ;now, R2=0x15585056, ..
;The contents of registers R0,R2, and R4 are stored into
;consecutive memory locations starting at an address given by R7.
```

```
;The R0 contents are stored into memory locations 0x12000-0x12003,
;the R2 contents  are stored into memory locations 0x12004 through
;0x12007, and so on. This is shown below.
;12000=(46)
;12001=(10)
;12002=(38)
;12003=(82)
;12004=(56)
;12005=(50)
;12006=(58)
;12007=(15)
;12008=(63)
;12009=(60)
;1200A=(68)
;1200B=(39)
```

STMDB **Store Multiple register and Decrement Before**

Flags: Unaffected.

Format: STMDB Rn,{Rx,Ry,…}

Function: Stores registers Rx, Ry,… into consecutive memory locations. The starting address of memory location is given by Rn register. The source registers are separated by comma and placed in braces. In the ARM Cortex, the default stack is descending meaning that as information are pushed onto stack the stack pointer is decremented. Since IA(Increment the address After each access) is the default we need to use DB (Decrement the address Before each access) is to overwrite the default. This instruction is widely used for Pushing (storing) multiple registers into Descending stack.

Example:

```
LDR     R7,=0x12000
LDR     R0,=0x39686063          ;R0=0x39686063
LDR     R2,=0x15585056          ;R2=0x15585056
LDR     R4,=0x82381046          ;R4,=0x82381046
STMDB R7,{R0,R2,R4}
;The contents of registers R0,R2, and R4 are stored into
;consecutive memory locations starting at an address given by R7.
```

```
;The R0 contents are stored into memory locations 0x11FFF-0x11FFC,
;the R2 contents  are stored into memory locations 0x11FFB
;through 0x11FF8, and so on. This is shown below.
;11FF4=(46)
;11FF5=(10)
;11FF6=(38)
;11FF7=(82)
;11FF8=(56)
;11FF9=(50)
;11FFA=(58)
;11FFB=(15)
;11FFC=(63)
;11FFD=(60)
;11FFE=(68)
;11FFF=(39)
```

STMEA **Store Multiple register Empty Ascending**

Flags: Unaffected.

Format: STMEA Rn,{Rx,Ry,…}

Function: This is same as STM.

STMIA Store Multiple register Empty Ascending

Flags: Unaffected.

Format: STMIA Rn,{Rx,Ry,…}

Function: This is same as STM.

STMFD **Store Multiple register Full Descending**

Flags: Unaffected.

Format: STMFD Rn,{Rx,Ry,…}

Function: This is another name for STMDB. The FD is for pushing onto Full Descending stacks

STR **Store Register**

Flags: Unaffected.

Format: STR Rd,[Rx] ;Store Rd into memory location pointed to be Rx

Function: Stores Rd register into four consecutive memory locations. The [Rx] points to starting address of memory location. This is widely used to store 32-bit register into memory locations.

Example:

```
    LDR     R1,=0x82381046          ;R1=0x82381046
    LDR     R0,=0x12000             ;R0=0x12000
    STR     R1,[R0]                 ;now,
    ;12000=(46)
    ;12001=(10)
    ;12002=(38)
    ;12003=(82)
```

STRB **Store Register Byte**

Flags: Unaffected.

Format: STRB Rd,[Rn]

Function: Stores the lowest byte of the Rd register into a single memory location indicated by Rn.

Example:

```
    LDR     R1,=0x82381046          ;R1=0x82381046
    LDR     R0,=0x12000             ;R0=0x12000
    STRB    R1,[R0]                 ;now, 12000=(46)
```

STRBT **Store Register Byte with Translation**

Flags: Unaffected.

Format: STRBT

Function: Stores the lowest byte of the Rd register into a single memory location indicated by Rn. This is the same as STRB but is used for unprivileged memory access. See ARM Cortex manual.

Example:

```
            LDR     R1,=0x82381046          ;R1=0x82381046
    LDR     R0,=0x12000             ;R0=0x12000
    STRBT   R1,[R0]
    ;now, 12000=(46)
```

STRD   **Store Register Double (two words)**

Flags: Unaffected.

Format: STRD Rd,[Rn]

Function: Stores two registers of Rd and Rd+1 into 8 consecutive memory locations indicated by Rn. Rd can be R0, R2, R4, R6, R8, R10, or R12.

Example:

```
    LDR       R2,=0x12000
    LDR       R0,=0x82381046              ;R0=0x82381046
    LDR       R1,=0x15585056              ;R1=0x15585056
    STRD      R0,R1,[R2]          ;store R0 and R1 into memory locations starting
              ;at an address given by R2. Now, we have:
    ;12000=(46)
    ;12001=(10)
    ;12002=(38)
    ;12003=(82)
    ;12004=(56)
    ;12005=(50)
    ;12006=(58)
    ;12007=(15)
```

STREX, STREXB, STREXH   **Store Register Exclusive**

Function: They are used with LDREX, LDREXB, and LDREXH instructions to perform CPU synchronization operation. See ARM Cortex manual.

STRH   **Store Register Halfword**

Flags: Unaffected.

Format: STRH Rd,[Rn]

Function: Stores the lower 2 bytes of the Rd register into two consecutive memory locations indicated by Rn.

Example:

```
    LDR       R1,=0x82381046              ;R1=0x82381046
    LDR       R0,=0x12000                  ;R0=0x12000
    STRB      R1,[R0]                        ;now, 12000=(46), and  12001=(10)
```

STRT   Store Register

Flags: Unaffected

Format: STRT Rx,[Rn]

Function: Stores Rx register into memory location pointed to by Rx. This is the same as STR but is used for unprivileged memory access. See ARM Cortex manual.

Example:

```
    LDR       R1,=0x82381046            ;R1=0x82381046
    LDR       R0,=0x12000               ;R0=0x12000
    STRT      R1,[R0]
    ;now, 12000=(0x82381046)
```

SUB   Subtract

Flags: Unaffected

Format: SUB Rd, Rn, Op2   ;Rd = Rn – Op2

Function: Subtracts the Op2 from the Rn and puts the result in the Rd. Has no effect on flags. The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Takes the 2’s complement of the Op2
2. Adds this to the Rn
3. Place the result in the Rd

The Rd and Op2 operands remain unchanged by this instruction.

Example:

```
    LDR       R0,=0x55555555            ;R0=0x55555555
    LDR       R1,=0x99999999            ;R1=0x99999999
    SUB       R2,R1,R0           ;R2=R1-R0
    ;For “SUB R2,R1,R0” we have:
    ;R2=R1-R0=0x99999999 - 0x55555555 =
    ;R2=0x99999999 + 2’s comp of 0x55555555
    ;R2=0x99999999 + 0xAAAAAAAB = 0x44444444
    ;   0x99999999
    ; - 0x55555555
    ;   ______
```

```
;   0x44444444
```

SUBS **Subtract**

Flags: Affected: V, N, Z, C.

Format: SUB Rd, Rn, Op2 ;Rd = Rn – Op2

Function: Subtracts the Op2 from the Rn and puts the result in the Rd. The SUBS updates the flags. The steps for subtraction performed by the internal hardware of the CPU are as follows:

1. Takes the 2's complement of the Op2
2. Adds this to the Rn
3. Place the result in the Rd

The Rd and Op2 operands remain unchanged by this instruction.

Example:

```
LDR     R0,=0x55555555          ;R0=0x55555555
LDR     R1,=0x99999999          ;R1=0x99999999
SUBS    R2,R1,R0         ;R2=R1-R0
;For "SUBS R2,R1,R0" we have:
;R2=R1 - R0 = 0x99999999 - 0x55555555 =
;R2=0x99999999 + 2's comp. of 0x55555555
;R2=0x99999999 + 0xAAAAAAAB = 0x44444444
;   0x99999999
; - 0x55555555
;   ______
;   0x44444444   C=1, Z=0, N=0, V=0
```

SVC **supervisor Call (Software Interrupt)**

Flags: Unaffected.

Format: SVC #imm_value

Function: It is used by application software to get services from operating systems (OS). This is like the SWI (software interrupt) instruction in ARM7.

SXTB **Sign Extend byte**

Flags: Unaffected.

Format: SXTB Rd,Rm

Function: Converts a signed byte in Rm into a signed word by copying the sign bit (D7) of Rm into all the bits of Rd. Used widely to convert a signed byte in Rm to a signed word to avoid the overflow problem in signed number arithmetic.

Example:

```
MOV     R1,#0xFB            ;R1=0xFB which is 2's complement of -5
SXTB     R0,R1     ;now, R0=0xFFFFFFFB
;R1= 0000 0000 0000 0000 0000 0000 1111 1011
;now R0=0xFFFFFFFB
;R0 = 1111 1111 1111 1111 1111 1111 1111 1011
```

SXTH   Sign Extend Halfword

Flags: Unaffected.

Format: SXTH Rd,Rm

Function: Converts a signed halfword in Rm into a signed word by copying the sign bit (D15) of Rm into all the bits of Rd. Used widely to convert a signed halfword (16-bit) in Rm to a signed word to avoid the overflow problem in signed number arithmetic.

Example:

```
;assume R1=0xFFFB which is 2's complement of -5
SXTH  R0,R1        ;now, R0=0xFFFFFFFB
;R1= 0000 0000 0000 0000 1111 1111 1111 1011
;now, R0=0xFFFFFFFB
;R0 = 1111 1111 1111 1111 1111 1111 1111 1011
```

TBB   Table Branch Byte

Flags: Unaffected.

Format: TBB [Rn, Rm]

Function: Branches forward using table of single byte offset using PC-relative addressing mode. Rn has starting address of the table and Rm is an index into the table. See ARM Cortex M3 manual.

TBH   Table Branch halfword

Flags: Unaffected.

Format: TBH [Rn, Rm, LSL #1]

Function: Branches forward using table of halfword offset using PC-relative addressing mode. Rn has starting address of the table and Rm is an index into the table. The "LSL # 1" shifts left the address once to make it halfword aligned address. See ARM Cortex M3 manual.

TEQ Test Equivalence

Flags: Affected: N and Z

Format: TEQ Rn,Op2 ;performs Rn Ex-OR Op2

Function: Performs a bitwise logical Ex-OR on Rn and Op2, setting flags but leaving the contents of both Rn and Op2 unchanged. While the EORS instruction changes the contents of the destination and the flag bits, the TEQ instruction changes only the flag bits. This is widely used to see if two registers are equal.

Example 1:

```
TEQ     R1,R2               ;check to see if R1=R2. If so Z=1. R1 and R2
;remain unchanged
```

Example 2:

```
TEQ     R2,#0x01            ;check to see if D0 of R2 is 1, if so Z=1. R2
;remains unchanged
```

Example 3:

```
TEQ     R1,#0xFF            ;check to see if D7_D0 of R1 are 1s,
;if so Z=1. R1 remains unchanged
```

TST Test

Flags: Affected: N and Z

Format: TST Rn,Op2 ;performs Rn AND Op2

Function: Performs a bitwise logical AND on Rn and Op2, setting flags but leaving the contents of both Rn and Op2 unchanged. While the ANDS instruction changes the contents of the destination and the flag bits, the TST instruction changes only the flag bits. To test whether a bit of Rn is 0 or 1, use the TST instruction with an Op2 constant that has that bit set to 1 and all other bits cleared to 0.

Example 1:

```
TST     R1,#0x01            ;check to see if D0 of R1 is zero, if so Z=1.
        ;R1 remain unchanged
```

Example 2:

```
TST     R1,#0xFF            ;check to see if any bits of R1 is zero, if so
;Z=1. R1 remain unchanged
```

UBFX Unsigned Bit filed extract

Flags: Unaffected.

Format: UBFX Rd,Rn,#LSB,#Width

Function: Extracts the bit field from the Rn register and then zero extends it and places in Rd. The #LSB indicates from which bit and #Width indicates how many bits.

Example 1:

```
LDR     R0,=0x00077555        ;R0=0x00077555
UBFX    R2,R0,#8,#4           ;now, R2=0x00000005
```

Example 2:

```
LDR     R0,=0x12345678        ;R0=0x12345678
UBFX    R2,R0,#8,#12          ;now, R2=0x00000456
```

UDIV **Unsigned Divide**

Flags: Unaffected

Format: UDIV Rd,Rn,Rm ;Rd= Rn/Rm

Function: Divides an unsigned integer word in Rn by another unsigned integer word in Rm. The quotient result is placed in Rd. If value in Rn register is not divisible by the value in Rm register, the result is rounded to zero and placed in Rd. Divide by zero causes exception interrupt.

Example 1:

```
LDR     R0,=100          ;R0=100
LDR     R1,=2000
UDIV    R2,R1,R0         ;now, R2=R1/R0=2000/100=20
```

Example 2:

```
LDR     R0,=20000        ;R0=20000
UDIV    R2,R0,#100       ;now, R2=2000/100=20
```

UMLAL **Unsigned Multiply with Accumulate**

Flags: Unaffected

Format: UMLAL RdLo,RdHi,Rn,Rm ;RdHi:RdLo=(Rm × Rn) + (RdHi:RdLo)

Function: Multiplies unsigned words in Rn and Rm register, adds the 64-bit result to RdHi:RdLo registers, and saves the final result in RdHi:RdLo. The RdLo (low) and RdHi(high) are the unsigned lower word and higher word of the 64-bit value.

Example:

```
LDR     R0,=20000        ;R0=20000
```

```
    LDR     R1,=1000
    LDR     R2,=5000
    LDR     R3,=4000
    UMLAL R2,R3,R0,R1       ;now, R3:R2= R1 × R0 + R3:R2
```

UMULL **Unsigned Multiply Long**

Flags: Unaffected

Format: UMULL RdLo,RdHi,Rn,Rm ;RdHi:RdLo = Rm × Rn

Function: Multiplies unsigned words in Rn and Rm registers, and saves the result in RdHi:RdLo. The RdLo (low) and RdHi(high) are the lower word and higher word of a 64-bit value.

Example:

```
    LDR     R0,=20000       ;R0=20000
    LDR     R1,=10000       ;R1=10000
    LDR     R2,=50000       ;R2=50000
    LDR     R3,=40000       ;R3=40000
    UMLAL R2,R3,R0,R1       ;now, R3:R2= R1 × R0
```

USAT **Unsigned Saturate**

Flags: Unaffected

Format: USAT Rd,#n,Rm,shift#

Function: Used for unsigned saturation operation. See ARM Cortex manual.

UXBT **Zero extend a byte**

Flags: Unaffected

Format: UXBT Rd,Rm

Function: Zero extends a byte in Rm and places in Rd. Used widely to convert a byte in Rm to word for signed number operations.

Example:

```
    MOV     R1,#0xFB            ;R1=0xFB
    UXBT     R0,R1      ;now, R0=0x00000000FB
    ;R1= 0000 0000 0000 0000 0000 0000 1111 1011
    ;now R0=0x000000FB
    ;R0 = 0000 00000 0000 0000 0000 0000 1111 1011
```

UXTH **Zero extend halfword**

Flags: Unaffected

Format: UXTH Rd,Rm

Function: Zero extends a halfword in Rm and places in Rd. Used widely to convert a halfword in Rm to word for signed number operations.

Example:

```
;assume R1=0xFFFB
UXTH  R0,R1       ;now, R0=0x00000FFFB
;R1= 0000 0000 0000 0000 1111 1111 1111 1011
;now, R0=0x0000FFFB
;R0 = 0000 0000 0000 0000 1111 1111 1111 1011
```

WFE **Wait for event**

Flags: Unaffected

Format: WFE

Function: Used by power management. See ARM Cortex M3 manual.

WFI **Wait for interrupt**

Flags: Unaffected

Format: WFI

Function: Suspends execution until one of the following events occurs:

1. a non-masked interrupt occurs and is taken,
2. an interrupt masked by PRIMASK becomes pending,
3. a Debug Entry request.

See ARM Cortex manual.

Appendix B: ARM Assembler Directives

Section B.1: List of ARM Assembler Directives

Section B.2: Description of ARM Assembler Directives

Directives, or as they are sometimes called, pseudo-ops or pseudo-instructions, are used by the assembler to translate Assembly language programs into machine language. Unlike the microprocessor's instructions, directives do not generate any opcode; therefore, no memory locations are occupied by directives in the final hex version of the assembly program. To summarize, directives give directions to the assembler program to tell it how to generate the machine code; instructions are assembled into machine code to give instructions to the CPU at execution time. The following are descriptions of the some of the most widely used directives for the ARM assembler. They are given in alphabetical order for ease of reference.

ALIGN

Format:

```
ALIGN   n          ;n is any power of 2 from 2^0 to 2^31
```

This is used to make sure data is aligned in 32-bit word or 16-bit half word memory address. If n is not specified, ALIGN sets the current location to the next word (four byte) boundary. The following uses ALIGN to make the data word and half word aligned:

```
ALIGN   4           ; The next instruction is word (4 bytes) aligned
ALIGN               ; The next instruction is word (4 bytes) aligned
ALIGN   2           ; The next instruction is half word (2 bytes) aligned
```

Notice that, this ALIGN directive should not be confused with the ALIGN attribute of the AREA directive.

AREA

Format:

```
AREA    sectionname     attribute, attribute, …
```

The AREA directive tells the assembler to define a new section of memory. The memory can be code or data and can have attributes such as ReadOnly, ReadWrite, and so on. This is widely used to define one or more blocks of indivisible memory for code or data to be used by the linker. Every assembly language program has at least one AREA.

The following line defines a new area named MY_ASM_PROG1 which has CODE and READONLY attributes:

```
AREA    MY_ASM_PROG1          CODE, READONLY
```

Among widely used attributes are CODE, DATA, READONLY, READWRITE, COMMON, and ALIGN. The following describes these widely used attributes.

CODE is an attribute given to an area of memory used for executable machine instruction. Since it is used for code section of the program it is by default READONLY memory. In ARM Assembly language we use this area to write our instructions.

DATA is an attribute given to an area of memory used for data and no instruction (machine instructions) can be placed in this area. Since it is used for data section of the program it is by default a READWRITE memory. In ARM Assembly language we use this area to set aside SRAM memory for scratch pad and stack.

READWRITE is an attribute given to an area of memory which can be read from and written to. Since it is READWRITE section of the program it is by default for DATA. In ARM Assembly language we use this area to set aside SRAM memory for scratch pad and stack.

READONLY is an attribute given to an area of memory which can only be read from. Since it is READONLY section of the program it is by default for CODE. In ARM Assembly language we use this area to write our instructions for machine code execution.

COMMON is an attribute given to an area of DATA memory section which can be used commonly by several program codes. We do not initialize the COMMON section of the memory since it is used by compiler exclusively. The compiler initializes the COMMON memory area with all zeros.

ALIGN is another attribute given to an area of memory to indicate how memory should be allocated according to the addresses. When the ALIGN is used for CODE and READONLY it aligned in 4-bytes address boundary by default since the ARM instructions are all 32-bit (4-bytes) word. The ALIGN attribute of AREA has a number after like ALIGN=3 which indicates the information should be placed in memory with addresses of 2^3, that is 0x50000, 0x50008, 0x50010, 0x50020, and so on. This ALIGN attribute of the AREA should not be confused with the ALIGN directive.

DCB directive (define constant byte)

Format:

```
label          DCB    n       ;n between -128 to 256 , byte or string
```

The DCB directive allocates a byte size memory and initializes the values for reading only.

```
MYVALUE      DCB    5                  ;MYVALUE = 5
MYMSAGE      DCB    "HELLO WORLD"          ;string
```

DCD directive (define constant word)

Format:

```
label          DCD    n
```

The DCD directive allocates a word size memory and initializes the values for reading only. The data is 32 bit aligned.

```
MYDATA       DCD    0x200000, 0xF30F5, 5000000, 0xFFFF9CD7
```

DCW directive (define constant half-word)

Format:

```
label           DCB     n
```

The DCW directive allocates a half-word size memory and initializes the values for reading only.

```
MYDATA          DCW     0x20, 0xF230, 5000, 0x9CD7
```

END

Every program must have an entry and an END point. The labels for the entry and end points must match. The END directive tells the assembler that it has reached the end of source file.

```
        AREA   PROG_C2, CODE, READONLY
  ENTRY
  …
  END
```

ENDP or ENDFUNC

The ENDFUNC or ENDP directive informs the assembler that it has reached the end of a function. ENDFUNC and ENDP are the same. See FUNCTION or PROC directives.

ENTRY

The ENTRY directive shows the entry point of a program to the assembler. Each program must have one entry point.

```
        AREA   PROG_C2, CODE, READONLY
  ENTRY
  …
  END
```

EQU (Equate)

To assign a fixed value to a name, one uses the EQU directive. The assembler will replace each occurrence of the name with the value assigned to it.

```
DATA1  EQU    0x39              ;the way to define hex value
PORTB  EQU    0xF0018000        ;SFR Port B address
SUM1   EQU    0x40000120        ;assign RAM loc to SUM1
```

Unlike data directives such as DCB, DCD, and so on, EQU does not assign any memory storage; therefore, it can be defined at any time and at any place, and can even be used within the code segment.

EXPORT or GLOBAL

To inform the assembler that a name or symbol will be referenced by other modules (in other files), it is marked by the EXPORT or GLOBAL directives. If a module is referencing a name outside itself, that name must be declared as EXTRN (or IMPORT). Correspondingly, in the module where the variable is defined, that variable must be declared as EXPORT or GLOBAL in order to allow it to be referenced by other modules. See the EXTRN directive for examples of the use of both EXTRN and EXPORT.

EXTRN (External)

The EXTRN directive is used to indicate that certain variables and names used in a module are defined by another module. In the absence of the EXTRN directive, the assembler would search for the definition and give an error when it couldn't find it. The format of this directive is:

```
    EXTRN   name
```

The following example shows how the EXPORT and EXTERN directives are used:

```
    ;from the main program:
    EXTRN   MY_FUNC
    …
    BL      MY_FUNC
    …
    ;______________
    ;MY_FUNC is located in a different file:
    AREA    OUR_EXAMPLE,CODE,READONLY
    EXTRN   DATA1
    EXPORT          MY_FUNC
MY_FUNC         FUNCTION
    …
    LDR     R1,=DATA1
    …
    ENDFUNC
```

Notice that the EXTRN directive is used in the main procedure to show that MY_FUNC is defined in another module. This is needed because MY_FUNC is not defined in that module. Correspondingly, MY_FUNC is defined as GLOABAL in the module where it is defined. EXTRN is used in the MY_FUNC module to declare that operand DATA1 has been defined in another module. Correspondingly, DATA1 is declared as GLOBAL in the calling module.

FUNCTION or PROC

Often, a group of Assembly language instructions will be combined into a procedure so that it can be called by another module. The FUNCTION and ENDFUNC directives are used to indicate the beginning and end of the procedure. See the following example:

```
MY_FUNC         FUNCTION
    …
    …
    ENDFUNC
```

INCLUDE

When there is a group of macros written and saved in a separate file, the INCLUDE directive can be used to bring them into another file.

RN (equate)

This is used to define a name for a register. The RN directive does not set aside a separate storage for the name, but associates a register with that name. The following code shows how we use RN:

```
VAL1     RN        R1        ;define VAL1 as a name for R1
VAL2     RN        R2        ;define VAL2 as a name for R2
SUM      RN        R3        ;define SUM as a name for R3

    AREA   PROG_2_1, CODE, READONLY
    ENTRY
    MOV     VAL1, #0x25                  ;R1 = 0x25
    MOV     VAL2, #0x34                  ;R2 = 0x34
    ADD      SUM, VAL1,VAL2              ;add R2 to R1 and place it in R3
HERE     B          HERE
    END
```

Appendix C: Macros

What is a macro and how is it used?

There are applications in Assembly language programming where a group of instructions performs a task that is used repeatedly. For example, you might need to add three registers together. So it does not make sense to rewrite them every time they are needed. Therefore, to reduce the time that it takes to write these codes and reduce the possibility of errors, the concept of macros was born. Macros allow the programmer to write the task (set of codes to perform a specific job) once only and to invoke it whenever it is needed, wherever it is needed.

MACRO definition

Every macro definition must have three parts, as follows:

```
    MACRO
[$label]        macroName parameter1,parameter2,…,parameterN
    …        …
    …        …
    MEND
```

The MACRO directive indicates the beginning of the macro definition and the MEND directive signals the end. What goes in between the MACRO and MEND directives is called the body of the macro. The name must be unique and must follow Assembly language naming conventions. The parameters are names, or parameters, or even registers that are mentioned in the body of the macro. After the macro has been written, it can be invoked (or called) by its name, and appropriate values are substituted for parameters. For example you might want to have an instruction that adds three registers. The following is a macro for the purpose:

```
    MACRO
    ADD3VAL     $DEST,$ARG1,$ARG2,$ARG3
    ADD         $DEST,$ARG1,$ARG2
    ADD         $DEST,$DEST,$ARG3
    MEND
```

The above code is the macro definition. Note that parameters $DEST, $ARG1, $ARG2, and $ARG3 are mentioned in the body of the macro. To distinguish parameters they must start with $. In the following example, the macro is invoked by its name with the user's actual data:

```
    AREA OURCODE,READONLY,CODE
    ENTRY
```

```
MOV     R1,#5
MOV     R2,#2
ADD3VAL         R0,R1,R2,#5
```

The instruction “ADD3VAL R0,R1,R2,#5” invokes the macro.

The assembler expands the macro by providing the following code in the .LST file:

```
3 00000008 E0810002       ADD    R0,R1,R2
4 0000000C E2800005       ADD    R0,R0,#5
```

Default Values for parameters

We can define default values for parameters as shown below:

```
MACRO
ADD3VAL         $DEST,$ARG1=R3,$ARG2,$ARG3=#5
ADD      $DEST,$ARG1,$ARG2
ADD      $DEST,$DEST,$ARG3
MEND
```

To use the default value we put a | instead of the parameter while invoking the macro:

```
ADD3VAL         R0,R1,R2,|
```

The above code uses the default value of $ARG3 which is set to #5.

Using labels in macros

In the discussion of macros so far, examples have been chosen that do not have a label or name in the body of the macro. This is because if a macro is expanded more than once in a program and there is a label in the label field of the body of the macro, the same label would be generated more than once and an assembler error would be generated. To address the problem we can give a unique label to the macro when we invoke it, as shown below:

```
 MACRO
$lbl   OUR_MACRO
 CMP     R1,#5
 BEQ      $lbl
 MOV    R1,#1
$lbl
 MEND
 AREA    OURCODE,READONLY,CODE
 ENTRY
 MOV    R1,#3
label1    OUR_MACRO
 MOV    R1,#5
label2    OUR_MACRO
HERE      B           HERE
```

The assembler expands the macro by providing the following code in the .LST file:

```
20 00000000                    AREA OURCODE,READONLY,CODE
21 00000000                 ENTRY
22 00000000 E3A01003        MOV  R1,#3
23 00000004         label1   OUR_MACRO
3 00000004 E3510005        CMP  R1,#5
4 00000008 0A000000        BEQ  label1
5 0000000C E3A01001        MOV  R1,#1
6 00000010         label1
24 00000010 E3A01005        MOV  R1,#5
25 00000014         label2   OUR_MACRO
3 00000014 E3510005        CMP  R1,#5
4 00000018 0A000000        BEQ label2
5 0000001C E3A01001        MOV R1,#1
6 00000020         label2
26 00000020 EAFFFFFE
HERE   B   HERE
```

In cases that there are more than one label in a macro the lines can be labeled as shown below:

```
   MACRO
$lbl        OUR_MACRO
   CMP      R1,#5
   BEQ       $lbl.equal
   MOV     R1,#1
   B           $lbl.next
```

```
$lbl.equal
    MOV     R1,#2
$lbl.next
    MEND

    AREA    OURCODE,READONLY,CODE
    ENTRY
    MOV     R1,#3
label1  OUR_MACRO
    MOV     R1,#5
label2  OUR_MACRO
HERE    B           HERE
```

The assembler expands the macro by providing the following code in the .LST file:

```
13 00000000                           AREA OURCODE,READONLY,CODE
14 00000000            ENTRY
15 00000000 E3A01003       MOV R1,#3
16 00000004        label1  OUR_MACRO
3 00000004 E3510005        CMP R1,#5
4 00000008 0A000001        BEQ label1equal
5 0000000C E3A01001        MOV R1,#1
6 00000010 EA000000        B   label1next
7 00000014         label1equal
8 00000014 E3A01002        MOV R1,#2
9 00000018         label1next
17 00000018 E3A01005       MOV R1,#5
18 0000001C        label2  OUR_MACRO
3 0000001C E3510005        CMP R1,#5
4 00000020 0A000001        BEQ label2equal
5 00000024 E3A01001        MOV R1,#1
6 00000028 EA000000        B   label2next
7 0000002C         label2equal
```

```
8 0000002C E3A01002       MOV R1,#2
9 00000030        label2next
19 00000030 EAFFFFFE
HERE   B   HERE
```

Conditional macros

We can pass condition into macros, as well:

```
     MACRO
$lbl         OurMacro$cond
     CMP     R1,#5
     B$cond               $lbl.equal
    MOV     R1,#1
$lbl.equal
     MEND

     AREA OURCODE,READONLY,CODE
     ENTRY
     MOV    R1,#3
label1    OurMacroEQ      ;in the macro check equality
     MOV    R1,#3
label2    OurMacroLO      ;in the macro check if is lower
HERE      B           HERE
```

The assembler expands the macro by providing the following code in the .LST file:

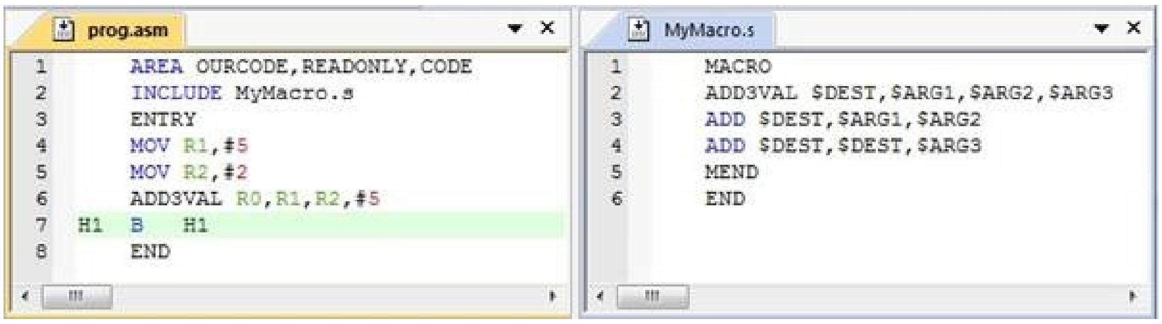
```
10 00000000              AREA OURCODE,READONLY,CODE
11 00000000              ENTRY
12 00000000 E3A01003       MOV   R1,#3
13 00000004         label1   OurMacroEQ
3 00000004 E3510005       CMP   R1,#5
4 00000008 0A000000       BEQ   label1equal
5 0000000C E3A01001       MOV   R1,#1
6 00000010        label1equal
14 00000010 E3A01003       MOV   R1,#3
```

```
15 00000014        label2  OurMacroLO
3 00000014 E3510005       CMP   R1,#5
4 00000018 3A000000       BLO   label2equal
5 0000001C E3A01001       MOV   R1,#1
6 00000020        label2equal
16 00000020 EAFFFFFE
      HERE     B    HERE
```

Notice that the first B$cond is substituted with BEQ while the second B$cond is substituted with BLO since the conditions EQ and LO are used respectively.

INCLUDE directive

Assume that there are several macros that are used in every program. Must they be rewritten every time? The answer is no if the concept of the INCLUDE directive is known. The INCLUDE directive allows a programmer to write macros and save them in a file, and later bring them into any file. For example, assuming that some widely used macros were written and then saved under the filename “MYMACRO1.S”, the INCLUDE directive can be used to bring this file into any “.asm” file and then the program can call upon any of the macros as many times as needed. In the following example the ADD3VAL macro is defined in the MyMACRO.s file and it is used in the example.asm file.


prog.asm

```
        AREA OURCODE,READONLY,CODE
        INCLUDE MyMacro.s
        ENTRY
        MOV R1,#5
        MOV R2,#2
        ADD3VAL R0,R1,R2,#5
H1      B    H1
        END
```

MyMacro.s

```
        MACRO
        ADD3VAL $DEST,$ARG1,$ARG2,$ARG3
        ADD $DEST,$ARG1,$ARG2
        ADD $DEST,$DEST,$ARG3
        MEND
        END
```

Figure C. 1: Defining a Macro in an Include File

Macros vs. subroutines

Macros and subroutines are useful in writing assembly programs, but each has limitations. Macros increase code size every time they are invoked. For example, if you call a 10-instruction macro 10 times, the code size is increased by 100 instructions; whereas, if you call the same subroutine 10 times, the code size is only that of the subroutine instructions. On the other hand, a function call takes 3 clocks and the return instruction takes 3 clocks to get executed. So, using functions adds around 6 clock cycles. The subroutines might use stack space as well when called, while the macros do not.

Appendix D: Flowcharts and Pseudocode

Flowcharts

If you have taken any previous programming courses, you are probably familiar with flowcharting. Flowcharts use graphic symbols to represent different types of program operations. These symbols are connected together into a flowchart to show the flow of execution of a program. The more commonly used symbols are as follows:

Start and End points

Start and End points are commonly represented as rounded rectangles or ovals containing the words “Start” or “End”. Small circle is another way to show them.

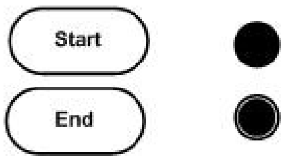


Figure D- 1: Start and End Points

Decisions

The conditions are represented in diamonds.

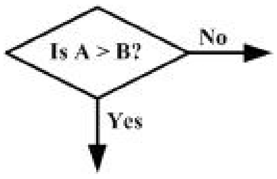


Figure D- 2: a Decision that Compares A with B

Process

The processing steps are represented using rectangles.

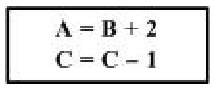


Figure D- 3: a Process Sample

Inputs and outputs

Inputs and outputs are represented as parallelogram.

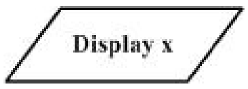


Figure D- 4: an Output Sample

Subroutines

Calling subroutines are represented as shown below

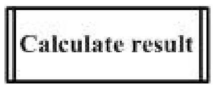


Figure D- 5: a Subroutine Call Sample

Pseudocode

Flowcharting has been standard practice in industry for decades. However, some find limitations in using flowcharts, such as the fact that you can't write much in the little boxes, and it is hard to get the "big picture" of what the program does without getting bogged down in the details. An alternative to using flowcharts is pseudocode, which involves writing brief descriptions of the flow of the code. Figures D-6 through D-10 show flowcharts and pseudocode for commonly used control structures.

Structured programming uses three basic types of program control structures: sequence, control, and iteration. Sequence is simply executing instructions one after another. Figure D-6 shows how sequence can be represented in pseudocode and flowcharts.

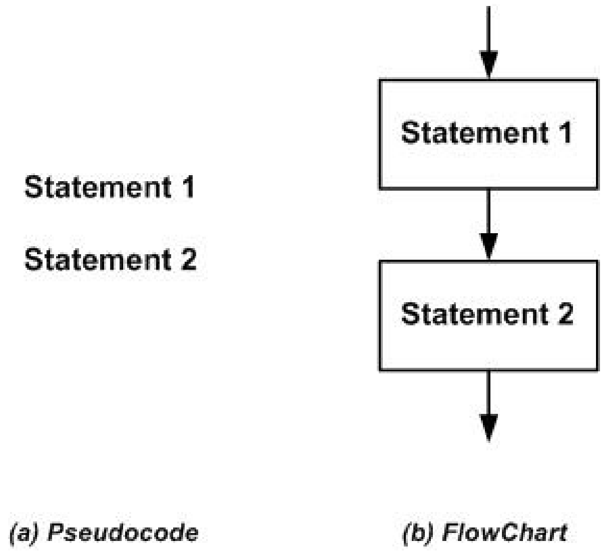


Figure D- 6: SEQUENCE Pseudocode versus Flowchart

Note in Figures D-6 through D-11 that "statement" can indicate one statement or a group of statements.

Figure D-7 through D-9 show two control programming structures: IF-THEN-ELSE and IF-THEN in both pseudocode and flowcharts.

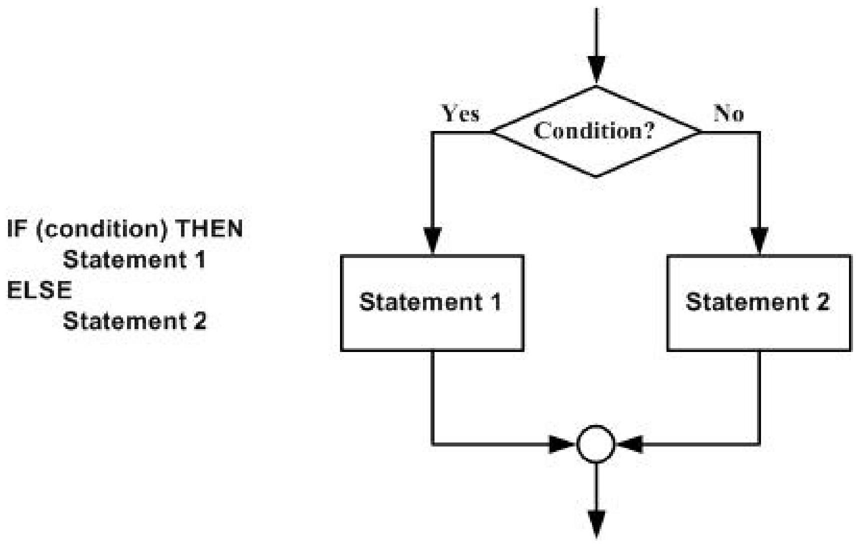
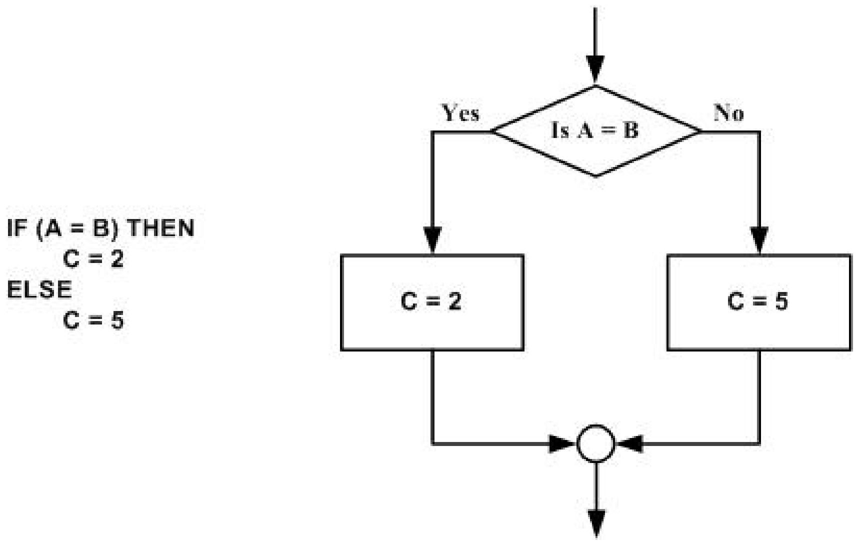
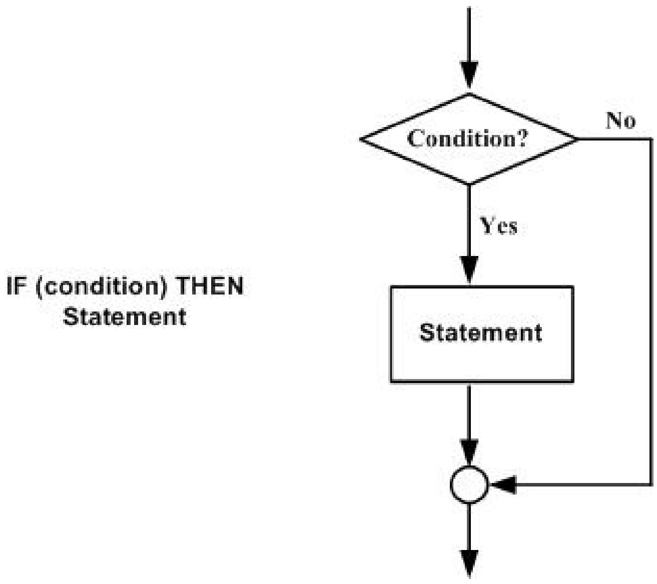


Figure D- 7: IF THEN ELSE Pseudocode versus Flowchart



```
IF (A = B) THEN
     C = 2
ELSE
     C = 5
```

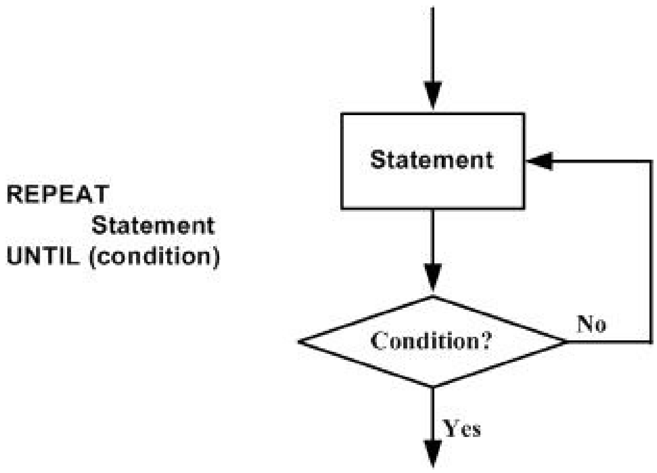
Figure D- 8: an IF THEN ELSE Sample



```
IF (condition) THEN
     Statement
```

Figure D- 9: IF THEN Pseudocode versus Flowchart

Figures D-10 and D-11 show two iteration control structures: REPEAT UNTIL and WHILE DO. Both structures execute a statement or group of statements repeatedly. The difference between them is that the REPEAT UNTIL structure always executes the statement(s) at least once, and checks the condition after each iteration, whereas the WHILE DO may not execute the statement(s) at all because the condition is checked at the beginning of each iteration.



```
REPEAT
     Statement
UNTIL (condition)
```

Figure D- 10: REPEAT UNTIL Pseudocode versus Flowchart

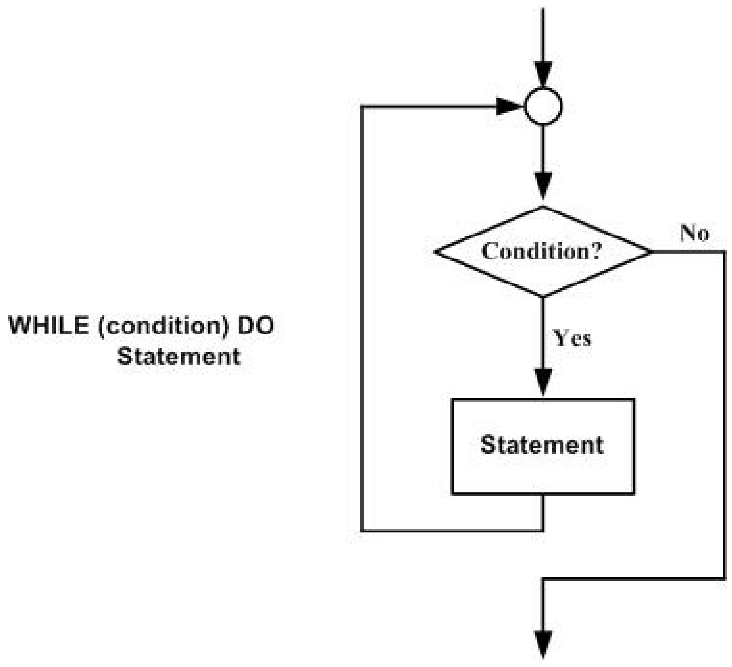


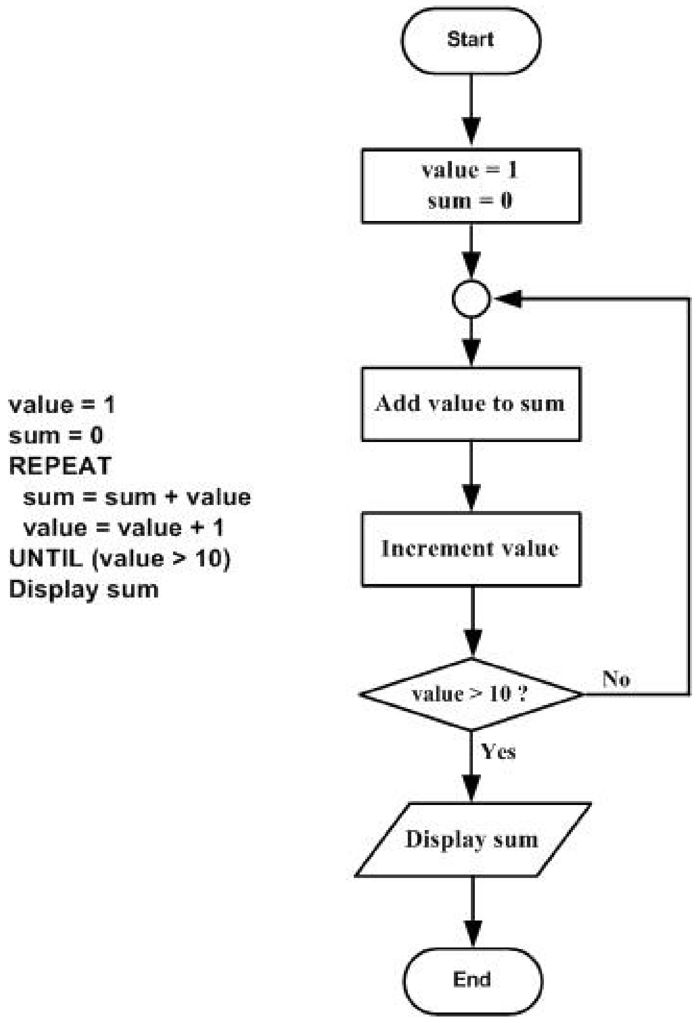
Figure D- 11: WHILE DO Pseudocode versus Flowchart

Program D-1 finds the sum of numbers between 1 and 10. Compare the flowchart versus the pseudocode for Program D-1 (shown in Figure D-12). In this example, more program details are given than one usually finds. For example, this shows steps for initializing and changing values. Another programmer may not include these steps in the flowchart or pseudocode. It is important to remember that the purpose of flowcharts or pseudocode is to show the flow of the program and what the program does, not the specific Assembly language instructions that accomplish the program's objectives. Notice also that the pseudocode gives the same information in a much more compact form than does the flowchart. It is important to note that sometimes pseudocode is written in layers, so that the outer level or layer shows the flow of the program and subsequent levels show more details of how the program accomplishes its assigned tasks.

Program D-1

```
int main ()
{
   int sum = 0;
   int value = 1;

   do{
   sum = sum + value;
   value++;
   }while(value <= 10);

```

```
    printf (“%d”, sum);
}
```



```
value = 1
sum = 0
REPEAT
  sum = sum + value
  value = value + 1
UNTIL (value > 10)
Display sum
```

Figure D-12: Pseudocode versus Flowchart for Program D-1

Appendix E: Passing Arguments into Functions

There are different ways to pass arguments to functions. Some of them are:

- through registers
- through memory using references
- using stack

E.1: Passing arguments through registers

In the following program the BIGGER function gets two values through R0 and R1. After comparing R0 and R1, it returns the bigger value through R2.

Program E-1

```
    AREA     OUR_PROG,CODE,READONLY
    ENTRY
    MOV     R0,#5     ;R0 = 5
    MOV     R1,#7     ;R1 = 7
    BL          BIGGER                ;BIGGER(5,7)
HERE     B             HERE      ;stay here

    ;======================================
    ;BIGGER returns the bigger value
    ;Parameters:
    ;               R0 and R1: the values to be compared
    ;Returns:
    ;               R2: containing the bigger value
    ;======================================
BIGGER
    CMP      R0,R1
    BHI         L1                          ;if R0 > R1 goto L1
    MOV     R2,R1     ;R2 = R1
    BX          LR                          ;return

L1           MOV     R2,R0     ;R2 = R0
    BX          LR                          ;return

    END
```

This is a fast way of passing arguments to the function.

E.2: Passing through memory using references

We can store the data in memory and pass its address through a register. In the following program the STR_LENGTH function gets the address of a zero-ended string through R0 and returns the length of the string through R1.

Program E-2

```
    AREA     OUR_PROG,CODE,READONLY
    ENTRY
    ADR       R0,OUR_STR      ;R0 = addr. of OUR_STR
    BL          STR_LENGTH      ;STR_LENGTH(&OUR_STR)
HERE      B              HERE       ;stay here

OUR_STR DCB   "HELLO!"
    ALIGN 4
    ;========================================
    ;STR_LENGTH returns the length of str
    ;Parameters:
    ;               R0: address of the string
    ;Returns:
    ;               R0: the length of string
    ;========================================
STR_LENGTH
    MOV     R2,#0
L_BEGIN
    LDRB      R5,[R0]
    CMP      R5,#0
    BXEQ     LR          ;return if 0
    ADD       R2,R2,#1             ;R2=R2+1
    ADD       R0,R0,#1             ;R0=R0+1
    B             L_BEGIN

    END

```

E.3: Passing arguments through stack

Passing through the stack is a flexible way of passing arguments. To do so, the arguments are pushed into the stack just before calling the function and popped off after returning. In Program E-3, the BIGGER function gets two arguments through the stack and returns the bigger value through the stack.

Program E-3

```
    AREA    OUR_PROG,CODE,READONLY
    ENTRY

    ;init stack pointer
    LDR SP,=(0x40000000+(16*1024))

    MOV     R0,#5
    PUSH    {R0}        ;push Arg1
    MOV     R0,#7
    PUSH    {R0}        ;push Arg2

    BL          BIGGER              ;BIGGER(5,7)
    LDR         R1,[SP,#0]          ;R1=return value
    POP     {R0}
    POP     {R0}
HERE    B           HERE        ;stay here

    ;=====================================
    ;BIGGER returns the bigger value
    ;Parameters:
    ;           values to be compared
    ;Returns:
    ;           the bigger value
    ;=====================================
BIGGER
    LDR         R0,[SP,#4]          ;R0 = arg1
    LDR       R1,[SP,#0]          ;R1 = arg2

    CMP     R0,R1
    BHI       L1                                          ;if R0 > R1 goto L1
    STR       R1,[SP,#0]          ;return R1
    BX        LR                         ;return

L1          STR         R0,[SP,#0]          ;return R0
    BX        LR                          ;return

    END
```

This method of passing arguments are used in x86 computers.

E.4: AAPCS (ARM Application Procedure Call Standard)

The AAPCS provides an standard for implementing the functions and the function calls so that the codes made by different compilers and different programmers can work with each other. Some of the rules of the standard are:

- The arguments must be sent through R0 to R4, respectively.
- The return value must be sent back through R0 and R1.
- The functions can use R5 to R8 for their internal calculations. But their values must be retrieved before returning. To do so, we push the registers before using them and pop them before returning from the function.
- The stack must be used as Full Descending

In Program E-1 most of the above rules are considered. But the return value must be in R0 instead of R2.

In Program E-4 the above rules are considered.

Program E-4

```
    AREA     OUR_PROG,CODE,READONLY
    ENTRY

    ;init stack pointer
        ;(change it according to your chip)
    LDR SP,=(0x40000000+(16*1024))

    MOV     R0,#20
    BL          DELAY   ;DELAY(20)
HERE      B             HERE      ;stay here

    ;=========================================
    ;DELAY waits for a while
    ;Parameters:
    ;               R0: the amount of wait
    ;Returns:
    ;               none
    ;=========================================
DELAY
```

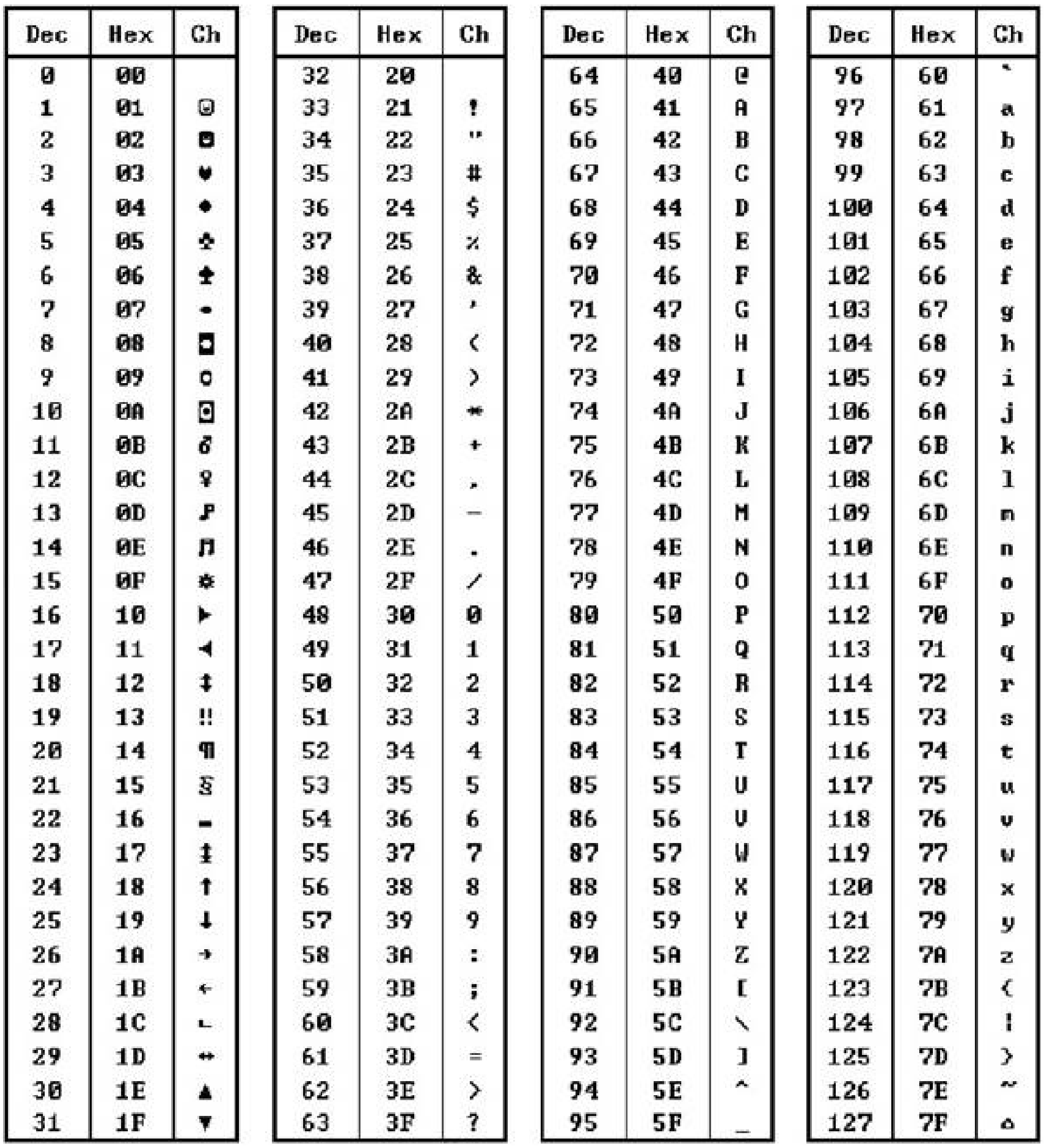
```
    CMP      R0,#0
    BXEQ     LR           ;return if zero

    PUSH     {R5}         ;save R5

    LDR      R5,=5000000      ;R5 = 5000000
L1          SUBS    R5,R5,#1              ;R5=R5-1
    BNE      L1           ;goto L1 if R5 is not zero

    POP      {R5}         ;restore R5
    BX       LR           ;return

    END
```

More information

For more information about AAPCS see the following article or search “AAPCS” on the Internet:

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

Appendix F: ASCII Codes



Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
0	00		32	20		64	40	@	96	60	`
1	01	☺	33	21	!	65	41	A	97	61	a
2	02	☻	34	22	"	66	42	B	98	62	b
3	03	♥	35	23	#	67	43	C	99	63	c
4	04	♦	36	24	$	68	44	D	100	64	d
5	05	♣	37	25	%	69	45	E	101	65	e
6	06	♠	38	26	&	70	46	F	102	66	f
7	07	•	39	27	'	71	47	G	103	67	g
8	08	◘	40	28	(	72	48	H	104	68	h
9	09	○	41	29	)	73	49	I	105	69	i
10	0A	◙	42	2A	*	74	4A	J	106	6A	j
11	0B	♂	43	2B	+	75	4B	K	107	6B	k
12	0C	♀	44	2C	,	76	4C	L	108	6C	l
13	0D	♪	45	2D	-	77	4D	M	109	6D	m
14	0E	♫	46	2E	.	78	4E	N	110	6E	n
15	0F	☼	47	2F	/	79	4F	O	111	6F	o
16	10	►	48	30	0	80	50	P	112	70	p
17	11	◄	49	31	1	81	51	Q	113	71	q
18	12	↕	50	32	2	82	52	R	114	72	r
19	13	‼	51	33	3	83	53	S	115	73	s
20	14	¶	52	34	4	84	54	T	116	74	t
21	15	§	53	35	5	85	55	U	117	75	u
22	16	▬	54	36	6	86	56	V	118	76	v
23	17	↨	55	37	7	87	57	W	119	77	w
24	18	↑	56	38	8	88	58	X	120	78	x
25	19	↓	57	39	9	89	59	Y	121	79	y
26	1A	→	58	3A	:	90	5A	Z	122	7A	z
27	1B	←	59	3B	;	91	5B	[	123	7B	{
28	1C	∟	60	3C	<	92	5C	\	124	7C	¦
29	1D	↔	61	3D	=	93	5D	]	125	7D	}
30	1E	▲	62	3E	>	94	5E	^	126	7E	~
31	1F	▼	63	3F	?	95	5F	_	127	7F	⌂