Ata Elahi · Trevor Arjeski

ARM Assembly Language with Hardware Experiments

Springer

ARM Assembly Language with Hardware Experiments

Ata Elahi • Trevor Arjeski

ARM Assembly Language with Hardware Experiments

Springer

Ata Elahi
Southern Connecticut State University
New Haven
Connecticut
USA

Trevor Arjeski
Southern Connecticut State University
New Haven
Connecticut
USA

ISBN 978-3-319-11703-4 ISBN 978-3-319-11704-1 (eBook)
DOI 10.1007/978-3-319-11704-1

Library of Congress Control Number: 2014955658

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

ARM is one of the leading suppliers of microprocessors for the entire world. ARM has designed and developed a CPU that partner companies can manufacture and add more peripherals to the processor. An ARM processor has a wide range of application in today's technology, such as mobile phones, tablets, televisions, and automobiles. Learning the ARM instruction set and ARM assembly programming is an essential tool in the development of low-level applications for the ARM processor. Engineers will benefit significantly from the understanding of computer architecture and assembly language, especially if they are working in an industry where performance is crucial or hardware is being developed.

Organization This book contains seven chapters. The reader does not require any background in ARM assembly language to understand material of this book.

Chapters one and two of this book form a foundation for the rest of the chapters.

Chapter 1 covers some necessary knowledge of digital signals, analog signals, number systems and transmission methods.

Chapter 2 covers logic gates, registers and an introduction to computer architecture.

Chapters 3 and 4 cover the ARM processor architecture with its instructions.

Chapter 5 covers ARM assembly language programming using Keil development tools.

Chapter 6 covers ARM Cortex-M3 processor architecture, the MBED NXP LPC1768 and basic GPIO Programming.

Chapter 7 covers lab experiments that include:

- Creating a binary counter using onboard LEDs
- Configuring an Analog-To-Digital Converter (ADC)
- Creating a voltmeter with an ADC
- Configuring Digital to Analog Converter (DAC)
- Converting binary to output for a hexadecimal display
- Configuring a Real-Time Clock (RTC)

Intended Audience This book is written primarily as an introduction to assembly language for students who are studying computer science, computer engineering,

or hobbyists who are simply interested in learning ARM assembly programming with hands-on experiments. This book can be used as a first course in computer system which covers numbers systems, Digital Logics, Introduction to Computer Architecture and Assembly language for computer science and computer technology students.

Contents

Chapter 1
Number Systems and Data Communication

1.1 Introduction

In order to understand network technology it is important to know how information is represented for transmission from one computer to another. Information can be transferred between computers in one of two ways: an analog signal or a digital signal.

1.2 Analog Signals

An analog signal is a signal whose amplitude is a function of time and changes gradually as time changes. Analog signals can be classified as non-periodic and periodic signals.

Non-Periodic Signal In a non-periodic signal there is no repeated pattern in the signal as shown in Fig. 1.1.

Periodic Signal A signal that repeats a pattern within a measurable time period is called a periodic signal and completion of a full pattern is called a *cycle*. The simplest periodic signal is a sine wave, which is shown in Fig. 1.2. In the time domain, a sine wave's amplitude $a(t)$ can be represented mathematically as $a(t) = ASin(\omega t + \theta)$ where A is the maximum amplitude, ω is the angular frequency and θ is the phase angle.

A periodic signal can also be represented in the frequency domain where the horizontal axis is the frequency and the vertical axis is the amplitude of signal. Figure 1.3 shows the Frequency domain representation of a sine wave signal.

Usually an electrical signal representing voice, temperature or a musical sound, is made of multiple waveforms. These signals have one fundamental frequency and multiple frequencies that are called harmonics.

Characteristics of Analog Signal The characteristics of a periodic analog signal are frequency, amplitude, and phase.

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_1

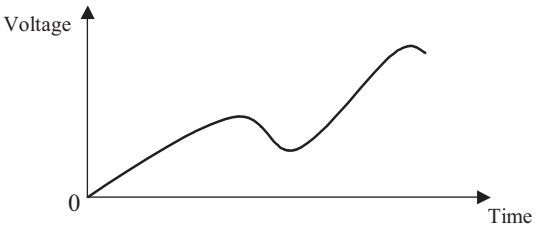


Fig. 1.1 Representation of a non-periodic analog signal

Frequency: Frequency (F) is the number of cycles in one second; $F = \frac{1}{T}$, represented in Hz (Hertz). If each cycle of an analog signal is repeated every one second, the frequency of the signal is one Hz. If each cycle of an analog signal is repeated 1000 times every second (once every millisecond) the frequency is:

$$f = \frac{1}{T} = \frac{1}{10^{-3}} = 1000Hz = 1kHz$$

Table 1.1 shows different values for frequency and their corresponding periods.

Amplitude: The Amplitude of an analog signal is a function of time as shown in Fig. 1.4 and may be represented in volts (unit of voltage). In other word, the amplitude is its voltage value at any given time. At the time of t_1, the amplitude of signal is V_1.

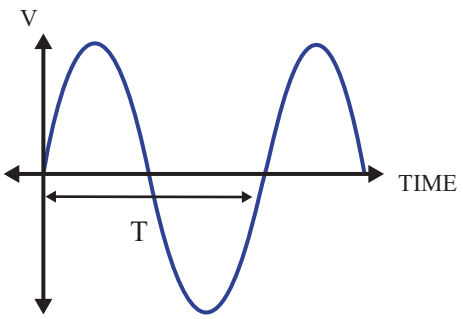


Fig. 1.2 Time domain representation of a sin wave

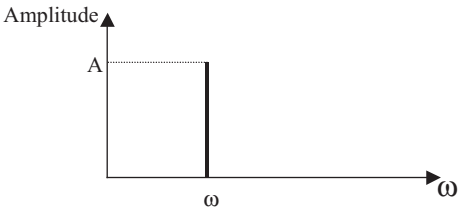


Fig. 1.3 Frequency representation of a sine wave

Phase: Two signals with the same frequency can differ in phase. This means that one of the signals starts at a different time from the other one. This difference can be represented by degree, from 0 to 360 degrees or by radians where $360^o = 2\pi$ radians. A sine wave signal can be represented by the equation $a(t) = ASin(\omega t + \theta)$ where A is the peak amplitude; ω (omega) is frequency in radians per second; t is time in seconds; and θ is the phase angle. Cyclic frequency f can be expressed in terms of ω according to $f = \frac{\omega}{2\pi}$. A phase angle of zero means the sine wave starts at time $t = 0$ and phase angle of 90° mean the signal start at 90° as shown in Fig. 1.5.

Example 1.1: Find the equation for a sine wave signal with frequency of 10 *Hz*, maximum amplitude of 20 V and phase angle of zero.

$$\omega = 2\pi f = 2 \times 3.1416 \times 10 = 62.83 \frac{rad}{\text{sec}}$$

$$a(t) = 20\sin(62.83t)$$

Table 1.1 Typical units of frequency and period

Units of frequency	Numerical value	Units of period	Numerical value
Hertz (Hz)	1 *Hz*	Second (s)	1 s
Kilo Hertz (k*Hz*)	10^3 *Hz*	Millisecond (ms)	10^{-3} s
Mega Hertz (M*Hz*)	10^6 *Hz*	Micro Second (μs)	10^{-6} s
Giga Hertz (G*Hz*)	10^9 *Hz*	Nanosecond (ns)	10^{-9} s
Tera Hertz (T*Hz*)	10^{12} *Hz*	Pico Second (*p*s)	10^{-12} s

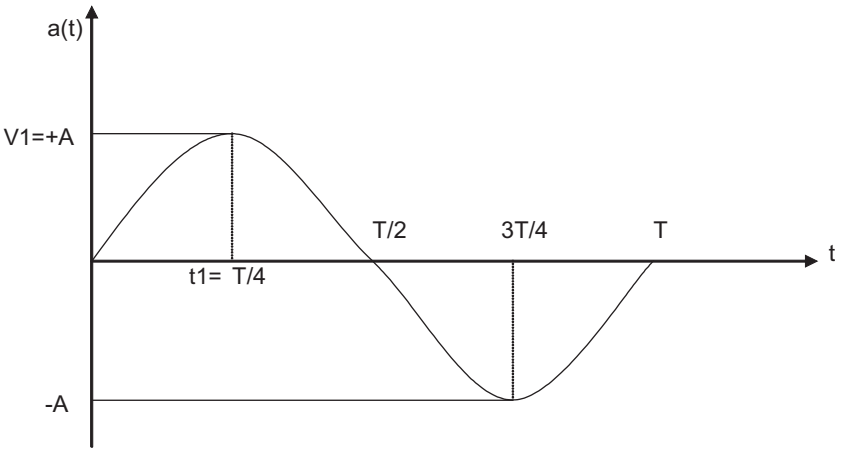


Fig. 1.4 A sine wave signal over one cycle

1.3 Digital Signals

Modern computers communicate by using digital signals. **Digital signals** are represented by two voltages: one voltage represents the number 0 in binary and the other voltage represents the number 1 in binary. An example of a digital signal is shown in Fig. 1.6, where 0 V represents 0 in binary and +5 V represents 1.

1.4 Number System

Numbers can be represented in different bases, consider the following number in decimal:

$$356 = 6 + 50 + 300 = 6*10^0 + 5*10^1 + 3*10^2$$

356 has a base of 10 or, more commonly called, decimal.

In general, a number can be represented in the form

$(a_5\ a_4\ a_3\ a_2\ a_1\ a_0\ .a_{-1}\ a_{-2}\ a_{-3})$ r where r is base of the number and a_i has to be less then r

Equation 1.1 can be used to converting a number in given base to decimal

$$(\underbrace{a_5\ a_4\ a_3\ a_2\ a_1\ a_0}_{\text{Integer}} \cdot \underbrace{a_{-1}\ a_{-2}\ a_{-3}}_{\text{Fraction}})_r$$
$$= a_0 \times r^0 + a_1 \times r^1 + a_2 \times r^2 + a_3 \times r^3 + \ldots + a_{-1} \times r^{-1} + a_{-2} \times r^{-2} + a_{-2} \times r^{-3} \ldots\ldots \quad (1.1)$$

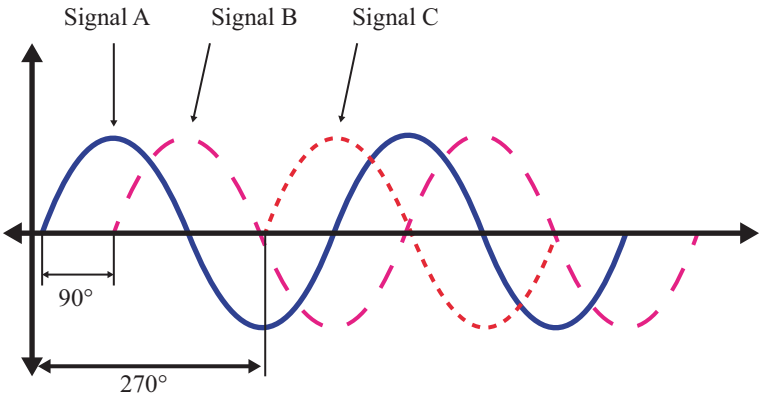
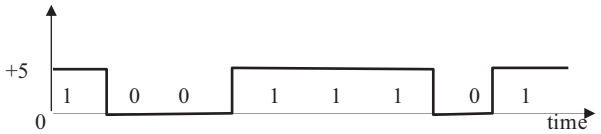


Fig. 1.5 Three sine waves with different phases

Fig. 1.6 Digital signal



Example 1.2 Converting $(27.35)_8$ to base 10

$$(27.35)_8 = 7*8^0 + 2*8^1 + 3*8^{-1} + 5*8^{-2} = 7 + 16 + .375 + .078125 = (23.45)_{10}$$

Example 1.3 Convert 1101111 to decimal

$$\begin{aligned}(1101111)_2 &= 1*2^0 + 1*2^1 + 1*2^2 + 1*2^3 + 0*2^4 + 1*2^5 + 1*2^6 \\ &= 1 + 2 + 4 + 8 + 32 + 64 = (111)_{10}\end{aligned}$$

Converting from Binary to Decimal Equation 1.2 represent any binary number.

$$(a_5\, a_4\, a_3\, a_2\, a_1\, a_0 \cdot a_{-1}\, a_{-2}\, a_{-3})_2 \tag{1.2}$$

where

a_i is a binary digit or bit (either 0 or 1)

Equation 1.2 can be converted to decimal number by using Eq. 1.1

$$\begin{aligned}&(\underbrace{a_5\, a_4\, a_3\, a_2\, a_1\, a_0}_{\text{Integer}} \cdot \underbrace{a_{-1}\, a_{-2}\, a_{-3}}_{\text{Fraction}})_2 = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 + a_3 \times 2^3 + \ldots\ldots + a_{-1} \\ &\times 2^{-1} + a_{-2} \times 2^{-2} + \ldots \left(a_5\, a_4\, a_3\, a_2\, a_1\, a_0 \cdot a_{-1}\, a_{-2}\, a_{-3}\right)_2 \\ &= a_0 + 2a_1 + 4a_2 + 8a_3 + 16a_4 + 32a_5 + 64a_6 + ½ * a_{-1} + ¼ * a_{-2} + ⅛ * a - 3\end{aligned} \tag{1.3}$$

Example 1.4: To convert $(110111.101)_2$ to decimal:

$$\begin{aligned}(110111.101)_2 &= 1*2^0 + 1*2^1 + 1*2^2 + 0*2^3 + 1*2^4 \\ &\quad + 1*2^{*5} + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 55.625\end{aligned}$$

Or

32	16	8	4	2	1		½	¼	⅛
1	**1**	**0**	**1**	**1**	**1**	.	**1**	**0**	**1**

$$32 + 16 + 0 + 4 + 2 + 1 + 1/2 + 0 + 1/8$$

Binary, or Base-2 numbers, are represented by 0's and 1's. A binary digit, 0 or 1, is called a bit. Eight bits are equal to one byte, and 4 bytes is called a word.

Converting From Decimal Integer to Binary: To convert an integer number from decimal to binary, divide the decimal number by the new base (2 for binary), which will result in a quotient and a remainder (either 0 or 1). The first remainder will be the least significant bit of the binary number. Continually divide the quotient by the new base, while taking the remainders as each subsequent bit in the binary number, until the quotient becomes zero.

Example 1.5: Convert 34 in decimal to binary.

	Quotient	Remainder
34/2 =	17	0 = a_0
17/2 =	8	1 = a_1
8/2	4	0 = a_2
4/2	2	0 = a_3
2/2	1	0 = a_4
1/2	0	1 = a_5

Therefore 34 = $(100010)_2$

Converting Decimal Fraction to Binary: A decimal number representation of $(0.XY)_{10}$ can be converted into base 2 resulting in the representation, $(0.a_{-1}\ a_{-2}\ a_{-3}\ldots)_2$.

The fraction number is multiplied by 2, the result of integer part is a_{-1} and fraction part multiply by 2 and then separate integer part from fraction, the integer part represent a_{-2}, this processes continues until the fraction becomes zero.

$(0.35)_{10}$ = ( )$_2$

0.35*2 =	0.7 =	0 + 0.7	a-1=0
0.7*2 =	1.4 =	1 + 0.4	a-2=1
0.4*2 =	0.8 =	0 + 0.8	a-3=0
0.8*2 =	1.6 =	1 + 0.6	a-4=1
0.6*2 =	1.2 =	1 + 0.2	a-5=1

Sometime the fraction does not reach zero and how many bits a decimal fraction should be represented depend on accuracy the user define.

The 0.35 = 0.01011 in binary

The hexadecimal number system has a base of 16, and therefore has 16 symbols (0 through 9, and A through F). Table 1.2 shows the decimal numbers, their binary values from 0 to 15, and their hexadecimal equivalents.

Table 1.2 Decimal numbers with binary and hexadecimal equivalents

Decimal	Binary (base 2)	Hexadecimal (Base 16) or HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting from Binary to Hex: Table 1.2 can also be used to convert a number from hexadecimal to binary and from binary to hexadecimal.

Example 1.5 Convert the binary number 001010011010 to hexadecimal. Each 4 bits are grouped from right to left. By using Table 2.2, each 4-bit group can be converted to its hexadecimal equivalent.

0010	1001	1010
2	**9**	**A**

Example 1.6: Convert $(3D5)_{16}$ to binary. By using Table 2.2, the result in binary is

3	D	5
0011	**1101**	**0101**

The resulting binary number is: 001111010101

Example 1.7: Convert 6DB from hexadecimal to binary. By using Table 1.2, the result in binary is

6	D	B
0110	1101	1011

The resulting binary number is: 011011011011

Example 1.8: **C**onvert $(110111.101)_2$ to decimal:

$$(110111.101)_2 = 1*2^0 + 1*2^1 + 1*2^2 + 0*2^3 + 1*2^4 + 1*2^{*5} + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 55.625$$

Binary Addition:

```
  1 +0 =1, 1+1=10,

Cary bits 11  1
          10101
        +  01101
        -------------
          100010
```

Complement and Two's Complement: The complement of 1 is zero and complement of 0 is one.

The complement of a binary number is calculated by complementing each bit of the number.

Example 1.9: The complement of 101101 is 010010

Two's Complement of a number = Complement of a number + 1

Example 1.10: The two's complement of 101011 is

$$010100 \text{ (complement)} + 1 = 010101$$

Example: Find the two's complement of 10000

$$01111 \text{ (complement)} + 1 = 10000$$

Subtraction using Two's Complement: Following procedure describe to subtract $B = b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ from $A = a_5\ a_4\ a_3\ a_2\ a_1\ a_0$.

1. Add Two's complement of B to the A
2. Check if result produce carry
   a. If result produce carry then discard the carry and result is positive
   b. If result does not produce carry, take two's complement of result and result is negative.

```
 Example:  Subtract B=101010 from A=110101
  Step1;find two's complement of B, ComplementB +1
      Two's complement of B = 010101+1 =010110
       Add 2's complementofB to A
        110101
     +  010110
     -----------
        10 01011
```

```
Carry overflow, discard the carry and result is   + 001011

Example1.11:   Subtract B= 110101 from A=101010

Two's complement of B is 001010+1= 001011
Add 2's complement of B to A

                 001011
             +   101010
                 ----------
                 110101
```

As we can see, adding two 6 bit number results in a 6 bits answer. There is no carry over flow so we just take the two's complement of the result.

Two's Complement of 110101 = 001010 + 1 = −001011

Unsigned, Signed Binary and Signed Two's Complement Numbers: In an unsigned number all bits are used to represent the number but in a signed number the most significant bit of the number represents the sign. A 1 represents a negative sign and 0 represents a positive sign. The unsigned number 1101 is 13

Signed Number: In a signed number the most significant bit represents the sign, where 1101 = −5 or 0101 = +5

Signed Two's Complement: A signed two's complement apply to negative number, if the sign bit of number is negative, the number is represented by signed two's complement.

Example 1.12: Representing −5 with 4 bits in signed two's complement.

−5 in signed number is 1101, the two's complement of 101 (5) is 011 then 1011 represent −5 in signed two's complement.

Example 1.13: Represent −23 with 8 bit signed two's complement

23 in binary is 10111,

23 in 8 bit signed number is 10010111, the two's complement (not including the sign) is

11101001

Binary Coded Decimal (BCD): In daily life we use decimal numbers where the largest digit is 9, which is represented by 1001 in binary. Table 1.3 shows decimal number and corresponding BCD code.

Table 1.3 Binary Coded Decimal (BCD)

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Example 1.14 Converting 345 to BCD.
Using the table: 0011 0100 0101.

Example 1.15 Converting $(10100010010)_{BCD}$ to decimal, separate each four bits from right to left and substituting the corresponding decimal number with BCD results in 512.

1.5 Coding Schemes

Since computers can only understand binary numbers (0 or 1), all information (such as numbers, letters and symbols) must be represented as binary data. One commonly used code to represent printable and non-printable characters is the American Standard Code for Information Interchange (ASCII).

ASCII Code Each character in ASCII code has a representation using 8 bits, where the most significant bit is used for parity bit. Table 2.3 shows the **ASCII code** and its hexadecimal equivalent.

Characters from hexadecimal 00 to 1F and 7F are control characters which are nonprintable characters, such as NUL, SOH, STX, ETX, ESC and DLE (data link escape).

Example 1.16 Convert the word "Network" to binary and show the result in hexadecimal. By using Table 1.4 each character is represented by seven bits and results in:

1001110	1100101	1110100	1110111	1101111	1110010	1101011
N	e	t	w	o	r	k

or in hexadecimal

4E 65 74 77 6F 72 6B

Table 1.4 American Standard Code for Information Interchange (ASCII)

Binary	Hex	Char	Binary	Hex	Char	Binary	Hex	Char	Binary	Hex	Char
0000000	00	***NUL***	0100000	20	**SP**	1000000	40	**@**	1100000	60	’
0000001	01	***SOH***	0100001	21	**!**	1000001	41	**A**	1100001	61	**a**
0000010	02	***STX***	0100010	22	**“**	1000010	42	**B**	1100010	62	**b**
0000011	03	***ETX***	0100011	23	**#**	1000011	43	**C**	1100011	63	**c**
0000100	04	***EOT***	0100100	24	**$**	1000100	44	**D**	1100100	64	**d**
0000101	05	***ENQ***	0100101	25	**%**	1000101	45	**E**	1100101	65	**e**
0000110	06	***ACK***	0100110	26	**&**	1000110	46	**F**	1100110	66	**f**
0000111	07	***BEL***	0100111	27	**‘**	1000111	47	**G**	1100111	67	**g**
0001000	08	***BS***	0101000	28	**(**	1001000	8	**H**	1101000	68	**h**
0001001	09	***HT***	0101001	29	**)**	1001001	49	**I**	1101001	69	**i**
0001010	0A	***LF***	0101010	2A	*	1001010	4A	**J**	1101010	6A	**j**
0001011	0B	***VT***	0101011	2B	+	1001011	4B	**K**	1101011	6B	**k**
0001100	0C	*FF*	0101100	2C	,	1001100	4C	**L**	1101100	6C	**l**
0001101	0D	**CR**	0101101	2D	-	1001101	4D	**M**	1101101	6D	**m**
0001110	0E	***SO***	0101110	2E	.	1001110	4E	**N**	1101110	6E	**n**
0001111	0F	**SI**	0101111	2F	/	1001111	4F	**O**	1101111	6F	**o**
0010000	10	**DLE**	0110000	30	**0**	1010000	50	**P**	1110000	70	**p**
0010001	11	**DC1**	0110001	31	**1**	1010001	51	**Q**	1110001	71	**q**
0010010	12	**DC2**	0110010	32	**2**	1010010	52	**R**	1110010	72	**r**
0010011	13	**DC3**	0110011	33	**3**	1010011	53	**S**	1110011	73	**s**
0010100	14	**DC4**	0110100	34	**4**	1010100	54	**T**	1110100	74	**t**
0010101	15	**NACK**	0110101	35	**5**	1010101	55	**U**	1110101	75	**u**
0010110	16	**SYN**	0110110	36	**6**	1010110	56	**V**	1110110	76	v
0010111	17	**ETB**	0110111	37	**7**	1010111	57	**W**	1110111	77	**w**
0011000	18	**CAN**	0111000	38	**8**	1011000	58	**X**	1111000	78	**x**
0011001	19	**EM**	0111001	39	**9**	1011001	59	**Y**	1111001	79	**y**
0011010	1A	**SUB**	0111010	3A	**:**	1011010	5A	**Z**	1111010	7A	**z**
0011011	1B	**ESC**	0111011	3B	**;**	1011011	5B	**[**	1111011	7B	**[**
0011100	1C	**FS**	0111100	3C	**<**	1011100	5C	\	1111100	7C	\
0011101	1D	**GS**	0111101	3D	=	1011101	5D	**]**	1111101	7D	}
0011110	1E	**RS**	0111110	3E	<	1011110	5E	**^**	1111110	7E	~
0011111	1F	**US**	0111111	3F	**?**	1011111	5F	**-**	1111111	7F	**DEL**

Universal Code or Unicode: Unicode is a new 16-bit character-encoding standard for representing characters and numbers in most languages such as Greek, Arabic, Chinese and Japanese. The ASCII code uses eight bits to represent each character in Latin, and it can represent 256 characters. The ASCII code does not support mathematical symbols and scientific symbols. Since Unicode uses 16 bits it can represent 65536 characters or symbols. A character in Unicode is represented by 16-bit binary, equivalent to four digits in hexadecimal. For example, the character B in Unicode is U0042H (U represents Unicode). The ASCII code is represented between $(00)_{16}$ to $(FF)_{16}$. For converting ASCII code to Unicode, two zeros are added to the left side of ASCII code; therefore, the Unicode to represent ASCII characters is between $(0000)_{16}$ to $(00FF)_{16}$. Table 1.5 shows some of the Unicode for Latin and Greek characters. Unicode is divided into blocks of code, with each block assigned to a specific language. Table 1.6 shows each block of Unicode for some different languages.

Table 1.5 Unicode values for some Latin and Greek characters

Latin		Greek	
Character	Code (Hex)	Character	Code (Hex)
A	U0041	φ	U03C6
B	U0042	α	U03B1
C	U0043	γ	U03B3
0	U0030	μ	U03BC
8	U0038	β	U03B2

Table 1.6 Unicode block allocations

Start Code(Hex)	End Code(Hex)	Block name
U0000	U007F	Basic Latin
U0080	U00FF	Latin supplement
U0370	U03FF	Greek
U0530	U058F	Armenian
U0590	U05FF	Hebrew
U0600	U06FF	Arabic
U01A0	U10FF	Georgian

1.6 Clock

0 and 1 continuously repeated is called clock as shown in Fig. 1.7.

Each cycle of clock consist of 1 and 0 and it is measured by time, if one cycle represented by T and unit of T is second then

F (Frequency) = 1/T the unit of frequency is Hertz (Hz) and unit of T is second

Example: What is frequency of a clock if one cyle of the clock equal to.5 ms

$$F = 1/T = 1/0.5 \times 10^{-3} = 2000 \text{ Hz}$$

1000 Hz	KHz (kilo Hertz)
10^6 Hz	MHz (Mega Hertz)
10^9 Hz	GHz (Giga Hertz)

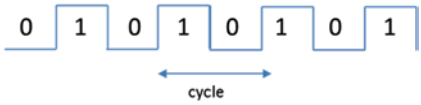


Fig. 1.7 Clock signals

1.7 Transmission Modes

When data is transferred from one computer to another by digital signals, the receiving computer has to distinguish the size of each signal to determine when a signal ends and when the next one begins. For example, when a computer sends a signal as shown in Fig. 1.8, the receiving computer has to recognize how many ones and zeros are in the signal. Synchronization methods between source and destination devices are generally grouped into two categories; Asynchronous and synchronous.

Asynchronous Transmission Asynchronous transmission occurs character by character and is used for serial communication, such as by a modem or serial printer. In asynchronous transmission each data character has a start bit which identifies the start of the character, and one or two bits which identifies the end of the character, as shown in Fig. 1.9. The data character is 7 bits. Following the data bits may be a parity bit, which is used by the receiver for error detection. After the parity bit is sent, the signal must return to high for at least one bit time to identify the end of the character. The new start bit serves as an indicator to the receiving device that a data character is coming and allows the receiving side to synchronize its clock. Since the receiver and transmitter clock are not synchronized continuously, the transmitter uses the start bit to reset the receiver clock so that it matches the transmitter clock. Also, the receiver is already programmed for the number of bits in each character sent by the transmitter.

Synchronous Transmission Some applications require transferring large blocks of data, such as a file from disk or transferring information from a computer to a printer. **Synchronous transmission** is an efficient method of transferring large blocks of data by using time intervals for synchronization.

One method of synchronizing transmitter and receiver is through the use of an external connection that carries a clock pulse. The clock pulse represents the data rate of the signal, as shown in Fig. 1.10, and is used to determine the speed of data transmission. The receiver of Fig. 2.9 reads the data as 01101, each bit width represented by one clock.

Fig. 1.8 Digital signals

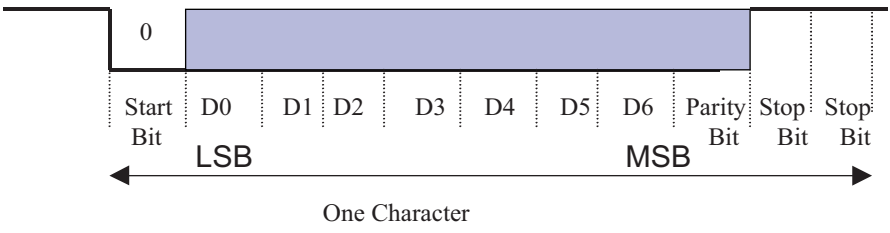


Fig. 1.9 Asynchronous transmission

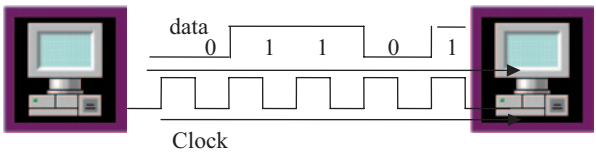


Fig. 1.10 Synchronous transmission

Figure 1.10 shows an extra connection is required to carry the clock pulse for synchronous transmission. In networking, one medium is used for transmission of both information and the clock pulse. The two signals are encoded in a way that the synchronization signal is embedded into the data. This can be done with Manchester encoding or Differential Manchester encoding.

1.8 Transmission Methods

There are two types of transmission methods used for sending digital signals from one station to another across a communication channel: serial transmission and parallel transmission.

Serial Transmission In **serial transmission**, information is transmitted one bit at a time over one wire as shown in Fig. 2.11.

Ground Line (Fig. 1.11)

Parallel Transmission In **parallel transmission**, multiple bits are sent simultaneously, one byte or more at a time, instead of bit by bit as in serial transmission. Figure 1.12 shows how computer A sends eight bits of information to computer B at the same time by using eight different wires. Parallel transmission is faster than serial transmission, at the same clock speed.

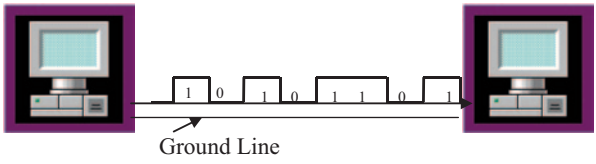


Fig. 1.11 Serial transmission

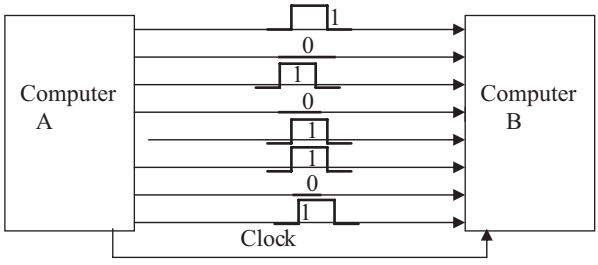


Fig. 1.12 Parallel transmission

Problems and Questions

1. Show an analog signal
2. Show a digital signal
3. Convert following decimal numbers to binary

    a. 35
    b. 85
    c. 23.25

4. Convert following binary numbers to decimal

    a. 1111101
    b. 1010111.1011
    c. 11111111
    d. 10000000

5. Convert following Binary numbers to Hexadecimal

    a. 1110011010
    b. 1000100111
    c. 101111.101

6. Convert following number to binary

    a. $(3FDA)_{16}$
    b. $(FDA.5F)_{16}$

7. Find two's complements of following numbers

    a. 11111111
    b. 10110000
    c. 10000000
    d. 00000000

8. Convert the word " LOGIC" to ASCII then represent each character in hex
9. Subtract following numbers using two's complement

    a. 11110011–11000011
    b. 10001101–11111000

10. List the types of transmission modes.
11. Why does a synchronous transmission require a clock?
12. What is frequency of an Analog signal repeated every 0.05 ms

Chapter 2
Logic Gates and Introduction to Computer Architecture

2.1 Introduction

The basic components of an Integrated Circuit (IC) is logic gates which made of transistors, in digital system there are three basic logic operations and they are called AND, OR and NOT.

2.2 Logic Gates

AND Logic The AND Logic is represented by ".". The most of the time, the period is left out. X.Y or XY is pronounced as X AND Y.

X AND Y = Z, Z = 1 if and only if X = 1 and Y = 1 otherwise Z = 0. The AND logic operation can represented by electrical circuit of Fig. 2.1.

Assume X and Y are switches and Z is the light, X = 0, Y = 0 means switches are open and light off means zero and light on means one, then we can make a Table 2.1 shows the operation of Fig. 2.1.

Figure 2.2 shows 2-Input AND gate and Table 2.2 show Truth table for AND gate. The output of AND gate is one when both inputs are one.

OR Logic The OR operation is represented by a + or V, where + is the most popular symbol used. X + Y is pronounced X OR Y.

$$X + Y = Z, \qquad Z = 1 \text{ if } X = 1 \text{ OR } Y = 1 \text{ or both } X = 1 \text{ and } Y = 1.$$

This OR operation can be represented by the electrical circuit in Fig. 2.3. In Fig. 2.3, the light is off when both switches are off, and light is on when at least one switch is close. Figure 2.4 shows 2-Input OR gate and Table 2.3 shows truth table for 2-Input OR gate.

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_2

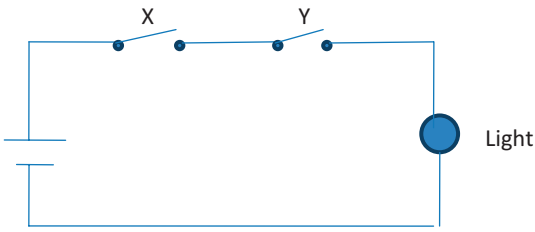


Fig. 2.1 Representation of AND operation

Table 2.1 Operation of Fig. 2.1

X	Y	Light
Off	Off	Off
Off	On	Off
On	Off	Off
On	On	On

Fig. 2.2 2-Input AND gate

Table 2.2 AND gate truth table

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

NOT Logic The NOT logic performs a complement, meaning it converts a 1 to 0 and 0 to 1. Also called an inverter, the NOT X is represented by X' or $\overline{X}$. Figure 2.5 shows NOT gate and Table 2.4 shows truth table for NOT gate (Inverter)

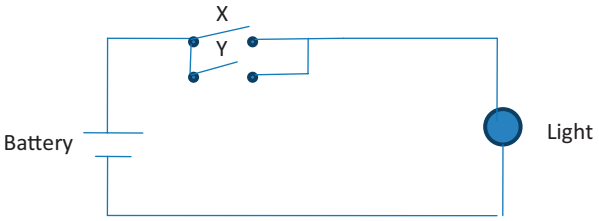


Fig. 2.3 Electrical circuit representation of OR operation

Fig. 2.4 2-Input OR gate

Table 2.3 Truth table of 2-Input OR gate

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

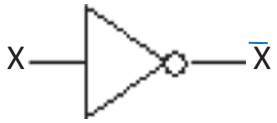


Fig. 2.5 NOT gate

Table 2.4 Truth table for not gate

X	X’
0	1
1	0

NAND Gate Figure 2.6 shows 2-input NAND gate, The NAND gate can be made from an AND and a NOT gate as shown in Fig. 2.7, Table 2.5 shows truth table of 2-Input NAND gate

Fig. 2.6 2-Input NAND gate

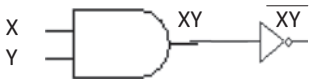


Fig. 2.7 AND-NOT

Table 2.5 Truth table of 2-Input NAND

X	Y	$\overline{XY}$
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 2.8 NOR gate

Table 2.6 Truth table for 2-Input NOR gate

X	Y	X+Y
0	0	1
0	1	0
1	0	0
1	1	0

NOR Gate Figure 2.8 shows a NOR logic gate. NOR gates are made of OR and NOT gates, Table 2.6 shows Truth table of 2-Input NOR gate.

Exclusive OR Gate Figure 2.9 shows an exclusive OR gate. Exclusive OR is represented by ⊙ and labeled XOR and Table 2.7 shows truth table for XOR gate.

Exclusive NOR Gate Figure 2.10 shows an exclusive NOR gate. Exclusive NOR is represented by ⊙ and labeled XNOR and Table 2.8 shows Truth Table for Exclusive NOR gate.

Fig. 2.9 2-Input XOR

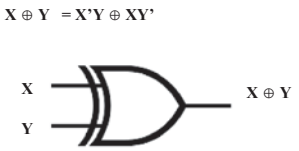


Table 2.7 Truth table for XOR gate

X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 2.10 Exclusive NOR gate

Table 2.8 Truth table for exclusive NOR gate

X	Y	X⊙Y
0	0	1
0	1	0
1	0	0
1	1	1

2.3 Integrated Circuit (IC) Classification

A transistor is a basic component of Integrated Circuits (IC). The Fig. 2.11 shows a transistor with an IC. Transistors act like a switch in Integrated Circuits. An Integrated circuit is made from 100 to millions transistors.

Integrated circuit classified based on number of the gates such SSI, MSI, LSI and VLSI.

Small Scale Integration (SSI) SSI refers to an IC that has less than 10 gates.

Medium Scale Integration (MSI) Refers to an IC that contains between 10 and 100 gates such as Decoders and Multiplexers.

Fig. 2.11 Transistor (*left*), IC (*right*)

Large Scale Integration (LSI) Refers to an IC that contains between 100 to 1000 gates.

Very Large Scale integration (VLSI) Refers to an IC that contains more than 1000 gates.

2.4 Registers

The registers are read/write memory that holds information inside the CPU. Each bit of a register is made of a D-flip flop as shown in Fig. 2.12 and Table 2.9 shows characteristic table for D-flip flop.

D Flip-Flop Operation As shown in Fig. 2.12, if the input of the flip flop is D = 0 then by applying a clock pulse the output is set to zero. If D = 1, applying a clock pulse sets the output to 1. The data will be stored in the flip-flop after applying a clock pulse. A register uses multiple D flip-flops that have a common clock pulse. Figure 2.13 shows 4 bit register.

If 32 D flip-flops use a common clock then it is called a 32-bit register.

Tri-State Device Figure 2.14 shows the diagram of tri-state device, the control line controls the operation of tri state device.

In Fig. 2.14 if control line set to zero there is no connection between input and output. If control line set to one the output value is equal to the input value.

2.5 Introduction to Computer Architecture

Just as the architecture of a building defines its overall design and functions, so computer architecture defines the design and functionality of a computer system. The components of a microcomputer are designed to interact with one another, and this interaction plays an important role in the overall system operation.

Fig. 2.12 D-Flip Flop

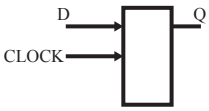


Table 2.9 Characteristic table of D-Flip Flop

CLOCK	D	Q
CK	0	0
CK	1	1

Fig. 2.13 4 bit register

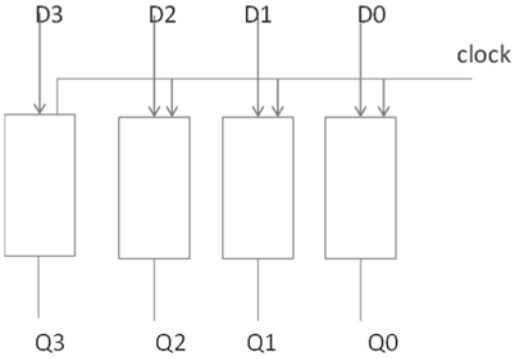
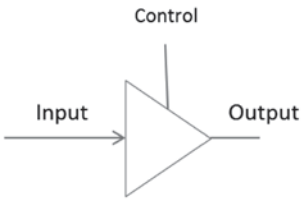


Fig. 2.14 Tri-State device



2.5.1 Components of a Microcomputer

A standard microcomputer consists of a microprocessor (CPU), buses, memory, parallel input/output, serial input/output, programmable I/O interrupt and direct memory access DMA. Figure 2.15 shows components of microcomputer.

Central Processing Unit (CPU) The **central processing unit (CPU)** is the "brain" of the computer and is responsible for accepting data from input devices, processing the data into information, and transferring the information to memory and output devices. The CPU is organized into the following three major sections:

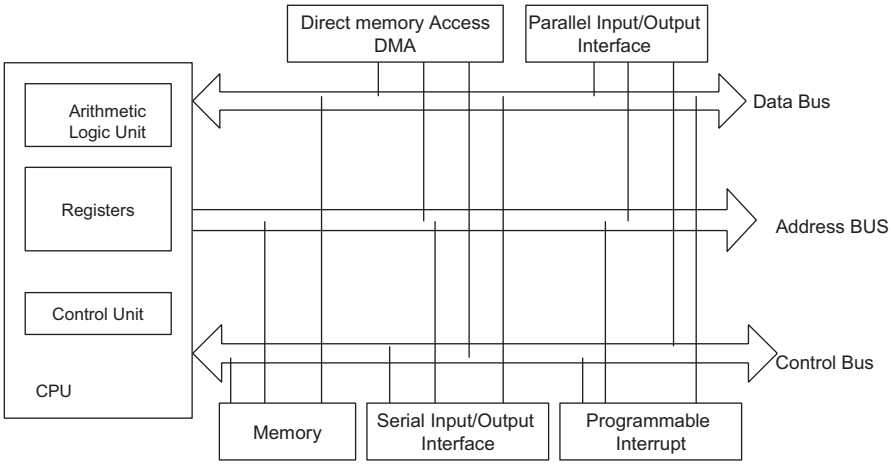


Fig. 2.15 Components of a microcomputer

1. Arithmetic Logic Unit (ALU)
2. Control Unit
3. Registers

A**rithmetic Logic Unit (ALU**: The function of the A**rithmetic Logic Unit (ALU)** is to perform arithmetic operations such as addition, subtraction, division and multiplication, and logic operations such as AND, OR and NOT. Figure 2.16 shows block diagram of ALU

Control Unit The function of the **control unit** is to control input/output devices, generate control signals to the other components of the computer such as read and write signals, and perform instruction execution. Information is moved from memory to the registers; the registers then pass the information to the ALU for logic and arithmetic operations.

It should be noted that the function of the microprocessor and CPU are the same. If the control unit, registers and the ALU are packaged into one integrated circuit (IC), then the unit is called a microprocessor, otherwise the unit is called a CPU. The difference in packaging is shown in Fig. 2.17.

There are two types of technology used to design a CPU: Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC).

CISC Architecture In 1978, Intel developed the 8086 microprocessor chip. The 8086 was designed to process a 16-bit data word; it had no instruction for floating point operations. At the present time, the Pentium processes 32-bit and 64-bit words and it can process floating-point instructions. Intel designed the Pentium processor in such a way that it can execute programs written for earlier 80×86 processors.

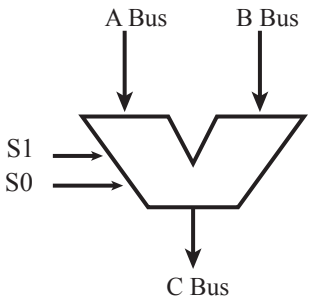


Fig. 2.16 Block diagram of ALU

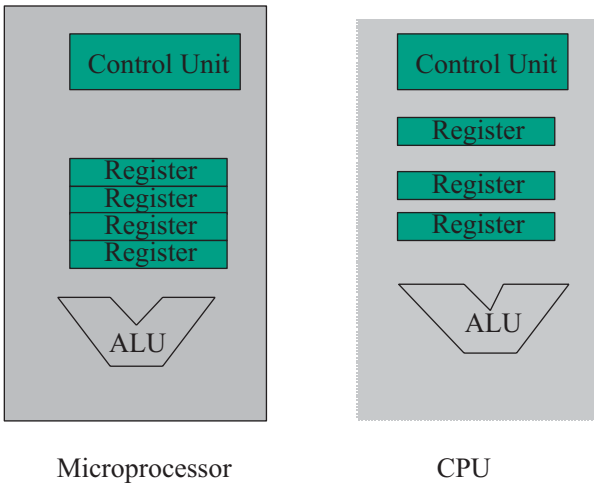


Fig. 2.17 Block diagram of microprocessor and CPU

The characteristics of 80 × 86 are called Complex Instruction Set Computers (CISC), which include instructions for earlier Intel processors. Another CISC processor is VAX 11/780, which can execute programs for the PDP-11 computer. The CISC processor contains many instructions with different addressing modes, for example: the VAX 11/780 has more than 300 instructions with 16 different address modes.

The major characteristics of CISC processor are:

1. A large number of instructions
2. Many addressing modes
3. Variable length of instructions
4. Most instruction can manipulate operands in the memory
5. Control unit is microprogrammed

RISC Architecture Until the mid-1990s, computer manufactures were designing complex CPUs with large sets of instructions. At that time, a number of computer manufacturers decided to design CPUs capable of executing only a very limited set of instructions.

One advantage of reduced-instruction set computer is that they can execute their instructions very fast because the instructions are simple. In addition, the RISC chip requires fewer transistors then the CISC chip. Some of the RISC processors are the PowerPC, MIPS processor, IBM RISC System/6000, ARM and SPARC.
The major characteristics of RISC processors are:

1. All instructions are the same length (they can be easily decoded)
2. Most instructions are executed in one machine clock cycle
3. Control unit is hardwired
4. Few address modes
5. A large number of registers

Computer Bus When more than one wire carries the same type of information, it is called a bus. The most common buses inside a microcomputer are the address bus, the data bus, and the control bus.

Address Bus The address bus defines the number of addressable locations in a memory IC by using the 2^n formula, where n represents the number of address lines. If the address bus is made up of three lines then there are $2^3 = 8$ addressable memory locations, as shown in Fig. 2.18. The size of the address bus directly determines the maximum numbers of memory locations that can be accessed by the CPU.

Data Bus The data bus is used to carry data to and from the memory and represents the size of each location in memory. In Fig. 2.14 each location can hold only four bits. If a memory IC has eight data lines, then each location can hold eight bits. The size of a memory IC is represented by **2n × m** where n is the number of address lines and m is the size of each location. In Fig. 3.3, where n = 3 and m = 4, the size of the memory is:

$$2^3 * 4 = 32 \text{ bits}$$

Control Bus The control bus carries control signals from the control unit to the computer components in order to control the operation of each component. In addition, the control unit receives control signals from computer components. Some of the control signals are as follows:

Read signal The read signal is used to read information from memory or input/output (I/O) devices.

Write signal The write line is used to write data into the memory.

Interrupt Indicates an interrupt request.

Bus request The device is requesting to use the computer bus.

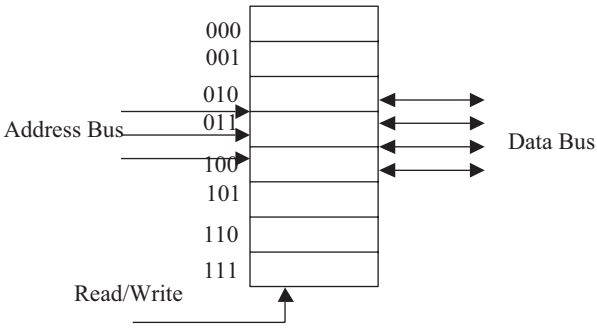


Fig. 2.18 A memory with three address lines and four data lines

Bus Grant Gives permission to the requesting device to use the computer bus.

I/O Read and Write I/O read and write is used to read from or write to I/O devices.

2.5.2 CPU Architecture

There are two types of CPU architecture and they are:

a. **Von Neumann Architecture**

A program is made of code (instructions) and data. Figure 2.19 shows a block diagram of the Von Neumann Architecture. Von Neumann uses the data bus to transfer data and instructions from the memory to the CPU.

b. **Harvard Architecture**

Harvard Architecture uses separate buses for instructions and data as shown in Fig. 2.20. The instruction address bus and instruction bus are used for reading instructions from memory. The address bus and data bus are used for writing and reading data to and from memory.

2.6 Memory

In general, memory can hold information either temporarily or permanently. The following are some types of memory:

- Semiconductor Memory or Memory IC
- Floppy disk and Hard disk
- Tape
- CD ROM (Compact Disk-Read Only Memory)
- Flash ROM

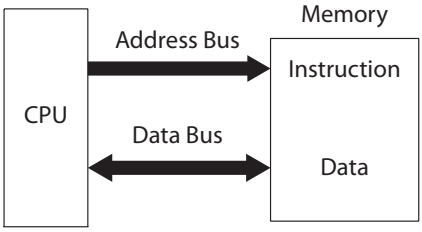


Fig. 2.19 Von Neumann architecture

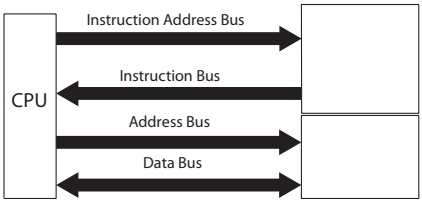


Fig. 2.20 Harvard architecture

Semiconductor Memory There are two types of **semiconductor memory**: Random Access Memory (RAM) and Read only Memory (ROM).

Memory Memory holds instruction and data. Figure 2.21 shows the block diagram of memory unit.

Memory is defined by the number of address lines it has (n) and size of each of its locations (M). The size of a memory is defined by $2^n \times M$.

Memory requires two control signals and they are:

Memory Write CPU writes data into memory by placing an address on the address bus and data on the data bus then activating the memory write signal. The data will then be stored in the specified memory location.

Memory Read CPU places the address on address bus and activates memory read signal. The data stored in memory is then placed on the data bus.

Data can be read from or written into **Random Access Memory (RAM)**. The RAM can hold the data as long as power is supplied to it.

There are many types of RAM, such as **Dynamic RAM (DRAM), Synchronous DRAM (SDRAM), EDO RAM, DDR SDRAM, RDRAM,** and **Static RAM (SRAM)**.

- **Dynamic RAM (DRAM)** is used in main memory. It needs to be refreshed (recharged) about every 1 ms. The CPU cannot read from or write to memory while the DRAM is being refreshed—this makes DRAM the slowest running memory. A DRAM comes in different types of packaging such as the SIMM (Single In-Line Memory Module) and the DIMM (Dual In-Line Memory Module). The SIMM is a small circuit board that holds several chips. It has a 32-bit data bus.

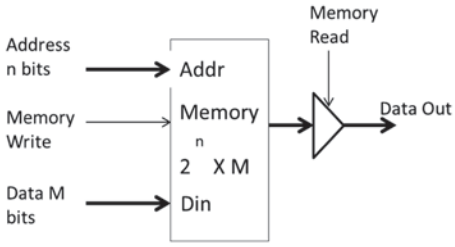


Fig. 2.21 Block diagram of a memory

The DIMM is a circuit board that holds several memory chips. A DIMM has a 64-bit data bus.

- **Synchronous DRAM (SDRAM)** technology uses DRAM and adds a special interface for synchronization. It can run at much higher clock speeds than DRAM. SDRAM uses two independent memory banks. While one bank is recharging, the CPU can read and write to the other bank. Figure 2.22 shows a block diagram of SDRAM.

Extended Data Out RAM (EDORAM) transfers blocks of data to or from memory.

- **Double Data Rate SDRAM (DDR SDRAM)** is a type of SDRAM that transfers data at both the rising edge and the falling edge of the clock.
- **Rambus DRAM (RDRAM)** was developed by Rambus corporation. It uses multiple DRAM banks with a new interface that enables DRAM banks to transfer multiple words and also transfer data at the rising edge and the falling edge of clock. The RDRAM refreshing is done by the interface. The second generation of RDAM is called DRDRAM (Direct RDRAM) and it can transfer data at a rate of 1.6 Gbps. Figure 2.23 shows a RDRAM module.

DRAM Packaging DRAM comes in different types of packaging such as: SIMMs (Single In-Line Memory Module) and DIMM (Dual-in Line Memory Module).

Figure 2.24 shows SIMM, which is a small circuit board that holds several chips. It has a 32 bit data bus.

DIMM is a circuit board that also holds several memory chips, but has a 64 bit data bus.

- **Static RAM (SRAM**) is used in cache memory. SRAM is almost twenty times faster than DRAM and is also much more expensive.
- **ROM (Read Only Memory)**

Like its name suggest, information can be ready only from **Read Only Memory (ROM).** ROM holds information permanently, even while there is no power to the ROM. Two types of ROM are listed below:

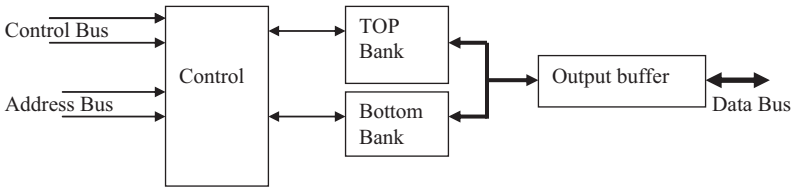


Fig. 2.22 Block diagram of SDRAM

Fig. 2.23 Rambus memory module. (Courtesy Samsung Corp)

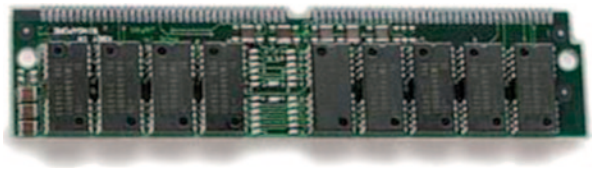


Fig. 2.24 DRAM SIMM

- **Erasable Programmable Read Only Memory (EPROM)**: EPROM can be erased with ultraviolet light and reprogrammed with a device called an EPROM programmer. Flash ROM is a type of EEPROM.
- **Electrically Erasable PROM (EEPROM)**: EEPROM can be erased by applying specific voltage to one of the pins and can be reprogrammed with an EPROM programmer.
- **Flash Memory:** Flash memory is a type of EEPROM that allows multiple memory location to be written or erased one operation but EEPROM only one memory location at a time to be erased or written

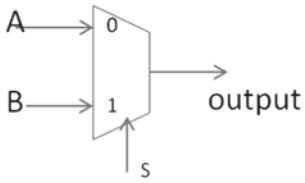
2.7 Multiplexer and Decoder

Multiplexer (MUX) Multiplexer has n inputs and one ouytput, Fig. 2.25 showes a 2*1 MUX, if S = 0 the outpout is A and if S = 1 then output is B.

Figure 2.26 shows 8*1 mux and Table 2.10 shows the function of multiplexer, S2 S1 S0 seclet the input to the MUX.

Decoder The function of decoder is to generate minterms of input at the ouput of decoder.

Fig. 2.25 2*1 MUX



A 2*4 decoder has 2 inputs and 4 outputs, outputs represent minterms of inputs Fig. 2.27 shows a block diagram of 2*4 decoder and Table 2.11 shows truth table of 2*4 decoder.

Fig. 2.26 8*1 MUX

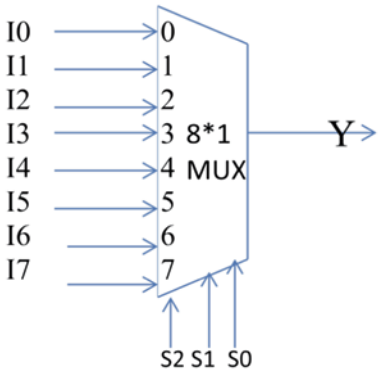


Table 2.10 Operation of 8*1 MUX

S2 S1 S0	Y
000	I0
001	I1
010	I2
011	I3
100	I4
101	I5
110	I6
111	I7

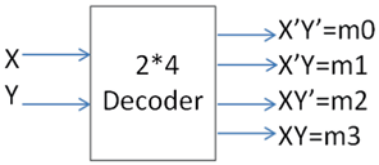


Fig. 2.27 Block diagram of 2*4 Decoder

Table 2.11 shows decoder truth table

X Y	m0	m1	m2	m3
00	1	0	0	0
01	0	1	0	0
10	0	1	0	0
11	0	0	0	1

Short Answer Questions

1. List the components of a microcomputer.
2. Explain the functions of a CPU.
3. List the functions of an ALU.
4. What is the function of a control unit?
5. What does RAM stand for?
6. What is SRAM? discuss its applications
7. Define DRAM and SDRAM and explain their applications.
8. Explain the function of an address bus and a data bus.
9. What does IC stand for?
10. What is the capacity of a memory IC with 10 address lines and 8 data buses?
11. What is ROM?
12. What does EEPROM stand for, and what is its application?
13. What does RDRAM stand for?
14. What is SIMM?
15. Explain the function of cache memory and give its location.
16. What is the application of a parallel port?
17. What is the application of a serial port?
18. Explain the difference between CISC processors and RISC processors

Explain difference between Von Neumann and Harvard Architecture.

Problems

1. If A = 11001011 and B = 10101110 then, what is the value of following operation
   a. a. A AND B
   b. b. A OR B
2. If A = 11001011 and B = 10101110, what is the value of following Operations
   a. A NOT
   b. A XOR B
   c. A AND 0F
   d. A AND F0
3. Draw logic circuit for following functions

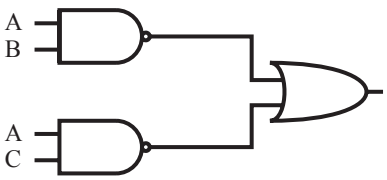
$$\text{A. } F(X,Y,Z) = X'Y' + XZ'$$

$$\text{B. } F(X,Y,Z) = (X + Y)(X + Z)$$

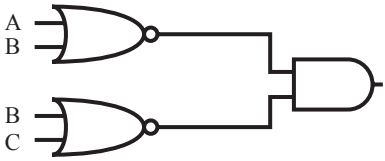
4. Find the truth table for following function

$$F(X,Y,Z) = XY' + YZ' + XZ'$$

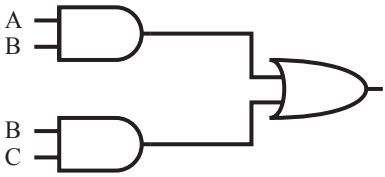
5. If A = 10110110 and B = 0110110011, then find
   a. A. A NAND B
   b. B. A NOR B
   c. C. A XOR B
6. Show output of following logic circuits



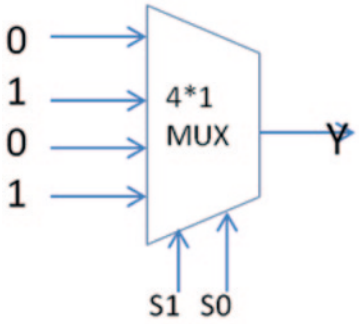
B.



C.



7. Following multiplexer is given show the output



S1	S0	Y
00		
01		
10		
11		

Chapter 3
ARM Instructions Part I

3.1 Introduction

Advanced RISC Machine (ARM) was developed by the Acorn Company. ARM is a leader supplier of microprocessors in the world, ARM develop the core CPU and thousand of suppliers add more functional units to the core. ARM uses two types instruction called Thumb and Thumb-2. Thumb instructions are 16 bits and thumb-2 instructions are 32 bits, currently most ARM processors uses 32 bit instructions.

ARM contains 15 registers called R0 through R15, R0 and R12 called general propose registers. ARM able to execute Thumb instructions (16 bit instructions) and Thumb-2 32 bits instruction, Thumb instructions use on R0 through R7 registers.

ARM is intended for applications that require power efficient processors, such as Telecommunications, Data Communication (protocol converter), Portable Instrument, Portable Computer and Smart Card. ARM is basically a 32-bit RISC processor (32-bit data bus and address bus) with fast interrupt response for use in real time applications. A block diagram of ARM7 processor is shown in Fig. 3.1.

Instruction Decoder and Logic Control: The function of instruction decoder and logic control is to decode instructions and generate control signals to other parts of processor for execution of instructions.

Address Register: To hold a 32-bit address for address bus.

Address Increment: It is used to increment an address by four and place it in address register.

Register Bank: Register bank contains thirty-one 32-bit registers and six status registers.

Barrel Shifter: It is used for fast shift operation.

ALU; 32-bit ALU is used for Arithmetic and Logic Operation.

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_3

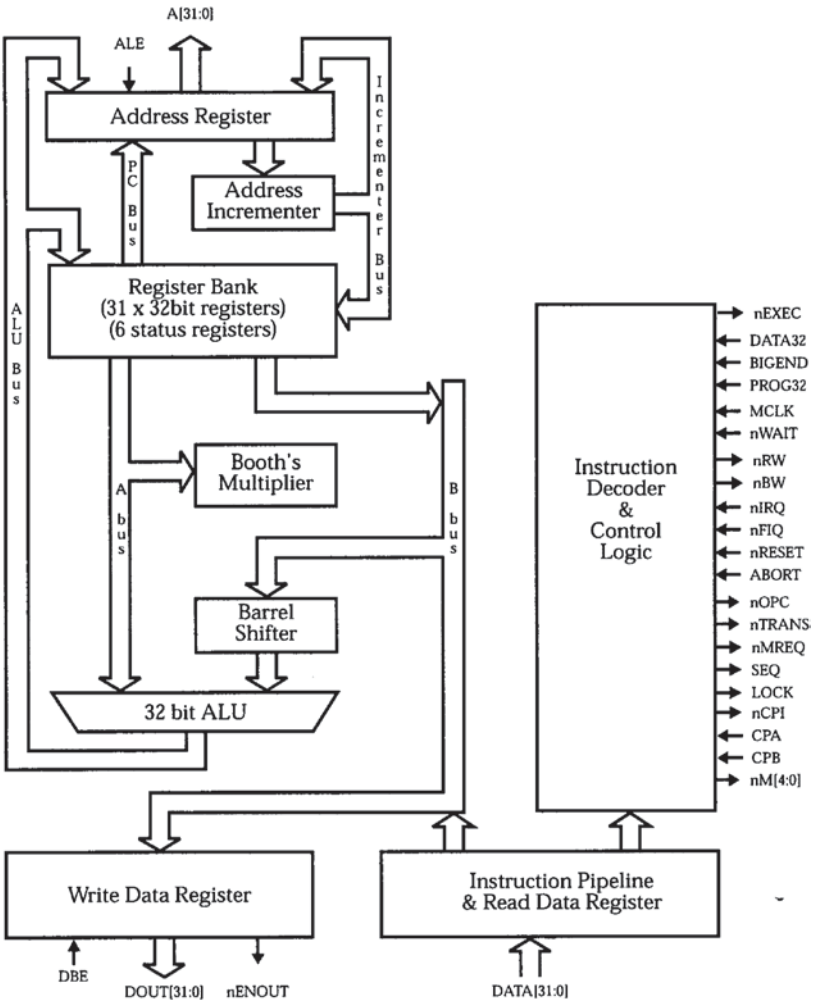


Fig. 3.1 Block diagram of ARM7 architecture

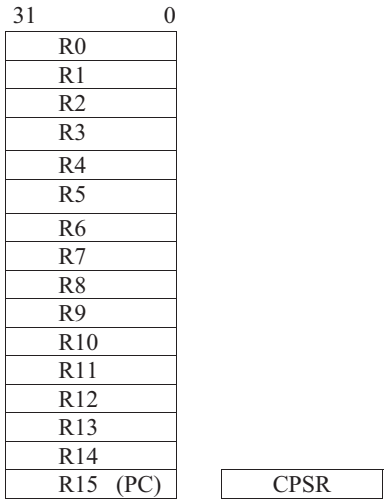
Write Data Register: The processor put the data in Write Data Register for write operation.

Read Data Register: When processor reads from memory it places the result in this register.

ARM Operation Mode: ARM can operates in one of the following mode:

1. *User Mode:* Use for normal operation.
2. *IRQ Mode:* This Interrupt mode is designed for handling interrupt operations.
3. *Supervisory Mode:* Used by operating system.
4. *FIQ Mode:* Fast Interrupt mode.

Fig. 3.2 User mode registers



31	30	29	28	27	7	6	5	4	3	2	1	0
N	**Z**	**C**	**V**	**Unused**	**I**	**F**	**T**	**M4**	**M3**	**M2**	**M1**	**M0**

Fig. 3.3 Storage format for CPSR

5. *Undefined Mode:* When an undefined instruction executed.
6. *Abort Mode:* This mode indicates that current memory access cannot be completed, such as when data is not in memory and processor require more time to access disk and transfer block of data to memory.

ARM Registers: ARM7 has 31 general registers and 6 status registers. At user mode only 16 registers and one Program Status Register (PSR) are available to programmers. The registers are labeled R0 through R15. R15 is used for Program Counter (PC), R14 is used for Link Register and R13 is used for Stack Pointer (SP). Figure 3.2 shows user mode registers.

Current Program Status Register (CPSR): Figure 3.3 shows the format of PSR. This register is used to store control bits and flag bits. The flag bits are N, Z, C and V, and the control bits are I, F, and M0 through M4. The flag bits may be changed during a logical, arithmetic and compare operation.

Flag Bits ***N*** *(negative):* N = 1 means result of an operation is negative and N = 0 means result of an operation is positive.

Z *(zero):* Z = 1 means result of an operation is zero and Z = 0 result of an operation is not zero.

C *(carry):* C = 1 means result of an operation generated a carry, and C = 0 means result of an operation did not produce a carry.

V (overflow): V = 1 means result of an operation generated an overflow and V = 0 means result of an operation did not generate an overflow.

Control Bits ***I (interrupt bit):*** When this bit set to one, it will disable the interrupt and this means the processor does not accept any software interrupt.

F bit is used to disable and enable *fast interrupt request mode* (FIQ) mode.

M4, M3, M2, M1 and M0 are mode bits and they are equal to 10000 for user mode.

T (State bit): T = 1 Processor executing thumb instructions, T = 0 processor executing ARM instructions

3.2 Instruction Set Architecture (ISA)

Manufacturers of CPUs publish a document that contains information about the processor such as list of registers, function of each register, size of data bus, size of address bus and list of instructions that can be executed by the CPU. Each CPU has a known instruction set that a programmer can use to write assembly language programs. Instruction sets are specific to each type of processor. That being said, Pentium processors use a different instruction set than ARM processors. The Instructions are represented in *mnemonic* form means abbreviation, for example, the Addition instruction represented by "ADD" Subtraction instruction represent by "SUB" for example, the addition instruction is represented by

ADD R1, R2, R3; means add contents of R2 with R3 and store results in R1. R1, R2, and R3 are called operands

A. Classification of Instruction base on number of operands
No Operand Instructions: The following are some of the instructions that do not require any operands:

HLT—Halt the CPU

NOP—No operation

PUSH operand: Push operand into top of the stack

POP operand: Remove the operand from top of the stack

One Operand Instructions: The following are some of the instructions that require one operand.

```
INC operand Example: INC R1 – Increment register R1 by 1
DEC operand Example: DEC R1 – Decrement register R1 by 1
J   target  Jump to memory location labeled by target
ADD operand Add operand to the accumulator (ACC) ACC ← ACC + operand
```

Two Operand Instructions: The following are some of the instructions that require two operands:

```
ADD   Rd, Rn              Example: ADD R1, R2-R1 ← R1+R2
Intel Instruction Set Architecture uses two operands.
MOV EAX, EBX  ; EAX ← EBX
```

Three Operand Instructions: Most modern processors use instructions with three operands, such ARM, MIPS and Itanium.

```
ADD R1, R2, R3 ; R1 ← R2 +R3
```

3.3 ARM Instructions

ARM Architecture support Thumb 16 bit and Thumb-2 32 bit instruction set. Most of the ARM instructions use three operands. These instructions are classified based on their instructions format and are listed as followings:

A. Data Processing Instructions
B. Single Data Swap
C. Shift and Rotate Instructions
D. Unconditional Instructions and Conditional Instructions:
E. Stack Operations
F. Branch
G. Multiply Instructions
E: Data Transfer

3.3.1 Data Processing Instructions

The data processing instructions are as follows: AND, EOR, SUB, RSB, ADD, ADC, SBC, RSB TST, TEQ, CMP, CMN, ORR, MOV, BIC and MNW. Data processing instructions use register operands and immediate operand. The general format of Data processing instructions is

Mnemonic {S}{Condition} Rd, Rn, operand2 Mnemonic: Mnemonic is abbreviation of an operation such as ADD for addition

{}: Commands inside the { } is optional such as S and condition

S: When an instruction contains S mean update the Processor Status Register (PSR) flag bits

Condition: Condition define the instruction will executed if meet the condition
Rd: Rd is destination register
Rn: Rn is operand1
Operand2: Operand2 can be register or immediate value

A. Registers Operands: The operands are in registers. First register is destination register, second register is operand1 and third register is operand2.

Following are Arithmetic and Logic operations Instructions with register operands

```
ADD R0, R1, R2    ;R0=R1+R2 Add contents of register R1 with
                  register R2 and place the result in
                  register R0.

ADC R0, R1, R2;   ;R0 = R1+R2 +C Add with carry C is carry
                  bit.

SUB R0, R2, R3    ;R0=R2-R3 where R2 is first operand and R3 is
                  second operand

SBC R0, R2, R3;   ;R0=R2-R3+C-1 SUB with carry.

RSB R0, R2, R5    ;R0= R5-R2 Reverse SUB.

RSC R0, R2, R5    ;R0=R5-R2+C-1 Reverse sub with carry.

AND R0, R3, R5    ;R0= R3 AND R5.

ORR R7, R3, R5;   ;R7=R3 OR R5.

EOR R0, R1, R2    ;R0 = R1 Exclusive OR with R2.

BIC R0, R1, R2    ;Bit clear. The one in second operand clears
                  corresponding bit in first operand and stores the
                  results in destination register.
```

Example 3.1: Assume contents of R1 is 1111111111011111 and R2 is 1000 0100 1110 0011 after execution of **BIC R0,R1, R2** the R0 contains 0111 101100011100

B. Immediate Operand: In immediate operand, operand2 is an immediate value and maximum can be 12 bits

```
ADD R1, R2, #0x25    ;R1=R2+&25, # means immediate and & means
                     the immediate value is in hexadecimal.

AND R2, R3, #0x45    ;R2 = R3 AND &45.

EOR R2, R3, #0x45    ;R2 = R3 Exclusive OR &45.
```

Example 3.2: What is contents of R1 after executing following instruction, assume R2 contains 0x12345678

```
ADD R1, R2, #0x345
```

The ADD instruction will add contains of R2 with 0x2345 and store the result in R1, then R1 = 0x123459BD

Setting Flag Bits of PSR: The above instructions do not affect the flag bit of PSR because the instructions do not have option S. By adding suffix S to the instruction, the instruction would affect the flag bit.

```
ADDS R1, R2, R3      ;The suffix S means set appropriate flag
                     bit.

SUBS R1, R2, R2;     ;The  will set zero flag to 1.
```

Compare and Test Instructions: ARM processor uses the compare and test instructions to set flag bits of PSR and following are Compare and Test instructions

CMP, CMN, TST, and TEQ: These instruction uses two operands for compare and test, the result of their operations do not write to any register

CMP Instruction (Compare Instruction): The CMP instruction has following format

CMP Operand 1, Operand2: The CMP instruction compares Operand1 with Operand2, this instruction subtract Operand2 from Operand 1 and sets the appropriate flag. The flag bit set based on the result of the operation as follow

Z flag set if Operand2 equal operand 1
N flag is set if operand1 less than operand2
C flag is set if result of operation generate carry

Example 3.4: Assume R1 contains 0x00000024 and R2 contains 0x00000078, the operation CMP R1, R2 will set N flag to 1

CMP Rd, immediate value, the immediate value can be 8 bits such as
CMP R1, #0xFF
CMN Compare Negate: The CMN has following format

CMN Operand1, Operand2: The instruction will add operand1 with operand 2 and set appropriate flag bit

Example 3.5: Assume R1 contains 0x00000024 and R2 contains 0x13458978, the operation CMN R1, R2 with result carry and set C flag to 1.

TST (Test Instruction): The test instruction has following format

TST Oprand1, Operand2: The Test Instruction performs AND operation between operand1 and Operand2 and set appropriate flag bit. The operand to can be immediate value or Register such as

```
TST R1, R2          ;This instruction performs R1 AND R2
                    operation and sets the appropriate flag.
```

OR

```
TST  R1, immediate, the immediate value can be 8 bits such as

TST R1, 0xFF
```

```
TEQ R1, R2          ;This instruction performs R1 Exclusive OR
                    R2.
```

If R1 equal to R2 then Z flag set to one

3.4 Register Swap Instructions (MOV and MVN)

The register swap instructions has following general formats

```
A. MOV{S} {condition} Rd, Rm; Move the contents of Rm to Rd
```

Example 3.6: What is contents of R1 after Execution of following instruction

Assume R2 contains 0X0000FFFF

```
a. MOV R1, R2;R1 ← R2
   R2=0x0000FFFF

b. MVN R1, R2; R1 ← NOT R2
 R2= 0xFFFF0000
```

MOV{S}{condition} Rd, immediate value

Immediate value is 16 bits, The range of immediate value if from 0x00000000 to 0x0000FFFF

Example 3.7: `MOV R2, # 0x45, the contents of R2 will be 0x00000045`

```
MOV Rn, Rm, lsl # n      ; shift Rm n  times to the left
and store the result  Rn
```

Conditional MOV

```
MOVEQ R2, 0x56        ; if zero bit is setthen executes MOVEQ
```

3.5 Shift and Rotate Instructions

ARM combined the Rotate and Shift operation with other Instructions, the ARM processor performs following shift operations

```
LSL        Logical Shift Left
LSR        Logical Shift Right
ASR        Arithmetic Shift Right
ROR        Rotate Right
```

Logical Shift Left (LSL): In logical shift left operations each bit of register shifted to the left as shown in Fig. 3.4 and a zero will placed in the least significant bit, the logical shift left multiply the contents of register by 2.

```
LSL R1, R1, n;shift to left R1 n times and store result in R1
```

Fig. 3.4 Logical shift left

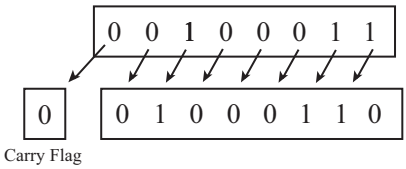
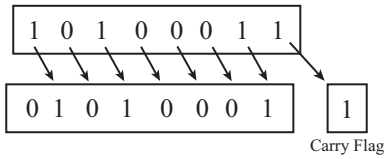


Fig. 3.5 Logical Shift Left



Example 3.8: What is contents of R1 after executing following Instruction, assume R1 contains 0x00000500.

LSL R1, R1, 8

R1= 0x00050000

Logical Shift Right (LSR): In logical shift right operations each bit of register shifted to the right as shown in Fig. 3.5 and a zero will placed in the most significant bit, the logical right divides the contents of register by 2.

```
LSR R1, R1, n ;shift to right R1 n times and store result in R1
```

Example 3.9 What is contents of R1 after executing following Instruction, assume R1 contains 0x00000500.

```
LSR R1, R1, 4

R1= 0x00000050
```

Arithmetic Shift Right (ASR): In Arithmetic shift right the most significant bit does not change and each bit shifted to the right as shown in Fig. 3.6.

Rotate Right: Figure 3.7 shows an eigth bit rgister and Fig. 3.7 shows the regiter after rotating one times

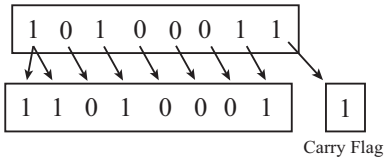


Fig. 3.6 Arithmetic shift right

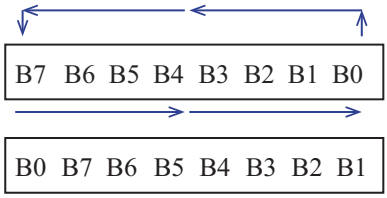


Fig. 3.7 Rotate right operation

Example 3.10 What is content of R1 after rotating 16 times, assum R1 contains 0x0000FFFF

```
ROR  R1, R1 , #16
R1=  0xFFFF0000
```

ARM combines data processing instructions and shift operation, shift operation is applied to the second operand of the instruction.

Example 3.11: Register R2 containes 0xEEEEFFFF, by executing

```
MOV R1, R2, ROR # 16  ;the R2 rotate 16 times and store results
in R1
```

by rotating 16 times the contains of R1 will be 0xFFFFEEE

```
ADD R1, R2, R3, LSL #4   ;R1= R2 + R3 x 2^4, R3 is shifted 4
                         times to the left and result is added to R3
                         and placed in R1.
```

Also a register can hold number of times the operand2 must be shifted.

```
ADD R1, R2, R3, LSL R4    ;R1= R2 + R3 X 2^R4, Number of times R3
                          to be shifted is in R4.

MOV R0, R1, LSL #3        ;Shift R1 to the left three times and move
                          the result to R0.
```

3.6 ARM Unconditional Instructions and Conditional Instructions

Figure 3.8 shows the general format of an ARM instruction. ARM instruction defines two types of instructions, namely:

1. Unconditional Instruction
2. Conditional Instruction

Condition code defines the type of conditions. If this field is set to 1110 then the instruction is an unconditional instruction, otherwise the instruction is a conditional instruction. To use an instruction as a conditional instruction, the condition will suffix to the instruction. The suffixes are:

Condition Code	Condition	
0000	EQ	Equal
0001	NE	Not equal
0010	CS	Carry set
0111	CC	Carry is clear
0100	MI	Negative (N flag is set)
0101	PL	Positive (N flag is zero)
0110	VS	Overflow set
0111	VC	Overflow is clear
1000	HI	Higher for unsigned number
1001	LS	Less than for unsigned number
1010	GT	Greater for signed number
1011	LT	Signed less than
1100	GT	Greater Than
1101	LE	Less than or equal
1110	AL	Unconditional instructions
1111	Unused code	

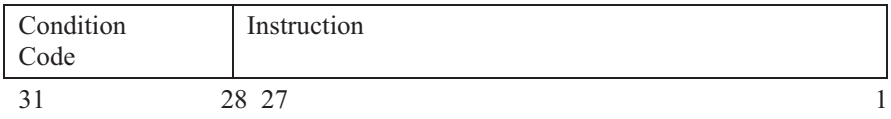


Fig. 3.8 General format of an ARM instruction

Processor checks condition flag before executing the conditional instruction. If it matches with the condition instruction then processor executes the instruction, otherwise skips the instruction.

```
ADDEQ R1, R2, R3     ;If zero flag is set and it will execute
                     this instruction.
```

Example 3.10: Convert the following HLL to ARM assembly language.

```
If R1=R2 then
ADD  R3, R4, R5
Endif
```

ARM assembly language for the above program would be:

```
CMP  R1, R2
ADDEQ  R3, R4, R5
```

Example 3.11: Convert the following HLL to ARM assembly language.

```
If R1 = R2 Then   R3= R4-R5
Else
If R1>R2   Then   R3=R4+R5
```

ARM assembly language for the above program would be:

```
CMP R1, R2
SUBEQ  R3, R4, R5
ADDGT  R3, R4, R5
```

3.7 ARM Data Processing Instruction Format

Figure 3.9 shows data processing instruction format.

Condition Code: To determine if the instruction is a conditional or a unconditional instruction

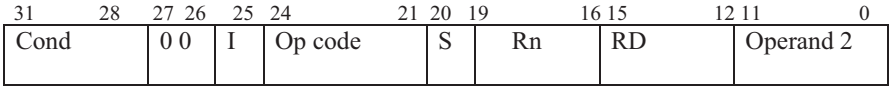


Fig. 3.9 Data processing instruction format

Fig. 3.10 Operand2's format when bit 4 is equal to 0

I bit I = 0 means the operand2 is a register, I = 1 means the operand 2 is an immediate value.

Op Code: To determine types of operation and they are as followings:

Instruction	Op Code
AND	0000
EOR	0001
SUB	0010
RSB	0011
ADD	0100
ADC	0101
SBC	0110
RSC	0111
TST	1000
TEQ	1001
CMP	1010
CMN	1011
ORR	1100
MOV	1101
BIC	1110
MVN	1111

S bit: S = 0 do not change flag bits of PSR register, S = 1 set condition flags of PSR register

Rn: Rn is first operand and it can be any of 16 registers, R0 through R15

Rd: Rd is destination register and it can be any of 16 registers, R0 through R15

Operand2: When I = 0 the operand2 is a register and Fig. 3.10 shows operand2's format.

 # Shift: To determine immediate value for number of times Rm must be shifted

SH: To determine types of shift operation

Rm: Second operand

Operation	SH value
LSL	00 Logical Shift Left
LSR	01 Logical Shift Right
ASR	10 Arithmetic Shift Right
ROR	11 Rotate Right

When bit 4 of operand2 is set to 1, the number of times Rm must be shifted is in a register.

Figure 3.11 shows format of operand2 of Fig. 3.9.

I=1 The operand 2 would have following format.

11	0
Immediate value	

Example 3.12: Convert the following instruction to machine code.

```
ADD R1, R2, R3, LSL #3
```

31 28	27 26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
Cond	0 0	I	Op code	S	Rn	RD	#Shift	SH		RM
1110		0	0100	0	0010	0001	0011	00	0	0011

Fig. 3.11 Format of Operand2 when bit 4 is equal to 1

11 8	7	6 5	4	3 0
RS	0	SH	1	Rm

3.8 Stack Operation and Instructions

Part of memory is used for temporary storage is called stack, the stack pointer holds the address of top of the stack as shown in Fig. 3.12

The register R13 assigned as Stack Pointer (SP), the stack uses following instructions

a. **Push {condition} Rn**: transfer the contains of Rn into stack and add 4 to the stack pointer

Example 3.12 Assume contains of R3 is 0x01234567, Fig. 3.13 show the contents of Stack after executing push R3.

Example 3.14 Shows contents of stack and SP in Fig. 3.13 after execution of Push R4, assume R4 contains 0x5645321F.

POP Instruction: The POP instruction has following format.

POP{condition} Rn

POP Rn: The pop instruction remove the word from top the stack and store it into register rn and automatically decrement stach pointer by 4.

Example 3.15: **S**how the contents of stack and SP of Fig. 3.14 after execution POP R0, the contents of will be R0=0x1FAD7856 and stack will look like as Fig. 3.15.

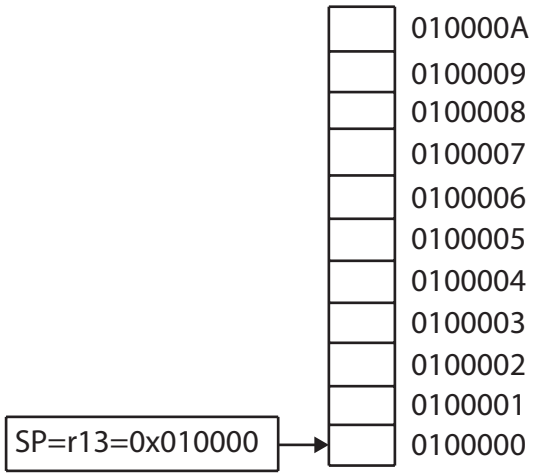


Fig. 3.12 Stack architecture

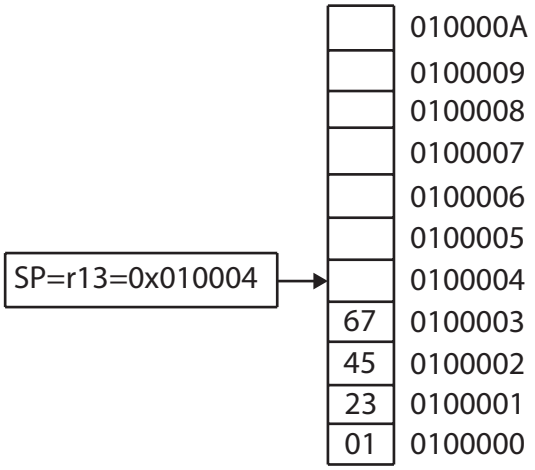


Fig. 3.13 Shows contents of stack after execution of Push R3

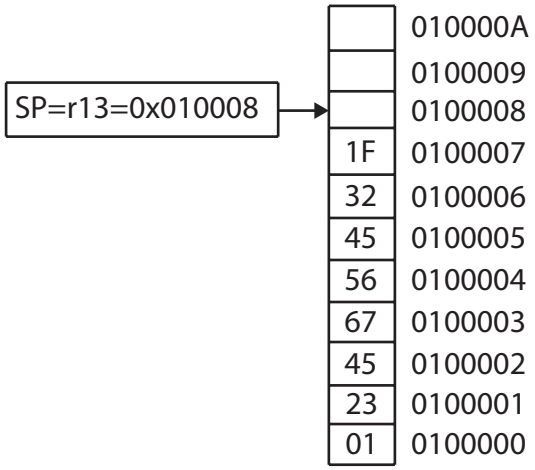


Fig. 3.14 Shows Stack after push operations

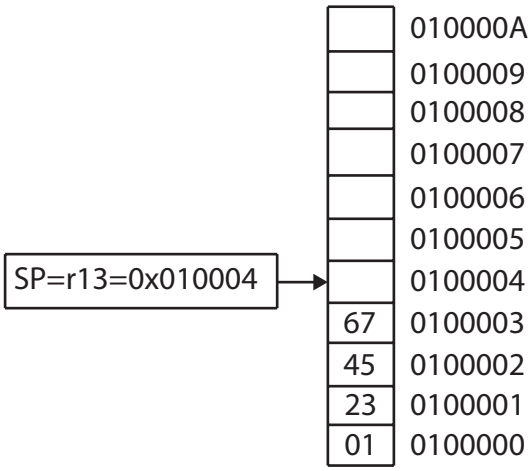


Fig. 3.15 Contains of stack after POP operation

3.9 Branch (B) and Branch with Link Instruction (BL)

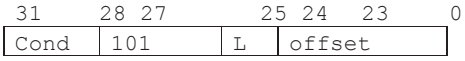
The Branch instruction has following general format
B{condition} label

```
B     label          ; branch to location label.

BEQ    label                          ; if flag bit Z=1 then
                    execute this instruction

BL Subroutine       ;it will branch to subroutine and save
                    contents of PC (R15) to R14 (link
                    register) for return from subroutine.
```

Example 3.16: Write a sub-routine to find the value of Y = 16X + 4, assume R1 holds the Y and R2 holds X.

```
BL       Funct


Funct      SUB R1, R1, R1
           ADD  R1, R1, R2, LSL4
           ADD R1, R1!, #04
           MOV R15, R14            ; Move return  address to PC
```

B and BL Instruction Format:



L = 0 means Branch and condition for branch can be set by Cond field.
L = 1 Mean Branch and Link

Instruction	
B	Branch always
BAL	Branch Always
BEQ	Branch if Equal
BNE	Branch if Not equal
BPL	Branch on positive
BMI	Branch on negative
BCC	Branch if carry flag is clear
BLO	Branch below for unsigned number
BCS	Branch carry flag is set
BHS	Branch if higher for unsigned number
BVC	Branch if Over flow flag is clear
BVS	Branch if Over flow flag is clear
BGT	Branch greater for signed number
BGE	Branch greater or equal for signed number
BLT	Branch Less than for signed number
BLE	Branch Less than for signed number
BLS	Branch less than or equal for unsigned number

Example 3.17: Rewrite following assembly language using conditional instructions.

```
CMP R1,R2
BEQ  Exit
ADD R1, R2, R3
Exit:
SUB R1, R5, R6
```

By using conditional instructions the above assembly language can be represented by

```
CMP R1,R2
SUBEQ    R1,R5,R6
ADDNE    R1,R2,R3
```

3.10 Multiply (MUL) and Multiply-Accumulate (MLA) Instructions

```
MUL instruction
MUL Rd, Rm,Rs           ;Rd= Rm*Rs

MLA  Multiply and Accumulate
MLA  Rd,Rm,Rs  , Rn       ; Rd= Rm*Rs +Rn
```

Multiply Instruction Format:

31–28	27–22	21	20	19–16	15–12	11–8	7–4	3–0
Cond	00000	A	S	Rd	Rn	RS	1001	Rm

A=0 MUL instruction
A=1 MLA instruction
S=0 Do not change flag bit
S=1 Set the flag bits
Rd is destination register
Rs, Rm and Rn are the operands

Problems

1. What is contents of R5 after execution of following instruction, assume R2 contains 0X34560701 and R3 contains 0X56745670

```
a. ADD R5, R2 , R3

b. AND R5, R 3, R2

c. XOR R5, R2,R3

d. ADD R5, R3, #0x45
```

2. What is contents of R1? assume R2 = 0x00001234

```
a. MOV  R1, R2, LSL #4

b. MOV  R1, R2,  LSR #4
```

3. What is difference between these two instructions?

```
a. SUBS   R1, R2, R2
b. SUB    R1,R2, R2
```

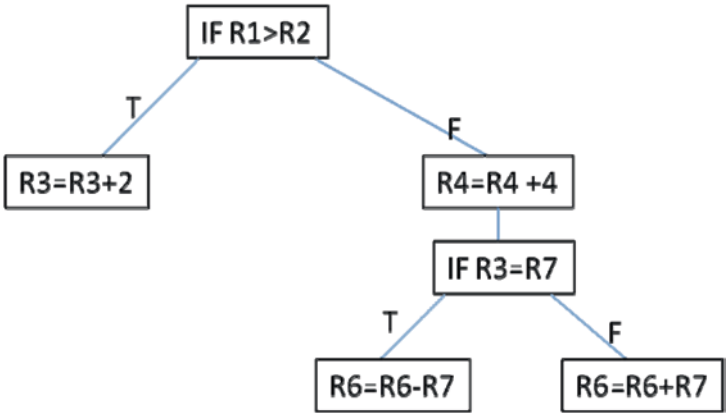
4. Convert following HLL language to ARM instructions

```
IF R1>R2 AND R3>R4 then

    R1= R1 +1
Else
R3=R3 +R3*8
Endif

if ( R1 > R2 && R3 > R4)
       R1 = R1+1;
else
       R3 = R3 + (R3 *8);
```

5. Convert following HLL language to ARM instructions

```
IF R1>R2 OR R3>R4 then
R1= R1 +1
Else
R3=R3 +R5*8
Endif
```

6. Convert following flowchart to ARM assembly language



7. Write a program to add ten numbers from 0 to 10 or Convert following C language to ARM assembly Language

```
int    sum;
          int i;
sum = 0;
for   (i  =  10 ;  i  > 0 ;   i  - - ){
sum  =  sum  +1
}
```

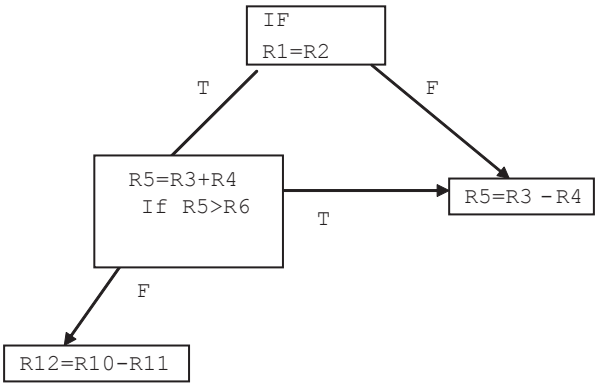
8. Write a program to convert following HLL to ARM assembly

```
a=  10;
b=45;
while ( a! =b ) {
       if ( a <  b)
              a   =   a  +5;
       else
              b= b+5;
 }
```

9. Convert following HLL to ARM assembly

```
IF R1>R2 AND R3>R4 then
     R1= R1 +1
Else
R3=R3 +R5*8
Endif
```

10. Convert Following Flow Chart to ARM Assembly



Chapter 4
ARM Instructions and Part II

4.1 ARM Data Transfer Instructions

The data transfer instructions are used to transfer data from memory to registers and from registers to memory and they are Load (LDR) and Store (STR) instructions.

4.1.1 Load Instructions (LDR)

The LDR instruction is used to load data to a register from memory and it has following general format.

LDR[type]{condition} Rd, Address Where "**type**" define following load instructions

LDR load 32 bits (word)

LDRB load 1 byte

LDRH Load 16 bits (Half word)

LDRS load signed byte

LDRSB Load sign extension

LDRSH Load Signed half word

LDM Load multiple words

Condition is an optional such as LDREQ load data if Z flag = 1 and Rd is destination register

Example 4.1 Assume R0 hold address 0000 and following memory is given, show the contains of R1 and R3 after executing following instructions (ARM Little Indian)

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_4

Address	Contents
0000	0x85
0001	0xF2
0002	0x86
0003	0xB6

```
LDRH R1, [R0]; R1=0x0000F285
LDRSH R3, [R0]; R3=0xFFFFF285
```

4.1.2 ARM Pseudo Instructions

ARM support multiple pseudo instructions, the pseudo instruction is used by the programmer and assembler convert the pseudo instruction to ARM instruction

ADR Pseudo Instruction ADR is used to load the address of memory location into a register and has following format

```
ADR Rd, Address
```

Example 4.2 The following instructions will read the address of data and then load the data into register R3

```
ADR R0, Table; Move address represented by Table
LDR R3, [R0]; R3=0x23456780
```

Address	Data
Table	0x23456780

LDR Pseudo Instruction LDR Pseudo instruction is use for loading a constant into a register. In order to move a 32 bits contestant into a register, The instruction MOV Rd, #value only can move 16 bits to the register Rd, The LDR Pseudo instruction has following format

```
LDR Rd,=Value
```

Example 4.3 The following instruction will load the R1 with 0x23456789

```
LDR R1,=0x23456789
```

4.1.3 Store Instructions (STR)

The STR instruction is used to transfer contents of a register to memory and have following general format

STR[type]{condition} Rd, [address] Where "**type**" define following instruction types

STR	Store 32 bits (word)
STRB	Store 1 byte
STRH	Store 16 bits (Half word)
STM	Store multiple words

Example 4.4 STR R5, [R3]
; Store contents of R5 in into the memory location that R3 holds the address, R3 is the base register.

4.2 ARM Addressing Mode

The ARM processor support indirect, pre-index and post-index addressing for loading data from memory to the registers and storing data the memory.

4.2.1 Register Indirect Addressing

In Register Indirect Addressing the register inside the brackets holds the address of data such as

LDR R0, [Rn]

$$\text{Effective Address (EA)} = \text{contents of Rn}$$

4.2.2 Pre-Index Addressing

The pre-index addressing uses following two format

A. LDR R0, [Rn, #Offset]

Where Rn is Base Register and the effective address (EA) is calculated by

$$EA = Rn + Offset$$

The offset can be immediate value or register or register with shift operation

A1. Pre-Index Addressing with Immediate Offset

Example 4.5 What is the effective address of following address assume R5 contains 0x00002345

[R5, #0x25]

$$EA = 0X000002345 + 0X25 = 0X0000236A$$

A2. Pre-index Addressing with Register Offset

Example 4.6 What is effective address of following Pre-index addressing, assume R5 = 0x00001542 and R2 = 0x00001000

[R5, R2]

$$EA = R5 + R2 = 0X00001542 + oX00001000 = 0X00002542$$

A3. Pre-Index Addressing with Register Shift operation

Example 4.7 What is EA of following instruction
LDR R0, [Rn, R2, LSL#2]

$$EA = Rn + R2 * 4$$

R2 shifted to the left twice (multiply by 4) and added to Rn

4.2.3 Pre-Index Addressing with Auto Index

The general format for Pre-index addressing with Auto-indexing is

[Rn, Offset]!

The Exclamation (!) character is used for auto-indexing; the offset can be immediate value or register or shifted register

A1. Offset is an Immediate Value

LDR R0, [R1,# -4]!
EA = R1 − 4 and R1 updated by R1 = R1 − 4.

Example 4.8 What is effective address and final value of R5 for following Instruction, assume the contents of R5 = 0x00002456.
LDR R0, [R5, #0x4]!

$$EA = R5 + 0x4 = 0x000245A$$

$$R5 = R5 + 0X4 = = 0x000245A$$

A2. Offset is a Register

LDR R0, [R1, R2]!

Example 4.9 What is effective address and final value of R5 of following Instruction, assume the contents of R5 = 0x00002456 and R2 = 0x00002222
LDR R0, [R5, R2]!

$$EA = R5 + R2 = 0x00004678$$

$$R5 = R5 + R2 = 0x00004678$$

4.2.4 Post -Index Addressing

The general format of Post-index addressing is

LDR R0, [Rn], Offset

Offset can be immediate value or register or shifted register

A. Offset is an Immediate Value

LDR R0, [Rn], #4

B. Offset is a Register

LDR R0, [Rn], Rm

$$\text{Effective address} = Rn \quad \text{and} \quad Rn = Rn + Rm$$

C. Offset is a shifted register

LDR R0, [Rn], Rm, SHL #4

$$\text{Effective address} = Rn \quad \text{and} \quad Rn = Rn + Rm * 16$$

4.3 Data Transfer Instruction Format

Figure 4.1 shows format of Data Transfer Instruction

Rd: Destination Register

Rn: Base Register

L = 0 Store to memory, L = 1 Load from Memory

W = 0 no write back (keep Base address the same value), W = 1 modify base address write back (auto indexing)

B = 0 transfer word, B = 1 transfer a byte

Up/Down bit; U = 0 subtract offset from base register, U = 1 add offset to the base register

P = 0 Post, add offset after transfer, P = 1 Pre, add offset before transfer

I = 0 offset is an immediate value

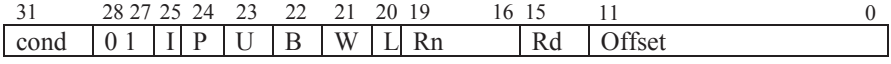
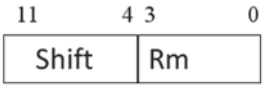


Fig. 4.1 Data transfer format

I = 1 Offset is a register and offset has following format



Where shift field determine number of time RM must be shifted

4.4 Block Transfer Instruction and Instruction Format

Block transfer instruction is used to load from memory to the registers or store contents of registers to memory (Fig. 4.2).

```
LDMIA R1, {R0,R2,R3};Load data from memory to registers R0,R2 and R3
                    ,R0=memory[R1], R2=memory[R1+4], and R3=memory
                    [R1+8].
STMIA  R0, {R2,R3}  ;Store R2 and R3 starting at memory location addressed by
                    R0.
```

4.5 Swap Memory and Register (SWAP)

The swap instruction combines the load and store instructions into one instruction and it has following format

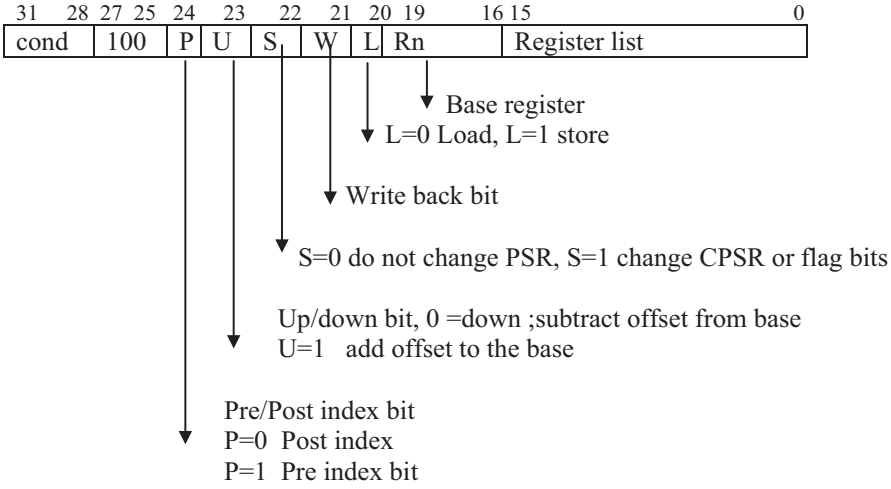


Fig. 4.2 Format of Block Transfer Instruction

31 28	27 23	22	21 20	19 16	15 12	11 4	3 0
Cond	00010	B	00	Rn	Rd	00001001	Rm

Fig. 4.3 SWP instruction format

SWP Rd, Rm, [Rn] The register Rd is destination register, Rm is the source register and Rn is base register.

The Swap instruction perform following functions (Fig. 4.3).

Rd ⟵ memory [Rn]; Load Rd from memory location [Rn]

[Rn] ⟵ Rm ;store the contents of Rm in memory location [Rd]

SWPB Rd ,Rm, [Rn] ;Swap one byte

4.6 Bits Field Instructions

ARM offers two bit field instructions and they are Bit Field Clear (BFC) and Bit Field Insertion (BFI).

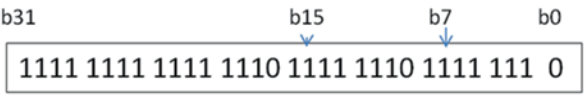
A. BFC (Bit Field Clear Instruction): BFC has following general format **BFC {cond} Rd, # lsb, #width**

Rd is destination register
lsb determine start of bit position in the source register (Rd) to be clear
Width determine number of bits to be clear from lsb to msb of the Rd register

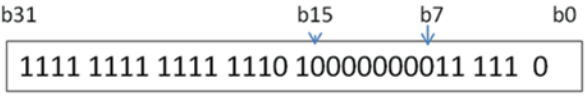
Example 4.10 Write an instruction to clear bits 7 through 15 of register R4, assumeR4 contains 0x FFFEFEFE

BFC R4, #7, #8 clear bit 7 through bit 15 (8 bits) of register R4

The initial value in R4 is



After clearing bit 7 through 15 of R4 results



B. BFI (Bit Insertion Instruction)

Bit insertion is used copy a set of bit from one register Rn into register Rd starting from lsb of Rd, BFI has following format

BFI{***cond***} ***Rd***, ***Rn***, #***lsb***, #***width***

Rd	is destination Reg
Rn	Source register
#lsb	starting bit from Rn
#width	number of bit starting from lsb of Rn

Example 4.11 Copy 8 bits of R3 starting from bit 4 to R4, assume R3 contains 0x FFFFEBCD and R4 contains 0xEE035007

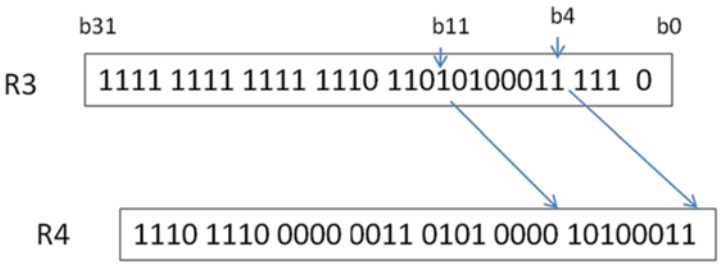
BFI R4, R3, #4, #8, this instruction will copy 8 bits from B4 to B11 of R3 into B0 through B7 of R4, The initial value of R3 in binary

11101110000000 11010100000000111

The initial value of R4 in binary is

1110 1110 0000 0011 0101 0000 0000 0111

The instruction will copy 8 bits from bit 4 of R3 into R4 starting from bit 0 of R4



4.7 Data Representation and Memory

ARM processors define a word as 4 bytes and a half word as 2 bytes. Data can be represented in the form of hexadecimal, decimal and binary.

a. Decimal numbers, such as 345
b. Hexadecimal numbers, such as 0x2345, where 'x' represents hexadecimal
c. Binary or base 2, such as 2_10111100

Memory holds data and code. Figure 4.4 shows a block diagram of memory. The address of memory defines the location of the data, where each location of ARM processor memory holds one byte. In assembly language a label, as shown in Fig. 4.5, represents the address of memory.

Figure 4.4 shows how each memory location holds one byte. Storing two bytes (half word) of data, such as 0x4563, can be stored two different ways called Big Endian and Little Endian.

Big Endian: In Big Endian the most significant byte (MSB) of data is stored first in memory.

The ARM 7 operates in Big and Little Endian; each memory location of ARM7 holds one byte and a word (4 bytes) can be store in memory in two different ways: Big Endian and Little Endian.

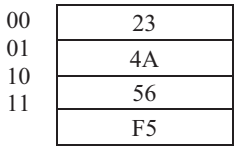


Fig. 4.4 Byte addressable memory

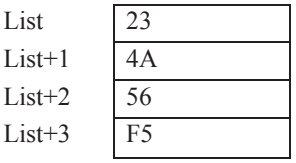


Fig. 4.5 Byte addressable memory using a label

Example 4.12: The 0x34569312 may be stored in Big Endian form as shown in Fig. 4.6.

Little Endian: In little Endian the least significant byte of a word is stored at the lowest address.

Example 4.13: Hex number 34569312 may be stored in Little Endian form as shown in Fig. 4.7.

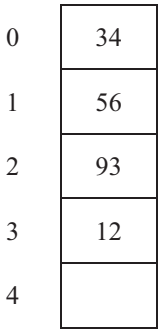


Fig. 4.6 Big Endian representation of hex number 34569312

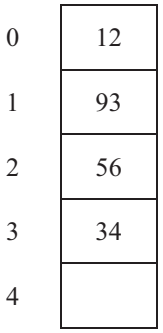


Fig. 4.7 Little Endian representation of hex number 34569312

Problem

1. Trace following instructions, assume list start at memory location 0x0000018 and using ARM Big Indian

```
ADR  R0, LIST ; Load R0 with address of memory location List
MOV  R10, #0x2

a.LDR R1, [R0]
b.LDR R2, [R0, #4]!
c.LDRB  R3, [R0] , #1
d.LDRB  R4, [R0 , R10]!
e.LDRSB R5, [R0],  #1
f.LDRSH R6, [R0]



LIST DCB      0x34, 0xF5, 0x32, 0xE5, 0x01, 0x02,0x8,0xFE
```

2. Work problem #1 part A and B using Litle Endian

3. What is contents of register R7 after execution following program

```
ADR R0, LIST

LDRSB R7, [R0]

LIST     DC   0xF5
```

4. What is contents of register Ri for following load Instructions, assume R0 hold the address of list using little Endian

```
a.LDR R1, [R0]
b.LDRH R2, [R0]
c.LDRB R3, [R0] , #1
d.LDRB R4, [R0]
e.LDRSB R5, [R0],  #1
f.LDRSH R6, [R0]


List DCB    0x34,0xF5, 0x32, 0xE5, 0x01, 0x02
```

5. Following memory is given, show the contents of each register, assume R1 = 0x0001000 and R2 = 0x00000004 (use Little Endian)

a. LDR R0, (R1)
b. LDR R0 , (R1, #4)
c. LDR R0 , (R1, R2)

d. LDR R0 , (R1, #4)!

1000	23
	13
	56
	00
1004	45
	11
	21
	88
1008	03
	08
	35
	89
100C	44
	93

6. What is effective address and contains of R5 after executing following instructions? assume R5 contains 0x18 and r6 contains 0x00000020

```
A. STR R4, [R5]
B. STR R4, [R5, #4]
C. STR R4, [R5, #8]
D. STR R4, [R5, R6]
E. STR R4, [R5], #4
```

Chapter 5
ARM Assembly Language Programming Using Keil Development Tools Introduction

5.1 Introduction

Manufacturers of CPUs publish a document that contains information about the processor such as list of registers, function of each register, size of data bus, size of address bus and list of instructions that can be executed by the CPU. Each CPU has a known instruction set that a programmer can use to write assembly language programs. Instruction sets are specific to each type of processor. That being said, Pentium processors use a different instruction set than ARM processors. Using Instructions of processor to write program is call assembly language and function of **Assembler** is to convert assembly language to machine code (binary) that the CPU.

Cross Assembler: The assembler which runs on a different CPU is called a cross assembler.

Development Tool: The development tool is a processor simulator that runs on a workstation using a Windows or Linux operating system and enables the programmer to write and test programs, then download the program to the processor target. The following development tools support ARM processors:

1. Keil Development: The programmer can download ARM Assembler from http://www.keil.com/download/list/arm.htm
2. IAR System Development tool: the evaluation tool is available for 30 days http//www.iar.com/website1/1.0.1.0/675/1
3. GNU ARM Assembler from http://www.gnu.org

5.2 Keil Development Tools for ARM Assembly

Download the Keil development tools from http://www.keil.com/demo/eval/armv4.htm, you need to register in order to download.

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_5

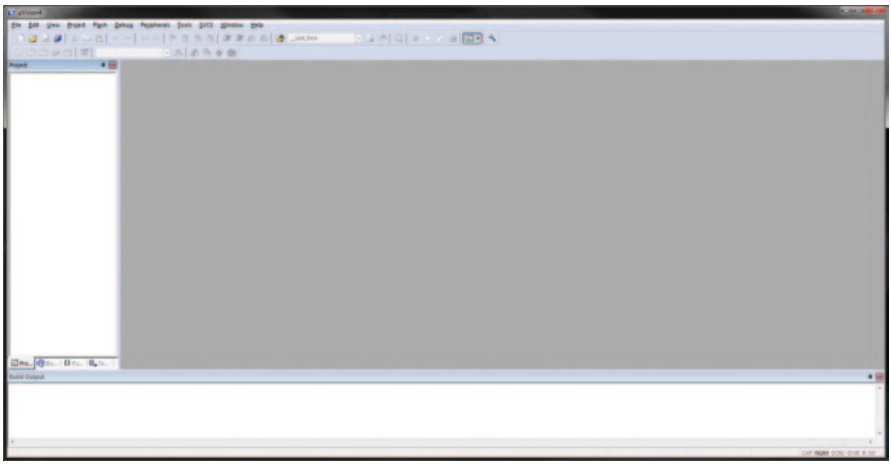


Fig. 5.1 Keil µVision window

The Keil development tools were selected for running assembly language throughout this book and following steps describe how to use Keil development tools for writing Assembly Language.

After installing Keil µVision to your computer, you will be able to begin creating programs for the ARM, Open µVision as shown in Fig. 5.1.

On µvision click on **project** and select **new project** and give a name, such as project2, to display Fig. 5.2 target device window.

From the target device window select NXP manufacture then LPC1768, press **ok**. Will display Fig. 5.3.

In Fig. 5.3 press yes and display Fig. 5.4 windows, in Fig. 5.4 click on Target and you will see start file is added to the target.

On Fig. 5.4 window click **File** and select **new file**, type following sample program. Save this file with the '.asm' extension.

```
          AREA  NAME, CODE, READONLY
          EXPORT SystemInit
          EXPORT __main
SystemInit



__main

     MOV R1, #0x25    ; program code
     MOV R2, #023
          END
```

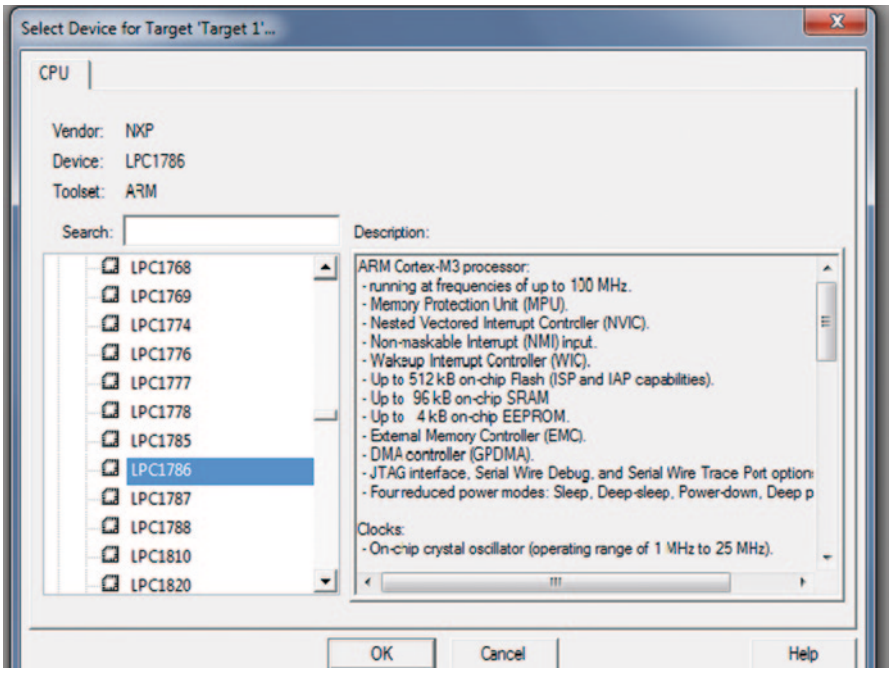


Fig. 5.2 Target device window

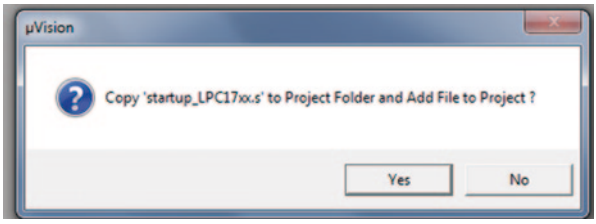


Fig. 5.3 Copy startup window

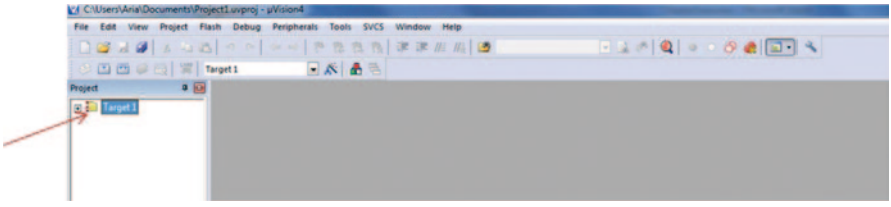


Fig. 5.4 Project window

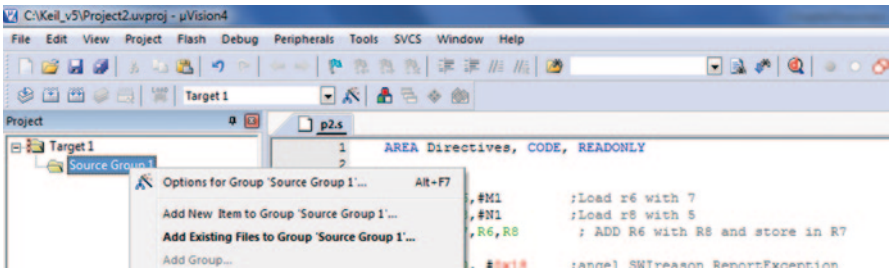
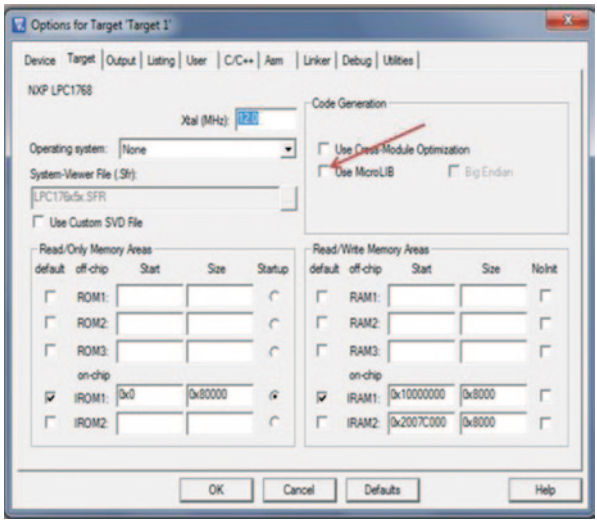


Fig. 5.5 Adding file to Group

Fig. 5.6 Option for target



Save the file in project directory with extension .s (p2.s).

Click on **target** then right click on **Source Group** 1 to display Fig. 5.5. Then select **Add Existing File to Group** and add the file to the group.

Select **Project** then select **options for target** to display Fig. 5.6, Now, select **Use MicoLIB** and click **ok**.

5.2.1 Building a Project

Once you have added a file to your project, and are ready to compile, you can either navigate to **Project→Build Target** or hit F7 on the keyboard. The **Build Output** panel on the bottom of the window will show any errors, warnings or if the project was built successfully as shown in Fig. 5.7.

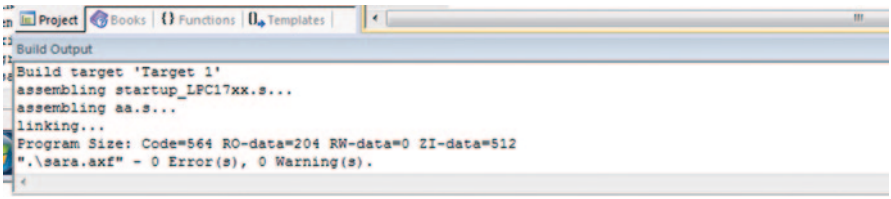


Fig. 5.7 Build output

5.2.2 Debugging a Program

Now that you have some compiled a piece of code, you will want to debug the code for testing. Navigate to **Debug→Start/Stop Debug Session** to switch to the debug environment as shown in Fig. 5.8.

Controls
Run—**F5**—Runs the program until it hits a breakpoint or the end of the program.
Step Into—**F11**—Steps through the code and follows into functions.
Step Over—**F10**—Steps through the code and jumps over functions.
Step Out—**Ctrl+F11**—Step out of a function.

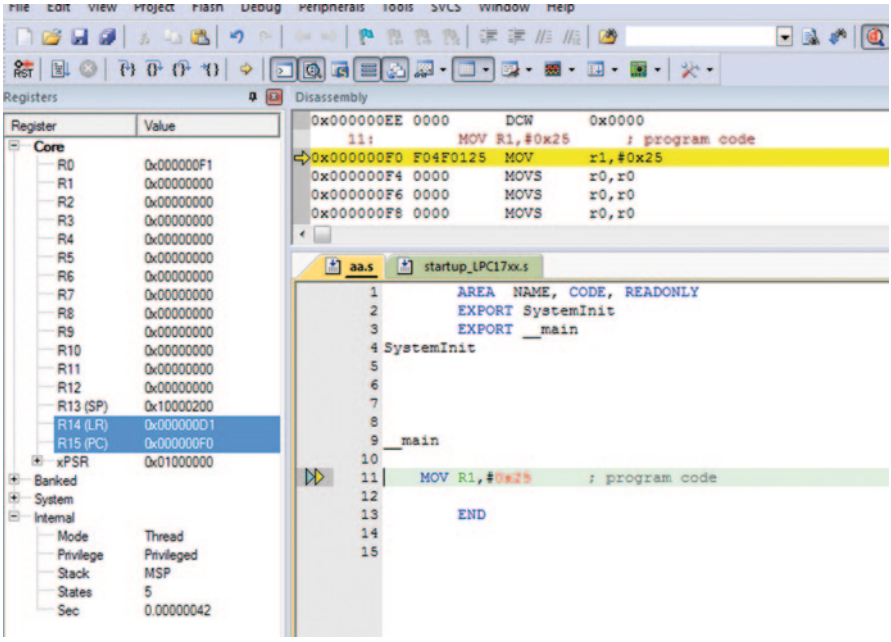


Fig. 5.8 Debug environment

Fig. 5.9 Use of breakpoint

```
 6                EXPORT __main
 7
 8  SystemInit
 9                MOV R1, #0x01
10
11  __main
12                ORR R1, R1, #0xF
13
14                END
15
16
```

Breakpoints You can add a breakpoint to a line of code that you would like the debugger to stop, or "break", at when reached. Once a breakpoint is reached, you can use the controls above to step through the code as shown in Fig. 5.9.

The debugger will monitor the CPU's registers and update their values in the register bank on the left side of the window as shown in Fig. 5.10.

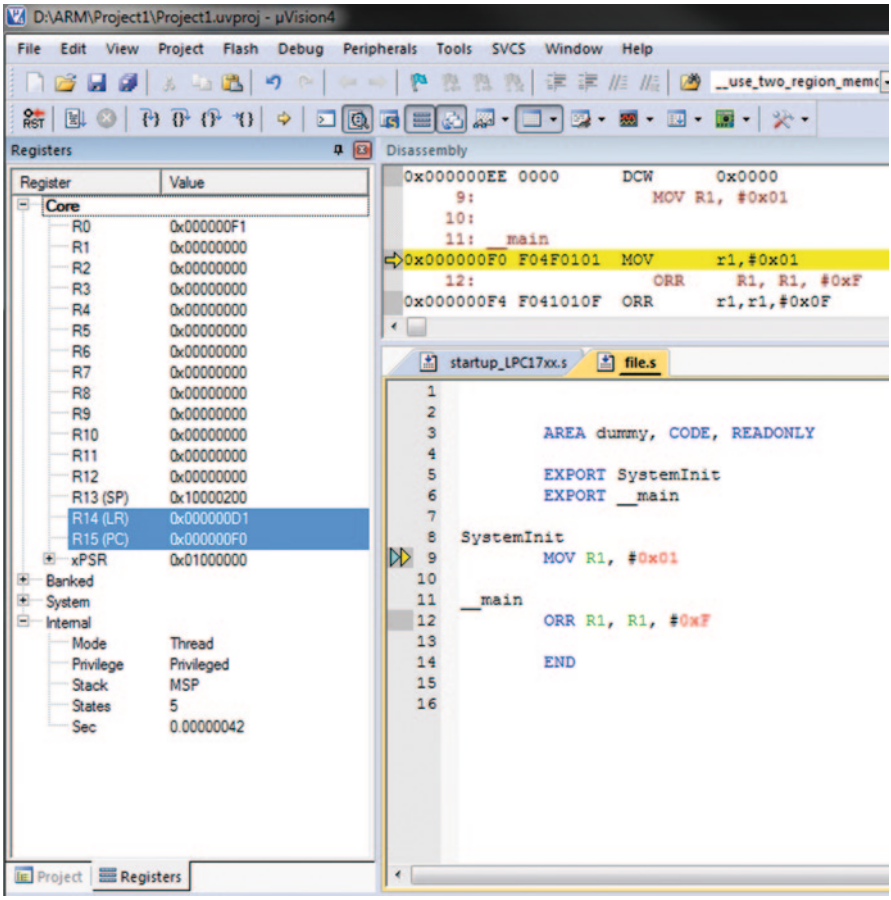


Fig. 5.10 Register bank

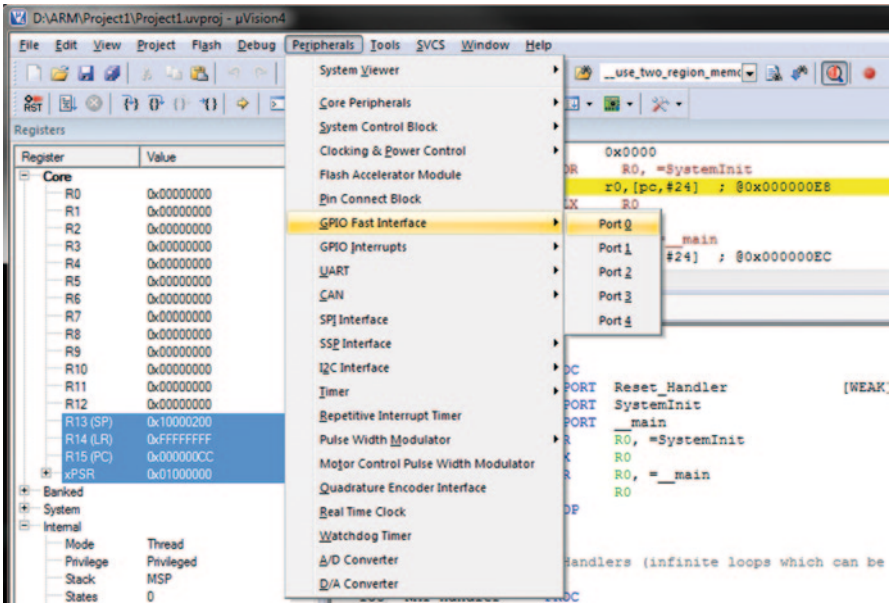


Fig. 5.11 LCP1768 Peripherals

While program in debug mode by selecting peripheral will display peripheral of the LPC1786 processor as shown in Fig. 5.11, in Fig. 5.11 the LCP 1786 contain GFIO fast Interface which consist of five ports p0 through p4.

Also while in debug mode by selecting view then memory windows then memory1 will display the contents of memory as shown in Fig. 5.12.

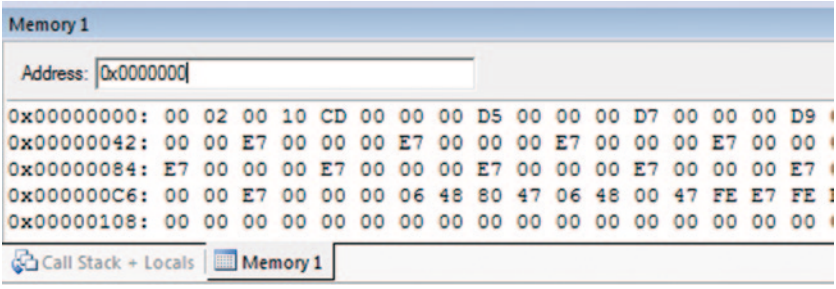


Fig. 5.12 Content of memory1

```
AREA  NAME, CODE, READONLY
              EXPORT SystemInit
              EXPORT __main
              ENTRY
SystemInit



__main

      MOV R1,#0x25   ; program code
      MOV R2,0x30
      ADD R1, R1, R2
      END
```

Fig. 5.13 Keil template for writing NXP Cotex-M3 assembly language

5.3 Program Template

Figure 5.13 shows the template that is used when writing assembly source code. The BOLD words are needed for all program and program code placed between **__main** and **END**.

5.4 Programming Rules

CSAE Rules Instructions, symbols and labels can be written in uppercase or lowercase but cannot be combined.

A generic line of assembly language has the following format:

Label Mnemonic Operand(s); Comment

Label A label is used to define a memory location. The assembler calculates its numerical value. ***Labels must start on the first column of each line of the source file***. Labels can be any string of characters with an unlimited size, but cannot begin with a number.

Mnemonic An instruction represented in mnemonic form. For example, ADD represents the instruction for addition and SUB represents the instruction for subtraction.

Operand(s) Each instruction may have one or more operand.

ADD R1, R1, R2; This instruction has three operands, R1, R2, and R3.

Comments The programmer can write comments after a semicolon (;)

MOV R1, R2; Moving contents of R1 to R2.

5.5 Directives

A directive is an assembler command that is executed by the assembler. Directives never produce any machine code. Directives are used to assign start of code, data and end of the program, A simple directive is END, which constitutes the end of a program. Here is a list of the most useful directives used by the ARM Assembler.

AREA Defines a segment of memory.
ENTRY Defines the start of the program.
EQU Used to assign a constant to a label.
Book EQU 0x25.

5.5.1 Data Directive

Data directives that define types and size of data and they are:
DCB (Define Constant Byte), DCW (Define Constant Half Word), DCD (Define Constant Word), and SPACE.

DCB (Define Constant Byte) DCB means define constant byte is used for allocating one or more than once byte in memory. Figure 5.14 shows how the List stored in memory.

Fig. 5.14 Byte addressable memory using a label

List	34
List+1	56
List+2	78
List+3	65

```
list DCB 0x34, 0x56, 0x78, 2_01100101
```

DCB also can be represented by hexadecimal, binary and decimal

```
Label DCB 0x23, 2_00011111, 23
```

Define Constant Half Word (DCW) DCW define constant word is used to define a half word (16 bits) and requires two memory locations per half word such as

```
List DCW 0x2345
```

```
Label2 DCW 0x2345, 0xFEEE, 0x4567
```

Define Constant Word (DCD) DCD is used to define a word and requires four memory locations per word such as

```
List DCD 0x23456789
```

Character Strings A sequence of characters is called a character string. In ARM Assembly, character strings are represented inside double quotation marks, followed by a comma and a zero. If there is a dollar sign ($) or double quotation (“) inside the string then the character must be repeated such as

```
List DCB “Assembly”,0
```

or

```
List DCB “I have $ 250.00”,0
```

Single Character When storing a single character in a register or memory location the character must be inside single quotation marks.

```
List DCB 0x23, ‘A’
```

or

```
MOV R1, #’A’
```

or

```
MOV R1, #0x41   ;0x41 is ASCII for the character 'A'
```

Reserving Memory SPACE is used to reserve memory locations for later use.

```
List SPACE 20      ;reserve 20 memory locations starting at the address of List
```

Problems

1. Write a program to add elements of List1 and store in List2.

```
List1   DCB  0x23, 0x45, 0X23, 0x11
List2    DCB  0x0
```

2. Write a program to find the largest number and store it in memory location List3.

```
List1     DCD     0x23456754
List2     DCD     0X34555555
List3     DCD       0x0
```

3. Write a program find the sum of data in memory location LIST and store the SUM in memory location Sum using loop.

```
List1   DCB  0x23, 0x45, 0X23, 0x11
List2    DCB  0x0
```

4. Show the content of registers R1 through R5 after execution of following program.

```
    AREA  NAME, CODE, READONLY
          EXPORT SystemInit
          EXPORT __main
          ENTRY
SystemInit

__main

          ADR R0, LIST1
          LDRB R1, [R0]
          LDRB R2, [R0, #1]!
          LDRB R3, [R0,#1]!
          LDRB R4, [R0,#1]!
          LDRB R5, [R0,#1]

List    DCB   0x23, 0x24, 0x67, 0x22, 0x99
       align
SUM     DCD   0x0
      align
     END
```

5. Write assembly language to clear bit position 0, 3, 5, and 6 of R12, the other bits must be unchanged (using ARM Instruction).

6. Write assembly language program for following HLL.

IF R1 = R0

Then

ADD R3, R0, #5

Else

SUB R3, R0, #5

7. Write a program to read memory location LIST1 and LIST2 and them then store the sum in LIST3.

```
LIST1   DCD 0x00002345
LIST2   DCD 0X00011111
LIST3   DCD 0x0
```

8. Write a program to multiplying two Numbers using subroutine.
9. Write a program to add 8 numbers using Indirect addressing.

```
LIST DCB 0x5, 0x2,0x6,0x7 ,0x9,0x1,0x2,0x08
```

10. Write a program to add 8 numbers using Post Index addressing.

```
LIST DCB 0x5, 0x2,0x6,0x7 ,0x9,0x1,0x2,0x08
```

11. Write a program to convert following HLL language to ARM instructions.

```
IF R1=R2 AND R3>R4 then

R1= R1 +1
Else
R3=R3 +R3*8
Endif
```

12. What is Contents of R4 after Execution of following Program.

```
          AREA  NAME, CODE, READONLY
              EXPORT SystemInit
              EXPORT __main
              ENTRY
SystemInit



__main

      LDR R1, =0xFF00FF
       ADR R0, LIST1
       LDR R2, [R0]
       AND R4, R2, R1

LIST1   DCD 0X45073487


     END
```

13. Write a program to convert following HLL to assembly language.

```
If  R1=R2 then
R3= R3+1
IF R1<R2 Then
R3=R3-1
If R1>R2 Then
R3=R3-5
```

14. Write a subroutine to calculate value of Y where Y = X * 2 + x + 5, assume x represented by List DCB 0x5

```
LIST DCB 0x5


LIST1    DCB  0x5
```

15. Write a program to rotate R1 16 times, assume R1 contains 0x12345678.
16. Write a program to compare two numbers and store largest number in a memory location LIST.

```
M1    EQU   5
N1    EQU   6
LIST2 DCB  0x0
```

17. Write a program to read a word memory location LIST and Clear bit position B4 through B7 of register R5, assume R5 contains 0XFFFFFFF.

```
LDR R0, =0x000000F0
LDR R5, =0xFFFFFFFF
```

18. Write program to load Register R1, R2, R3, and R4 from memory location LIST.

```
LIST DCD LIST DCD 0x12345AAA, 0x0000BBBB, 0x0000CCCC, 0X0000DDD

LIST DCD 0x12345AAA, 0x0000BBBB, 0x0000CCCC , 0X0000DDDD
    END
```

Chapter 6
ARM Cortex-M3 Processor and MBED NXP LPC1768

6.1 Introduction

ARM offers variety of the core processor base on their applications and they are:

Cortex A series: Cortex A series is a High performance processor for open operating system, the Cortex-A50 is a 64 bit process, application of Cortex-A series are Smart phones, Netbook, Digital TV, and eBook readers

Cortex-R series: Cortex-R series is design for real time application such as automobile braking, mass storage controller, printers and networking

ARM Secure Processor: This is an ultra-low power processor and it is used for SIMs cards, smart cards and electronics passport. Figure 6.1 shows the general Architecture of ARM Processor

ARM Cortex M series: ARM Cortex M series is used as microcontroller for applications such as smart sensors, automobile control system, motor control, smart meters and airbags. The ARM Corporation develop ARM core processor and ARM developer partners add more Peripherals to the ARM processor such as A/D, D/A, CAN, Ethernet and USB.

The Cortex-M3 is based on Harvard Architecture with 3 stage pipeline Architecture.

The ARM cortex is a low power processor and it is designed for embedded application with following features

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_6

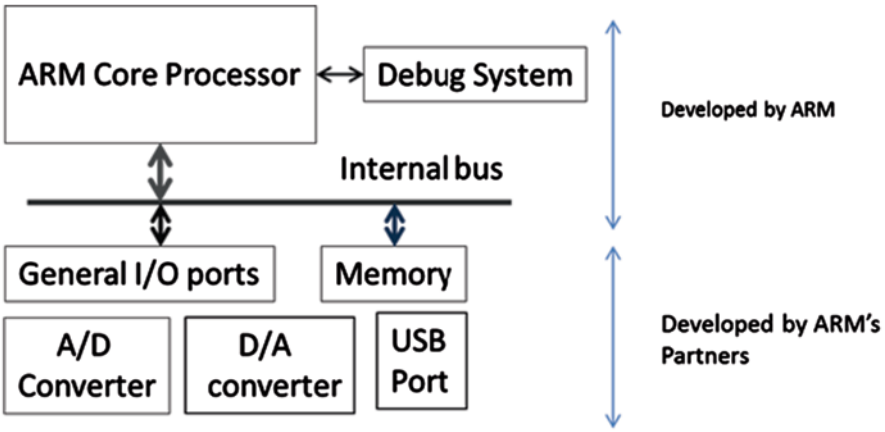


Fig. 6.1 Block diagram of ARM processor with peripherals

ARM Cortex-M3 Specifications

ARM Cortex-M3 Features	
ISA Support	Thumb®/Thumb-2
Pipeline	3-stage
Performance Efficiency	3.32 CoreMark/MHz* −1.25 to 1.50 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Up to 240 Wake-up Interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability.Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Integrated Instructions & Bit Banding
Enhanced Instructions	Hardware Divide (2–12 Cycles), Single-Cycle (32 × 32) Multiply, Saturated Math Support.
Debug	Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints.
Trace	Optional Instruction Trace (ETM), Data Trace (DWT), and Instrumentation Trace (ITM)

Figure 6.2 shows Internal components of ARM Cortex-M3 (http://www.microsemi.com/products/fpga-soc/soc-processors/arm-cortex-m3)

Nested Vector Interrupt Controller (NIVC): The NVIC supports up to 240 Priority interrupts The main purpose of the NVIC is to handle low-latency exceptions and interrupts, and control the CPU's power management. The NVIC supports

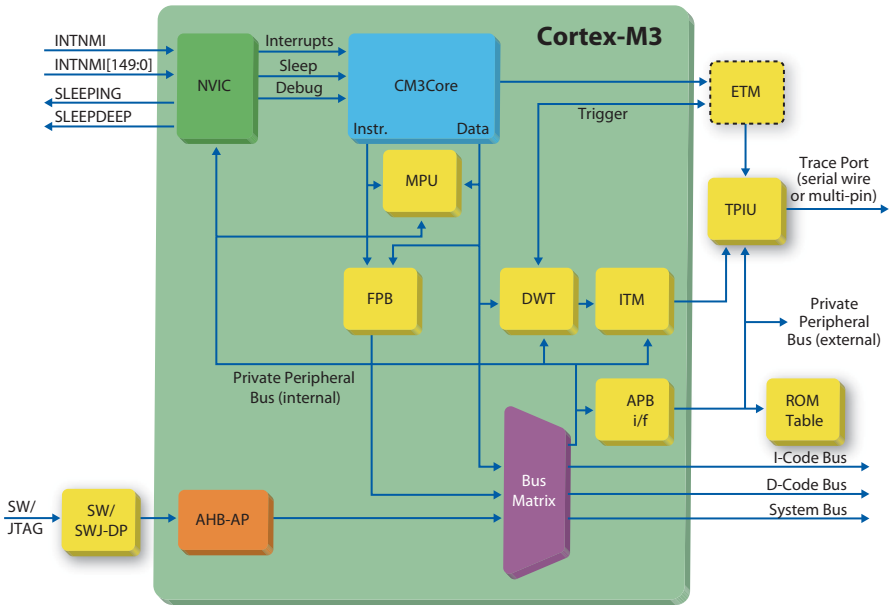


Fig. 6.2 Internal components of ARM Cortex-M3

nested interrupts by maintaining knowledge of the stack, which allows for tail-chaining interrupts, the NVIC also supports interrupt masking.

Bus Matrix: The bus matrix connects the processor and debug interface to the external buses and it is interfaces with the following external buses.

I-Code Bus: I-Code Bus is used to fetch Instruction fetch from memory.

D-Code Bus: It is used for data load/store and debug accesses to code space.

System Bus: For instruction and vector fetches, data load/store and debug accesses to system space.

Memory Protection Unit (MPU): The MPU provides support for protecting memory regions, overlapping protection regions, memory access permissions, and exporting memory attributes to the system. The MPU can be used for enforcing privilege rules, separating processes, and enforcing access rules.

Flash Patch and Breakpoint (FPB) Unit: The FPB implements hardware breakpoints and can be used to patch code and data from code space to system space.

Data Watch Point and Trace Unit (DWT): The DWT is a unit that performs debugging functions.

AHB-AP: The AHB-AP is an optional debug access port for the Cortex-M3 system, and provides access to all memory and registers in the system.

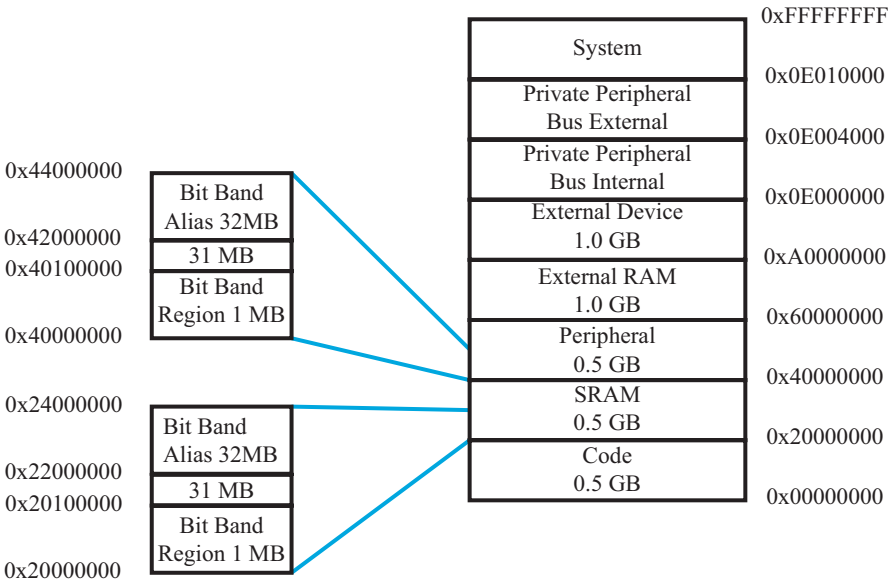


Fig. 6.3 Cortex-M3 Memory Map

Memory: Cortex-M3 support 32 address lines which enable it to access 2^{32} memory location and each memory location holds one byte, the cortex-M3 can have 4 Gigabytes of Memory and Fig. 6.3 shows ARM Cortex Memory Map.

6.2 MBED NXP LPC1768

For the hardware experiments we selected MBED NXP LPC1768 which is one of the most popular microcontroller with ARM Cortex-M3 processor, Fig. 6.4 shows MBED NXP LPC1768 pin out and its peripherals

The MBED contains following components

1. **ARM Cortex-M3 Processor with following features**
   A. Clock Operation 100MHZ
   B. Nested Vector Interrupt Controller
   C. Weak up Interrupt Controller
   D. Reduced power mode
2. **Memory**
   A. 512 KB of Flash memory
   B. 64 K bytes of SRAM (Static RAM)
3. **Three Universal Asynchronous Receiver/Transmitter (Serial Input/Output)**

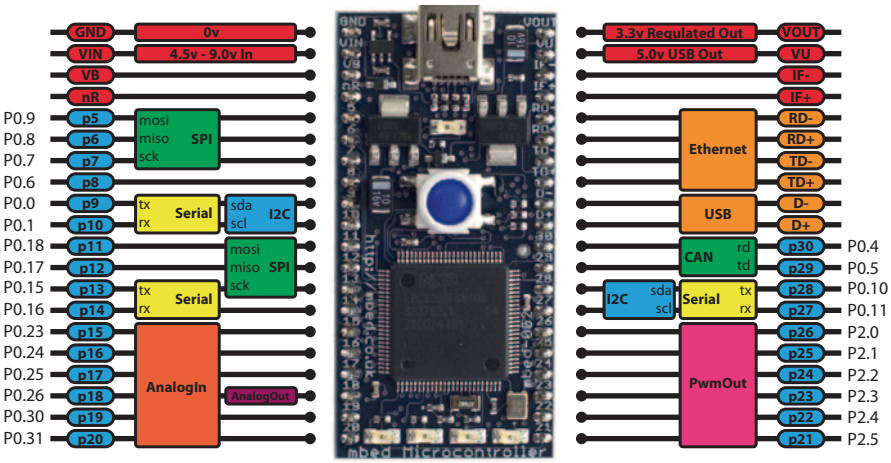


Fig. 6.4 MBED Block diagram [2],image from http://www.nxp.com/documents/leaflet/LPC1768.pdf

4. **USB port**
5. **CAN (Controlled Area network)**: CAN is a two-wire serial bus communications originally developed for the automotive industry
6. **SPI (Serial Peripheral Interface)**: SPI is a synchronous serial data link
7. **I²C (Inter-Integrated Circuit)**: it is a multi-master serial single-ended computer bus and it used for connecting low-speed peripherals to a microcontroller
8. **12 bit Analog to Digital (A/D) Convert with 8 channels**
9. **10 bit Digital to Analog (D/A) Convert**
10. **PWM (Motor Control):**
11. **Timer/Counter**
12. **General Purpose Input and outputs**
13. **4 LEDs**

Figure 6.4 shows MBED LPC 1768, MEBD uses Cortex-M3 processor, the Cortex-M3 has 4 ports and it is called P0, P1, P2 and P3, each port has 32 pins and each pin represented by Px.y, where x represented port number and y represent pin number. The MBED board uses some of the ports of Cortex M3 not all of them. There are four LEDs on the MBED board which are connected to the following ports

LED1	LED2	LED3	LED4
P1.18	P1.20	P1.21	P1.23

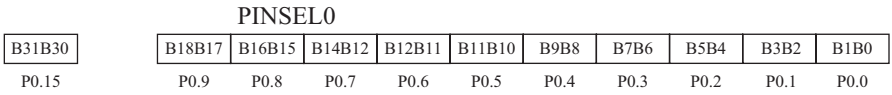


Fig. 6.5 PINSEL0 Register

6.3 Basic GPIO Programming

The following steps describe how to program a General Purpose Input/output pin using assembly language. Some applications of GPIO pins include driving LEDs, controlling external devices, sensing digital inputs, and waking up the device. In this guide, the GPIO pins will be used as simple I/O by manipulating the registers dedicated to configuring GPIO Port 0.

A. Setting the Pin Function
B. Setting the Pin Direction
C. Setting Fast GPIO Port Mask Register
D. Setting output Pin to logic high
E. Clearing a Pin
F. Reading a Pin Value

A. Setting the Pin Function

Each pin of MBED can be used for multiple functions such as Input/output, Serial Receiver (RX) or Serial Transmitter (TX) and some pins can have four different functions. Two bits are used to select function of each pin, therefore 64 bits are required to selecting function of a 32 bit port such as port P0. This 64 bits is represented by two 32 bit registers called PINSEL0 and PINSEL1 each register define by an address, for selecting port P0 pins function. Figure 6.5 show PINSEL0 and corresponding pins of P0/y

Bits B1B0 select function of P0.0, For using P0.0 as Input/output the B1B0 must set to 00 result P0.0 can be used as I/O

Table 6.1 shows PINSELX registers with their corresponding address and corresponding pins and Table 6.2 shows PINSEL0 function bits.

Table 6.1 PINSELX Register with corresponding Address and Port function bits

Register Name	Address	Port function bits
PINSEL0	0X4002C000	P0.15-P0.0
PINSEL1	0X4002C004	P0.31-P0.16
PINSEL2	0X4002C008	P1.15-P1.0
PINSEL3	0X4002C00C	P1.31-P1.16
PINSEL4	0X4002C010	P2.15-P2.0
PINSEL4	0X4002C01C	P2.31-P2.16

Table 6.2 PINSEL0 function bits (0x4002C000)

Bits	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset Value
1:0	P0.0	GPIO Pin 0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Pin 1	TD1	RXD3	SCL1	00
5:4	P0.2	GPIO Pin 2	TXD0	AD0.7	Reserved	00
7:6	P0.3	GPIO Pin 3	RXD0	AD0.6	Reserved	00
9:8	P0.4	GPIO Pin 4	I2SRX_CLK	RD2	CAP2.0	00
11:10	P0.5	GPIO Pin 5	I2SRX_WS	TD2	CAP2.1	00
13:12	P0.6	GPIO Pin 6	I2SRX_SDA	SSEL1	MAT2.0	00
15:14	P0.7	GPIO Pin 7	I2STX_CLK	SCK1	MAT2.1	00
17:16	P0.8	GPIO Pin 8	I2STX_WS	MISO1	MAT2.2	00
19:18	P0.9	GPIO Pin 9	I2STX_SDA	MOSI1	MAT2.3	00
21:20	P0.10	GPIO Pin 10	TXD2	SDA2	MAT3.0	00
23:22	P0.11	GPIO Pin 11	RXD2	SCL2	MAT3.1	00
29:24	–	Reserved	Reserved	Reserved	Reserved	0.0
31:30	P0.15	GPIO Pin 15	TXD1	SCK0	SCK	00

B. Setting the Pin Direction

The Fast GPIO Direction Register (FIOxDR) is used to set direction of I/O pins as input or output of a port, X in FIOXDR represent the port number, each port has one FIOxDR register and each register is represented an address

FIO0DR with the address 0x2009C000 is used to set P0.0 through P0.31 as input or output

FIO1DR with the address 0x2009C020 is used to set P1.0 through P1.31 as input or output

FIO2DR with the address 0x2009C040 is used to set P2.0 through P2.31 as input or output

The direction of the pin determines if the pin will act as an input or an output. Each bit in the FIOxDR Register corresponds to a GPIO pin. Figure 6.6 shows FDIO0DR Register with Address FIO0DR = 0x2009C00

In the above configuration P0.0 is set to output and P0/1 is set to input.

C. Fast GPIO Port Mask register (FIOxMASK)

There is a mask register that is dedicated to selecting which pins on the port will and will not be affected by write accesses. This register will also filter the ports contents when reading inputs. Writing a 0 in this register's bits enables read and write access to the corresponding pin. If a bit's value is 1 then the corresponding pin will not be changed with a write access and that pin will not be updated in the register that holds pins' values. Every Port assigned as FIOxMASK register, the Table 6.3 shows FIOxMASK and correspond address for port 0 and port1 with address 0x2009C000. Figure 6.7 show FIO0MASK with b1b0 equal 00.

Value	Description
0	Input
1	Output

Fig. 6.6 FIOxDR Register format

Table 6.3 FIOxMASK with corresponding address and port

Register	Address	Port number
FIO0MASK	0x0x2009C010	P0
FIO1MASK	0x0x2009C030	P1

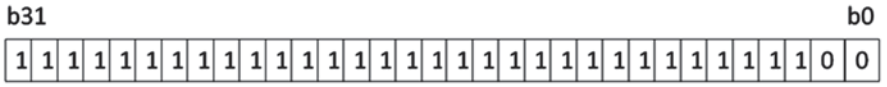


Fig. 6.7 FIO0MASK with b1b0 = 00

D.Setting Output Pin (FIOxSET)

Once you have a pin configured as an output, it will be a very simple to modify the value of the pin. There is a pin set register that produces a high level output on the pins selected (again, they must be in output mode). Writing a 1 to any bit in this register will produce a high level output on the corresponding pins. Also, if the pin is configured as an input, writing a 1 will have no effect.

Each port has a FIOxSET register, Table 6.4 shows FIOxSET address and corresponding port

Figure 6.8 shows FIO0SET register with b0 set to one and the default reset value for this register is 0x0.

E. Clearing a Pin (FLOxCLR)

There is a register dedicated to producing a low level output to a pin. Writing a 1 to this register will produce a low level output to the corresponding pin. Writing a 0 will have no effect, and pins that are not configured as outputs will remain unchanged. Each port has one Cleraring register which represent by FIOxCLR, The FLO0CLR with address—0x2009C01C is used for P0 (port zero). Figure 6.9 show that P0/1 is set to zero and The default reset value for this register is 0x0.

F. Reading a Pin Value (FIOxPIN)

There is a register that provides the value of pins configured as digital inputs or outputs. When this register is read it will return the logic value of the pin regardless of its configuration, as long as it is a digital I/O. Writing to this register will store

Table 6.4 FIOxSET Register with corresponding Address and Port

Register	Address	Port number
FIO0SET	0x0x2009C018	P0
FIO1SET	0x0x2009C038	P1

Fig. 6.8 FIO0SET with b0 set to 1

Fig. 6.9 FLO0CLR register (0x2009C01C)

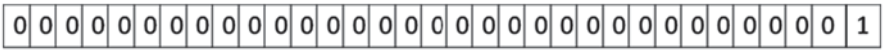


Fig. 6.10 FIO0PIN register Contents

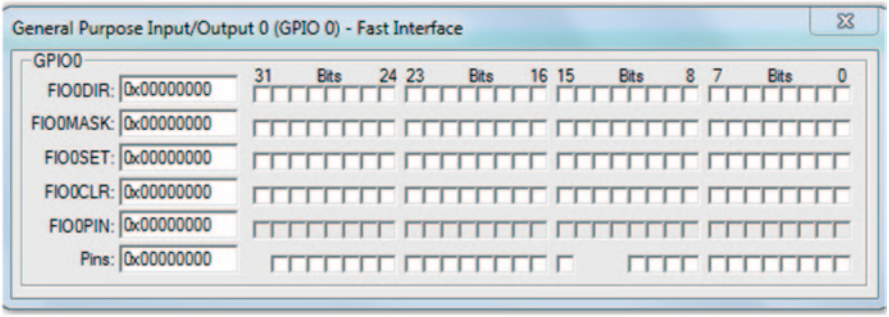
the values written to the pins. When you write to this register, although bypassing the need for the SET and CLR register, it affects the entire ports pins, therefore the pins should be properly masked. Each port has one FIOxPIN register and Fig. 6.10 shows the contents of FIO0PIN register after reading

The above register has a value of 0x1 and pin 0 has a high level value. The default reset value for this register is 0x0.

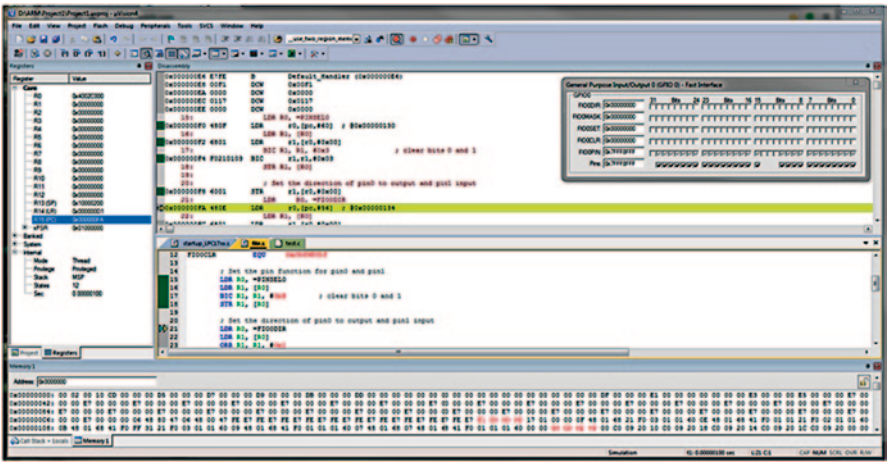
Example: In this example the P0/0 is set as output with logic high and P0/1 set to input

```
        AREA gpio, CODE, READONLY
        EXPORT SystemInit
        EXPORT __main
SystemInit

PINSEL0             EQU         0x4002C000
FIO0DIR             EQU         0x2009C000
FIO0MASK            EQU         0x2009C010
FIO0PIN             EQU         0x2009C014
FIO0SET             EQU         0x2009C018
FIO0CLR             EQU         0x2009C01C

        ; Set the pin function for pin0 and pin1
        LDR R0, =PINSEL0
        LDR R1, [R0]
        BIC R1, R1, #0x3       ; clear bits 0 and 1
        STR R1, [R0]

        ; Set the direction of pin0 to output and pin1 input
        LDR   R0, =FIO0DIR
        LDR R1, [R0]
        ORR R1, R1, #0x1
        BIC R1, R1, #0x2
        STR R1, [R0]

        ; Set the mask to only allow R/W to pins0 and1
        LDR R0, =FIO0MASK
        LDR R1, [R0]
        ORR R1, R1, #0xFFFFFFFF
        BIC R1, R1, #0x3
        STR R1, [R0]

__main
; Write out a 1 to pin0
LDR   R0, =FIO0SET
LDR R1, [R0]
ORR R1, R1, #0x1
STR R1, [R0]

; Read pin value of pin1
LDR R0, =FIO0PIN
LDR R1, [R0]           ; value of pin1 is in R1

; Clear pin 0 to 0 (low level)
LDR R0, =FIO0CLR
LDR R1, [R0]
ORR R1, R1, #0x1
STR R1, [R0]

END
```

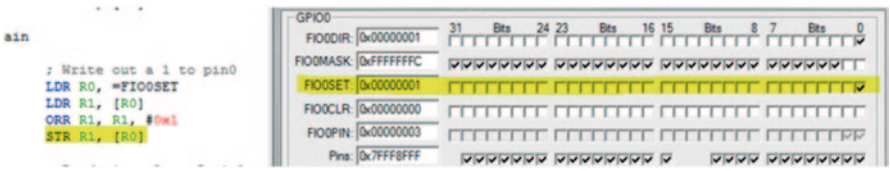
LCP 1768 Peripheral: Another way to configure the value for GPIO control register is to used Peripheral simulator for I/O port, on uvision window click on debug then start while uvision in debug mode click on Peripheral and select GPIO fast interface port 0 will display following windows



Stepping Through Code and Debugging: Once you have the code in Keil uVision, press **F7** to compile and then **Ctrl-F5** to enter debugging mode. Now, go to **Peripherals- > GPIO Fast Interface- > Port 0** to bring up all the registers for GPIO Port 0. When stepping through the program, we will be able to see the registers change when written to. Step through **(F11)** the initialization code and observe the changes in the corresponding registers.

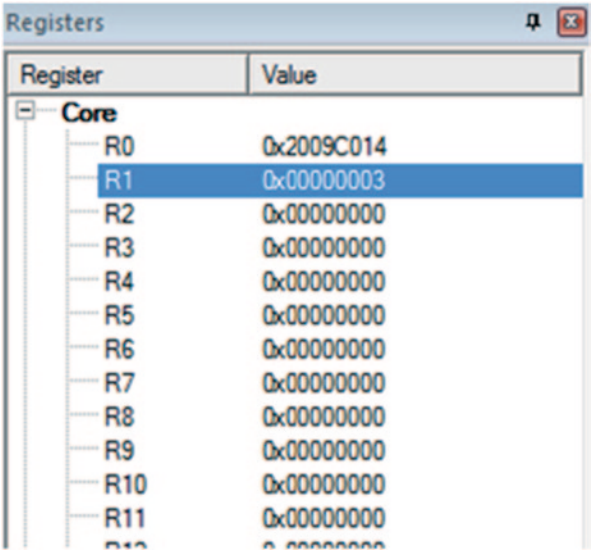


Outputting a High Signal/Writing a 1 to a pin: When you get to the STR instruction, the value generated is written to the specified register. In this case, you will see the checkbox appear for bit 0 on the FIO0SET register.



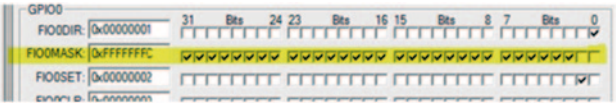
Reading the value of a pin: Before we step over the line LDR R1, [R0], let us manually set the value on pin 1 to HIGH by checking the checkbox for bit 1 in the FIO0SET register on the peripheral window.

Now, we can step over LDR R1, [R0] to read the value of the pins into R1. Keep in mind that all 32 pin values will be in register R1. In the register bank you will see that R1 has the value 0x3 (11 in binary), which means that pin 0 and pin 1 are both HIGH.



Clearing a Pins Value: The next code block sets pin 0 to LOW by writing a 1 to the FIO0CLR register. Notice that after the STR instruction, the checkbox is cleared on bit 0.

```
; Clear pin 0 to 0 (low level)
LDR R0, =FIO0CLR
LDR R1, [R0]
ORR R1, R1, #0x1
STR R1, [R0]
```



6.4 Flashing the NXP LPC1768

There are some prerequisites that need to be met before you can flash the NXP LPC1768 with a program.

Follow the guide to update the unit's firmware: http://mbed.org/handbook/Firmware-LPC1768-LPC11U24

Once that firmware update is complete, you are ready to flash the device with software. If you navigate to the directory where your project is save, you will notice that there is a.axf file with the same name as your project. This file needs to be converted from ELF format to a binary that can be run on the device. Luckily, Keil provides a tool with uVision, called fromelf, that we can use for this conversion. The binary 'fromelf' is located inside the install directory of Keil uVision in ARM/ARMCC/bin/. We will have uVision automatically run the command to create the appropriate binary file after project building.

Open a new project, or a current project in uVision and go to Project->Options for Target 'project-name' then the 'User' tab.

Add this line to 'Run #1' under Run User Programs After Build/Rebuild:
Be sure to use the correct path to the 'fromelf' executable and also the correct file name for the input file (after-bin) to match your project name. The output file name can be anything you wish plus the.bin.

Chapter 7
Lab Experiments

7.1 Introduction

The objectives of these labs are to use assembly language to program peripherals of microcontroller, you can use simulator of Keil development tools to observe result of your programs or use of MBED microcontroller.

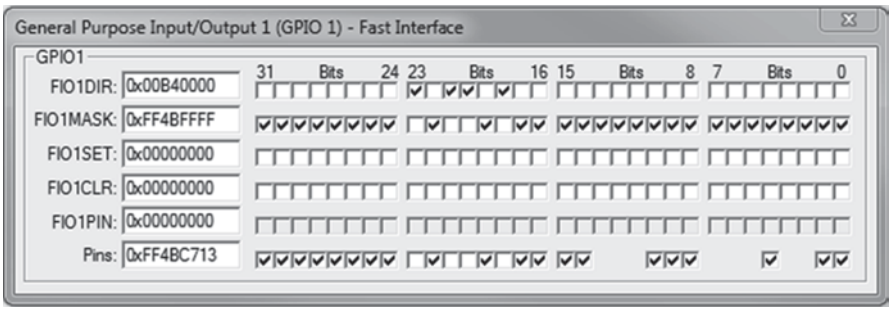
7.2 Lab#1 Binary Counter Using Onboard LEDs

The Objective of this lab is design a counter to count from 0000 to 1111 and display the result of the count on LEDs of MBED.

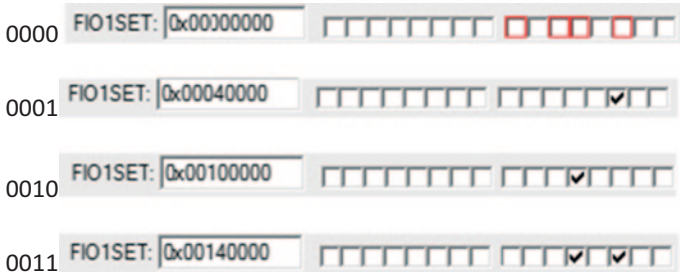
Now that we know how to program GPIO pins and how to flash the device, we can use that knowledge to toggle the onboard LEDs on our MBED LPC1768. If you don't have the physical device, you can still follow along using the simulator in uVision and watch the pins count in binary on the simulator's peripheral viewer. Enter debug mode, Ctrl-F5 (On uvision window select debug then start), Navigate to **Peripherals- > GPIO Fast Interface-** > **Port 1**.

Each onboard LED will represent a bit in a binary counter that will count from 0 to 15 (0xF/0b1111). Since the LEDs are mapped to GPIO pins that are not all in order, we will need to figure out the values that will output the correct binary number on the LEDs. Bits 23 (most significant bit), 21, 20, and 18 (least significant bit) of GPIO Port 1 are the LED bits.

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments,*
DOI 10.1007/978-3-319-11704-1_7

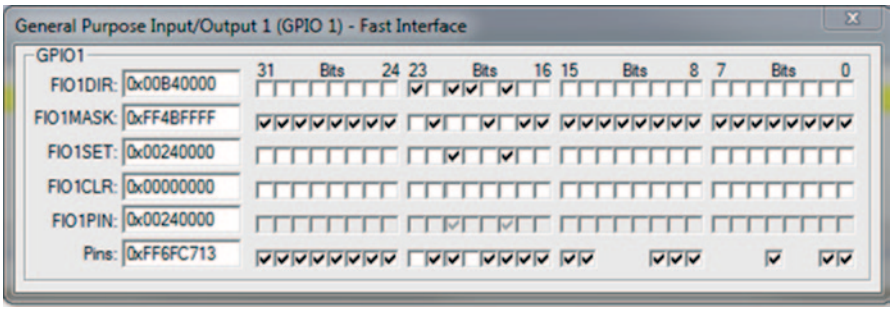


We will start counting from 0 to 1111, in binary. Then map the values to the 32 bit register so they reflect correctly on the LED bits. You may use the Keil peripheral viewer to convert the values to hex (toggle the checkboxes in FIO1SET).



The complete list of values is provided in the coding example. We will loop through the list to count in binary with the LEDs.

```
          AREA LED_counter, CODE, READONLY
          ENTRY
          EXPORT SystemInit
          EXPORT __main

; List of binary 0-1111 for LED display
Nums DCD  0x0, 0x40000, 0x100000, 0x140000, 0x200000, 0x240000,\
0x300000, 0x340000, 0x800000, 0x840000, 0x900000, 0x940000,
0xA00000,\ 0xA40000, 0xB00000, 0xB40000

PINSEL3            EQU       0x4002C00C
FIO1DIR            EQU       0x2009C020
FIO1MASK           EQU       0x2009C030
FIO1PIN            EQU       0x2009C034
FIO1SET            EQU       0x2009C038
FIO1CLR            EQU       0x2009C03C

   SystemInit
          ; Set the pin function for p1/18, P1/20,P1/21,andP1/23
          LDR R0, =PINSEL3
          LDR R1, [R0]
          MOV R2, #0x6F30
          BIC R1, R1, R2
          STR R1, [R0]

          ; Set the direction of p1/18, P1/20,p1/21, and P1/23 to output
          LDR  R0, =FIO1DIR
          LDR R1, [R0]
          ORR R1, R1, #0xB40000
          STR R1, [R0]

          ; Set the mask to only allow R/W to pin18, 20,21, 23 of port1
          LDR R0, =FIO1MASK
          LDR R1,[R0]
          ORR R1, R1, #0xFFFFFFFF
          BIC R1, R1, #0xB40000
          STR R1, [R0]

          MOV R4, #0x200000  ; wait counter


   __main
          ; Load address of FIO1PIN
          LDR  R0, =FIO1PIN
          LDR R1, [R0]
refresh
          MOV R5, #0x10           ; counter
          LDR R6, =Nums           ; Load address of Nums
          ; Beging counting in binary
count_loop
          LDR R1, [R6], #0x4      ; Load value of Nums and increment
                                  ; offset by 4 bytes to get next
                                  ; value
          STR R1, [R0]            ; Write to pins

          BL   wait               ; Add a delay so you can see LEDs
change
          SUBS R5, R5, #0x1       ; Subtract from counters
          BEQ  refresh            ; If counter is at 0 then refresh
counter
          B    count_loop

; Delay function that does subtraction for a little while
wait
          SUBS R4, R4, #0x1
          BNE   wait
          MOV R4, #0x200000  ; Reset wait counter
          BX   LR

          END
```



The uVision simulator interface for GPIO port 1, running the binary counter program.

7.3 Lab2: Configuring the Real-Time Clock (RTC)

The objective of this lab to set up initial value of the LPC1768 Real –Time Clock.

A real-time clock is a peripheral on a computer or embedded system that keeps track of the current time even when the system is off. Real-time clocks are more accurate at keeping time than other methods and also free the system by being a piece of hardware. The LPC17XX contains a Real Time Clock and user able to set this clock for operation, The RTC has following specifications.

Specifications

- Provides seconds, minutes, hours, day of the month, month, year, day of the week, and day of the year.
- Low power consumption. Less than 1 microamp for battery operation. Uses CPU power when present.
- 32 kHz oscillator.
- Calibration adjustment to ±1 s/day with 1 s resolution.
- Interrupts can be generated by increments of any field of the time registers.
- Interrupts: Interrupts can be used to control the RTC state (wake-up, sleep, power-down).
- General purpose registers to store data during system power off.

Configuration Following registers must be configured in order Real Time Clock initialize with time and date

1. Power Control for Peripherals register (PCONP—address 0x400F C0C4)
2. Clock Control Register (CCR-0x40024008)
3. Table 7.1 shows Registers that must loaded with Initial values and they are Read/Write Registers and their contents do not change by resetting the processor

Table 7.1 Clock registers

Register Name	Description	Address
SEC	Second Register	0x4002 4020
MIN	Minute Register	0x4002 4024
HOUR	Hours Register	0x4002 4028
DOM	Day of Month Register	0x4002402C
DOW	Day of Week Register	0x4002 4030
DOY	Day of Year Register	0x4002 4034
MONTH	Months Register	0x4002 4038
YEAR	Years Register	0x4002403C

Table 7.2 Clock Control Register (CCR)—0x40024008 [1]

Bit	Symbol	Value	Description	Reset value
0	CLKEN		Clock enable	NC
		1	Time counters enabled	
		0	Time counters disabled	
1	CTCRST		CTC Reset	0
		1	Resets oscillator divider	
		0	No Effect	
3:2	–	–	Must be 0	NC
4	CCALEN		Calibration enable	NC
		1	Calibration counter disabled	
		0	Calibration counter is enabled and counting	
31:5	–	–	Reserved	–

- Power Control for Peripherals register (PCONP—address 0x400F C0C4)

By setting of bit 9 of PCOP will enable RTC

- Clock Control Register (CCR)—0x40024008

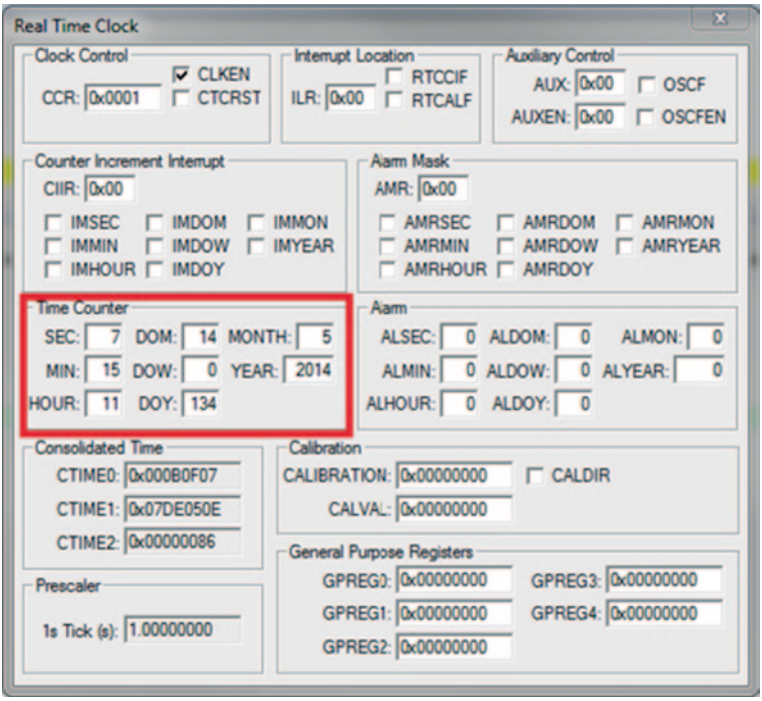
Table 7.2 shows CCR fields, The clock will be enable by set ting bit zero (b0) of CCR one.

Programming Example—Set Date and Time in RTC

```
The objective of this program to set RTC to 14 May 2014
at 11:15:00

         AREA rtc_config, CODE, READONLY
         EXPORT SystemInit
         EXPORT __main


PCONP_R             EQU         0x400FC0C4
CCR                 EQU         0x40024008

SEC_R               EQU         0x40024020
MIN_R               EQU         0x40024024
HOUR_R              EQU         0x40024028
DOM_R               EQU         0x4002402C
DOW_R               EQU         0x40024030
DOY_R               EQU         0x40024034
MONTH_R             EQU         0x40024038
YEAR_R              EQU         0x4002403C
YEAR_2014           EQU         0x7DE

SystemInit

   ; Enable power for RTC
   LDR   R0, =PCONP_R
   LDR R1, [R0]
   ORR   R1, R1, #0x200
   STR R1, [R0]

   ; Initialize the Clock Control Register
   ; by setting the CLKEN bit to 1
   LDR   R0, =CCR
   SUB R1, R1,R1
   ORR   R1, R1, #0x1
   STR R1, [R0]

__main

   ; Set time and date to Sunday, 14 May 2014 at  11:15:00

   ; Seconds set to 0
   LDR   R0, =SEC_R
   SUB R1, R1, R1          ; clear R1
   ORR   R1, R1, #0x0
   STR R1, [R0]            ;Set Second Counter to zero
```

```
; Minutes set to 15
LDR   R0, =MIN_R
SUB R1, R1,R1
ORR   R1, R1, #0xF
STR R1, [R0]

; Hour set to 11
LDR      R0, =HOUR_R
   SUB R1, R1,R1
   ORR   R1, R1, #0xB
   STR R1, [R0]

   ; DOM set to 14
   LDR   R0, =DOM_R
   SUB R1, R1,R1
   ORR   R1, R1, #0xE
   STR R1, [R0]

   ; DOW set to 0
   LDR   R0, =DOW_R
   SUB R1, R1,R1
   ORR   R1, R1, #0x0
   STR R1, [R0]

   ; DOY set to 134
   LDR   R0, =DOY_R
   LDR R1, [R0]
   ORR   R1, R1, #0x86
   STR R1, [R0]

   ; MONTH set to 5
   LDR   R0, =MONTH_R
   SUB R1, R1,R1
   ORR   R1, R1, #0x5
   STR R1, [R0]

   ; YEAR set to 2014
   LDR   R0, =YEAR_R
   SUB R1, R1,R1
   LDR R2, =YEAR_2014
   ORR   R1, R1, R2
   STR R1, [R0]


   END
```

In the simulator you will see the values being set in the RTC registers. Let the program run and you will see the seconds count up and eventually trigger the minutes.



7.4 Lab#3 Configuring Analog-To-Digital Converter (ADC)

Objective of this lab is to become familiarize with operation of ADC and how to program A/D converter for operation

Introduction The function of ADC is to convert analog signal to digital, Fig. 7.1 shows block diagram of 3 bits ADC.

Vref (Voltage Reference) It is used to compare input voltage with Vref, and also to determine the maximum amplitude voltage of input signal.

Outputs B2B1B0 are output binary numbers representing input voltage.

Assume Vref is 8 v and the resolution of ADC is $8/2^3 = 1$ V. This means, when the input changes 1 V then the output will change 1 bit.

As seen in Table 7.3 if input voltage 1.5 V the output will be 001. In order to have less error the number of outputs need to be increased. Most ADC converters come with 8, 12, 16, and 24 bits output. If an 8-bit ADC were used then the resolution will be $8/2^8 = 8/258 = 0.03125$ V, this mean that input voltage from 0-<0.03125 will

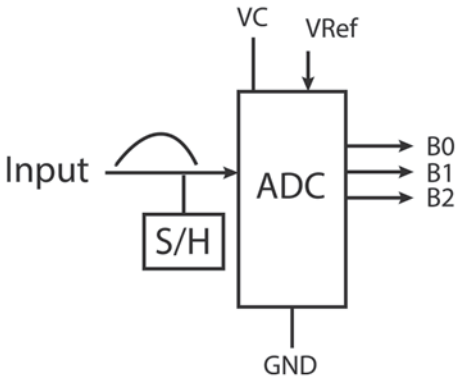


Fig. 7.1 Block diagram og ADC

Table 7.3 The Input voltage range and binary outputs

Input Voltage	Binary output
0-<1	000
1-<2	001
2-<3	010
3-<4	011
4-<5	100
5-<6	101
6-<7	110
7-<8	111

represent 00000000 in binary. Another way to have better a resolution is to decrease voltage reference, but it is important that the VRef should not be less than the maximum of input voltage Vin.

If Vref equals 4 V with a 3 bits ADC, then the resolution will be $4/2^3 = 0.5$ V, therefore smaller the resolution the smaller margin of error.

S/H (Sample and Hold) The function of S/H is to take samples of Input signals then have ADC convert it to binary, but the question is how many sample per second must take by S/H? According to Neyquest's theorem the sample rate must be at least twice the frequency of the input signal. If frequency of the input to the ADC be 8KHZ then the sampling rate should be 8000 or more sample per second.

Most A/D converter offers Multiple Inputs by using Analog Multiplexer, Fig. 7.2 shows A/D converter with 4 analog inputs, the function of S1 S0 is to select the input to the ADC converter

The A/D converter of NXP LPC 1768 has 8 inputs but MBED uses only 6 of them as shown in Fig. 6.3 Chap. 6.

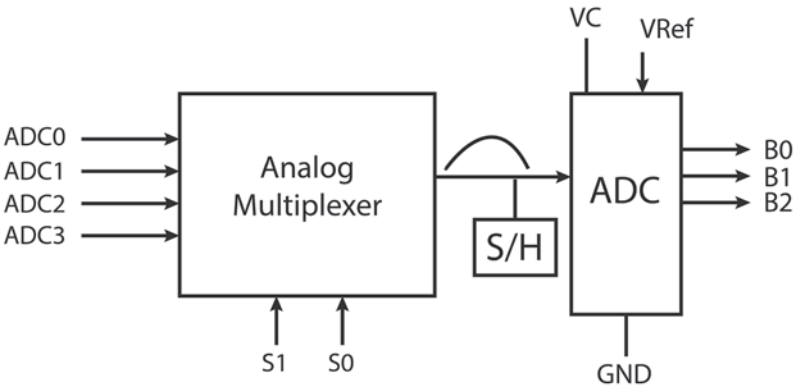


Fig. 7.2 Block diagram of A/D converter with 4 inputs

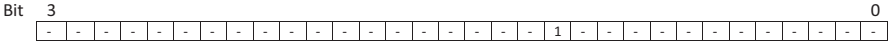


Fig. 7.3 Power control for peripherals register

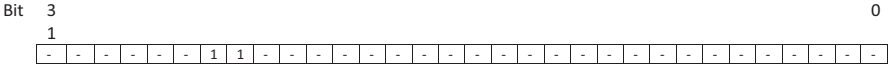


Fig. 7.4 Peripheral clock selection—PCLKSEL0–0x400F C1A8

MBED ADC Specifications

- 12-bit analog to digital converter.
- Input multiplexing among 6 pins.
- Power-down mode.
- Burst conversion mode

Configuration Flowing steps describe the configuration of ADC

- Power: PCONP—set PCADC bit
- Clock: PCLKSEL0—set bit PCLK_ADC
- Control Register: AD0CR—control the A/D
- Pins: PINSEL—select ADC0 pin

Power Control for Peripherals register—PCONP—0x400F C0C4 Set bit 12 to enable power/clock on ADC0. Disabled by default. Must clear the PDN pin in AD0CR before clearing this bit, and set this bit before setting PDN as shown in Fig. 7.3

Peripheral Clock Selection—PCLKSEL0–0x400F C1A8 Set bits 24 and 25 to enable the clock on the ADC, disabled by default as shown in Fig. 7.4.

Table 7.4 A/D Control Register (AD0CR)—0x40034000 [1]

Bit	Symbol	Value	Description	Reset value
7:0	SEL		Selects A/D pins to be sampled and converted. Bit 0 = AD0.0 on the board, and bit 7 = AD0.7.	0x01
8:15	CLKDIV		PCLK_ADC0 is divided by this to produce the A/D clock. <= 13 MHz.	0
16	BURST	1	A/D converter does repeated conversions of the pins selected in SEL.	0
20:17	-		Reserved	-
21	PDN	1	A/D converter is operational.	0
		0	A/D converter is in power-down mode.	
23:22	-		Reserved	-
26:24	START	**000**		0
		001	When BURST is disabled, these bits control the A/D conversion.	
		010	No Start.	
		011	Start conversion now.	
		100	Start conversion when the edge selected occurs on P2.10.	
		101	Start conversion when the edge selected occurs on the P1.27.	
		110 111	Start conversion when the edge selected occurs on MAT0.1. Start conversion when the edge selected occurs on MAT0.3.	
			Start conversion when the edge selected occurs on MAT1.0.	
			Start conversion when the edge selected occurs on MAT1.1.	
27	EDGE	1	Only significant when the START field contains 010–111	0
		0	Start on falling edge	
			Start on rising edge	
31:28	–		Reserved	–

A/D Control Register (AD0CR)—0x40034000: A/D control register is used to set up operation of A/D converter such as selecting input to A/D and clock as shown in Table 7.4

We will first use this register to put the ADC in operational mode by setting the PDN bit to 1. This register will then be used to select what A/D pins will be sampled for conversion using the SEL register.

Table 7.5 A/D Pin Selection—Pin Select Register—PINSEL0—0x4002C000

Bits	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset Value
1:0	P0.0	GPIO Pin 0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Pin 1	TD1	RXD3	SCL1	00
5:4	P0.2	GPIO Pin 2	TXD0	AD0.7	Reserved	00
7:6	P0.3	GPIO Pin 3	RXD0	AD0.6	Reserved	00
9:8	P0.4	GPIO Pin 4	I2SRX_CLK	RD2	CAP2.0	00
11:10	P0.5	GPIO Pin 5	I2SRX_WS	TD2	CAP2.1	00
13:12	P0.6	GPIO Pin 6	I2SRX_SDA	SSEL1	MAT2.0	00
15:14	P0.7	GPIO Pin 7	I2STX_CLK	SCK1	MAT2.1	00
17:16	P0.8	GPIO Pin 8	I2STX_WS	MISO1	MAT2.2	00
19:18	P0.9	GPIO Pin 9	I2STX_SDA	MOSI1	MAT2.3	00
21:20	P0.10	GPIO Pin 10	TXD2	SDA2	MAT3.0	00
23:22	P0.11	GPIO Pin 11	RXD2	SCL2	MAT3.1	00
29:24	-	Reserved	Reserved	Reserved	Reserved	0.0
31:30	P0.15	GPIO Pin 15	TXD1	SCK0	SCK	00

Table 7.6 A/D Global Data Register (AD0GDR)—0x40034004 [1]

Bit	Symbol	Description	Reset Value
3:0	–	Reserved	–
15:4	RESULT	When DONE is 1, this field contains a binary fraction representing the voltage on the pin selected by SEL in the control register.	–
23:16	–	Reserved	–
26:24	CHN	The bits contain the channel that the RESULT bits were converted from.	–
29:27	–	Reserved	–
30	OVERRUN	This bit is 1 in burst mode if the results of one or more conversions were lost and overwritten.	0
31	DONE	This bit is set to 1 when the A/D conversion is completed. It is cleared when this register is read and when the ADCR is written.	0

A/D Pin Selection—Pin Select Register—PINSEL0–0x4002C000 The PINSEL0 register is used to select function of the input pins as shown in Table 7.5 [1].

A/D Global Data Register (AD0GDR)—0x40034004 This register keeps the latest conversion done by the A/D converter. When the done bit set to one means conversion completed and result available on the bits b4 through b15 of this register. Each input channel allocated a Data register, Table 7.6 shows A/D global Data Register

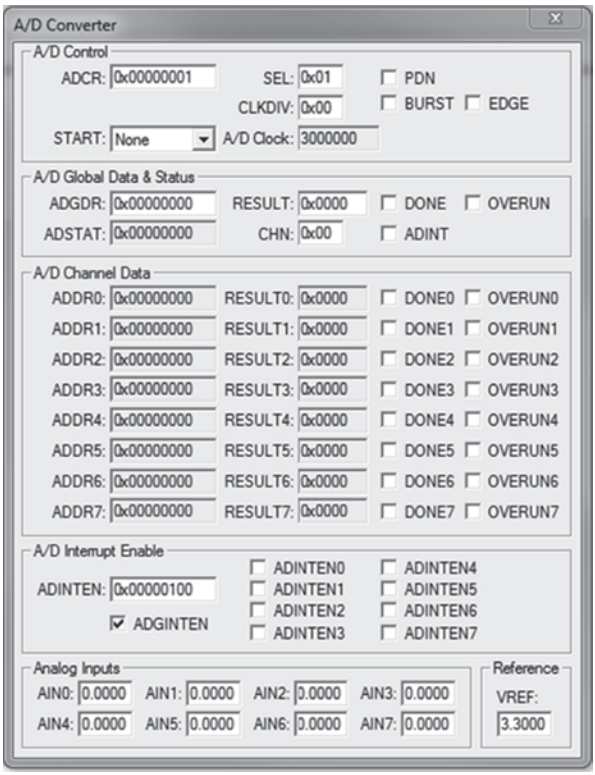


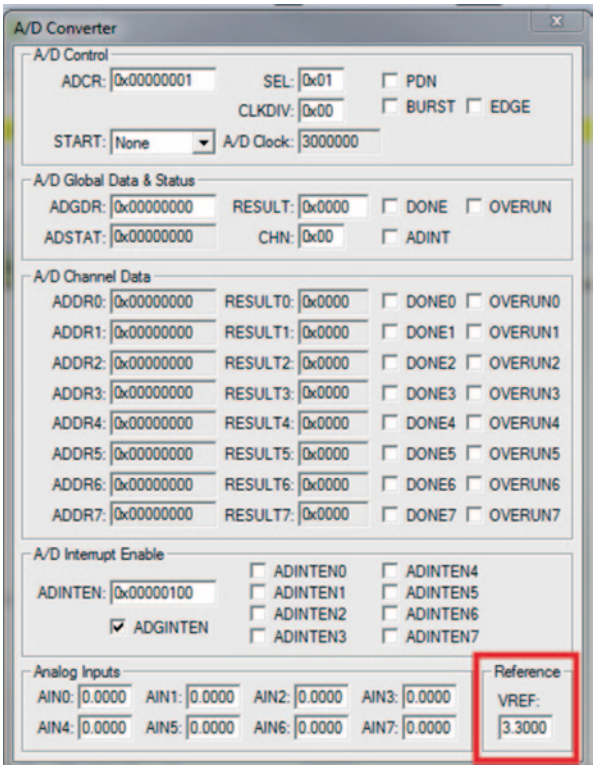
Fig. 7.5 LPC1768 Simulator A/C peripheral view

Figure 7.5 show LPC1786 A/D simulation, it can be observed the contents of A/D simulator while program in debug mode.

Programming Example—Voltmeter This program is best run on the actual NXP device, with a potentiometer connected to ADC0. The LEDs will light up based on the input voltage of the ADC pin.

The program may be run in debug mode by opening the ADC peripheral (**Peripheral- > A/D Converter).**

Manipulate the virtual input voltage by changing the value in the VREF box before stepping through the *read* section of the code. Observe the value of RESULT.



```
        AREA AD_Converter, CODE, READONLY
        ENTRY
        EXPORT SystemInit
        EXPORT __main

; ADC Registers
PCONP_R             EQU         0x400FC0C4
PCLKSEL0_R          EQU         0x400FC1A8
PINSEL1_R           EQU         0x4002C004

AD0CR_R             EQU         0x40034000
AD0GDR_R            EQU         0x40034004

VOLTAGE_3           EQU         0xE8B
VOLTAGE_2           EQU         0x9B2
VOLTAGE_1           EQU         0x4D9
VOLTAGE_0           EQU         0x000

; LED Registers
PINSEL3             EQU         0x4002C00C
FIO1DIR             EQU         0x2009C020
FIO1MASK            EQU         0x2009C030
FIO1PIN             EQU         0x2009C034
FIO1SET             EQU         0x2009C038
FIO1CLR             EQU         0x2009C03C

LED_1               EQU         0x800000
LED_2               EQU         0x200000
LED_3               EQU         0x100000

SystemInit

   ; ADC CONFIG
   ; Enable power for ADC
   LDR  R0, =PCONP_R
   LDR R1, [R0]
   ORR  R1, R1, #0x1000
   STR R1, [R0]

   ; Enable operational mode by setting Power Down (PDN) bit in AD0CR
   LDR  R0, =AD0CR_R
   LDR R1, [R0]
   ORR  R1, R1, #0x200000
   STR R1, [R0]

   ; Enable peripheral clock for ADC
   LDR  R0, =PCLKSEL0_R
   LDR R1, [R0]
   ORR  R1, R1, #0x30000
   STR R1, [R0]

; Select pin P0.23 to use for ADC reading.  Alt function will be
; AD0.0
   LDR  R0, =PINSEL1_R
   LDR R1, [R0]
      ORR  R1, R1, #0x4000
      STR R1, [R0]

      ; Set AD0.0 in the SEL bits fo AD0CR_R
      LDR  R0, =Ad0CR_R
```

```
LDR R1, [R0]
   BIC R1, R1, #0xFF
   ORR  R1, R1, #0x1
   STR R1, [R0]

   ; GPIO CONFIG
   ; Set the pin function for pin18, 20, 21, 23
   LDR R0, =PINSEL3
   LDR R1, [R0]
   MOV R2, #0x6F30
   BIC R1, R1, R2
   STR R1, [R0]

   ; Set the direction of pin18, 20, 21, 23 to output
   LDR  R0, =FIO1DIR
   LDR R1, [R0]
   ORR R1, R1, #0xB40000
   STR R1, [R0]

   ; Set the mask to only allow R/W to pin18, 20, 21, and 23
   LDR R0, =FIO1MASK
   LDR R1, [R0]
   ORR R1, R1, #0xFFFFFFFF
   BIC R1, R1, #0xB40000
   STR R1, [R0]

__main

   LDR  R0, =AD0CR_R
   LDR  R2, =AD0GDR_R
   LDR  R7, =FIO1PIN
   LDR R8, [R7]
   MOV R6, #0xFFF        ; for isolating RESULT
   ; Start conversion
start
   ; Set the START bits to 001 to commence an A/D conversion
   LDR R1, [R0]
   ORR  R1, R1, #0x1000000
   STR R1, [R0]
   NOP
   NOP

read
   LDR R5, [R2]
   ; get RESULT into R5
   LSR  R5, R5, #4
   AND R5, R5, R6

   ; Toggle LED based on Voltage
   ; VOLTAGE_3 >= 3 Volts
   ; VOLTAGE_2 >= 2 Volts
   ; VOLTAGE_1 >= 1 Volts

   MOV  R1, #VOLTAGE_3
```

```
CMP     R5, R1
LDRGE   R1, =LED_3
STRGE   R1, [R7]
BGE             start
; greater than or equal to 2V
MOV R1, #VOLTAGE_2
CMP     R5, R1
LDRGE   R1, =LED_2
STRGE   R1, [R7]
BGE             start
; greater than or equal to 1V
MOV R1, #VOLTAGE_1
CMP     R5, R1
LDRGE   R1, =LED_1
STRGE   R1, [R7]
BGE             start
; less than 1V
MOV     R1, #0x0
STR     R1, [R7]
B       start


END
```

7.5 Lab #4: Digital to Analog Converter (DAC)

The function of DAC is to convert digital to Analog, DAC has wide range applications such as Audio Amplifier, Voice over IP, motor control, and CD player, Fig. 7.6 shows block diagram of 4 bit DAC

Reference Voltage (Vrf) Reference voltage determine the maximum Analog output voltage,

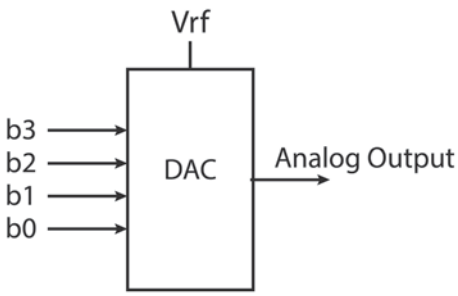


Fig. 7.6 Block diagram of 4 bit DAC

Resolution The resolution of DAC depend on number of inputs, the resolution for 4 bit DAC is define by

$R = Vrf/2^N$

Example The resolution of a 4 bit DAC with Vrf of 4 V is

$R = 4/2^4 = 0.25$ V,

This means that when input change from 0000 to 0001 the output change by 025 V

MBED DAC MBED contains 10 bit DAC and P0.26 represent the analog output pin, the voltage reference for DAC is 3.3 V. Following steps describe how to set DAC for operation

1. **Pin Function Select Register 1 (PINSEL1—0x4002 C004)**

Pin P0, 26 is used for DAC output and by setting bits b21b20 of PINSEL1 register to 01 will set P0, 26 as output of DAC

2. **Peripheral Clock Selection register 0 (PCLKSEL0—address 0x400F C1A8)**

The b23b22 is used to select clock for DAC

00	PCLK_peripheral = CCLK/4 00
01	PCLK_peripheral = CCLK
10	PCLK_peripheral = CCLK/2
11	PCLK_peripheral = CCLK/8,

3. **D/A Converter Register (DACR—0x4008 C000)**

The b6 through b15 holds the digital value to be converted to Analog.

Following Program will set DAC for operation; the user can check output of DAC by Accessing DAC peripheral of Uvision simulator

```
AREA DAC_config, CODE, READONLY
        EXPORT SystemInit
        EXPORT __main

PINSEL1       EQU   0x4002C004
PCLKSEL0      EQU   0x400FC1A8


DACR          EQU    0x4008C000
CONST1        EQU    0x00100000
CONST2        DCD    0x000FFC3;   data Converted to Analog
SystemInit
__main


   LDR   R0, =PINSEL1
   LDR R1, =CONST1
   STR R1, [R0]

   ; Initialize the Clock Control Register
   ; by setting the CLKEN bit to 1
   LDR   R0, =PCLKSEL0
   LDR R1, [R0]
   ORR   R1, R1, #0x1
   STR R1, [R0]

   LDR   R0, =DACR
   ADR    R2, CONST2
   LDR R3, [R2]
SUB R4, R4, R4
   ADD R4, R4, R3, LSL #6   ; shift data 6 times to be in bits b6-b15
STR R4, [R0]
   END
```

The result of the above program is shown by Fig. 7.7 DAC simulation, as show in this figure

A. The 10 bits converted to analog is 0x3FF
B. The voltage reference is 3.3 V
C. The analog output is 3.2968 V
D. Error 3.3 − 3.2968 = 0.0032 V

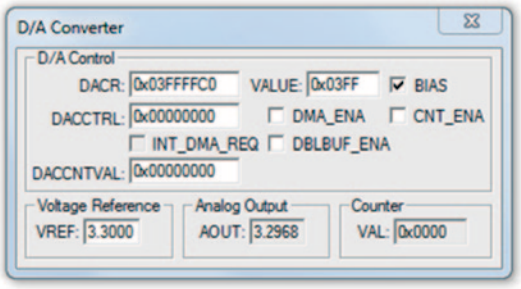


Fig. 7.7 D/A Converter simulation

7.6 Experiment #5: Binary to Hexadecimal Display

The objective of this lab is to read in a 4 bit binary number and display the number in hexadecimal on a 7-segment display.

Figure 7.8 shows the 4 inputs to Port P0 and the 7 outputs from Port P0 to the 7-segment display. A 7-segment display consists of 7 LEDs that can be turned on with a logical one. The pin P0/4 will be connected to the LED marked 'a' on the 7-segment display, pin P0/5 will be connected to 'b'…and pin P0/11 will be connected to 'g'.

Table 7.7 shows the input values and output values of P0 with the corresponding display values.

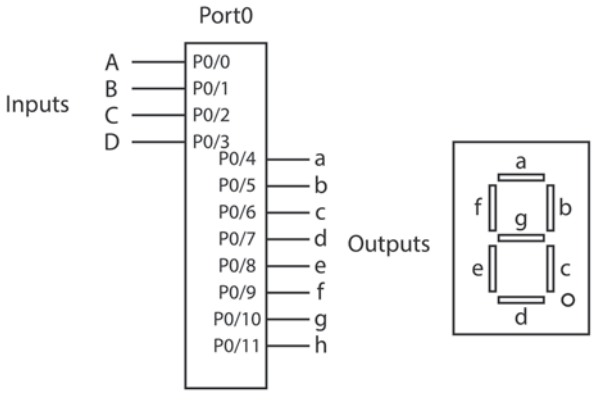


Fig. 7.8 Connection between port and 7-segment display

Table 7.7 Binary Input and port output

Input	Output of P0	Display
ABCD	g f e d c b a	
0000	0 1 1 1 1 1 1 (0x3F)	0
0001	0 0 0 0 1 1 0 (0x30)	1
0010	1 0 1 1 0 1 1 (0x5B)	2
0011	1 0 0 1 1 1 1 (0x4F)	3
0100	1 1 0 0 1 1 0 (0x66)	4
0101	1 1 0 1 1 0 1 (0x6D)	5
0110	1 1 1 1 1 0 1 (0x7D)	6
0111	0 0 0 0 1 1 1 (0x07)	7
1000	1 1 1 1 1 1 1 (0x7F)	8
1001	1 1 0 1 1 1 1 (0x6F)	9
1010	1 1 1 0 1 1 1 (0x77)	A
1011	1 1 1 1 1 0 0 (0x7C)	b
1100	0 1 1 1 0 0 1 (0x39)	c
1101	1 0 1 1 1 1 0 (0x5E)	d
1110	1 1 1 1 0 0 1 (0x79)	E
1111	1 1 1 0 0 0 1 (0x71)	F

```
          AREA  SEGMENT, CODE, READONLY
          EXPORT SystemInit
          EXPORT __main
          ENTRY
   ; HOW-TO
   ; Set input switches, run program, see number shown on seven segment
display

   PINSEL0            EQU        0x4002C000
   FIO0DIR            EQU        0x2009C000
   FIO0MASK           EQU        0x2009C010
   FIO0PIN            EQU        0x2009C014
   FIO0SET            EQU        0x2009C018
   FIO0CLR            EQU        0x2009C01C

   values DCB 0x3F,\
              0x30, \
              0x5B, \
              0x4F, \
              0x66, \
              0x6D, \
              0x7D, \
              0x07, \
              0x7F, \
              0x6F, \
              0x77, \
              0x7C, \
              0x39, \
              0x5E, \
              0x79, \
              0x71  \

   SystemInit

      ; Set the pin functions for pin0-11
      LDR R0, =PINSEL0
      LDR R1, [R0]
      MOV R2, #0xFFFFFF
      BIC R1, R1, R2    ; clear bits 0-11 to set to GPIO function
      STR R1, [R0]

      ; Set the direction of pin0-3 to input and pin4-10 input
      LDR  R0, =FIO0DIR
      LDR R1, [R0]
      BIC R1, R1, #0xF; set outputs
      ORR R1, R1, #0x7F0  ; set inputs
      STR R1, [R0]

      ; Set the mask to only allow R/W to pins0-10
      LDR R0, =FIO0MASK
      LDR R1, [R0]
      ORR R1, R1, #0xFFFFFFFF
      MOV R2, #0x7FF
      BIC R1, R1, R2
```

7.7 Universal Asynchronous Receiver/Transmitter (UART)

Introduction UART is a peripheral that handles serial communication between devices without using a clock for synchronization. It converts received serial data to parallel data and also translates parallel data to serial data for transmission; Fig. 7.9 shows the block diagram of a UART.

Receiving Data Bits (RX) The serial Data In register is populated by left shifting bits into itself based on the data on the input line, then the CPU reads the data in parallel from the register.

Transmitting Data Bit (TX) The CPU stores a byte of data in the Data Out Register and shift right bit by bit to the output line.

UART Baud Rate Defined by number of bits transmitted in one second. Popular baud rates are 9600 bits/second and 115200 bits/second.

The common standard for the UART is RS232 or EIA 232, the Voltage level for RS-232 are ±3 to ±15 V, where +15 represent logical 0 and +15 V represents a logical 1.

Figure 7.10 shows UART frame format, it shows one start bit and two stop bits and 8 data bits

Start Bit Indicates start of transmission

Data Bits can be 5 to 8 bits

P (Parity bit) P is used for error detection

Stop bits It can be one or two bits represent end of data frame

Figure 7.11 shows connection between two devices using UART

Most Microcontrollers come with at least one UART. This experiment demonstrates how to configure a UART for operation on the MBED NXP LPC1768, which is equipped with 3 UARTs.

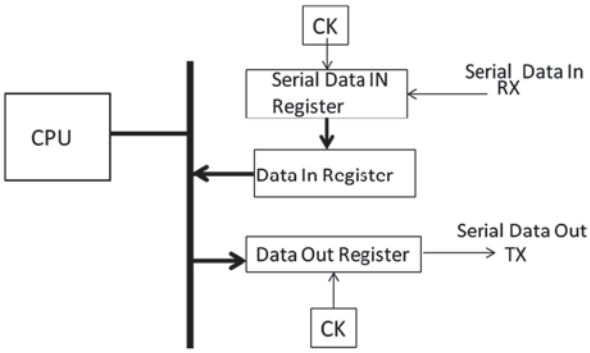


Fig. 7.9 Block diagram of UART

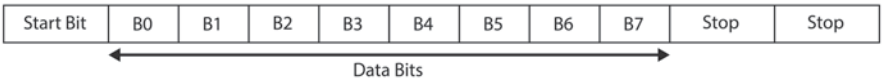


Fig. 7.10 USRT frame format

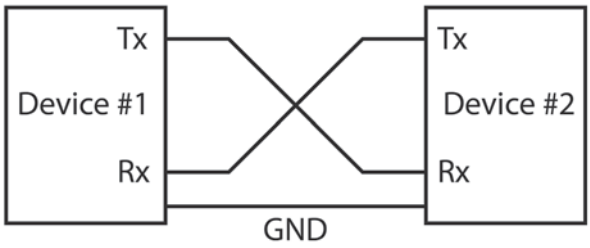


Fig. 7.11 Connection between two devices using USRT

Configuring Universal Asynchronous Receiver/Transmitter (UART) The following steps show how to configure UART0 in MBED

A. Pin Select Register—PINSEL0–0 × 4002C000
B. Power Control for Peripherals register (PCONP—0x400F C0C4)
C. Peripheral Clock Selection register (PCLKSEL0—0x400F C1A8)
D. UART Line Control Register (U0LCR)—0x4000 C00C
E. Setting the Baud rate of UART:
F. Configuring UART0 FIFO Control Register (U0FCR)—0x4000C008

Pin Select Register—PINSEL0–0 × 4002C000 Refereeing to Table 7.5 show pin selection register PINSEL0, the UART0, the UART0 uses P0.2 for TXD0 and pin P0.3 for RXDo, in order to uses these pins bits 4, 5, 6, and 7 of PINSEL0 Register must set to 0101

- **Power Control for Peripherals register (PCONP—0x400F C0C4)**

The PCONP register allows turning on and off selected peripheral function for the purpose of saving power. The bit thee (b3) of register PCONP is used for UART0, if this bit set to one the UART0 is enabled.

- **Peripheral Clock Selection Register (PCLKSEL0—0x400F C1A8)**

The bits b7b6 of PCLKSEL0 register is used to select clock rate for UART0 and offers following clock rates

B7b6	Clock Rate
00	CCLK/4
01	CCLK
10	CCLK/2
11	CCLK/8

Table 7.8 U0LCR register fields for UART0 [1]

Bit	Symbol	Value	Description	Reset Value
1:0	Word Length	00	5-bit	0
		01	6-bit	
		10	7-bit	
		11	8-bit	
2	Stop Bit	0	1 stop bit	0
		1	2 stop bits	
3	Parity Enable	0	Disable	0
		1	Enable	
5:4	Parity Select	00	Odd parity	0
		01	Event Parity	
		10	Forced "1"	
		11	Forced "0"	
6	Break Control	0	Disable	0
		1	Enable	
7	Divisor Latch	0	Disable	0
		1	Enable	
31:8	–	–	Reserved	–

- **UART Line Control Register (U0LCR)—0x4000 C00C**

The U0LCR is used to selecting format of the data such as number of bits in data, number of stop bits, and parity bit, each UART has one UARTn Line Control register, Table 7.8 show U0LCR register fields for UART0

By setting bit 7 to one will enable access to DLL and DLM register for setting the baud rate.

Setting the Baud Rate of UART The baud rate is calculated using following equation

Baud rate = System Clock/16 (256 * U0DLM + U0DLL)

U0DLM and U0SLM are called UART divisor latch and they use to decrease system clock to obtain proper baud rate. U0DLM and U0SLl each are 8 bits and combination of this two register are 16 bits. The baud rate and system clock are given, this equation is used to find the value for U0DLM and U0DLL

Example What are the value of UART latches for transmit date at 115200 baud, assume system clock of 8 MHZ

$115200 = 8 * 10^6/16$ (UART Divisor)

Table 7.9 UART0 FIFO Control Register (U0FCR)—0x4000C008

Bit	Symbol	Value	Description	Reset Value
0	FIFO Enable	0	Disabled	0
		1	Active Enable	
1	RX FIFO reset	0	No Impact	0
		1	Clear all bytes in Rx FIFO	
2	TX FIFO reset	0	No impact	0
		1	Clear all bytes in Tx FIFO	
3	DMA Mode Select		Selected by bit 0 (FIFO Enable)	0
5:4	–		Reserved	–
7:6	RX Trigger Level		Determines how many UART FIFO chars must be written	0
		00	before an interrupt or DMA request is activated.	
		01	1 character	
		10	4 characters	
		11	8 characters	
			14 characters	
31:8	–	–	Reserved	-

UART divisor=434, this number is converted to binary and the result of binary number is divided to MSB and LSB, U0DLL holds the LSB and U0DLM hold MSB

UART Divisor Latch LSB Register (U0DLL)—0x4000C000

Bit	Symbol	Description	Reset Value
7:0	DLLSB	Baud Rate	0×01

UART Divisor Latch MSB Register (U0DLL)—0x4000C004

Bit	Symbol	Description	Reset Value
7:0	DLMSB	Baud Rate	0×00

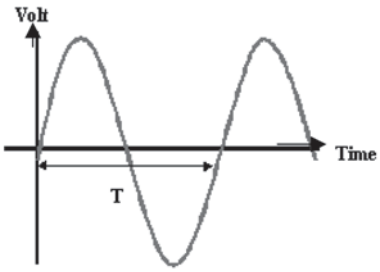
UART0 FIFO Control Register (U0FCR)—0x4000C008 Most UART has buffer can holds multiple byte f or transmission, the buffer operates based on First-In- First Out. Table 7.9 shows the U0FCR register

Solution to the Problems and Questions

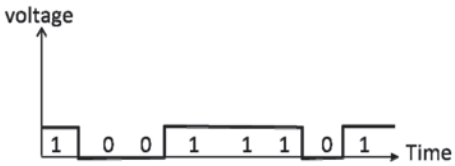
Chapter 1

Problems and Questions

1. Show an analog signal



2. Show a digital signal



A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments*,
DOI 10.1007/978-3-319-11704-1

3. Convert following decimal numbers to binary

```
a. 35

   100011
b. 85
   1010101
c. 23.25
   10111.01
```

4. Convert following binary numbers to decimal

```
a. 1111101
   125
b. 1010111.1011
   87.6875
c. 11111111
   2^8-1  = 255
d. 10000000
   128
```

5. Convert following Binary numbers to Hexadecimal

```
a. 1110011010
   39A
b. 1000100111
   227
c. 101111.101
   2F.A
```

6. Convert following number to binary

   a. $(3FDA)_{16}$ = 0011 1111 1101 1010
   b. $(FDA.5F)_{16}$ = 1111 1101 1010.0101 1111

7. Find two's complements of following numbers

```
a.  11111111
    00000001
b.  10110000
    01010000
c.  10000000
    10000000
d.  00000000
    00000000
```

8. Convert the word "LOGIC" to ASCII then represent each character in hex

```
  L         O          G          I         C
1001100 1001111   1000111   1001001  1000011   ASCII
 4C        4F              47       49      43       Hex
```

9. Subtract following numbers using two's complement

```
a.11110011 – 11000011
  Two's complement of     11000011    is   00111101
  11110011 + 00111101 = 1 00110000, discard carry then result is + 00110000
b.10001101 – 11111000
  Tows complement of  11111000 =   00001000
  10001101 + 00001000 = 10010101 result does not produce carry then
  Tow's complement of  10010101  =  - 01101011
```

10. List the types of transmission modes.
    Asynchronous Transmission and Synchronous Transmission
11. Why does a synchronous transmission require a clock?
    Synchronous transmission use clock for synchronization (clock is used to represent speed of data)
12. What is frequency of an Analog signal repeated every 0.05 ms
    $F = 1/T = 1/0.05 * 10^{-3} = 20$ KHz

Chapter 2

Problem

1. If A= 11001011 and B= 10101110 then, what is the value of following operation

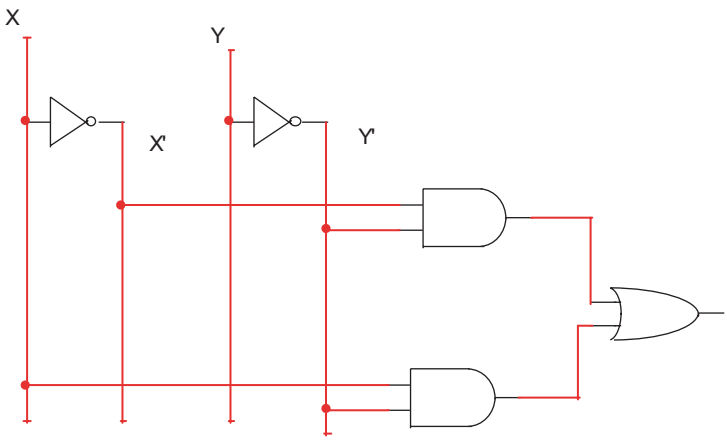
```
a. A AND B
    10001010
b. A OR B
    11101111
```

2. If A=11001011 and B=10101110, what is the value of following Operations

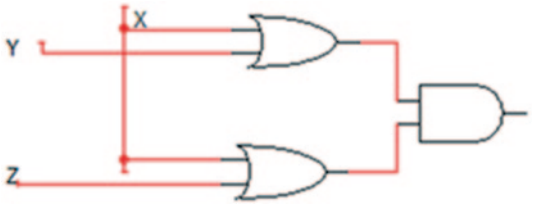
```
a.  A NOT
    00110100
b.  A XOR B
    0110010
c.  A AND OF
    00001011
d.  A AND F0
    11000000
```

3. Draw logic circuit for following functions

A. F (X,Y,Z) = X'Y' +X Z'



B. F(X, Y,Z) = (X + Y) (X+Z)



4. Find the truth table for following function

$$F(X,Y,Z) = XY' + YZ + XZ'$$

XYZ	XY'	YZ	XZ'	F
000	0	0	0	0
010	0	0	0	0
011	0	0	0	0
100	1	0	1	1
101	1	0	1	1
110	0	1	0	1
111	0	1	0	1

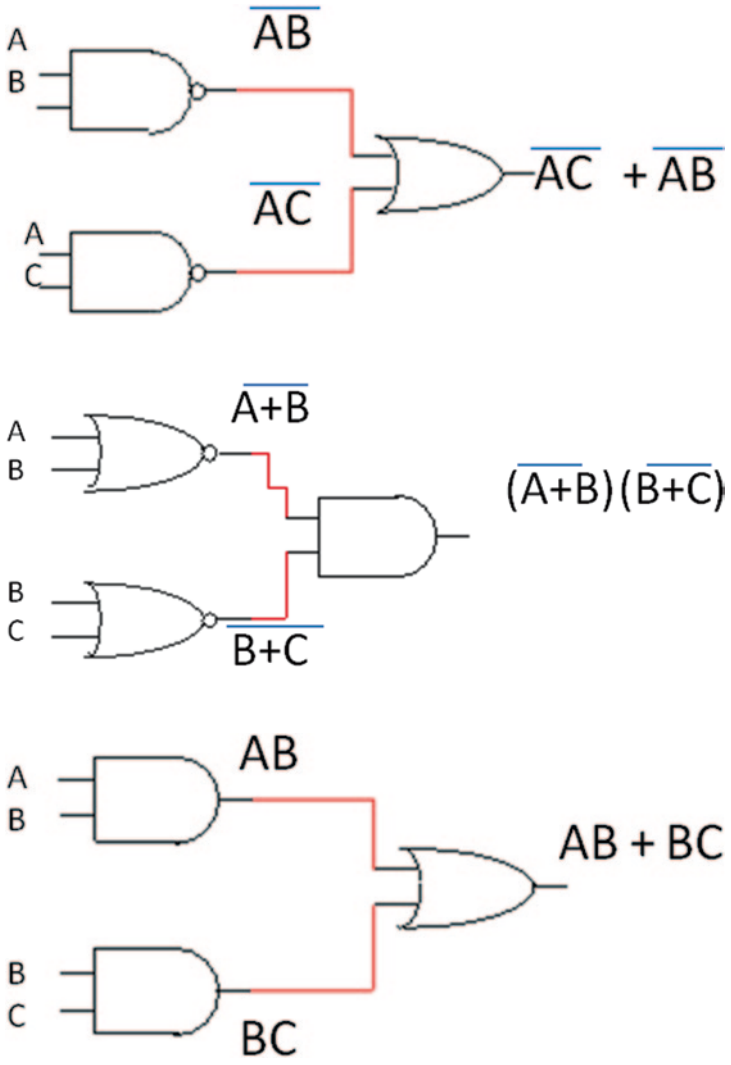
5. If A=10110110 and B= 01101100, then find

A. A NAND B
11011011

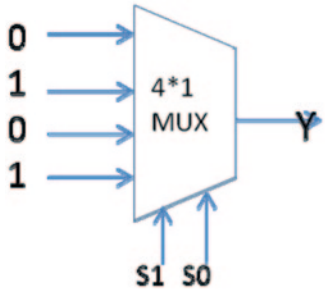
B. A NOR B
0000001

C. A XOR B
11011010

1. Show output of following logic circuits



1. Following multiplexer is given show the output



S1	S0	Y
0	0	0
0	1	1
1	0	0
1	1	1

Short Answer Questions

1. List the components of a microcomputer.
   CPU, Memory, Parallel port, Serial Port , and DMA
2. Explain the functions of a CPU.
   CPU execute instruction and control other components in a computer
3. List the functions of an ALU.
   Arithmetic and logic operation
4. What is the function of a control unit?
   Generates Control signal and execute instruction
5. What does RAM stand for?
   Random Access Memory
6. What is SRAM ? discuss its applications
   Static RAM and it used in Cache memory
7. Define DRAM and SDRAM and explain their applications.
   DRAM is Dynamic RAM
   SDRAM is Synchronous DRAM
8. Explain the function of an address bus and a data bus.
   The address BUS carry Address and data BUS carry data
9. What does IC stand for?
10. What is the capacity of a memory IC with 10 address lines and 8 data buses?
    2^{10} *8 bits or 2^{10} =1024 bytes
11. What is ROM?
    Read Only Memory
12. What does EEPROM stand for, and what is its application?
    Electrically Erasable ROM. It used in flash drive
13. What does RDRAM stand for?
    Rambus DRAM
14. What is SIMM?
    Single In-Line Memory Module

15. Explain the function of cache memory and give its location.
    Cache memory is fastest memory and reside in CPU
16. What is the application of a parallel port?
    Printer with parallel port
17. What is the application of a serial port?
    COM1, RS232
18. Explain the difference between CISC processors and RISC processors
    CISC has variable instrucrion format, less registers, Control Uni is microcode
    RISC has fixed instruction format, control unit is hardware, uses only Load and store instructions to access memory
19. Explain difference between Von Neumann and Harvard Architecture
    Van Neumann uses of BUS for transferring Data and Instruction
    Harvard Architecture uses separate BUS for Data and Instruction

Chapter 3

Problems

1. What is contents of R5 after execution of following instruction, assume R2 contains 0X34560701 and R3 contains 0X56745670

```
a.  ADD R5, R2, R3
    R5=0x8ACA5D71

b.  AND R5, R3, R2
    R5= 0x14540600

c.  XOR R5, R2,R3
    R5=0x66225171
d.  ADD R5, R3, #0x45
    R5=0X567456B5
```

2. What is contents of R1? assume R2= 0x00001234

```
a.  MOV  R1, R2,  LSL #4
   R1=  0x00012340

b.  MOV  R1, R2,  LSR #4
   R1 = 0x00000123
```

3. What is difference between these two instructions?

```
a. SUBS   R1, R2, R2
b. SUB R1,R2, R2
```

Question a does not change bits in PSR register, question b will change bits in PSR

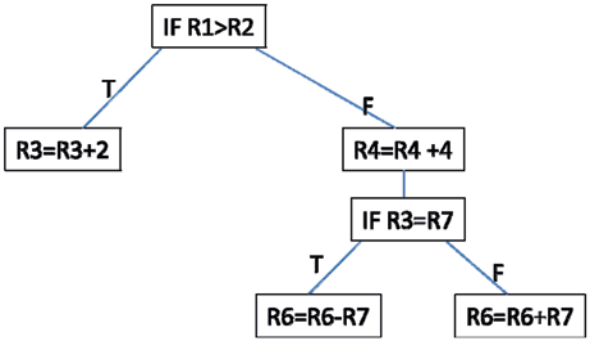
4. Convert following HLL language to ARM instructions

```
IF R1>R2 AND R3>R4 then

 R1= R1 +1
Else
R3=R3 +R5*8
Endif
```

5. Convert following HLL language to ARM instructions

```
IF R1>R2 OR R3>R4 then
R1= R1 +1
Else
R3=R3 +R5*8
Endif
```

6. Convert following flowchart to ARM assembly language



7. Write a program to add ten numbers from 0 to 10 or Convert following C language to ARM assembly Language

```
int    sum;
       int i;
sum = 0;
for   (i  =  10 ;  i  > 0 ;   i  - -  ){
sum  =  sum  +1
}
```

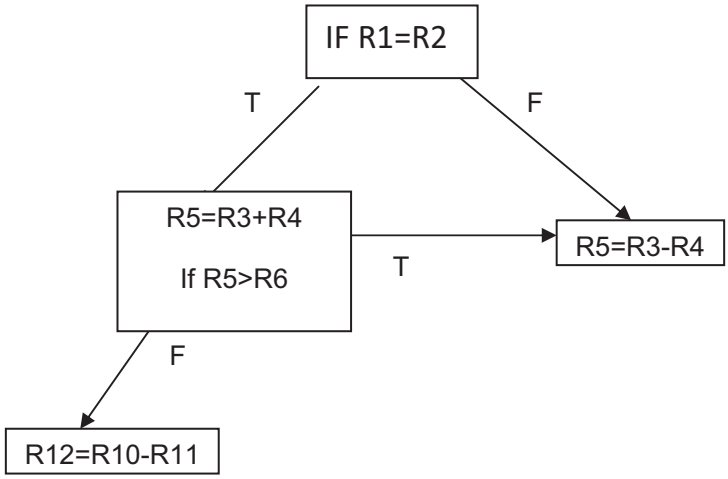
8. Write a program to convert following HLL to ARM assembly

```
a= 10;
b=45;
while ( a! =b ) {
if   a <b    then;
a   =   a  +5;
else ;
b= b+5;
 }
```

9. Convert following HLL to ARM assembly

```
IF R1>R2 AND R3>R4 then
R1= R1 +1
Else
R3=R3 +R5*8
Endif
```

10. Convert Following Flow Chart to ARM Assembly



Chapter 4

Problem

1. Trace following instructions, assume list start at memory location 0x0000018 and using ARM Big Indian

```
ADR  R0, LIST  ; Load R0 with address of memory location List

MOV  R10, #0x2

a. LDR R1, [R0]                  ;R0= 0x18          R1=0x34F532E5

b. LDR R2, [R0, #4]!             ;R0= 0x1C          R2=0x010208FE

c. LDRB  R3, [R0] , #1           ;R0= 0x19          R3=0x34

d. LDRB  R4, [R0 , R10]!         ;R0= 0x1A          R4=0X32

e. LDRSB R5, [R0],  #1           ;R0= 0x19          R5=0x34

f. LDRSH R6, [R0]                ;R0= 0x18          R6=0x34F5


LIST  DCB      0x34, 0xF5, 0x32, 0xE5, 0x01, 0x02,0x8,0xFE
```

2. Work problem #1 part A and B using Little Endian

```
a. R1= 0xE532F534
b. R2= 0xFE080201
```

3. What is contents of register R5 after execution following program

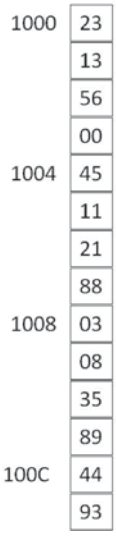
```
ADR R0, LIST

LDRSB R7, [R0]

LIST   DC  0xF5

R7= 0xFFFFFFF5
```

4. What is contents of register Ri for following load Instructions, assume R0 hold the address of list using little Endian

```
a. LDR R1, [R0]                          ;R1=0xE532F534
b. LDRH R2, [R0]                         ;R2=0x0000F534
c. LDRB  R3, [R0] , #1                   ;R3=0x00000034
d. LDRB  R4, [R0]                        ;R4=0x000000F5
e. LDRSB R5, [R0],  #1                   ;R5=0xFFFFFFF5
f. LDRSH  R6, [R0]                       ;R6=0xFFFFE532


List DCB   0x34, 0xF5, 0x32, 0xE5, 0x01, 0x02
```

5. Following memory is given, show the contents of each register, assume R1 = 0x0001000 and R2 = 0x00000004 (use Little Endian)

```
a. LDR   R0,  ( R1)          R0 = 0x00561323
b. LDR   R0  , (R1, #4)      R0=0x88211145
c. LDR   R0  , (R1, R2)      R0=0x88211145

d. LDR   R0 , (R1, #4)!      R0= 0x88211145      R1=0x1004
```



6. What is effective address and contains of R5 after executing following instructions ? assume R5 contains 0x 18 and r6 contains 0X00000020

```
A. STR   R4, [R5]            EA=  0x18
B. STR   R4, [R5, #4]        EA= 0x18 + 4= 0x1C
C. STR   R4, [R5, #8]        EA=0x18 +8=0x20
D. STR   R4, [R5, R6]        EA= 0x18 +0x20 = 0x38
E. STR   R4, [R5], #4        EA= 0x18 , R5=0x18 +4=0x1C
```

Chapter 5

1. Write a program to add elements of list1 and store in List2

```
LIST 1  DCB  0x23, 0x45, 0X23  ,0x11
```

2. Write a program to find the largest number and store it in memory location LIST3, Assume Numbers are in location LIST1 and LIST2

3. Write a program to add data in memory location LIST and store the SUM in memory location Sum.

4. Write a program to Add two number , the number represented by

```
N1    EQU   5
M1    EQU   7
```

5. Write assembly language for following HLL

```
IF R1 = R0
Then
ADD R3, R0, #5
Else
SUB R3, R0, #5
```

6. Write a program to read memory location LIST1 and LIST2 and them then store LIST3

7. Move two 32 bits number to R1 and R2 and add the result

```
LDR  R1, =0x22222222
LDR R2, =0x33333333
ADD R3, R1,R2
```

8. Write a program to multiplying two numbers

9. Write a program to add 8 numbers using Indirect addressing

```
LIST DCB 0x5, 0x2,0x6,0x7 ,0x9,0x1,0x2,0x08
```

10. Write a program to add 8 numbers using Post Index addressing

```
LIST DCB 0x5, 0x2,0x6,0x7 ,0x9,0x1,0x2,0x08
```

11. Write a program to convert following HLL language to ARM instructions

```
IF R1=R2 AND R3>R4 then

R1= R1 +1
Else
R3=R3 +R3*8
Endif
```

12. What is Contents of R4 after Execution of following Program

```
AREA  NAME, CODE, READONLY
            EXPORT SystemInit
            EXPORT __main
            ENTRY
SystemInit


__main
      LDR R1, =0xFF00FF00
      ADR R0, LIST1
      LDR R2, [R0]
      AND R4, R2, R1


LIST1   DCD 0X45073487
   END
```

13. Write a program to convert following HLL to assembly language

```
If  R1=R2 then

R3= R3+1

IF R1<R2 Then

R3=R3-1

If R1>R2 Then

R3=R3-5
```

14. Write a subroutine to calculate value of Y where Y = X*2 + x + 5, assume x represented by

N1 EQU 0x5

15. Write a program to rotate R1 16 times assume R1 contains 0x12345678

16. Write a program to compare two numbers and store largest number in a memory location LIST

17. Write a program to read a word memory location LIST and Clear bit position B4 through B7 of register R5, assume R5 contains 0XFFFFFFF

```
LDR R0, =0x000000F0

LDR R5, =0xFFFFFFFF
```

18. Write program to load Register R1, R2, R3, and R4 from memory location LIST

References

1. NXP Corp LPC16XX user manual
2. http://infocenter.arm.com. ARM V7 manual
3. Keil Corp, Uvission Development tool
4. NXP Cop, Rapid prototyping for the LPC1768 MCU
5. MBED Microcontroller. https://mbed.org
6. ARM Cortex-M3 Technical Reference Manual
7. Furber SB (2000) ARM system-on-chip architecture. Adison Wesly
8. Holm W (2009) ARM assembly language. CRC Press
9. Schindler K (2013) Introduction to microprocessor based system using the ARM processor. Pearson
10. Clements A (2014) Computer organization and architecture themes and variations. Cengage Learning
11. Valvano JW (2011) Embeded systems real-time interfacing to the ARN Cortex-M3
12. Lewis D (2013) Fundamentals of embedded software with ARMCotex-M3. Pearson
13. Gibson R (2007) ARM assembly language–an introduction. Lulu

A. Elahi, T. Arjeski, *ARM Assembly Language with Hardware Experiments*,
DOI 10.1007/978-3-319-11704-1