Chapter 1

Introduction

1.1 Introduction

The Intel 4004, developed in 1972 by Intel Corporation, was the first commercial microprocessor. The 4004 was a 4-bit device and it opened up a whole new era in terms of advances in technology. Many other microprocessors were later developed by companies such as Zilog, Motorola, Intel, Texas Instruments, and many other companies which realized the enormous potential of the microprocessor. Today we can find a microprocessor in almost any device imaginable, from a simple and inexpensive arithmetic calculator to the most sophisticated and powerful computer available in the market.

Being such a ubiquitous device, it is not a coincidence that more and more people today need to understand the basics of how a microprocessor works in order to harness and exploit its capacity to do almost anything we want. Among these people we find a range of interested individuals starting with the casual hobbyists, electronic technology and engineering students, up to the experienced computer professionals and engineers.

This book is written with the dual purpose that it will be easy to understand by the most casual reader and at the same time be a very useful reference to the most experienced professional. Throughout the years we have encounter many good microprocessor books. However, most of these books share the same basic problem: they were written for one specific microprocessor and its usefulness is thus limited to the scope of the device they present. Knowing this, we have set out to write a microprocessor and microcomputer book that will stand the test of time. Each chapter will be full of concepts and practices that can be used regardless of the specific device. At the end of each chapter we will present sections specific to several of the most used devices today. The latter is done so that our reader has the chance to put in practice what is presented in the book in order to reinforce the concepts learned.

Having said this we have only one more thing to say: Prepare for a most gratifying experience while learning how to unleash the power of the micropro-


1.2 Number Systems

In order to understand how a microprocessor and a microcomputer work it is necessary to understand the way they do things. In this section we present several number systems that are commonly used with microprocessors, i.e. the decimal or traditional number system, the binary number system, the octal number system, and the hexadecimal number system. Only unsigned numbers are discussed when presenting each of the number systems listed above.

1.2.1 Decimal Number System

The traditional number system is most commonly known as the decimal system because it has 10 as its base or radix. This system has 10 digits, e.g. 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Numbers are composed of combinations of these ten digits. When we run out of single-digit numbers we just add another digit and form double-digit numbers, and when we run out of double-digit numbers we form three-digit numbers, and so on, i.e. 0, 1, ..., 9, 10, 11, ..., 99, 100, and so on. Any number in the decimal number system can be expressed as a sum of powers of ten, e.g.

\[
11 = 1 \times 10^1 + 1 \times 10^0,
\]

\[
32 = 2 \times 10^1 + 3 \times 10^0,
\]

\[
578 = 8 \times 10^2 + 7 \times 10^1 + 5 \times 10^0,
\]

etc. Note that this is just the same as saying that a number in decimal can be decomposed in terms of its units, tens, hundreds, thousands, etc.

When a number has a fixed number of digits we say that the number has n-digits. For example, if n is 4 we can obtain any positive number between 0001 and 9999 and the number 0000. Humans usually discard the zeros to the left and thus 0001 becomes 1, 0002 becomes 2, and so on. By appending a minus sign to a positive number we can obtain the negative of a number. Thus minus one is represented as -1, minus 2 as -2, etc. Though we may not have known it, this way of producing negative numbers out of positive numbers is called the sign-magnitude method. The sign-magnitude method of expressing negative numbers is very handy and natural for humans, but as we shall see later, it is not fit for computers.

Arithmetic operations on the set of numbers is allowed. This arithmetic operations on the numbers usually result in other numbers. We use the word usually because there are some arithmetic operations whose results can not represented by any of the allowable numbers, e.g. division by zero.

It is very important to understand that, although humans are very generous in extending n whenever necessary, a microprocessor, and thus a microcomputer, has a fixed number of digits that can not be extended beyond certain point.
1.2. NUMBER SYSTEMS

Thus, if \( n \) is 4 and we add 1 to the number 9999, as humans we usually do not care about the resulting overflow and just say that the result of the previous addition is 10000. As mentioned earlier, a microcomputer which has a fixed number of digits, simply can not do this at will and, if \( n \) were 4 the result of the previous operation will not be 10000, but some other number, i.e. 0 if using addition modulo 10000,

\[
\begin{array}{c}
9999 \\
+ 1 \\
\hline
0
\end{array}
\]

or 1 if the addition is modulo 9999, i.e.

\[
\begin{array}{c}
9999 \\
+ 1 \\
\hline
1
\end{array}
\]

1.2.2 Binary Number System

The binary number system has a radix \( r = 2 \) and thus it has only two binary digits or bits, i.e. 0 and 1. Everything in a computer is stored and handled as a binary sequence regardless of the way it was written. This means that all numbers, characters, symbols, etc. are all stored and handled as binary sequences. For now we will only focus on the numbers.

Numbers in the binary number system can be expressed pretty much in the same way as we did with the decimal number, just remember that the base is 2 and that we only have two digits. Thus, decimal 0 is expressed as 0, decimal 1 as 1, 2 is expressed as 10, 3 as 11, 4 as 100, and so on. In order to avoid the ambiguity between decimal 10 and the binary number 10 (decimal 2), decimal 11 and the binary number 11 (decimal 3), etc, we shall use the convention of appending the base as a subscript to the number. In case we do not append the base to the number, we shall assume that base 10 is meant.

Thus

\[
\begin{align*}
0 &= 0_{10} = 0_2, \\
2 &= 2_{10} = 10_2, \\
3 &= 3_{10} = 11_2,
\end{align*}
\]

and so on. Note that just as we did with the decimal number system, we just add a digit once we run out of single digit binary numbers, another digit once we run out of double-digit binary numbers, and so on. Thus, in binary we would count 0, 1, 10, 11, 100, etc. Note that even numbers in the binary number system have a 0 at their least significant bit (lsb) position, whereas odd numbers have a 1.

In order to convert a binary number into its equivalent decimal number we can just write the binary number as a sum of powers of 2. Thus

\[
\begin{align*}
11_2 &= 1 \times 2^0 + 1 \times 2^1 = 1 + 1 = 2, \\
100_2 &= 0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 = 0 + 0 + 4 = 4,
\end{align*}
\]
\[ 111010_2 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6 = 1 + 0 + 4 + 0 + 16 + 32 + 64 = 117. \]

There are analogous procedures for converting a number from decimal into its equivalent binary sequence. A procedure that could be used here is to divide the original decimal number by two, record the remainder of the division as the least significant binary digit or bit. Now take the residue and divide it by two, record the remainder as the next significant bit. This procedure has to be repeated until the residue is zero. Thus, if we want to convert number 4 into a binary number we would divide 4 by 2. The result is a residue of two (2) and a remainder of zero (0). Dividing next the residue of 2 by 2 we obtain a residue of one (1) and a remainder of zero (0). Now divide this residue by 2 and we get a residue of 0 and a remainder of one (1). If we put together the binary sequence as explained above we obtain 100 as the equivalent binary sequence of the decimal number four (4), i.e. \( 4 = 100_2 \).

1.2.3 Octal Number System

The octal number system has a radix \( r = 8 \) and thus it has eight octal digits, i.e. 0, 1, 2, \ldots, 7. Numbers in the octal number system can be expressed pretty much in the same way as we did before, but remember that the base is 8 and that we have eight digits. Thus, decimal 0 is expressed as 0, decimal 1 as 1, 2 is expressed as 2, until we arrive at decimal numbers above seven (7). Here we have to remember what we do with decimal and binary numbers once we run out of single-digit numbers, double-digit numbers, etc. Thus \( 8 = 10_8 \), \( 9 = 11_8 \), \( 10 = 100_8 \), and so on.

In order to convert a octal number into its equivalent decimal number we can just write the octal number as a sum of powers of 8. Thus

\[
10_8 = 0 \times 8^0 + 1 \times 8^1 = 0 + 8 = 8, \\
11_8 = 1 \times 8^0 + 1 \times 8^1 = 1 + 8 = 9, \\
1765_8 = 5 \times 8^0 + 6 \times 8^1 + 7 \times 8^2 + 1 \times 8^3 \\
= 5 + 48 + 448 + 512 = 1013.
\]

There are also analogous procedures for converting a number from decimal into its equivalent octal sequence. We could perform a procedure similar to the procedure we used for converting a decimal number into a binary number. However, an easier method is available. Note that \( 8 = 2^3 \). This means that every octal digit is equivalent to three binary digits. Thus, we could convert the decimal number into a binary number and then convert the binary number into its equivalent octal number.

For example, to convert number 26 into an octal number we find its binary equivalency, i.e. 11010\(_2\), and now form groups of three bits starting from the least significant bit (from right to left). When attempting to do this we shall soon realize that we will end up with a group of three bits (010) and a group of two bits (11). At this point we need to remember that there are many zeroes to
1.2. NUMBER SYSTEMS

the left of the leftmost 1 in 11, we just need to use one of them. Thus, 11010₂
for our purposes becomes

\[
\begin{array}{c}
\begin{array}{c}
\text{3} \\
\text{011} \\
\text{010}
\end{array}
\end{array}
\]

This means that we have two groups of three bits each, namely 011 and 010.
Now note that 011₂ = 3, and 010₂ = 2. Thus, 2₆ = 3₂₈. We can check this
result: 3₂₈ = 2 * 8⁰ + 3 * 8¹ = 2 + 24 = 2₆.

1.2.4 Hexadecimal Number System

The hexadecimal number system has a radix \( r = 16 \) and thus it has sixteen
hexadecimal digits, i.e. 0, 1, 2, \ldots, 9, A, B, C, D, E, F. Note that since there is
no single digit number above 9, we are using letter A as number 10, B as number
11, C as number 12, D as number 13, E as number 14, and F as number 15.
Numbers in the hexadecimal number system can be expressed pretty much in
the same way as we did before, but remember that the base is 16 and that
we have sixteen digits. Thus, decimal 0 is expressed as 0, decimal 1 as 1, 2 is
expressed as 2, until we arrive at decimal numbers above 15 (F). Here we have
to remember what we do with decimal numbers once we run out of single-digit
numbers, double-digit numbers, etc. Thus

\[
\begin{align*}
16 &= 10_{16} \\
17 &= 11_{16} \\
26 &= 1A_{16}
\end{align*}
\]

and so on.

In order to convert a hexadecimal number into its equivalent decimal number
we can just write the hexadecimal number as a sum of powers of 16. Thus

\[
\begin{align*}
10_{16} &= 0 \times 16^0 + 1 \times 16^1 = 0 + 16 = 16, \\
11_{16} &= 1 \times 16^0 + 1 \times 16^1 = 1 + 16 = 17, \\
1A_{16} &= A \times 16^0 + 1 \times 16^1 = A + 16 = 10 + 16 = 26.
\end{align*}
\]

There are also analogous procedures for converting a number from decimal
into its equivalent hexadecimal sequence. We could perform a procedure similar
to the procedure we used for converting a decimal number into a binary number.
However, an easier method is available. Note that 16 = 2⁴. This means that
every hexadecimal digit is equivalent to four binary digits. Thus, we could
convert the decimal number into a binary number and then convert the binary
number into its equivalent hexadecimal number.

For example, to convert number 26 into a hexadecimal number we find its
binary equivalency, i.e. 11010₂, and now form groups of four bits starting from
the least significant bit (from right to left). Remember that there are many
zeroes to the left of the leftmost 1, we just need to use three of them. Thus, $11010_2$ for our purposes becomes

$$
1001 \ 1010 \quad_{10}
$$

This means that we have two groups of four bits each, namely 0001 and 1010. Now note that $0001_2 = 1$, and $1010_2 = 10$. Thus, $26 = 1A_{16}$ which is the same result we got before.

The following table shows the relationship between the first 17 numbers in each of the number systems used in this chapter so far. A bold entree indicates that the number is also one of the single digit numbers that make up the corresponding system.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

### 1.3 2’s Complement Numbers

Up to this moment we have only studied how to represent non-negative numbers in all the different number systems we have introduced so far. However, we need to learn how to represent negative numbers in the binary number system since, regardless of the number system used, all computations performed in a microprocessor based system are performed in the binary number system. Negative numbers in the binary number system can be represented using the 2’s complement system.

In the 2’s complement system we take the most significant bit (msb) position in the binary stream of bits and use it to indicate whether the number is positive.
1.3. 2’S COMPLEMENT NUMBERS

(the msb is zero) or negative (the msb is one). Thus, if \( A \) is a binary number of \( n \) bits, then both \( A \) and its negative \(-A\) can be represented as follows

\[
\begin{align*}
A &= (\overset{\text{msb}}{m} \ a_{n-2}a_{n-3}\ldots a_1a_0)_2 \\
-A &= (\overset{\text{msb}}{m} \ \bar{a}_{n-2}\bar{a}_{n-3}\ldots \bar{a}_1\bar{a_0})_2 + 1
\end{align*}
\]

where \( a_i \) is used to represent bit position \( i \) in number \( A \) and \( \bar{a}_i = 1 - a_i, 0 \leq i \leq n - 2 \). Using the definition of the complement of a bit, the negative (or complement) of a number in 2’s complement is found to be

\[ -A = 2^n - A. \tag{1.1} \]

In order to prove the above result we will use the most general case of the negative of a number in \( r \)’s complement, where \( r \) is the radix of the number system. In this case \( a_i = (r - 1) - a_i \).

**Proof:**

\[
egin{align*}
-A &= (\overset{\text{msb}}{m} \ (r - 1)a_{n-2}a_{n-3}\ldots (r - 1)a_1(a_0)_r + 1 \\
\quad &= (r - 1)r^{n-1} + [(r - 1) - a_{n-2}]r^{n-2} + [(r - 1) - a_{n-3}]r^{n-3} \\
\quad &\quad + \ldots + [(r - 1) - a_1]r^1 + [(r - 1) - a_0]r^0 + 1 \\
\quad &= (r - 1)r^{n-1} + (r - 1)r^{n-2} + (r - 1)r^{n-3} + \ldots + (r - 1)r^1 + (r - 1)r^0 - A \\
\quad &\quad + \frac{[\alpha_{n-2}]r^{n-2} + [\alpha_{n-3}]r^{n-3} + \ldots + [\alpha_1]r^1 + [\alpha_0]r^0 + 1}{\frac{r^{n-1}}{r - 1} + A + 1} \\
\quad &= (r - 1) \left[ \frac{r^{n-1} + r^{n-2} + r^{n-3} + \ldots + r^1 + r^0}{r - 1} \right] - A + 1 \\
\quad &= r^n - A + 1 - A + 1 \\
\quad &= r^n - A \tag{1.2}
\end{align*}
\]

If we now substitute \( r = 2 \) we get equation 1.1 which completes our proof. Note that, in equation 1.2, \( r^n - 1 - A \) is the formula for obtaining the \( (r-1)’s \ complement \), or the 1’s complement if \( r = 2 \). Thus, if we use equation 1.1 to find the negative of a number \( A \) in the 2’s complement representation, we first find the 1’s complement and then add one.

Say we were looking for the negative of \( A = 5 \) for \( n = 4 \) and \( r = 2 \). Using equation 1.2 we just need to subtract 5 from 2\(^4\) to obtain the result 16 - 5 = 11. Thus, the negative representation of -6 in the 2’s complement form is the same as the binary representation for number 11, i.e. 1011\(_2\) for \( n = 4 \), and \( r = 2 \). I.e.

\[
\begin{align*}
-A &= 2^4 - A \\
\quad &= 16 - 5 \\
\quad &= 11 \implies 1011\(_2\) = -5
\end{align*}
\]
If we first compute the 1’s complement to find the 2’s complement representation of -5 we would note that $A = 0101_2$, then complement each bit in $A$ to obtain $1010_2$ and then add 1 to obtain $-A = 1011$. I.e.

$$-A = 0101 + 1 = 1010 + 1 = 1011_2 = -5$$

which is the same result we obtained before.

In order to obtain the value of a number $A$ in 2’s complement we can use the following equation

$$A_v = \text{value of } A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i2^i \quad (1.3)$$

**Proof:**

$A > 0 : a_{n-1} = 0$

$$A = 0a_{n-2}a_{n-3} \ldots a_1a_0$$

$$A_v = 0 \times 2^{n-1} + a_{n-2}2^{n-2} + a_{n-3}2^{n-3} + \ldots + a_12^1 + a_02^0$$

$$= \sum_{i=0}^{n-2} a_i2^i$$

$A < 0 : a_{n-1} = 1$

$$A = 1a_{n-2}a_{n-3} \ldots a_1a_0$$

$$A_v = -(2^n - (a_{n-1} \times 2^{n-1} + a_{n-2}2^{n-2} + a_{n-3}2^{n-3} + \ldots + a_12^1 + a_02^0))$$

$$= -(2^n - a_{n-1}2^{n-1} - \sum_{i=0}^{n-2} a_i2^i)$$

$$= -(2^n - 2^{n-1} - \sum_{i=0}^{n-2} a_i2^i)$$

$$= -(2^{n-1} - \sum_{i=0}^{n-2} a_i2^i)$$

$$= 2^{n-1} + \sum_{i=0}^{n-2} a_i2^i$$

$$= -1 \times 2^{n-1} + \sum_{i=0}^{n-2} a_i2^i$$

$$= -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i2^i$$
Using this result we can now find the value of the 2's complement number \( A = 1011_2 \) as follows:

\[
A_v = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
= -8 + 3 = -5
\]

which is the same result we obtained using the other methods. Now let us find the value of the 2's complement number \( A = 10110111_2 \).

\[
A_v = -2^7 + 2^6 + 2^4 + 2^2 + 1 = -73
\]

Note that we could speed things up by forming groups of more than one bit and then applying equation 1.3. For example, if using groups of four bits each:

\[
A = \overline{11011011} \\
A_v = -(2^4 - 11) \times 2^4 + 7 = -73
\]

We could have also used groups of two bits each:

\[
A = \overline{1011} \overline{0111} \\
A_v = -(2^2 - 2) \times 2^6 + 3 \times 2^4 + 1 \times 2^2 + 3 = -73
\]

1.4 The MSP430 from Texas Instruments

There are many microprocessor units (MPUs) and microcontroller units (MCUs) that we could use in this text. We have already explained that this book is written in such a way that its contents will be useful for years to come and that the way we chose to do that was by writing a general book with some specific applications using some off-the-shelf MCU or MPU. Our choice is the MSP430x1xx family of ultra low power microcontrollers.

The MSP430 comes in a variety of choices: the 11x, 11x1, 11x2, 12x, 12x2, 13x, and 14x devices. These devices share the following features and capabilities:

- Ultra low-power architecture
  - Low current (\( \mu A \)) and low voltage (1.8V - 3.6V) range operation
  - 6\( \mu \)s wake-up from standby mode
  - Interrupt capability to relieve the need for polling

- Processing capabilities
  - Seven source addressing modes
  - Four destination addressing modes
  - 27 core instructions
CHAPTER 1. INTRODUCTION

- Prioritized-nested interrupts
- No interrupt or subroutine level limits
- Large register-file
- RAM execution capability
- Efficient table processing
- Fast hexadecimal to decimal conversion

- Memory-mapped peripheral set
  - Integrated analog-to-digital (AD) converter
  - Integrated precision comparator
  - Multiple timers and pulse width modulation (PWM) capabilities
  - Slope AD conversion
  - Integrated universal synchronous asynchronous receiver transmitter (USART)
  - Watchdog timer
  - Multiple I/O with interrupt capability
  - Integrated programmable oscillator
  - 32kHz crystal oscillator
  - 450kHz-8MHz crystal oscillator on some devices

- Development tools
  - Simulator which include peripheral and interrupt simulation
  - C compiler
  - Assembler
  - Linker
  - Emulators
  - Flash emulation tool (FET)
  - Device programmer
  - Application notes
  - Example code

The MSP430x1xx controllers are very easy to use and for teaching microcomputers basics. This is the main reason we chose this family of microcontrollers for our device specific sections within each chapter.
\[ P_v = A_v \ast B_v = a_{m-1}b_{n-1}2^{m-n-2} + \sum_{i=0}^{m-2} \sum_{j=0}^{n-2} a_i b_j 2^{i+j} \]
\[ + \ 2^{m-1}( -2^n + 2^{n-1} + a_{m-1}2^{n-1} + \sum_{j=0}^{n-2} a_{m-1} b_j 2^j ) \]
\[ + \ 2^{n-1}( -2^m + 2^{m-1} + b_{n-1}2^{m-1} + b_{n-1} + \sum_{i=0}^{m-2} b_{n-1} a_i 2^i ) . \]

1.5 Exercises

1. Convert the following decimal (base 10) numbers into their equivalent binary (base 2) and hexadecimal (base 16) numbers.
   (a) 10
   (b) 32
   (c) 40
   (d) 64
   (e) 156
   (f) 244

2. Write the following unsigned binary (base 2) numbers as decimal (base 10) numbers.
   (a) 010000
   (b) 11111
   (c) 011101
   (d) 1111111
   (e) 10111111
   (f) 10000000

3. Write the following unsigned binary (base 2) numbers as hexadecimal (base 16) numbers.
   (a) 10
   (b) 101
   (c) 1011
   (d) 1010111
   (e) 11101010
   (f) 10000000
4. Convert the following unsigned hexadecimal (base 16) numbers into their equivalent decimal (base 10) and binary (base 2) numbers.

   (a) 1C
   (b) 7ABCD
   (c) 1234
   (d) 1FD53
   (e) 9D23A
   (f) 0A1B2C

5. Express 2’s complement -1 in each of the following formats

   (a) 4-bit binary (base 2) and hexadecimal (base 16) number
   (b) 8-bit binary (base 2) and hexadecimal (base 16) number
   (c) 16-bit binary (base 2) and hexadecimal (base 16) number

6. Write the following negative numbers as their 2’s complement representation in the binary and hexadecimal number systems using 8 and 16 bits.

   (a) -12
   (b) -68
   (c) -128

7. Find the corresponding decimal numbers for each of the following 2’s complement numbers.

   (a) 1000
   (b) 1111
   (c) 11000110
   (d) 10111110
   (e) A2
   (f) BE

8. Using the ASCII code find the binary streams represented by following character strings.

   (a) abcd
   (b) ABCD
   (c) 123aB
   (d) ASCII
   (e) Microprocessors and Microcontrollers
   (f) YoUr nAMe
9. Convert the following binary streams into their equivalent character strings using the ASCII code.

(a) "How Dy"
(b) "university"
(c) "012abCD"
(d) "blank space"

10. In order for a computer or host to be able to connect to the Internet it must have a number "logical address" known as an IP address. IP addresses are expressed using the dotted decimal notation where the decimal equivalent of each byte is separated from the following using a dot (.). Thus, when using the IPv4 version of the Internet Protocol (IP) a 32-bit IP address is expressed as four (4) decimal numbers separated by dots. Thus, IP address 127.0.0.1, known as the loopback or localhost IP address, would be expressed in binary as 01111111.00000000.00000000.00000001. Express each of the following IPv4 IP addresses as four (4) binary numbers separated by dots and vice versa.

(a) 240.0.0.1
(b) 10.240.68.11
(c) 192.168.40.163
(d) 11111111.11111111.11110000.00000000
(e) 00001010.00000000.00000000.00000001
(f) 10101100.10110010.00001111.11111111
Chapter 2

Assembly Language Programming

The following is the binary machine language representation of a program that toggles an LED on pin 0 of port 1 for the microprocessor within the MSP430F1232 microcontroller. Machine language is really the only language that a computer, or in this case the microprocessor, is able to understand. It is easy to see from this example that writing programs in binary machine language is a tedious, time consuming, and error prone task. Each line in Listing 2.1 below represents an instruction in binary machine language. Note that instructions are not all of the same length.

Listing 2.1: Binary Machine Language

```
1 00110001010000000000000000000011
2 101100100100000010000000010110100010000000000001
3 11010010110100110010001000000000
4 11010010111000110010000100000000
5 00111111110000000101000011000011
6 000111111100000011
7 1111111100010001
8 1111100100111111
```

Next we present in Listing 2.2 the hexadecimal machine language representation for the program shown in Listing 2.1. We see that, although more compact than its binary equivalent, writing programs in hexadecimal machine language is still a task not very well suited for humans. Again, each line represents an instruction only this time it is shown in hexadecimal.

Listing 2.2: Hexadecimal Machine Language

```
1 31400003
2 B240805A2001
3 D2D32200
```
The next program in Listing 2.3 is the assembly language representation for the above example in Listing 2.1 and Listing 2.2. Note that, although you still need to understand the details of the underlying architecture, there are certain acronyms of the code you are able to read because they look a lot like the English word they represent, e.g., mov (from move), xor (from exclusive-or), dec (from decrement), and jmp (from jump). Instructions in assembly language are in a one-to-one correspondence with instructions in machine language. This is the reason you see eight assembly language instructions just as there are eight machine language instructions. Assembly language allows us to also add labels and comments to document what we are doing. In the example below labels are written at the very beginning of the line and comments are written after the semicolon (;). We could not use labels and comments in the machine language versions.

```
Listing 2.3: Assembly Language Listing
1 RESET mov .w #300h, SP ; Initialize stack pointer
2 StopWDT mov .w #WDT PW + WDT HOLD, &WDT C TL ; Stop WDT
3 SetupP1 bis .b #001h, &P1DIR ; P1.0 output
4 Mainloop xor .b #001h, &P1OUT ; Toggle P1.0
5 Wait mov .w #050000, R15 ; Delay to R15
6 L1 dec .w R15 ; Decrement R15
7 jnz L1 ; Delay over?
8 jmp Mainloop ; Again
```

Finally, the same program will now be shown in the C high level language in Listing 2.4. The relationship between instructions in any high level language and instructions in machine or assembly language cannot be established as easy as the relationship between instructions in assembly language and instructions in machine language which is always a one-to-one correspondence. Although we see seven instructions in the C language version of our example we cannot say anything more in terms of the corresponding instructions in assembly language, at least not unless we get more information. One thing is certain, most high level language instructions require more than one assembly language instruction.

```
Listing 2.4: C Language Listing
1 #include <msp430x12x.h>
2 void main (void)
3 { WDTCFL = WDT PW + WDT HOLD; /* Stop watchdog timer */
4 P1DIR |= 0x01; /* Set P1.0 to output direction */
5 for (; ;)
```
2.1. FETCHING, DECODING, AND EXECUTING AN INSTRUCTION

```c
{ unsigned int i;
    P1OUT ^= 0x01;  /* Toggle P1.0 using ex-or */
    i = 50000;     /* Delay */
    do ( i--);
    while ( i != 0);
}
```

As we will find later lines 1, 2, and 6 in 2.4 are called directives and thus they are not instructions. Directives do not generate instructions from the instruction set.

In order to effectively use a microprocessor it is necessary to understand its internal structure. Figure 2.1 is a block diagram showing the typical components of a microprocessor unit which includes: the general purpose registers (R1-RN), the special purpose registers (PC, IR, MAR, MDR, etc.), the instruction decoder, the control unit, the address bus, the data bus, the control bus, and the arithmetic and logic unit (ALU).

What distinguishes the microprocessor from any other piece of hardware is its ability to process instructions. Although we usually think of these instructions in terms of the source code written by humans in the form of assembly language instructions and/or high level language instructions, the only instructions that a microprocessor, or any computer for that matter, can process are machine language instructions, i.e. instructions written in the native language of the machine.

As mentioned above, humans typically write their programs (sequence of instructions) using a language other than the native language of the microprocessor. These programs must be translated into machine language using an assembler, compiler, or interpreter and then loaded into the microprocessor’s memory. In order for a microprocessor to process an instruction the machine language version of the instruction must be brought from the microprocessor’s memory into the Instruction Register (IR). The microprocessor’s memory could be located outside or inside the microprocessor. We should now explain the process followed by the microprocessor to carry out (fetch, decode, execute) an instruction.

2.1 Fetching, Decoding, and Executing an Instruction

Assume the instruction to be processed by the microprocessor is already stored in the microprocessor memory as a machine language instruction, Figure 2.1. The address of the memory location where the instruction starts at must somehow be placed in the Program Counter (PC) register. Think of the PC as a special purpose register whose contents is always pointing to the next instruction to be executed in
Figure 2.1: Main Components of a Typical Microprocessor Unit.
2.1. FETCHING, DECODING, AND EXECUTING AN INSTRUCTION

Figure 2.2: Main Components of a Typical Microprocessor Unit.
Figure 2.3: Fetch, Decode, Execute.
sequence. The Control Unit (CU) will tell the PC to place its contents at the 
PC’s output and simultaneously tell the Memory Address Register (MAR) to 
read in the value output by the PC. Then the CU will instruct the MAR to place 
its contents in the address bus. The CU will also send a Read control signal 
out via the control bus so that the memory location whose address is indicated 
at the address bus will place its contents (the machine language instruction) at 
the data bus.

The Memory Data Register (MDR) will be instructed by the CU to read in 
the data bus. After this the CU will instruct the MDR to place its contents at 
the MDR’s output and at the same time will tell the IR to read in this value. 
The machine language instruction is now stored as the contents of the IR. The 
CU will now tell the IR to send its contents to the instruction decoder where the 
machine code will be decoded in order to find out the specific instruction from 
the instruction set that must be executed along with the operands specified by 
the instruction. It is at this point that the microprocessor is ready to carry out 
the steps involved in the “processing” of the instruction which usually include 
having the ALU act upon the contents of some general purpose registers and or 
memory locations and storing the result of the operation either as the contents of 
a register or memory location. At this time the address of the memory location 
holding the next machine code instruction will be stored in the PC.

The steps involved from the point when the CU tells the PC to place the 
address of the memory location holding the instruction to the point where the 
contents (instruction in the form of machine code) of the memory location is 
stored in the MDR is usually referred to as fetching the instruction or the fetch 
phase. From the point the CU tells the MDR to place its contents at the MDR 
output to the point where the instruction decoder has acted upon it is called the 
decoding phase. From the point the instruction decoder has decoded the 
instruction to the point where all the steps involved in processing the instruction 
are carried out is usually referred to as the execute phase. It should be clear by 
now that when people say executing an instruction they usually mean, though 
they might not be aware or it, fetching-decoding-executing.

2.2 The Instruction Set and Addressing Modes

The set of all the different instructions a microprocessor is able to execute 
is called the microprocessor instruction set. The number of instructions in the 
instruction set of a microprocessor varies with the microprocessor ranging from 
less than thirty (30) to several hundreds. Microprocessors or machines with a 
small numbers of instructions are usually called Reduced Instruction Set Com-
puters or RISC whereas those machines exhibiting a large number of instruc-
tions are called Complex Instruction Set Computers or CISC. Table TABLE 
SHOWING THE INSTRUCTION SET shows the instruction set for our microprocessor.

In order to be able to access the contents of the memory locations using the 
instructions provided in the instruction set of a particular microprocessor the
architects of the microprocessor also provide the user with several addressing modes for each of the instructions in the instruction set. The addressing modes allow the user (programmer) to specify the location of the data item of items in any of several applicable ways.

The most common addressing modes found are: immediate, register, absolute or direct, indirect, register indirect, and relative. The format for an assembly language instruction is

\[
\text{mnemonic operand1, operand2}
\]

and in order to illustrate the different addressing modes we will use the `mov` instruction which is used to load or bring a data item into one of the registers inside the microprocessor. The syntax of the `mov` instruction is as shown below.

\[
\text{mov source, destination}
\]

The semantics (meaning) of the above instruction is: replace the value of the operand specified by the contents of destination with the data item specified by the contents of the source operand, i.e.

\[
(\text{destination}) \leftarrow (\text{source})
\]

Note that (destination), i.e. destination enclosed within parenthesis, means the contents of the operand indicated by the destination and (source),i.e. source enclosed within parenthesis, means the contents of the operand indicated by the source. The following examples illustrate each of the applicable addressing modes for the `mov` instruction:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
<th>Addressing Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov #01,r1</td>
<td>((R1) \leftarrow 01_{10})</td>
<td>immediate and register</td>
</tr>
<tr>
<td>mov &amp;01,r1</td>
<td>((R1) \leftarrow (01))</td>
<td>absolute and register</td>
</tr>
<tr>
<td>mov 01,r1</td>
<td>((R1) \leftarrow ((01)))</td>
<td>indirect and register</td>
</tr>
<tr>
<td>mov @r2,r1</td>
<td>((R1) \leftarrow ((R2)))</td>
<td>indexed and register</td>
</tr>
<tr>
<td>mov 01(r2),r1</td>
<td>((R1) \leftarrow ((01) + (R2)))</td>
<td>relative and register</td>
</tr>
</tbody>
</table>

It should be mentioned that, although we chose, for the sake of clarity, to use the register addressing mode for the second operand in the examples shown above, any of the addressing modes could be used for the second operand with the exception of the immediate addressing mode.

## 2.3 Programming Basics

In order to be able to exploit the full power of a microprocessor we must be able to program it using instructions. We have already seen that an instruction is an order that we give the microprocessor when we want it to accomplish a specific command. Usually, what we need to tell the microprocessor to do will take more than just one instruction. A program is a logical sequence of
2.3. PROGRAMMING BASICS

instructions written in the specific order in which we want a specific task to be accomplished. It is important to point out the difference between a series of instructions and a program. A program is much more than just a mere series of instruction put together. The instructions that constitute a program must be logically interwoven to accomplish a specific task to solve a specific problem usually with the goal of optimizing one or more resources, e.g., memory used, time, power, etc.

It is probably clear by now that a microprocessor executes instructions in sequence. This is what we have shown when we explained the way a machine code instruction pointed to by the PC is fetched, decoded, and then executed while the PC is being updated to point to the next instruction to be processed after the current instruction is executed.

When we program a microprocessor we use the available registers and memory locations. Regardless of how many registers a microprocessor has it is always a finite, usually less than two dozens, number of registers whose names we can remember rather easily. On the other hand, the number of memory locations although also finite, is usually in the order of thousands or millions making it impossible for humans to keep track of all of them. To solve this problem programmers usually give names to the memory locations whose content they are interested in using. A variable is the name given in a program to a specific memory location. Whenever a variable is used within a program it means that the programmer is referencing the contents of the corresponding memory location. This concept is similar to that of a variable in algebra.

As already mentioned in a previous section, a microprocessor can only understand and process instructions written in its very own machine language. Machine language instructions, however, are very difficult for humans to write since they consist of only zeroes (0’s) and ones (1’s) making it also extremely error prone. To circumvent this problem humans developed assemblers. An assembler is a program written to take as input the source program written using assembly language instructions and translate it to produce machine language instructions. We have already seen what assembly language instructions look like:

\[
mnemonic \ operand1, operand2
\]

and given examples using our assembly language for the mov instruction. The mnemonic is usually an abbreviated form of some command, e.g., mov, and, or, add, etc. The first and second operands, operand1 and operand2, are used to indicate the source and destination of the data items, respectively.

The following piece of code adds the content of the memory locations indicated by mem1, mem2, and mem3, divide the result by 3, and store it in register R5. Note, however, that, for the sake of clarity, we are omitting the declarations and initialization of variables mem1, mem2, and mem3.

\[
\begin{align*}
mov & \ r5, \ mem1; \ (mem1) \ leftarrow \ (R5) \\
add & \ r5, \ mem2; \ (mem2) \ leftarrow \ (R5) + (mem2) \\
add & \ r5, \ mem3; \ (mem3) \ leftarrow \ (R5) + (mem3) \\
xor & \ #3, \ r5; \ (R5) \ leftarrow \ (R5) \ exor \ 3
\end{align*}
\]
In the above piece of code the semicolon, i.e. ;, is used to indicate the beginning of a comment. Comments are used for documentation purposes only and are not considered by the assembler as part of the code. As soon as the assembler sees the ; the rest of the line is ignored. Comments may appear anywhere in a line, but if a ; shows up before the instruction is finished then the assembler will flag an error. For example, in the following piece of code the assembler will indicate there is an error in the second line, but will not say anything about the third line because that line has a ; right at the beginning, hence the third line will be completely ignored by the assembler.

\[
\begin{align*}
\text{mov} & \ r5 , \text{mem1}; \ (\text{mem1}) <= (R5) \\
\text{add} & : \ r5 , \text{mem2}; \ (\text{mem2}) <= (R5) + (\text{mem2}) \\
; \text{sum} & : r5 , \text{mem3}; \ (\text{mem3}) <= (R5) + (\text{mem3}) \\
\text{xor} & \ #3 , r5; \ (R5) <= (R5) \text{ xor } 3
\end{align*}
\]

2.3.1 Program Design

It is time to pause and introduce the concept of program design. It is of upmost importance to understand the fact that, before we try to solve a problem, we should make sure that we understand the problem, that the procedure we are using to find the solution to the problem is valid and efficient, and that in the future someone else will be able to figure out what we did. A good example is building a house: In this case the problem is usually that there is a need for shelter and a dwelling place with some specific requirements in terms of size, number of bedrooms, etc. and we should not attempt to build the house until we are sure we understand the specific needs and requirements; once we understand the type and requirements of the house that needs to be built we now have to engage in the process of designing it and for that we use blueprints which must include everything needed to build the house and everything needed for future maintenance of the house. If the person who is designing the house did not fully understand the needs and requirements then something is bound to be missing and thus some future work will have to be performed on the house after it is finished. If the blueprints used to build the house did not accurately reflect what needed to be included and how it needed to be done or if the person in charge of the construction did not accurately follow the blueprints then any future enhancements or maintenance procedures of the house may result in a nightmare for the owner.

Writing a program is analogous to building a house. The problem the program is to solve must be understood so that all the needs and requirements are included. Next the process that will be used to solve the problem must be spelled out in some way. The process that leads to the solution of the problem is called an algorithm and is analogous to the blueprints. The algorithm must include all the information needed to meet the needs and requirements of the problem to be solved. Finally, the last phase in solving the problem would be
2.4 MICROCONTROLLER PROGRAMMING MODEL

2.4. MICROCONTROLLER PROGRAMMING MODEL

to write the code which should include enough documentation for future maintenance and enhancements. It is tempting for beginners to jump to the last phase, i.e. start writing the code, without going through the first two phases of program design especially when the programs they are writing are toy programs. This mistake must be avoided at all cost.

There are different techniques and tools that can be used during program design as well as there are many different languages that can be used for coding. We shall make use of flowcharts for program design, but the reader must be aware that other tools are available some of which would even produce the flowcharts.

2.3.2 Using Flowcharts for Program Design

Flowcharts are one of the oldest and most widely used tools available for program design. Though flowcharts are not good for detailed explanations of the code to be produced, they are extremely good at providing a good general idea of what needs to be done because flowcharts aid in the visualization of the process they represent. Several block symbols are available for building flowcharts: start, end, computation or assignment, decision, connector, and subprocess.

2.4 Microcontroller Programming Model

Our programming model will be that of the MSP430 microcontroller. The MSP430 has sixteen (16) registers of sixteen (16) bits each. The names of these registers are R0 thru R15. Four (4) of these registers are used for special purposes, i.e. R0 is used as the program counter (PC), R1 is used as the stack pointer (SP), R2 is used as the status register, and R3 is used as the constant generator. Except for the constant generator register (R3) and part of the status register (R2), all registers are accessible using the MSP430 instruction set. The sixteen (16) registers are shown in Figure: FIGURE SHOWING THE SIXTEEN REGISTERS IN THE MSP430 WITH THE DETAILS FOR THE FOUR SPECIAL REGISTERS R0-R3.

The MSP430 system configuration shown in Figure: FIGURE SHOWING THE MSP430 SYSTEM CONFIGURATION BLOCK DIAGRAM shows all the components available for programming the MSP430 microcontroller. Besides the CPU and the sixteen (16) registers there are also available several input/output (I/O) ports, an analog to digital converter (ADC), two timers (TA and TB), and a comparator.

2.5 Anatomy of an Assembly Program

The program in Listing 2.5 is an example of a program structure of a complete assembly program for the MSP430. The first line we find is line 1 which indicates the file where the definitions for the constant symbols written in all
capital letters are found. Next we find line 3, an ORG (origin) directive in-structing the assembler to place the first byte it finds, i.e. the first byte of the machine code for line 5 at the memory location whose address is specified after the ORG mnemonic.

Lines up to but not including line 29 constitute the main program. Any valid assembly language instruction can be used as part of the main program. The main program could also be the only program which will be the case if we write monolithic code. Monolithic code or program means we are not using any subprocess but have decided to include all our instructions within the same programming unit.

Listing 2.5: Assembly Language Program Example

```assembly
#include <msp430x12x2.h>

; ******************************************************************************
ORG 0E000h ; Program Start

RESET
mov.w #300h, SP ; Initialize stack pointer

; Stop WDT Setup TA
StopWDT
mov.w #MDIPW, WDHOLD, &WDTCTL

; SMCLK, clear TAR Setup C0
mov.w #TASSEL1, TACL, &TACTL
mov.w #CCIE, CCTL0 ; CCR0 interrupt enabled
mov.w #CC0, &CCR0;

SetupP1
bis.b #001h, &P1DIR ; P1.0 output

; Start Timer_a in continuous mode
bis.w #MCI, &TACTL
eint ; Enable interrupts

; Mainloop
bis.w #CPUOFF, SR ; CPU off
nop ; Required for C-spy

; TA0_ISR; Toggle P1.0

xor.b #001h, &P1OUT; Toggle P1.0

; Add Offset to CCR0
add.w #50000, &CCR0
reti

; Interrupt Vectors Used MSP430x12x(2)

ORG 0FFFEh ; MSP430 RESET Vector
DW RESER;
ORG 0FFF2h ; Timer_A0 Vector
DW TA0_ISR;
END
```
Beginning in line 29 we find an interrupt handler or interrupt service routine. Interrupt service routines (ISR) are an example of a subprocess or module. Other subprocesses or modules are usually called subroutines or functions. Modularizing your code is strongly encouraged because it leads to a more readable, easier to code, easier to debug, and easier to maintain program. In the example shown in Listing 2.5 case we only have one subprocess, but a typical program will have one main program and several subprocesses.

Lines 30 and 32 are also assembler directives just as lines 3 and 29. The assembler directive DW used in lines 30 and 32 is the "define word" directive. The define word directive instructs the assembler to reserve or set apart the next word (two bytes). In the case of line 30 the define word directive tells the compiler to reserve a word beginning at location 0FFFEh and initialize it with the word corresponding to the address associated with label RESET in line 5. Line 32 tells the compiler to reserve a word beginning at location 0FFF2h and initialize it with the word corresponding to the address associated with label TA0_ISR, i.e. the address of the instruction beginning in line 22.

Whenever our assembler sees a semicolon, i.e. ; it knows that the rest of the line is to be ignored. This is independent of where the semicolon is placed, whether this be after an instruction, before an instruction, in the middle of an instruction, at the end or beginning of a line.

Thus, in general the structure of an assembly language program can be summarized as shown below. The reader is warned, though, to bear in mind that this is just one way of writing code and not the only way of writing code.

Indeed, the user will soon find himself adopting his own style for writing programs.

### 2.6 Assembly Language Programming Techniques

Writing monolithic code is tempting when you are new to programming, but this programming style is highly discouraged and should be avoided whenever possible because it is difficult to understand, debugged, and maintained. Of course, if your code is just a few lines long and there is really no need for a subprocess or module, then it is okay to just use one module. Nevertheless, you should always strive to write modularized code.

Comments are very important because they allow us to document our programs. Without comments, if our code is sufficiently long and or complicated, we will soon find ourselves lost trying to remember what it was we were doing. The frontend of program documentation is your algorithmic language, in our case we have chosen to use flowcharts and the vehicle to spell out our algorithm. The next step is to document our code as we write it. Do not fall in the trap of first writing your code to document it after it is free from errors.
Indenting and aligning your code is another very useful technique. It is customary to write labels beginning in column one (1) whenever the line begins with a label, then indent the beginning of the instruction, and finally use a comment to document or explain the instruction. If you always align the instructions, comments, etc., your code will be more readable and easier to understand by yourself and others.

Although most assemblers are case insensitive, try to always write your instructions in lowercase letters and leave uppercase words for labels, constants, and directives.

2.7 Exercises

1. Explain the difference between machine language, assembly language, and high level language.

2. Explain why an assembler and a compiler are needed.

3. Describe in your own words the process a machine code instruction undergoes in order to be executed.

4. What would happen if the instruction the CPU fetches to be executed turns out to be an instruction written in assembly language or in a high level language?

5. If a program has 700 machine language instructions, how many assembly language instructions will it have?

6. Describe the function of the PC (Program Counter), MAR (Memory Address Register), MDR (Memory Data Register), IR (Instruction Register), and the general purpose registers within a CPU.

7. Describe the function of the address bus, data bus, ALU, control unit, and the instruction decoder.

8. Explain the main difference between a RISC and a CISC computer.

9. What is the difference between a microprocessor and a microcontroller.

10. Enumerate the addressing modes in the MSP430.

11. Explain what an embedded system is and give some examples.

12. Why should we avoid writing monolithic code?

13. Using a flowchart, develop an algorithm that will take a temperature reading in degrees Fahrenheit stored in variable degF and convert it to degrees Celsius stored in variable degC.
14. Assume that we are using a letter grade based system as follows: A=4.00, B=3.00, C=2.00, D=1.00, F=0.0. Using a flowchart, develop the algorithm that asks a person the number of credits with A, B, C, D, and F and computes the general point average or GPA.

15. Using the bit assignment in the MSP430 status register SR (see Figure (FIGURE SHOWING MSP430 SR)), show the piece of code in the MSP430 assembly language instruction to determine if the N (negative) bit is set to a one (1) and the GIE (Global Interrupt Enable) bit is set to a zero (0).

In this chapter we learn the basics of how to use the C high level language. The need for a high level language arose when it was needed to separate the programmer from having to know all the details of the hardware inside the computer in order to program it. In doing so, the programmer was able to program faster. In the opinion of some people, as a result of the development of high level languages, programming became less cumbersome and more people began to program. An instruction in a high level language would typically correspond to several instructions in machine code allowing for faster programming, see Listings 2.1, 2.2, and 2.4. On the downside since the programmer is not aware of the mapping of the high level language into machine code, the programmer has in effect lost some control of the hardware.

C evolved from BCPL (1967) and B (1970) both of which were type-less languages. The C language was developed by Dennis Ritchie in 1972 and was first implemented on a Digital Equipment Corporation PDP-11. C, a strongly-typed language, is the development language of the Unix and Linux operating systems and provides for dynamic memory allocation through the use of pointers.

3.1 C Program Structure and Design

The structure of a C program is shown below:

```c
preprocessor directives
global variables;
functions
{
    local variables;
    program block;
}
```
int main(void)
{
    local variables;
    program block;
}

As an example of this structure we show the following program. Note that function hello in the program below does not have local variables.

Listing 3.1: First C Language Program Structure Example
#include <stdio.h> /* note no semicolon ; */
#define MAX 15 /* note no ; */
void hello(int k) /* note no ; */
{
    printf("%d\n",k);
}
main() /* note no ; */
{
    int i = MAX;
    hello(i);
} /* COMMENTS IN C MUST BE
ENCLOSED LIKE THIS ONE */

An alternate structure for a C program is as follows:

preprocessor directives
global variables;
function prototypes;
int main(void)
{
    local variables;
    program block;
}

functions
{
    local variables;
    program block;
}

An example of the above program structure is shown in the following code

Listing 3.2: Second C Language Program Structure Example
#include <stdio.h>
#define MAX 15
void hello(int k); /* note the use of the ; */
Again, note that function hello in the above code does not have any local variables. It is easy to distinguish a directive in C because directives in the C language must always begin with 
. In the examples above we have used the \#include directive to let the preprocessor know that we wish to include a specific header file as part of our code and the \#define directive in order to tell the compiler that a particular string must always be substituted by the constant following it. In our examples we have included the stdio.h header file which includes, among others, the declaration for the printf function and had defined the MAX constant so that each time it is found in the code the compiler will replace it with the constant integer 15.

A program written in the C language must always include the main function. A C program always begin execution by executing function main. In the examples above a local variable i was declared to be of type int and initialized with the value 15 in the first of the two instructions contained by the main function. The second instruction is a call or invocation to function printf which will print the value of variable i.

We should point out that since different C compilers are more often than not written by different people you will find variations of the above structures implemented depending on the C compiler that was used. This is especially true for compilers specifically written for embedded systems. For example, some C compilers will not require you to write the type of function main, i.e. int, nor will they require you to write the word void within the parenthesis. In other words, some compilers will accept main() instead of int main(void). There is nothing wrong with these C compilers, we are just bringing this up now so that the reader may be aware of this fact.

In section 2.3 we introduced the reader to programming basics and using flowcharts as part of the program design. The reader is encouraged to review this section because the same applies when coding is performed using a high level language such as C. We shall also make use of flowcharts as part of the program design when coding is going to be implemented using C as the programming vehicle.
3.2 Memory and I/O Access using C

If we are going to use the C language to program a microprocessor or a microcontroller we should make sure that we understand how to access memory and input/output (I/O) units using C. Actually, this is very easy to do in C since the only thing we need to know is the address of the memory location or input/output unit we want to access.

For both memory and I/O accesses using C we just need to know the address of the memory location or the I/O unit. Once the address of the memory location is known we can access its contents using pointers. We will see specific examples of how this is done in the MSP430 in section 3.3.

3.3 Programming the MSP430 in C

The program shown in Listing 3.3 is written in the C language program and will be used as an example of how to program the MSP430 microcontroller using C.

Listing 3.3: Accessing Memory using C

```c
/* Written by Mark Buccini */
#include <msp430x12x.h>

void main (void) {

    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= 0x01; // Set P1.0 to output direction

    for (; ;)
    {
        unsigned int i;
        P1OUT ^= 0x01; // Toggle P1.0 using exclusive-OR
        i = 50000; // Delay
        do (i--);
        while (i != 0);
    }
}
```

Let us now explain in detail what the example above is meant to do. There are several symbols used in this example: WDTCTL, WDTPW, WDTHOLD, P1DIR, P1OUT, and i. The symbol i is a variable of type unsigned int. This means that i cannot take on negative values and that it can only take on integer values. Most people use the convention of writing variables in their C language programs in lower case and constants in upper case. If we follow this convention we can easily figure out that the rest of the symbols used in the example above are meant to be constants. In the example, the values for the constants are all defined in the header file msp430x12x.h included using the preprocessor directive #include.
Some of the constant values found in the header file are associated with special function registers, i.e. WDTCTL (0x0120), P1DIR (0x0022), and P1OUT (0x0021) whereas others are just constant values used in arithmetic and/or logical operations, i.e. WDTPW (0x5A00), and WDTHOLD (0x0080). The values enclosed within parenthesis are the values associated with the constant names in file msp430x12x.h. Thus the instruction WDTCTL = WDTPW + WDTHOLD; adds 0x0080 to the value 0x5A00 to effectively take the watchdog timer in the MSP430 out of action. This is accomplished by setting to one (1) bits 14, 12, 11, 9, and 7 in the watchdog timer control register, see Figure FIGURE FOR THE WATCHDOG TIMER CONTROL REGISTER.

Instruction P1DIR |= 0x01; is equivalent to the following instruction in the C programming language P1DIR = P1DIR | 0x01; which takes the current value of P1DIR and performs a bit-wise OR with the value 0x01. In the context of the MSP430 this effectively sets bit 0 to the value 1, i.e. bit 0 will be used for output purposes, whereas bits 1 thru 7 are left unchanged since 1 OR 0 = 1 and 0 OR 0 = 0. P1DIR in the MSP430 is used to define whether a bit in port 1 is used as an input pin (0) or as an output pin (1). For the purpose of this program an LED is expected to be connected to pin 0 in port 1 of the MSP430 in such a way that when a 1 is output the LED will light and when a 0 is output the LED will turn off.

Within the body of the infinite loop established by the for (; ; ) instruction we find the declaration of i explained previously followed by the P1OUT ^=0x01; instruction. The latter instruction EX-ORs the value of P1OUT with 0x01 which has the effect of toggling the state of bit 0 in port 1 which in turn toggles the state of the LED. This is true since 1 EX-OR 1 = 0 and 0 EX-OR 1 = 1. We find next instruction i = 50000; followed by the loop established by instructions do (i--); and while (i!=0);. These instructions load 50000 into the memory location assigned by the compiler to variable i and then decrement its value by one until the contents of variable i is zero. What this does is that it introduces a delay equivalent to however long it takes the CPU to count from 50000 to zero. Once the value of zero is reached it is time to go back up to the first instruction within the body of the for loop, i.e. P1OUT ^=0x01; repeating the whole loop again. Thus, for each time the for loop iterates the do–while loop iterates 50000 times.

Example: Assume that you have a 12-key keypad connected to the MSP430 as shown in FIGURE OF KEYPAD CONNECTED TO MSP430. We need to write a program that will correctly scan the keypad and identify the key that was pressed. For the sake of simplicity we and since the keypad is mechanical, we will hide the details of the need to de-bounce the keys and will assume an error-free de-bouncing function is available.

Solution: SOLUTION
3.4 C-Cross Compilers Vs. C-Compilers

Just as it is necessary to have some means of translating assembly language instructions into machine language instructions it is also necessary to translate high level language instructions into machine code. In the case of assembly language it is the job of the assembler to translate into machine language. When we program in a high level language we need a compiler to make the translation into machine language.

There are many products available which are labeled as compilers. Some of these products come included as part of an Integrated Development Environment or IDE. Some “compilers” are provided with the ability to produce code other than machine code, e.g. translate from one high level language into another high level language or from a high level language into assembly language. Since we have defined a compiler as a translator from a high level language into machine language these so called “compilers” must be something else. The term cross-compiler is probably more appropriate for these type of translators. C-compilers are no exception and thus we will find some C-compilers which produce machine code and some others which produce code in some language other than machine language.

The main reason for our taking the time to differentiate between a compiler, e.g. a C-compiler, and a cross-compiler, e.g. a C-cross compiler, is the fact that a cross-compiler will most probably produce code that, when translated into machine language, might not be as efficient as the machine language produced by a compiler. This is regardless of how many optimizations the cross compiler might have available and is directly related to how much time and effort was put in its development.

**Example and explanation of the machine code produced by a compiler for some program and the machine code produced by a cross-compiler for the same program**

As explained at the beginning of this chapter a similar comparison can be made between a compiler and an assembler where it is most probably that the assembler will produce the most efficient machine code.

3.5 Real-time Programming using C

When it comes to using any programming language for real-time programming we have to make sure what we are dealing with. Real-time refers to the fact of having to respond to inputs and signals in a timely manner, i.e. within a specified amount of time because responding otherwise would harm the system or make it useless. One very simple example of a real-time application would be a clock system. In this example it is crucial that the system responds to the signal that tells it that a second (or a 100th of a second) has elapsed. Other real-time systems we encounter each day are found in automobiles where they are used for engine, brakes, and transmission control. Another real-time system which is much closer to home is the microwave oven. A more critical real-time
3.5. REAL-TIME PROGRAMMING USING C

system could be a pacemaker. In general, real-time systems deal with time crit-
ical applications where it is imperative to process and respond to the inputs
within a precise time slot.

After talking about real-time systems we are in a better position to talk
about real-time programming. Real-time programming is the processing of sig-
als in a time-critical application using software. In order to be able to process
signals in a real-time manner it may be necessary to abandon, at least in part,
our traditional way of writing programs. What we mean with the latter state-
ment is the following: We were taught to write programs in a modular fashion,
i.e. using program modules that are invoked each time they are needed and
avoid entirely the use of unconditional branch instructions better known as
GO-TOs. However, we probably did not realize that each time we invoke (call)
a module a context switch needs to occur in which the instructions that were
to be executed next will have to wait, the current state of the program is saved,
instructions from the invoked module are brought to memory, and then begin
to be executed. After executing the module the previous state of the program
needs to be restored so that execution can continue from the instruction imme-
diately following the module invocation. There is nothing wrong with this when
we are writing non real-time applications, but in a real-time environment this
is a lot of overhead that must be avoided as much as possible or all together.
So now we need to try to avoid invoking modules unless we really have to and
yes, we should use GO-TOs because they are much faster for altering the flow
of the execution of instructions.

Another thing that we probably were told when it comes to writing programs
in C was to take advantage of pointers and avoid using arrays. In real-time pro-
gramming we can still use pointers when writing programs in C but will probably
now prefer to use arrays, because calculating the address of an array element
can be performed at compile-time (unless subscript expressions
using values not known at compile-time are used) and thus a fixed (and short)
amount of time is needed to reach it, whereas we may not know in advance how
long it will take to traverse a chain of pointers to reach some specific value at
run-time).

Yet another example comes to mind: interrupts. We normally set up an
interrupt handler or interrupt service routine to respond to some event and
allow the event to interrupt the processor. The way we usually do this is very
similar to the way we are accustomed to handle modules in that, each time we
have to reach the interrupt handler and then exit it to go back to the part of the
code we were executing at the moment of the interruption, much overhead is
incurred. This overhead is measured in terms of cycles, e.g. in the MSP430 six
(6) cycles are used for the first part of the interrupt sequence (reaching the first
instruction in the service routine) and five (5) for the second part (returning
from the interrupt service routine). A better way to handle the interrupts could
be to poll the interrupt flags in order to service an interrupt using a GO-TO
before we exit the current interrupt handler. We will go back to real-time
programming in Chapter ??.
CHAPTER 3. HIGH LEVEL LANGUAGE PROGRAMMING WITH C

3.6 Special Processor Features in C Programs

What exactly goes in here?

3.7 Summary

3.8 Exercises

1. What is the name of the program modules in the C language?

2. What is the minimum number of functions that a C program must have? Identify them.

3. Can you explain why it is that you can not use the exact same compiler on some specific PC to produce code to be run on a PC and also to produce code to be run on some embedded system?

4. What does it mean when we say that C is a strongly-typed language?

5. What are the main parts of a C program?

6. Why is a compiler needed?

7. What type of code does a compiler produce?

8. What is a cross-compiler?

9. Explain why a compiler will normally produce code that is more efficient than a cross-compiler.

10. Name several preprocessor directives in the C language.

11. What is a header file?

12. What do you need to know in order to be able to use memory locations and I/O units in your C language programs?

13. Write a small C language program that will set the lower four bits of port 3 (P3) in the MSP430 as input bits and the upper four bits as output bits. Assume P3DIR is defined in the msp430x12x.h file.

14. Write a C language program that will read 8 input bits from port 2 (P2) rotate them twice and send them out port 4 (P4) of the MSP430. Assume P2DIR, P4DIR, P2IN, P4OUT are defined in the msp430x14x.h file.

15. Angel and Carlos wanted to allow more time for the LED used to test the program in Listing 3.3 to be on and off. Thus, they increased the loop count from 50000 to 75000. When testing the program they found out, however, that the LED would now turn on and off at a faster rate. Explain what happened. Hint: Variables in the MSP430 are 16-bit long.
16. When can a context switch occur during the execution of a program?

17. Explain in your own words the term overhead as it applies to both real and non-real time programming. Is it possible to execute an overhead-free program?

18. Choose which are real-time applications:
   (a) Anti-lock Brake System (ABS) in a car.
   (b) Car cruise control.
   (c) Sending an electronic mail (email) message.
   (d) Generating a report of how many people entered a particular location.
   (e) Hearing aid system.
   (f) University online registration system.
   (g) Word processor system.