
This may be reproduced and distributed in its entirety (including this authorship, copyright and permission notice), provided that no charge is made for the document itself, without the author’s consent. This includes “fair use” excerpts like reviews and advertising, and derivative works like translations.

Note that this restriction is not intended to prohibit charging for the service of printing or copying the document.

Instructors are encouraged to use this document as a class resource; however, the author would appreciate being notified in this case.
Contents

Preface v

1 Introduction 1

1.1 Number Systems 1

1.1.1 Decimal 1

1.1.2 Binary 1

1.1.3 Hexadecimal 3

1.2 Computer Organization 4

1.2.1 Memory 4

1.2.2 The CPU 5

1.2.3 The 80x86 family of CPUs 6

1.2.4 8086 16-bit Registers 7

1.2.5 80386 32-bit registers 8

1.2.6 Real Mode 8

1.2.7 16-bit Protected Mode 9

1.2.8 32-bit Protected Mode 10

1.2.9 Interrupts 10

1.3 Assembly Language 11

1.3.1 Machine language 11

1.3.2 Assembly language 11

1.3.3 Instruction operands 12

1.3.4 Basic instructions 12

1.3.5 Directives 13

1.3.6 Input and Output 16

1.3.7 Debugging 16

1.4 Creating a Program 18

1.4.1 First program 18

1.4.2 Compiler dependencies 22

1.4.3 Assembling the code 22

1.4.4 Compiling the C code 23

1.4.5 Linking the object files 23

1.4.6 Understanding an assembly listing file 23
## CONTENTS

1.5 Skeleton File .................................................. 25

2 Basic Assembly Language ........................................ 27
   2.1 Working with Integers ....................................... 27
      2.1.1 Integer representation ................................ 27
      2.1.2 Sign extension ........................................ 30
      2.1.3 Two's complement arithmetic ......................... 33
      2.1.4 Example program ...................................... 35
      2.1.5 Extended precision arithmetic ....................... 36
   2.2 Control Structures .......................................... 37
      2.2.1 Comparisons ........................................... 37
      2.2.2 Branch instructions .................................. 38
      2.2.3 The loop instructions ................................ 41
   2.3 Translating Standard Control Structures .................. 42
      2.3.1 If statements ........................................ 42
      2.3.2 While loops ........................................... 43
      2.3.3 Do while loops ....................................... 43
   2.4 Example: Finding Prime Numbers ............................ 43

3 Bit Operations .................................................. 47
   3.1 Shift Operations ........................................... 47
      3.1.1 Logical shifts ........................................ 47
      3.1.2 Use of shifts .......................................... 48
      3.1.3 Arithmetic shifts ..................................... 48
      3.1.4 Rotate shifts .......................................... 49
      3.1.5 Simple application ................................... 49
   3.2 Boolean Bitwise Operations ................................ 50
      3.2.1 The AND operation .................................... 50
      3.2.2 The OR operation ..................................... 50
      3.2.3 The XOR operation .................................... 51
      3.2.4 The NOT operation ................................... 51
      3.2.5 The TEST instruction ................................ 51
      3.2.6 Uses of bit operations ............................... 52
   3.3 Avoiding Conditional Branches ............................... 53
   3.4 Manipulating bits in C ..................................... 56
      3.4.1 The bitwise operators of C ........................... 56
      3.4.2 Using bitwise operators in C ......................... 56
   3.5 Big and Little Endian Representations ..................... 57
      3.5.1 When to Care About Little and Big Endian .......... 59
   3.6 Counting Bits ............................................... 60
      3.6.1 Method one ............................................ 60
      3.6.2 Method two ............................................ 61
      3.6.3 Method three ......................................... 62
CONTENTS

6.2.3 Multiplication and division ........................................ 123
6.2.4 Ramifications for programming ................................. 124
6.3 The Numeric Coprocessor ........................................... 124
  6.3.1 Hardware .......................................................... 124
  6.3.2 Instructions ...................................................... 125
  6.3.3 Examples .......................................................... 130
  6.3.4 Quadratic formula ............................................... 130
  6.3.5 Reading array from file ...................................... 133
  6.3.6 Finding primes ................................................. 135

7 Structures and C++ ......................................................... 143
  7.1 Structures .............................................................. 143
    7.1.1 Introduction .................................................. 143
    7.1.2 Memory alignment ........................................... 145
    7.1.3 Bit Fields ...................................................... 146
    7.1.4 Using structures in assembly ............................. 150
  7.2 Assembly and C++ ..................................................... 150
    7.2.1 Overloading and Name Mangling ............................ 151
    7.2.2 References .................................................... 153
    7.2.3 Inline functions ............................................ 154
    7.2.4 Classes .......................................................... 156
    7.2.5 Inheritance and Polymorphism ............................ 166
    7.2.6 Other C++ features ....................................... 171

A 80x86 Instructions ...................................................... 173
  A.1 Non-floating Point Instructions ............................... 173
  A.2 Floating Point Instructions .................................. 179

Index ................................................................. 181
Preface

Purpose

The purpose of this book is to give the reader a better understanding of how computers really work at a lower level than in programming languages like Pascal. By gaining a deeper understanding of how computers work, the reader can often be much more productive developing software in higher level languages such as C and C++. Learning to program in assembly language is an excellent way to achieve this goal. Other PC assembly language books still teach how to program the 8086 processor that the original PC used in 1981! The 8086 processor only supported real mode. In this mode, any program may address any memory or device in the computer. This mode is not suitable for a secure, multitasking operating system. This book instead discusses how to program the 80386 and later processors in protected mode (the mode that Windows and Linux runs in). This mode supports the features that modern operating systems expect, such as virtual memory and memory protection. There are several reasons to use protected mode:

1. It is easier to program in protected mode than in the 8086 real mode that other books use.

2. All modern PC operating systems run in protected mode.

3. There is free software available that runs in this mode.

The lack of textbooks for protected mode PC assembly programming is the main reason that the author wrote this book.

As alluded to above, this text makes use of Free/Open Source software: namely, the NASM assembler and the DJGPP C/C++ compiler. Both of these are available to download from the Internet. The text also discusses how to use NASM assembly code under the Linux operating system and with Borland’s and Microsoft’s C/C++ compilers under Windows. Examples for all of these platforms can be found on my web site: http://www.drpaulcarter.com/pcasm. You must download the example code if you wish to assemble and run many of the examples in this tutorial.
Be aware that this text does not attempt to cover every aspect of assembly programming. The author has tried to cover the most important topics that all programmers should be acquainted with.

Acknowledgements

The author would like to thank the many programmers around the world that have contributed to the Free/Open Source movement. All the programs and even this book itself were produced using free software. Specifically, the author would like to thank John S. Fine, Simon Tatham, Julian Hall and others for developing the NASM assembler that all the examples in this book are based on; DJ Delorie for developing the DJGPP C/C++ compiler used; the numerous people who have contributed to the GNU gcc compiler on which DJGPP is based on; Donald Knuth and others for developing the \TeX and \LaTeX\ typesetting languages that were used to produce the book; Richard Stallman (founder of the Free Software Foundation), Linus Torvalds (creator of the Linux kernel) and others who produced the underlying software the author used to produce this work.

Thanks to the following people for corrections:

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D’Imperio
- Jeremiah Lawrence
- Ed Beroset
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates
- Mik Mifflin
- Luke Wallis
- Gaku Ueda
- Brian Heward
• Chad Gorshing
• F. Gotti
• Bob Wilkinson
• Markus Koegel
• Louis Taber
• Dave Kiddell
• Eduardo Horowitz
• Sébastien Le Ray
• Nehal Mistry
• Jianyue Wang
• Jeremias Kleer
• Marc Janicki

Resources on the Internet

<table>
<thead>
<tr>
<th>Resource</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author’s page</td>
<td><a href="http://www.drpaulcarter.com/">http://www.drpaulcarter.com/</a></td>
</tr>
<tr>
<td>NASM SourceForge page</td>
<td><a href="http://sourceforge.net/projects/nasm/">http://sourceforge.net/projects/nasm/</a></td>
</tr>
<tr>
<td>DJGPP</td>
<td><a href="http://www.delorie.com/djgpp">http://www.delorie.com/djgpp</a></td>
</tr>
<tr>
<td>Linux Assembly</td>
<td><a href="http://www.linuxassembly.org/">http://www.linuxassembly.org/</a></td>
</tr>
<tr>
<td>The Art of Assembly</td>
<td><a href="http://webster.cs.ucr.edu/">http://webster.cs.ucr.edu/</a></td>
</tr>
<tr>
<td>USENET</td>
<td>comp.lang.asm.x86</td>
</tr>
<tr>
<td>Intel documentation</td>
<td><a href="http://developer.intel.com/design/Pentium4/documentation.htm">http://developer.intel.com/design/Pentium4/documentation.htm</a></td>
</tr>
</tbody>
</table>

Feedback

The author welcomes any feedback on this work.

E-mail: pacman128@gmail.com
WWW: http://www.drpaulcarter.com/pcasm
Chapter 1

Introduction

1.1 Number Systems

Memory in a computer consists of numbers. Computer memory does not store these numbers in decimal (base 10). Because it greatly simplifies the hardware, computers store all information in a binary (base 2) format. First let’s review the decimal system.

1.1.1 Decimal

Base 10 numbers are composed of 10 possible digits (0-9). Each digit of a number has a power of 10 associated with it based on its position in the number. For example:

\[ 234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 \]

1.1.2 Binary

Base 2 numbers are composed of 2 possible digits (0 and 1). Each digit of a number has a power of 2 associated with it based on its position in the number. (A single binary digit is called a bit.) For example:

\[ 11001_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
\[ = 16 + 8 + 1 \]
\[ = 25 \]

This shows how binary may be converted to decimal. Table 1.1 shows how the first few numbers are represented in binary.

Figure 1.1 shows how individual binary digits (i.e., bits) are added. Here’s an example:
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 1.1: Decimal 0 to 15 in Binary

<table>
<thead>
<tr>
<th>No previous carry</th>
<th>Previous carry</th>
</tr>
</thead>
</table>
| \[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
+0 & +1 & +0 & +1 \\
\hline
0 & 1 & 0 & 0 \\
c & c & c & c
\end{array} \] | \[ \begin{array}{cccc}
0 & 0 & 1 & 1 \\
+0 & +1 & +0 & +1 \\
\hline
0 & 0 & 1 & 1 \\
c & c & c & c
\end{array} \] |

Figure 1.1: Binary addition (c stands for carry)

\[
\begin{array}{c}
11011_2 \\
+10001_2 \\
\hline
101100_2
\end{array}
\]

If one considers the following decimal division:

\[
1234 \div 10 = 123 \, r \, 4
\]

he can see that this division strips off the rightmost decimal digit of the number and shifts the other decimal digits one position to the right. Dividing by two performs a similar operation, but for the binary digits of the number. Consider the following binary division\[1\]

\[
1101_2 \div 10_2 = 110_2 \, r \, 1
\]

This fact can be used to convert a decimal number to its equivalent binary representation as Figure 1.2 shows. This method finds the rightmost digit first, this digit is called the least significant bit (lsb). The leftmost digit is called the most significant bit (msb). The basic unit of memory consists of 8 bits and is called a byte.

\[1\]The 2 subscript is used to show that the number is represented in binary, not decimal
1.1. NUMBER SYSTEMS

1.1.1 Decimal to Binary

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 (\div 2 = 12\ r\ 1)</td>
<td>11001 (\div 10 = 110 \ r\ 1)</td>
</tr>
<tr>
<td>12 (\div 2 = 6\ r\ 0)</td>
<td>1100 (\div 10 = 110 \ r\ 0)</td>
</tr>
<tr>
<td>6 (\div 2 = 3\ r\ 0)</td>
<td>110 (\div 10 = 11 \ r\ 0)</td>
</tr>
<tr>
<td>3 (\div 2 = 1\ r\ 1)</td>
<td>11 (\div 10 = 1 \ r\ 1)</td>
</tr>
<tr>
<td>1 (\div 2 = 0\ r\ 1)</td>
<td>1 (\div 10 = 0 \ r\ 1)</td>
</tr>
</tbody>
</table>

Thus \(25_{10} = 11001_2\)

Figure 1.2: Decimal conversion

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>589 (\div 16 = 36 \ r\ 13)</td>
<td>2BD (\div 16 = 24D)</td>
</tr>
<tr>
<td>36 (\div 16 = 2 \ r\ 4)</td>
<td>16</td>
</tr>
<tr>
<td>2 (\div 16 = 0 \ r\ 2)</td>
<td>16</td>
</tr>
</tbody>
</table>

Thus \(589 = 24D_{16}\)

Figure 1.3:

1.1.3 Hexadecimal

Hexadecimal numbers use base 16. Hexadecimal (or hex for short) can be used as a shorthand for binary numbers. Hex has 16 possible digits. This creates a problem since there are no symbols to use for these extra digits after 9. By convention, letters are used for these extra digits. The 16 hex digits are 0-9 then A, B, C, D, E and F. The digit A is equivalent to 10 in decimal, B is 11, etc. Each digit of a hex number has a power of 16 associated with it. Example:

\[
2BD_{16} = 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0
\]

\[
= 512 + 176 + 13
\]

\[
= 701
\]

To convert from decimal to hex, use the same idea that was used for binary conversion except divide by 16. See Figure 1.3 for an example.

The reason that hex is useful is that there is a very simple way to convert
between hex and binary. Binary numbers get large and cumbersome quickly. Hex provides a much more compact way to represent binary.

To convert a hex number to binary, simply convert each hex digit to a 4-bit binary number. For example, \(24_{16}\) is converted to \(0010\ 0100\ 1101_2\). Note that the leading zeros of the 4-bits are important! If the leading zero for the middle digit of \(24_{16}\) is not used the result is wrong. Converting from binary to hex is just as easy. One does the process in reverse. Convert each 4-bit segments of the binary to hex. Start from the right end, not the left end of the binary number. This ensures that the process uses the correct 4-bit segments.

Example:

\[
\begin{array}{c}
110 & 0000 & 0101 & 1010 & 0111 & 1110_2 \\
6 & 0 & 5 & A & 7 & E_{16}
\end{array}
\]

A 4-bit number is called a nibble. Thus each hex digit corresponds to a nibble. Two nibbles make a byte and so a byte can be represented by a 2-digit hex number. A byte’s value ranges from 0 to 11111111 in binary, 0 to FF in hex and 0 to 255 in decimal.

1.2 Computer Organization

1.2.1 Memory

The basic unit of memory is a byte. A computer with 32 megabytes of memory can hold roughly 32 million bytes of information. Each byte in memory is labeled by a unique number known as its address as Figure 1.4 shows.

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>2A</td>
<td>45</td>
<td>B8</td>
<td>20</td>
<td>8F</td>
<td>CD</td>
<td>12</td>
<td>2E</td>
</tr>
</tbody>
</table>

Figure 1.4: Memory Addresses

Often memory is used in larger chunks than single bytes. On the PC architecture, names have been given to these larger sections of memory as Table 1.2 shows.

All data in memory is numeric. Characters are stored by using a character code that maps numbers to characters. One of the most common character codes is known as ASCII (American Standard Code for Information Interchange). A new, more complete code that is supplanting ASCII is Unicode. One key difference between the two codes is that ASCII uses

\[2\] If it is not clear why the starting point makes a difference, try converting the example starting at the left.
one byte to encode a character, but Unicode uses two bytes (or a word) per character. For example, ASCII maps the byte 4116 (6510) to the character capital \textit{A}; Unicode maps the word 004116. Since ASCII uses a byte, it is limited to only 256 different characters\footnote{In fact, ASCII only uses the lower 7-bits and so only has 128 different values to use.}. Unicode extends the ASCII values to words and allows many more characters to be represented. This is important for representing characters for all the languages of the world.

\subsection*{1.2.2 The CPU}

The Central Processing Unit (CPU) is the physical device that performs instructions. The instructions that CPUs perform are generally very simple. Instructions may require the data they act on to be in special storage locations in the CPU itself called \textit{registers}. The CPU can access data in registers much faster than data in memory. However, the number of registers in a CPU is limited, so the programmer must take care to keep only currently used data in registers.

The instructions a type of CPU executes make up the CPU’s \textit{machine language}. Machine programs have a much more basic structure than higher-level languages. Machine language instructions are encoded as raw numbers, not in friendly text formats. A CPU must be able to decode an instruction’s purpose very quickly to run efficiently. Machine language is designed with this goal in mind, not to be easily deciphered by humans. Programs written in other languages must be converted to the native machine language of the CPU to run on the computer. A \textit{compiler} is a program that translates programs written in a programming language into the machine language of a particular computer architecture. In general, every type of CPU has its own unique machine language. This is one reason why programs written for a Mac can not run on an IBM-type PC.

Computers use a \textit{clock} to synchronize the execution of the instructions. The clock pulses at a fixed frequency (known as the \textit{clock speed}). When you buy a 1.5 GHz computer, 1.5 GHz is the frequency of this clock\footnote{Actually, clock pulses are used in many different components of a computer. The other components often use different clock speeds than the CPU.}. The clock does not keep track of minutes and seconds. It simply beats at a constant frequency.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
word & 2 bytes \\
\hline
double word & 4 bytes \\
\hline
quad word & 8 bytes \\
\hline
paragraph & 16 bytes \\
\hline
\end{tabular}
\caption{Units of Memory}
\end{table}

\textit{GHz stands for gigahertz or one billion cycles per second. A 1.5 GHz CPU has 1.5 billion clock pulses per second.}
rate. The electronics of the CPU uses the beats to perform their operations correctly, like how the beats of a metronome help one play music at the correct rhythm. The number of beats (or as they are usually called cycles) an instruction requires depends on the CPU generation and model. The number of cycles depends on the instructions before it and other factors as well.

1.2.3 The 80x86 family of CPUs

IBM-type PC’s contain a CPU from Intel’s 80x86 family (or a clone of one). The CPUs in this family all have some common features including a base machine language. However, the more recent members greatly enhance the features.

8088,8086: These CPU’s from the programming standpoint are identical. They were the CPU’s used in the earliest PC’s. They provide several 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. They only support up to one megabyte of memory and only operate in real mode. In this mode, a program may access any memory address, even the memory of other programs! This makes debugging and security very difficult! Also, program memory has to be divided into segments. Each segment can not be larger than 64K.

80286: This CPU was used in AT class PC’s. It adds some new instructions to the base machine language of the 8088/86. However, its main new feature is 16-bit protected mode. In this mode, it can access up to 16 megabytes and protect programs from accessing each other’s memory. However, programs are still divided into segments that could not be bigger than 64K.

80386: This CPU greatly enhanced the 80286. First, it extends many of the registers to hold 32-bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) and adds two new 16-bit registers FS and GS. It also adds a new 32-bit protected mode. In this mode, it can access up to 4 gigabytes. Programs are again divided into segments, but now each segment can also be up to 4 gigabytes in size!

80486/Pentium/Pentium Pro: These members of the 80x86 family add very few new features. They mainly speed up the execution of the instructions.

Pentium MMX: This processor adds the MMX (MultiMedia eXtensions) instructions to the Pentium. These instructions can speed up common graphics operations.
1.2. COMPUTER ORGANIZATION

AX

\[
\begin{array}{c|c}
AH & AL \\
\end{array}
\]

Figure 1.5: The AX register

**Pentium II:** This is the Pentium Pro processor with the MMX instructions added. (The Pentium III is essentially just a faster Pentium II.)

1.2.4 8086 16-bit Registers

The original 8086 CPU provided four 16-bit general purpose registers: AX, BX, CX and DX. Each of these registers could be decomposed into two 8-bit registers. For example, the AX register could be decomposed into the AH and AL registers as Figure 1.5 shows. The AH register contains the upper (or high) 8 bits of AX and AL contains the lower 8 bits of AX. Often AH and AL are used as independent one byte registers; however, it is important to realize that they are not independent of AX. Changing AX’s value will change AH and AL and *vice versa*. The general purpose registers are used in many of the data movement and arithmetic instructions.

There are two 16-bit index registers: SI and DI. They are often used as pointers, but can be used for many of the same purposes as the general registers. However, they can not be decomposed into 8-bit registers.

The 16-bit BP and SP registers are used to point to data in the machine language stack and are called the Base Pointer and Stack Pointer, respectively. These will be discussed later.

The 16-bit CS, DS, SS and ES registers are *segment registers*. They denote what memory is used for different parts of a program. CS stands for Code Segment, DS for Data Segment, SS for Stack Segment and ES for Extra Segment. ES is used as a temporary segment register. The details of these registers are in Sections 1.2.6 and 1.2.7.

The Instruction Pointer (IP) register is used with the CS register to keep track of the address of the next instruction to be executed by the CPU. Normally, as an instruction is executed, IP is advanced to point to the next instruction in memory.

The FLAGS register stores important information about the results of a previous instruction. These results are stored as individual bits in the register. For example, the Z bit is 1 if the result of the previous instruction was zero or 0 if not zero. Not all instructions modify the bits in FLAGS, consult the table in the appendix to see how individual instructions affect the FLAGS register.
1.2.5 80386 32-bit registers

The 80386 and later processors have extended registers. For example, the 16-bit AX register is extended to be 32-bits. To be backward compatible, AX still refers to the 16-bit register and EAX is used to refer to the extended 32-bit register. AX is the lower 16-bits of EAX just as AL is the lower 8-bits of AX (and EAX). There is no way to access the upper 16-bits of EAX directly. The other extended registers are EBX, ECX, EDX, ESI and EDI.

Many of the other registers are extended as well. BP becomes EBP; SP becomes ESP; FLAGS becomes EFLAGS and IP becomes EIP. However, unlike the index and general purpose registers, in 32-bit protected mode (discussed below) only the extended versions of these registers are used.

The segment registers are still 16-bit in the 80386. There are also two new segment registers: FS and GS. Their names do not stand for anything. They are extra temporary segment registers (like ES).

One of definitions of the term word refers to the size of the data registers of the CPU. For the 80x86 family, the term is now a little confusing. In Table 1.2, one sees that word is defined to be 2 bytes (or 16 bits). It was given this meaning when the 8086 was first released. When the 80386 was developed, it was decided to leave the definition of word unchanged, even though the register size changed.

1.2.6 Real Mode

In real mode, memory is limited to only one megabyte ($2^{20}$ bytes). Valid address range from (in hex) 00000 to FFFFF. These addresses require a 20-bit number. Obviously, a 20-bit number will not fit into any of the 8086’s 16-bit registers. Intel solved this problem, by using two 16-bit values to determine an address. The first 16-bit value is called the selector. Selector values must be stored in segment registers. The second 16-bit value is called the offset. The physical address referenced by a 32-bit selector:offset pair is computed by the formula

$$16 \times \text{selector} + \text{offset}$$

Multiplying by 16 in hex is easy, just add a 0 to the right of the number. For example, the physical addresses referenced by 047C:0048 is given by:

\[
\begin{array}{c}
047C0 \\
+0048 \\
\hline
04808 \\
\end{array}
\]

In effect, the selector value is a paragraph number (see Table 1.2).

Real segmented addresses have disadvantages:

So where did the infamous DOS 640K limit come from? The BIOS required some of the 1M for its code and for hardware devices like the video screen.
1.2. COMPUTER ORGANIZATION

- A single selector value can only reference 64K of memory (the upper limit of the 16-bit offset). What if a program has more than 64K of code? A single value in CS can not be used for the entire execution of the program. The program must be split up into sections (called segments) less than 64K in size. When execution moves from one segment to another, the value of CS must be changed. Similar problems occur with large amounts of data and the DS register. This can be very awkward!

- Each byte in memory does not have a unique segmented address. The physical address 04808 can be referenced by 047C:0048, 047D:0038, 047E:0028 or 047B:0058. This can complicate the comparison of segmented addresses.

1.2.7 16-bit Protected Mode

In the 80286's 16-bit protected mode, selector values are interpreted completely differently than in real mode. In real mode, a selector value is a paragraph number of physical memory. In protected mode, a selector value is an index into a descriptor table. In both modes, programs are divided into segments. In real mode, these segments are at fixed positions in physical memory and the selector value denotes the paragraph number of the beginning of the segment. In protected mode, the segments are not at fixed positions in physical memory. In fact, they do not have to be in memory at all!

Protected mode uses a technique called virtual memory. The basic idea of a virtual memory system is to only keep the data and code in memory that programs are currently using. Other data and code are stored temporarily on disk until they are needed again. In 16-bit protected mode, segments are moved between memory and disk as needed. When a segment is returned to memory from disk, it is very likely that it will be put into a different area of memory that it was in before being moved to disk. All of this is done transparently by the operating system. The program does not have to be written differently for virtual memory to work.

In protected mode, each segment is assigned an entry in a descriptor table. This entry has all the information that the system needs to know about the segment. This information includes: is it currently in memory; if in memory, where is it; access permissions (e.g., read-only). The index of the entry of the segment is the selector value that is stored in segment registers.

One big disadvantage of 16-bit protected mode is that offsets are still 16-bit quantities. As a consequence of this, segment sizes are still limited to at most 64K. This makes the use of large arrays problematic! One well-known PC columnist called the 286 CPU "brain dead."
1.2.8 32-bit Protected Mode

The 80386 introduced 32-bit protected mode. There are two major differences between 386 32-bit and 286 16-bit protected modes:

1. Offsets are expanded to be 32-bits. This allows an offset to range up to 4 billion. Thus, segments can have sizes up to 4 gigabytes.

2. Segments can be divided into smaller 4K-sized units called pages. The virtual memory system works with pages now instead of segments. This means that only parts of segment may be in memory at any one time. In 286 16-bit mode, either the entire segment is in memory or none of it is. This is not practical with the larger segments that 32-bit mode allows.

In Windows 3.x, standard mode referred to 286 16-bit protected mode and enhanced mode referred to 32-bit mode. Windows 9X, Windows NT/2000/XP, OS/2 and Linux all run in paged 32-bit protected mode.

1.2.9 Interrupts

Sometimes the ordinary flow of a program must be interrupted to process events that require prompt response. The hardware of a computer provides a mechanism called interrupts to handle these events. For example, when a mouse is moved, the mouse hardware interrupts the current program to handle the mouse movement (to move the mouse cursor, etc.) Interrupts cause control to be passed to an interrupt handler. Interrupt handlers are routines that process the interrupt. Each type of interrupt is assigned an integer number. At the beginning of physical memory, a table of interrupt vectors resides that contain the segmented addresses of the interrupt handlers. The number of interrupt is essentially an index into this table.

External interrupts are raised from outside the CPU. (The mouse is an example of this type.) Many I/O devices raise interrupts (e.g., keyboard, timer, disk drives, CD-ROM and sound cards). Internal interrupts are raised from within the CPU, either from an error or the interrupt instruction. Error interrupts are also called traps. Interrupts generated from the interrupt instruction are called software interrupts. DOS uses these types of interrupts to implement its API (Application Programming Interface). More modern operating systems (such as Windows and UNIX) use a C based interface.

Many interrupt handlers return control back to the interrupted program when they finish. They restore all the registers to the same values they had before the interrupt occurred. Thus, the interrupted program runs as if nothing happened (except that it lost some CPU cycles). Traps generally do not return. Often they abort the program.

\footnote{However, they may use a lower level interface at the kernel level.}
1.3 Assembly Language

1.3.1 Machine language

Every type of CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own unique numeric code called its operation code or opcode for short. The 80x86 processor’s instructions vary in size. The opcode is always at the beginning of the instruction. Many instructions also include data (e.g., constants or addresses) used by the instruction.

Machine language is very difficult to program in directly. Deciphering the meanings of the numerical-coded instructions is tedious for humans. For example, the instruction that says to add the EAX and EBX registers together and store the result back into EAX is encoded by the following hex codes:

03 C3

This is hardly obvious. Fortunately, a program called an assembler can do this tedious work for the programmer.

1.3.2 Assembly language

An assembly language program is stored as text (just as a higher level language program). Each assembly instruction represents exactly one machine instruction. For example, the addition instruction described above would be represented in assembly language as:

add eax, ebx

Here the meaning of the instruction is much clearer than in machine code. The word add is a mnemonic for the addition instruction. The general form of an assembly instruction is:

\[ \text{mnemonic operand(s)} \]

An assembler is a program that reads a text file with assembly instructions and converts the assembly into machine code. Compilers are programs that do similar conversions for high-level programming languages. An assembler is much simpler than a compiler. Every assembly language statement directly represents a single machine instruction. High-level language statements are much more complex and may require many machine instructions.

Another important difference between assembly and high-level languages is that since every different type of CPU has its own machine language, it also has its own assembly language. Porting assembly programs between

\[ \text{It took several years for computer scientists to figure out how to even write a compiler!} \]
different computer architectures is much more difficult than in a high-level language.

This book’s examples uses the Netwide Assembler or NASM for short. It is freely available off the Internet (see the preface for the URL). More common assemblers are Microsoft’s Assembler (MASM) or Borland’s Assembler (TASM). There are some differences in the assembly syntax for MASM/-TASM and NASM.

1.3.3 Instruction operands

Machine code instructions have varying number and type of operands; however, in general, each instruction itself will have a fixed number of operands (0 to 3). Operands can have the following types:

register: These operands refer directly to the contents of the CPU’s registers.

memory: These refer to data in memory. The address of the data may be a constant hardcoded into the instruction or may be computed using values of registers. Address are always offsets from the beginning of a segment.

immediate: These are fixed values that are listed in the instruction itself. They are stored in the instruction itself (in the code segment), not in the data segment.

implied: These operands are not explicitly shown. For example, the increment instruction adds one to a register or memory. The one is implied.

1.3.4 Basic instructions

The most basic instruction is the MOV instruction. It moves data from one location to another (like the assignment operator in a high-level language). It takes two operands:

\[
\text{mov dest, src}
\]

The data specified by src is copied to dest. One restriction is that both operands may not be memory operands. This points out another quirk of assembly. There are often somewhat arbitrary rules about how the various instructions are used. The operands must also be the same size. The value of AX can not be stored into BL.

Here is an example (semicolons start a comment):
1.3. ASSEMBLY LANGUAGE

mov eax, 3 ; store 3 into EAX register (3 is immediate operand)
mov bx, ax ; store the value of AX into the BX register

The ADD instruction is used to add integers.

add eax, 4 ; eax = eax + 4
add al, ah ; al = al + ah

The SUB instruction subtracts integers.

sub bx, 10 ; bx = bx - 10
sub ebx, edi ; ebx = ebx - edi

The INC and DEC instructions increment or decrement values by one. Since the one is an implicit operand, the machine code for INC and DEC is smaller than for the equivalent ADD and SUB instructions.

inc ecx ; ecx++
decl dl ; dl--

1.3.5 Directives

A directive is an artifact of the assembler not the CPU. They are generally used to either instruct the assembler to do something or inform the assembler of something. They are not translated into machine code. Common uses of directives are:

- define constants
- define memory to store data into
- group memory into segments
- conditionally include source code
- include other files

NASM code passes through a preprocessor just like C. It has many of the same preprocessor commands as C. However, NASM’s preprocessor directives start with a % instead of a # as in C.

The equ directive

The equ directive can be used to define a symbol. Symbols are named constants that can be used in the assembly program. The format is:

symbol equ value

Symbol values can not be redefined later.
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Unit</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>B</td>
</tr>
<tr>
<td>word</td>
<td>W</td>
</tr>
<tr>
<td>double word</td>
<td>D</td>
</tr>
<tr>
<td>quad word</td>
<td>Q</td>
</tr>
<tr>
<td>ten bytes</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 1.3: Letters for RESX and DX Directives

The %define directive

This directive is similar to C’s #define directive. It is most commonly used to define constant macros just as in C.

%define SIZE 100

    mov eax, SIZE

The above code defines a macro named SIZE and shows its use in a MOV instruction. Macros are more flexible than symbols in two ways. Macros can be redefined and can be more than simple constant numbers.

Data directives

Data directives are used in data segments to define room for memory. There are two ways memory can be reserved. The first way only defines room for data; the second way defines room and an initial value. The first method uses one of the RESX directives. The X is replaced with a letter that determines the size of the object (or objects) that will be stored. Table 1.3 shows the possible values.

The second method (that defines an initial value, too) uses one of the DX directives. The X letters are the same as those in the RESX directives.

It is very common to mark memory locations with labels. Labels allow one to easily refer to memory locations in code. Below are several examples:

L1    db   0 ; byte labeled L1 with initial value 0
L2    dw   1000 ; word labeled L2 with initial value 1000
L3    db   110101b ; byte initialized to binary 110101 (53 in decimal)
L4    db   12h ; byte initialized to hex 12 (18 in decimal)
L5    db   17o ; byte initialized to octal 17 (15 in decimal)
L6    dd   1A92h ; double word initialized to hex 1A92
L7    resb 1 ; 1 uninitialized byte
L8    db   "A" ; byte initialized to ASCII code for A (65)

Double quotes and single quotes are treated the same. Consecutive data definitions are stored sequentially in memory. That is, the word L2 is stored immediately after L1 in memory. Sequences of memory may also be defined.
1.3. ASSEMBLY LANGUAGE

L9    db    0, 1, 2, 3 ; defines 4 bytes
L10   db    "w", "o", "r", 'd', 0 ; defines a C string = "word"
L11   db    'word', 0 ; same as L10

The DD directive can be used to define both integer and single precision floating point constants. However, the DQ can only be used to define double precision floating point constants.

For large sequences, NASM's TIMES directive is often useful. This directive repeats its operand a specified number of times. For example,

L12   times 100 db 0 ; equivalent to 100 (db 0)'s
L13   resw 100 ; reserves room for 100 words

Remember that labels can be used to refer to data in code. There are two ways that a label can be used. If a plain label is used, it is interpreted as the address (or offset) of the data. If the label is placed inside square brackets ([ ]), it is interpreted as the data at the address. In other words, one should think of a label as a pointer to the data and the square brackets dereferences the pointer just as the asterisk does in C. (MASM/TASM follow a different convention.) In 32-bit mode, addresses are 32-bit. Here are some examples:

1   mov    al, [L1] ; copy byte at L1 into AL
2   mov    eax, L1 ; EAX = address of byte at L1
3   mov    [L1], ah ; copy AH into byte at L1
4   mov    eax, [L6] ; copy double word at L6 into EAX
5   add    eax, [L6] ; EAX = EAX + double word at L6
6   add    [L6], eax ; double word at L6 += EAX
7   mov    al, [L6] ; copy first byte of double word at L6 into AL

Line 7 of the examples shows an important property of NASM. The assembler does not keep track of the type of data that a label refers to. It is up to the programmer to make sure that he (or she) uses a label correctly. Later it will be common to store addresses of data in registers and use the register like a pointer variable in C. Again, no checking is made that a pointer is used correctly. In this way, assembly is much more error prone than even C.

Consider the following instruction:

mov    [L6], 1 ; store a 1 at L6

This statement produces an operation size not specified error. Why? Because the assembler does not know whether to store the 1 as a byte, word or double word. To fix this, add a size specifier:

mov    dword [L6], 1 ; store a 1 at L6

---

6Single precision floating point is equivalent to a float variable in C.
This tells the assembler to store an 1 at the double word that starts at L6. Other size specifiers are: BYTE, WORD, QWORD and TWORD.\footnote{TWORD defines a ten byte area of memory. The floating point coprocessor uses this data type.}

1.3.6 Input and Output

Input and output are very system dependent activities. It involves interfacing with the system’s hardware. High level languages, like C, provide standard libraries of routines that provide a simple, uniform programming interface for I/O. Assembly languages provide no standard libraries. They must either directly access hardware (which is a privileged operation in protected mode) or use whatever low level routines that the operating system provides.

It is very common for assembly routines to be interfaced with C. One advantage of this is that the assembly code can use the standard C library I/O routines. However, one must know the rules for passing information between routines that C uses. These rules are too complicated to cover here. (They are covered later!) To simplify I/O, the author has developed his own routines that hide the complex C rules and provide a much more simple interface. Table 1.4 describes the routines provided. All of the routines preserve the value of all registers, except for the read routines. These routines do modify the value of the EAX register. To use these routines, one must include a file with information that the assembler needs to use them. To include a file in NASM, use the \texttt{\%include} preprocessor directive. The following line includes the file needed by the author’s I/O routines\footnote{The \texttt{asm_io.inc} (and the \texttt{asm_io} object file that \texttt{asm_io.inc} requires) are in the example code downloads on the web page for this tutorial, \url{http://www.drpaulcarter.com/pcasm}}:

\texttt{\%include "asm_io.inc"}

To use one of the print routines, one loads EAX with the correct value and uses a \texttt{CALL} instruction to invoke it. The \texttt{CALL} instruction is equivalent to a function call in a high level language. It jumps execution to another section of code, but returns back to its origin after the routine is over. The example program below shows several examples of calls to these I/O routines.

1.3.7 Debugging

The author’s library also contains some useful routines for debugging programs. These debugging routines display information about the state of the computer without modifying the state. These routines are really macros...
1.3. ASSEMBLY LANGUAGE

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>prints out to the screen the value of the integer stored in EAX</td>
</tr>
<tr>
<td>print_char</td>
<td>prints out to the screen the character whose ASCII value stored in AL</td>
</tr>
<tr>
<td>print_string</td>
<td>prints out to the screen the contents of the string at the address stored in EAX. The string must be a C-type string (i.e. null terminated).</td>
</tr>
<tr>
<td>print_nl</td>
<td>prints out to the screen a new line character.</td>
</tr>
<tr>
<td>read_int</td>
<td>reads an integer from the keyboard and stores it into the EAX register.</td>
</tr>
<tr>
<td>read_char</td>
<td>reads a single character from the keyboard and stores its ASCII code into the EAX register.</td>
</tr>
</tbody>
</table>

Table 1.4: Assembly I/O Routines

that preserve the current state of the CPU and then make a subroutine call. The macros are defined in the `asm_io.inc` file discussed above. Macros are used like ordinary instructions. Operands of macros are separated by commas.

There are four debugging routines named `dump_regs`, `dump_mem`, `dump_stack` and `dump_math`; they display the values of registers, memory, stack and the math coprocessor, respectively.

**dump_regs** This macro prints out the values of the registers (in hexadecimal) of the computer to **stdout** (i.e. the screen). It also displays the bits set in the FLAGS register. For example, if the zero flag is 1, ZF is displayed. If it is 0, it is not displayed. It takes a single integer argument that is printed out as well. This can be used to distinguish the output of different `dump_regs` commands.

**dump_mem** This macro prints out the values of a region of memory (in hexadecimal) and also as ASCII characters. It takes three comma delimited arguments. The first is an integer that is used to label the output (just as `dump_regs` argument). The second argument is the address to display. (This can be a label.) The last argument is the number of 16-byte paragraphs to display after the address. The memory displayed will start on the first paragraph boundary before the requested address.

**dump_stack** This macro prints out the values on the CPU stack. (The stack will be covered in Chapter 4.) The stack is organized as double words and this routine displays them this way. It takes three comma

---

9Chapter 2 discusses this register
delimited arguments. The first is an integer label (like \texttt{dumpRegs}). The second is the number of double words to display \textit{below} the address that the EBP register holds and the third argument is the number of double words to display \textit{above} the address in EBP.

\textbf{dump.math} This macro prints out the values of the registers of the math coprocessor. It takes a single integer argument that is used to label the output just as the argument of \texttt{dumpRegs} does.

\section*{1.4 Creating a Program}

Today, it is unusual to create a stand alone program written completely in assembly language. Assembly is usually used to key certain critical routines. Why? It is \textit{much} easier to program in a higher level language than in assembly. Also, using assembly makes a program very hard to port to other platforms. In fact, it is rare to use assembly at all.

So, why should anyone learn assembly at all?

1. Sometimes code written in assembly can be faster and smaller than compiler generated code.

2. Assembly allows access to direct hardware features of the system that might be difficult or impossible to use from a higher level language.

3. Learning to program in assembly helps one gain a deeper understanding of how computers work.

4. Learning to program in assembly helps one understand better how compilers and high level languages like C work.

These last two points demonstrate that learning assembly can be useful even if one never programs in it later. In fact, the author rarely programs in assembly, but he uses the ideas he learned from it everyday.

\subsection*{1.4.1 First program}

The early programs in this text will all start from the simple C driver program in Figure 1.6. It simply calls another function named \texttt{asm\_main}. This is really a routine that will be written in assembly. There are several advantages in using the C driver routine. First, this lets the C system set up the program to run correctly in protected mode. All the segments and their corresponding segment registers will be initialized by C. The assembly code need not worry about any of this. Secondly, the C library will also be available to be used by the assembly code. The author’s I/O routines take
1.4. CREATING A PROGRAM

```c
int main()
{
    int ret_status;
    ret_status = asm_main();
    return ret_status;
}
```

Figure 1.6: driver.c code

advantage of this. They use C’s I/O functions (printf, etc.). The following shows a simple assembly program.

```asm
; file: first.asm
; First assembly program. This program asks for two integers as
; input and prints out their sum.
;
; To create executable using djgpp:
; nasm -f coff first.asm
; gcc -o first first.o driver.c asm_io.o
%
#include "asm_io.inc"
;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output
;
prompt1 db "Enter a number: ", 0 ; don’t forget null terminator
prompt2 db "Enter another number: ", 0
outmsg1 db "You entered ", 0
outmsg2 db " and ", 0
outmsg3 db ", the sum of these is ", 0
;
; uninitialized data is put in the .bss segment
;
segment .bss
;
; These labels refer to double words used to store the inputs
;
input1 resd 1
input2 resd 1

; code is put in the .text segment

segment .text
    global _asm_main

_asm_main:
    enter 0,0 ; setup routine
    pusha

    mov eax, prompt1 ; print out prompt
    call print_string

    call read_int ; read integer
    mov [input1], eax ; store into input1

    mov eax, prompt2 ; print out prompt
    call print_string

    call read_int ; read integer
    mov [input2], eax ; store into input2

    mov eax, [input1] ; eax = dword at input1
    add eax, [input2] ; eax += dword at input2
    mov ebx, eax ; ebx = eax

    dump_regs 1 ; print out register values
    dump_mem 2, outmsg1, 1 ; print out memory

    ; next print out result message as series of steps
    ;
    mov eax, outmsg1
    call print_string ; print out first message
    mov eax, [input1]
    call print_int ; print out input1
    mov eax, outmsg2
    call print_string ; print out second message
    mov eax, [input2]
    call print_int ; print out input2
    mov eax, outmsg3
Line 13 of the program defines a section of the program that specifies memory to be stored in the data segment (whose name is .data). Only initialized data should be defined in this segment. On lines 17 to 21, several strings are declared. They will be printed with the C library and so must be terminated with a null character (ASCII code 0). Remember there is a big difference between 0 and '0'.

Uninitialized data should be declared in the bss segment (named .bss on line 26). This segment gets its name from an early UNIX-based assembler operator that meant “block started by symbol.” There is also a stack segment too. It will be discussed later.

The code segment is named .text historically. It is where instructions are placed. Note that the code label for the main routine (line 38) has an underscore prefix. This is part of the C calling convention. This convention specifies the rules C uses when compiling code. It is very important to know this convention when interfacing C and assembly. Later the entire convention will be presented; however, for now, one only needs to know that all C symbols (i.e., functions and global variables) have a underscore prefix appended to them by the C compiler. (This rule is specifically for DOS/Windows, the Linux C compiler does not prepend anything to C symbol names.)

The global directive on line 37 tells the assembler to make the _asm_main label global. Unlike in C, labels have internal scope by default. This means that only code in the same module can use the label. The global directive gives the specified label (or labels) external scope. This type of label can be accessed by any module in the program. The asm_io module declares the print_int, et.al. labels to be global. This is why one can use them in the first.asm module.
1.4.2 Compiler dependencies

The assembly code above is specific to the free GNU-based DJGPP C/C++ compiler. This compiler can be freely downloaded from the Internet. It requires a 386-based PC or better and runs under DOS, Windows 95/98 or NT. This compiler uses object files in the COFF (Common Object File Format) format. To assemble to this format use the -f coff switch with nasm (as shown in the comments of the above code). The extension of the resulting object file will be .o.

The Linux C compiler is a GNU compiler also. To convert the code above to run under Linux, simply remove the underscore prefixes in lines 37 and 38. Linux uses the ELF (Executable and Linkable Format) format for object files. Use the -f elf switch for Linux. It also produces an object with an .o extension.

Borland C/C++ is another popular compiler. It uses the Microsoft OMF format for object files. Use the -f obj switch for Borland compilers. The extension of the object file will be .obj. The OMF format uses different segment directives than the other object formats. The data segment (line 13) must be changed to:

    segment _DATA public align=4 class=DATA use32

The bss segment (line 26) must be changed to:

    segment _BSS public align=4 class=BSS use32

The text segment (line 36) must be changed to:

    segment _TEXT public align=1 class=CODE use32

In addition a new line should be added before line 36:

    group DGROUP _BSS _DATA

The Microsoft C/C++ compiler can use either the OMF format or the Win32 format for object files. (If given a OMF format, it converts the information to Win32 format internally.) Win32 format allows segments to be defined just as for DJGPP and Linux. Use the -f win32 switch to output in this mode. The extension of the object file will be .obj.

1.4.3 Assembling the code

The first step is to assemble the code. From the command line, type:

    nasm -f object-format first.asm

---

10 GNU is a project of the Free Software Foundation (http://www.fsf.org)
11 http://www.delorie.com/djgpp
where object-format is either coff, elf, obj or win32 depending on what C compiler will be used. (Remember that the source file must be changed for both Linux and Borland as well.)

1.4.4 Compiling the C code

Compile the driver.c file using a C compiler. For DJGPP, use:

```
gcc -c driver.c
```

The -c switch means to just compile, do not attempt to link yet. This same switch works on Linux, Borland and Microsoft compilers as well.

1.4.5 Linking the object files

Linking is the process of combining the machine code and data in object files and library files together to create an executable file. As will be shown below, this process is complicated.

C code requires the standard C library and special startup code to run. It is much easier to let the C compiler call the linker with the correct parameters, than to try to call the linker directly. For example, to link the code for the first program using DJGPP, use:

```
gcc -o first driver.o first.o asm_io.o
```

This creates an executable called first.exe (or just first under Linux).

With Borland, one would use:

```
bcc32 first.obj driver.obj asm_io.obj
```

Borland uses the name of the first file listed to determine the executable name. So in the above case, the program would be named first.exe.

It is possible to combine the compiling and linking step. For example,

```
gcc -o first driver.c first.o asm_io.o
```

Now gcc will compile driver.c and then link.

1.4.6 Understanding an assembly listing file

The -l listing-file switch can be used to tell nasm to create a listing file of a given name. This file shows how the code was assembled. Here is how lines 17 and 18 (in the data segment) appear in the listing file. (The line numbers are in the listing file; however notice that the line numbers in the source file may not be the same as the line numbers in the listing file.)
CHAPTER 1. INTRODUCTION

The first column in each line is the line number and the second is the offset (in hex) of the data in the segment. The third column shows the raw hex values that will be stored. In this case the hex data correspond to ASCII codes. Finally, the text from the source file is displayed on the line. The offsets listed in the second column are very likely not the true offsets that the data will be placed at in the complete program. Each module may define its own labels in the data segment (and the other segments, too). In the link step (see section 1.4.5), all these data segment label definitions are combined to form one data segment. The new final offsets are then computed by the linker.

Here is a small section (lines 54 to 56 of the source file) of the text segment in the listing file:

```
94 0000002C A1[00000000] mov eax, [input1]
95 00000031 0305[04000000] add eax, [input2]
96 00000037 89C3 mov ebx, eax
```

The third column shows the machine code generated by the assembly. Often the complete code for an instruction cannot be computed yet. For example, in line 94 the offset (or address) of input1 is not known until the code is linked. The assembler can compute the op-code for the mov instruction (which from the listing is A1), but it writes the offset in square brackets because the exact value cannot be computed yet. In this case, a temporary offset of 0 is used because input1 is at the beginning of the part of the bss segment defined in this file. Remember this does not mean that it will be at the beginning of the final bss segment of the program. When the code is linked, the linker will insert the correct offset into the position. Other instructions, like line 96, do not reference any labels. Here the assembler can compute the complete machine code.

**Big and Little Endian Representation**

If one looks closely at line 95, something seems very strange about the offset in the square brackets of the machine code. The input2 label is at offset 4 (as defined in this file); however, the offset that appears in memory is not 00000004, but 04000000. Why? Different processors store multibyte integers in different orders in memory. There are two popular methods of storing integers: big endian and little endian. Big endian is the method

*Endian is pronounced like*
that seems the most natural. The biggest (i.e. most significant) byte is stored first, then the next biggest, etc. For example, the dword 00000004 would be stored as the four bytes 00 00 00 04. IBM mainframes, most RISC processors and Motorola processors all use this big endian method. However, Intel-based processors use the little endian method! Here the least significant byte is stored first. So, 00000004 is stored in memory as 04 00 00 00. This format is hardwired into the CPU and can not be changed. Normally, the programmer does not need to worry about which format is used. However, there are circumstances where it is important.

1. When binary data is transferred between different computers (either from files or through a network).

2. When binary data is written out to memory as a multibyte integer and then read back as individual bytes or vice versa.

Endianness does not apply to the order of array elements. The first element of an array is always at the lowest address. This applies to strings (which are just character arrays). Endianness still applies to the individual elements of the arrays.

1.5 Skeleton File

Figure 1.7 shows a skeleton file that can be used as a starting point for writing assembly programs.
skel.asm

#include "asm_io.inc"

section .data
;
; initialized data is put in the data segment here
;

section .bss
;
; uninitialized data is put in the bss segment
;

section .text
global _asm_main

_asm_main:
  enter 0,0 ; setup routine
  pusha

; code is put in the text segment. Do not modify the code before
; or after this comment.
;
  popa
  mov eax, 0 ; return back to C
  leave
  ret

Figure 1.7: Skeleton Program
2.1 Working with Integers

2.1.1 Integer representation

Integers come in two flavors: unsigned and signed. Unsigned integers (which are non-negative) are represented in a very straightforward binary manner. The number 200 as an one byte unsigned integer would be represented as by 11001000 (or C8 in hex).

Signed integers (which may be positive or negative) are represented in a more complicated ways. For example, consider −56. +56 as a byte would be represented by 00111000. On paper, one could represent −56 as −111000, but how would this be represented in a byte in the computer’s memory. How would the minus sign be stored?

There are three general techniques that have been used to represent signed integers in computer memory. All of these methods use the most significant bit of the integer as a sign bit. This bit is 0 if the number is positive and 1 if negative.

Signed magnitude

The first method is the simplest and is called signed magnitude. It represents the integer as two parts. The first part is the sign bit and the second is the magnitude of the integer. So 56 would be represented as the byte 00111000 (the sign bit is underlined) and −56 would be 10111000. The largest byte value would be 01111111 or +127 and the smallest byte value would be 11111111 or −127. To negate a value, the sign bit is reversed. This method is straightforward, but it does have its drawbacks. First, there are two possible values of zero, +0 (00000000) and −0 (10000000). Since zero is neither positive nor negative, both of these representations should act the same. This complicates the logic of arithmetic for the CPU. Secondly,
general arithmetic is also complicated. If 10 is added to −56, this must be recast as 10 subtracted by 56. Again, this complicates the logic of the CPU.

One's complement

The second method is known as one's complement representation. The one's complement of a number is found by reversing each bit in the number. (Another way to look at it is that the new bit value is $1 - \text{old bit value}$.) For example, the one’s complement of 00111000 (+56) is 11000111. In one’s complement notation, computing the one’s complement is equivalent to negation. Thus, 11000111 is the representation for −56. Note that the sign bit was automatically changed by one’s complement and that as one would expect taking the one’s complement twice yields the original number. As for the first method, there are two representations of zero: 00000000 (+0) and 11111111 (−0). Arithmetic with one’s complement numbers is complicated.

There is a handy trick to finding the one’s complement of a number in hexadecimal without converting it to binary. The trick is to subtract the hex digit from F (or 15 in decimal). This method assumes that the number of bits in the number is a multiple of 4. Here is an example: +56 is represented by 38 in hex. To find the one’s complement, subtract each digit from F to get C7 in hex. This agrees with the result above.

Two’s complement

The first two methods described were used on early computers. Modern computers use a third method called two’s complement representation. The two’s complement of a number is found by the following two steps:

1. Find the one’s complement of the number
2. Add one to the result of step 1

Here’s an example using 00111000 (56). First the one’s complement is computed: 11000111. Then one is added:

\[
\begin{array}{c}
\text{11000111} \\
+ \text{1} \\
\hline
\text{11001000}
\end{array}
\]

In two complement’s notation, computing the two’s complement is equivalent to negating a number. Thus, 11001000 is the two’s complement representation of −56. Two negations should reproduce the original number. Surprising two’s complement does meet this requirement. Take the two’s
2.1. WORKING WITH INTEGERS

<table>
<thead>
<tr>
<th>Number</th>
<th>Hex Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>-128</td>
<td>80</td>
</tr>
<tr>
<td>-127</td>
<td>81</td>
</tr>
<tr>
<td>-2</td>
<td>FE</td>
</tr>
<tr>
<td>-1</td>
<td>FF</td>
</tr>
</tbody>
</table>

Table 2.1: Two’s Complement Representation

The complement of 11001000 by adding one to the one’s complement.

\[
\begin{array}{c}
00110111 \\
+ 1 \\
00111000
\end{array}
\]

When performing the addition in the two’s complement operation, the addition of the leftmost bit may produce a carry. This carry is not used. Remember that all data on the computer is of some fixed size (in terms of number of bits). Adding two bytes always produces a byte as a result (just as adding two words produces a word, etc.) This property is important for two’s complement notation. For example, consider zero as a one byte two’s complement number (0000000). Computing its two complement produces the sum:

\[
\begin{array}{c}
11111111 \\
+ 1 \\
c 00000000
\end{array}
\]

where \( c \) represents a carry. (Later it will be shown how to detect this carry, but it is not stored in the result.) Thus, in two’s complement notation there is only one zero. This makes two’s complement arithmetic simpler than the previous methods.

Using two’s complement notation, a signed byte can be used to represent the numbers −128 to +127. Table 2.1 shows some selected values. If 16 bits are used, the signed numbers −32,768 to +32,767 can be represented. +32,767 is represented by 7FFF, −32,768 by 8000, -128 as FF80 and -1 as FFFF. 32 bit two’s complement numbers range from −2 billion to +2 billion approximately.

The CPU has no idea what a particular byte (or word or double word) is supposed to represent. Assembly does not have the idea of types that a high level language has. How data is interpreted depends on what instruction is used on the data. Whether the hex value FF is considered to represent a signed −1 or a unsigned +255 depends on the programmer. The C language
defines signed and unsigned integer types. This allows a C compiler to
determine the correct instructions to use with the data.

2.1.2 Sign extension

In assembly, all data has a specified size. It is not uncommon to need
to change the size of data to use it with other data. Decreasing size is the
easiest.

Decreasing size of data

To decrease the size of data, simply remove the more significant bits of
the data. Here's a trivial example:

\[
\text{mov ax, 0034h} \quad ; \text{ax = 52 (stored in 16 bits)} \\
\text{mov cl, al} \quad ; \text{cl = lower 8-bits of ax}
\]

Of course, if the number can not be represented correctly in the smaller
size, decreasing the size does not work. For example, if AX were 0134h (or
308 in decimal) then the above code would still set CL to 34h. This method
works with both signed and unsigned numbers. Consider signed numbers,
if AX was FFFFh (\(-1\) as a word), then CL would be FFh (\(-1\) as a byte).
However, note that this is not correct if the value in AX was unsigned!

The rule for unsigned numbers is that all the bits being removed must
be 0 for the conversion to be correct. The rule for signed numbers is that
the bits being removed must be either all 1's or all 0's. In addition, the first
bit not being removed must have the same value as the removed bits. This
bit will be the new sign bit of the smaller value. It is important that it be
same as the original sign bit!

Increasing size of data

Increasing the size of data is more complicated than decreasing. Consider
the hex byte FF. If it is extended to a word, what value should the word
have? It depends on how FF is interpreted. If FF is a unsigned byte (255
in decimal), then the word should be 00FF; however, if it is a signed byte
\((-1\) in decimal), then the word should be FFFF.

In general, to extend an unsigned number, one makes all the new bits
of the expanded number 0. Thus, FF becomes 00FF. However, to extend
a signed number, one must extend the sign bit. This means that the new
bits become copies of the sign bit. Since the sign bit of FF is 1, the new
bits must also be all ones, to produce FFFF. If the signed number 5A (90
in decimal) was extended, the result would be 005A.
There are several instructions that the 80386 provides for extension of numbers. Remember that the computer does not know whether a number is signed or unsigned. It is up to the programmer to use the correct instruction.

For unsigned numbers, one can simply put zeros in the upper bits using a MOV instruction. For example, to extend the byte in AL to an unsigned word in AX:

\[
\text{mov} \quad \text{ah}, \ 0 \quad ; \text{zero out upper 8-bits}
\]

However, it is not possible to use a MOV instruction to convert the unsigned word in AX to an unsigned double word in EAX. Why not? There is no way to specify the upper 16 bits of EAX in a MOV. The 80386 solves this problem by providing a new instruction MOVZX. This instruction has two operands. The destination (first operand) must be a 16 or 32 bit register. The source (second operand) may be an 8 or 16 bit register or a byte or word of memory. The other restriction is that the destination must be larger than the source. (Most instructions require the source and destination to be the same size.) Here are some examples:

\[
\begin{align*}
\text{movzx} & \quad \text{eax, ax} \quad \text{;} \text{ extends ax into eax} \\
\text{movzx} & \quad \text{eax, al} \quad \text{;} \text{ extends al into eax} \\
\text{movzx} & \quad \text{ax, al} \quad \text{;} \text{ extends al into ax} \\
\text{movzx} & \quad \text{ebx, ax} \quad \text{;} \text{ extends ax into ebx}
\end{align*}
\]

For signed numbers, there is no easy way to use the MOV instruction for any case. The 8086 provided several instructions to extend signed numbers. The CBW (Convert Byte to Word) instruction sign extends the AL register into AX. The operands are implicit. The CWD (Convert Word to Double word) instruction sign extends AX into DX:AX. The notation DX:AX means to think of the DX and AX registers as one 32 bit register with the upper 16 bits in DX and the lower bits in AX. (Remember that the 8086 did not have any 32 bit registers!) The 80386 added several new instructions. The CWDE (Convert Word to Double word Extended) instruction sign extends AX into EAX. The CDQ (Convert Double word to Quad word) instruction sign extends EAX into EDX:EAX (64 bits!). Finally, the MOVZX instruction works like MOVZX except it uses the rules for signed numbers.

**Application to C programming**

Extending of unsigned and signed integers also occurs in C. Variables in C may be declared as either signed or unsigned (int is signed). Consider the code in Figure 2.1. In line 3, the variable a is extended using the rules for unsigned values (using MOVZX), but in line 4, the signed rules are used for b (using MOVZX).
(1) unsigned char uchar = 0xFF;
(2) signed char schar = 0xFF;
(3) int a = (int) uchar; /* a = 255 (0x000000FF) */
(4) int b = (int) schar; /* b = -1 (0xFFFFFFFF) */

Figure 2.1:

char ch;
while( (ch = fgetc(fp)) != EOF ) {
    /* do something with ch */
}

Figure 2.2:

There is a common C programming bug that directly relates to this subject. Consider the code in Figure 2.2. The prototype of fgetc() is:

    int fgetc( FILE * );

One might question why does the function return back an int since it reads characters? The reason is that it normally does return back an char (extended to an int value using zero extension). However, there is one value that it may return that is not a character, EOF. This is a macro that is usually defined as -1. Thus, fgetc() either returns back a char extended to an int value (which looks like 000000FF in hex) or EOF (which looks like FFFFFFFF in hex).

The basic problem with the program in Figure 2.2 is that fgetc() returns an int, but this value is stored in a char. C will truncate the higher order bits to fit the int value into the char. The only problem is that the numbers (in hex) 000000FF and FFFFFFFF both will be truncated to the byte FF. Thus, the while loop can not distinguish between reading the byte FF from the file and end of file.

Exactly what the code does in this case, depends on whether char is signed or unsigned. Why? Because in line 2, ch is compared with EOF. Since EOF is an int value, ch will be extended to an int so that two values being compared are of the same size. As Figure 2.1 showed, where the variable is signed or unsigned is very important.

If char is unsigned, FF is extended to be 000000FF. This is compared to EOF (FFFFFFF) and found to be not equal. Thus, the loop never ends!

---

1 It is a common misconception that files have an EOF character at their end. This is not true!
2 The reason for this requirement will be shown later.
2.1. WORKING WITH INTEGERS

If char is signed, FF is extended to FFFFFFFF. This does compare as equal and the loop ends. However, since the byte FF may have been read from the file, the loop could be ending prematurely.

The solution to this problem is to define the ch variable as an int, not a char. When this is done, no truncating or extension is done in line 2. Inside the loop, it is safe to truncate the value since ch must actually be a simple byte there.

2.1.3 Two’s complement arithmetic

As was seen earlier, the add instruction performs addition and the sub instruction performs subtraction. Two of the bits in the FLAGS register that these instructions set are the overflow and carry flag. The overflow flag is set if the true result of the operation is too big to fit into the destination for signed arithmetic. The carry flag is set if there is a carry in the msb of an addition or a borrow in the msb of a subtraction. Thus, it can be used to detect overflow for unsigned arithmetic. The uses of the carry flag for signed arithmetic will be seen shortly. One of the great advantages of 2’s complement is that the rules for addition and subtraction are exactly the same as for unsigned integers. Thus, add and sub may be used on signed or unsigned integers.

\[
\begin{array}{c c c}
002C & + & 44 \\
+ FFFF & + & (-1) \\
002B & + & 43 \\
\end{array}
\]

There is a carry generated, but it is not part of the answer.

There are two different multiply and divide instructions. First, to multiply use either the MUL or IMUL instruction. The MUL instruction is used to multiply unsigned numbers and IMUL is used to multiply signed integers. Why are two different instructions needed? The rules for multiplication are different for unsigned and 2’s complement signed numbers. How so? Consider the multiplication of the byte FF with itself yielding a word result. Using unsigned multiplication this is 255 times 255 or 65025 (or FE01 in hex). Using signed multiplication this is \(-1 \times -1 = 1\) (or 0001 in hex).

There are several forms of the multiplication instructions. The oldest form looks like:

\texttt{mul source}

The source is either a register or a memory reference. It can not be an immediate value. Exactly what multiplication is performed depends on the size of the source operand. If the operand is byte sized, it is multiplied by the byte in the AL register and the result is stored in the 16 bits of AX. If the source is 16-bit, it is multiplied by the word in AX and the 32-bit result
CHAPTER 2. BASIC ASSEMBLY LANGUAGE

<table>
<thead>
<tr>
<th>dest</th>
<th>source1</th>
<th>source2</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg/mem8</td>
<td>reg/mem16</td>
<td>AX = AL*source1</td>
<td></td>
</tr>
<tr>
<td>reg/mem16</td>
<td>reg/mem32</td>
<td>DX:AX = AX*source1</td>
<td></td>
</tr>
<tr>
<td>reg/mem16</td>
<td>reg/mem32</td>
<td>EDX:EAX = EAX*source1</td>
<td></td>
</tr>
<tr>
<td>reg16</td>
<td>reg/mem16</td>
<td>dest *= source1</td>
<td></td>
</tr>
<tr>
<td>reg32</td>
<td>reg/mem32</td>
<td>dest *= source1</td>
<td></td>
</tr>
<tr>
<td>reg16</td>
<td>immed8</td>
<td>dest *= immed8</td>
<td></td>
</tr>
<tr>
<td>reg32</td>
<td>immed8</td>
<td>dest *= immed8</td>
<td></td>
</tr>
<tr>
<td>reg16</td>
<td>immed16</td>
<td>dest *= immed16</td>
<td></td>
</tr>
<tr>
<td>reg32</td>
<td>immed32</td>
<td>dest *= immed32</td>
<td></td>
</tr>
<tr>
<td>reg16</td>
<td>reg/mem16, immed8</td>
<td>dest = source1*source2</td>
<td></td>
</tr>
<tr>
<td>reg32</td>
<td>reg/mem32, immed8</td>
<td>dest = source1*source2</td>
<td></td>
</tr>
<tr>
<td>reg16</td>
<td>reg/mem16, immed16</td>
<td>dest = source1*source2</td>
<td></td>
</tr>
<tr>
<td>reg32</td>
<td>reg/mem32, immed32</td>
<td>dest = source1*source2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: imul Instructions

is stored in DX:AX. If the source is 32-bit, it is multiplied by EAX and the
64-bit result is stored into EDX:EAX.

The IMUL instruction has the same formats as MUL, but also adds some
other instruction formats. There are two and three operand formats:

\[
\begin{align*}
\text{imul} & \quad \text{dest, source1} \\
\text{imul} & \quad \text{dest, source1, source2}
\end{align*}
\]

Table 2.2 shows the possible combinations.

The two division operators are DIV and IDIV. They perform unsigned
and signed integer division respectively. The general format is:

\[
\text{div} \quad \text{source}
\]

If the source is 8-bit, then AX is divided by the operand. The quotient is
stored in AL and the remainder in AH. If the source is 16-bit, then DX:AX
is divided by the operand. The quotient is stored into AX and remainder
into DX. If the source is 32-bit, then EDX:EAX is divided by the operand
and the quotient is stored into EAX and the remainder into EDX. The IDIV
instruction works the same way. There are no special IDIV instructions like
the special IMUL ones. If the quotient is too big to fit into its register or the
divisor is zero, the program is interrupted and terminates. A very common
error is to forget to initialize DX or EDX before division.

The NEG instruction negates its single operand by computing its two’s
complement. Its operand may be any 8-bit, 16-bit, or 32-bit register or
memory location.
2.1. WORKING WITH INTEGERS

2.1.4 Example program

```asm
; include "asm_io.inc"
segment .data ; Output strings
prompt db "Enter a number: ", 0
square_msg db "Square of input is ", 0
cube_msg db "Cube of input is ", 0
cube25_msg db "Cube of input times 25 is ", 0
quot_msg db "Quotient of cube/100 is ", 0
rem_msg db "Remainder of cube/100 is ", 0
neg_msg db "The negation of the remainder is ", 0

segment .bss
input resd 1

segment .text
    global _asm_main

_asm_main:        enter 0,0 ; setup routine
    pusha

    mov eax, prompt
    call print_string

    call read_int
    mov [input], eax

    imul eax ; edx:eax = eax * eax
    mov ebx, eax ; save answer in ebx
    mov eax, square_msg
    call print_string
    mov eax, ebx
    call print_int
    call print_nl

    mov ebx, [input] ; ebx *= [input]
    imul ebx, [input] ; edx:eax = ebx * [input]
    mov eax, cube_msg
    call print_string
    mov eax, ebx
    call print_int
    call print_nl
```

---

This program includes a segment for output strings, a bss section for input storage, and a text section containing the main assembly code. The program prompts the user to enter a number, then calculates the square and cube of the input, and performs additional calculations based on the cube.
imul    ecx, ebx, 25  ; ecx = ebx*25
mov     eax, cube25_msg
call    print_string
mov     eax, ecx
call    print_int
call    print_nl

mov     eax, ebx
       ; initialize edx by sign extension
cdq
mov     ecx, 100  ; can’t divide by immediate value
idiv    ecx  ; edx:eax / ecx
mov     ecx, eax  ; save quotient into ecx
mov     eax, quot_msg
call    print_string
mov     eax, ecx
call    print_int
call    print_nl
mov     eax, rem_msg
call    print_string
mov     eax, edx
call    print_int
call    print_nl

neg     edx  ; negate the remainder
mov     eax, neg_msg
call    print_string
mov     eax, edx
call    print_int
call    print_nl

popa
mov     eax, 0  ; return back to C
leave
ret

2.1.5 Extended precision arithmetic

Assembly language also provides instructions that allow one to perform
addition and subtraction of numbers larger than double words. These in-
structions use the carry flag. As stated above, both the ADD and SUB instruc-
tions modify the carry flag if a carry or borrow are generated, respectively.
This information stored in the carry flag can be used to add or subtract large numbers by breaking up the operation into smaller double word (or smaller) pieces.

The ADC and SBB instructions use this information in the carry flag. The ADC instruction performs the following operation:

\[
\text{operand1} = \text{operand1} + \text{carry flag} + \text{operand2}
\]

The SBB instruction performs:

\[
\text{operand1} = \text{operand1} - \text{carry flag} - \text{operand2}
\]

How are these used? Consider the sum of 64-bit integers in EDX:EAX and EBX:ECX. The following code would store the sum in EDX:EAX:

```assembly
1    add    eax, ecx ; add lower 32-bits
2    adc    edx, ebx ; add upper 32-bits and carry from previous sum
```

Subtraction is very similar. The following code subtracts EBX:ECX from EDX:EAX:

```assembly
1    sub    eax, ecx ; subtract lower 32-bits
2    sbb    edx, ebx ; subtract upper 32-bits and borrow
```

For really large numbers, a loop could be used (see Section 2.2). For a sum loop, it would be convenient to use ADC instruction for every iteration (instead of all but the first iteration). This can be done by using the CLC (CLear Carry) instruction right before the loop starts to initialize the carry flag to 0. If the carry flag is 0, there is no difference between the ADD and ADC instructions. The same idea can be used for subtraction, too.

## 2.2 Control Structures

High level languages provide high level control structures (e.g., the if and while statements) that control the thread of execution. Assembly language does not provide such complex control structures. It instead uses the infamous goto and used inappropriately can result in spaghetti code! However, it is possible to write structured assembly language programs. The basic procedure is to design the program logic using the familiar high level control structures and translate the design into the appropriate assembly language (much like a compiler would do).

### 2.2.1 Comparisons

Control structures decide what to do based on comparisons of data. In assembly, the result of a comparison is stored in the FLAGS register to be
used later. The 80x86 provides the *CMP* instruction to perform comparisons. The *FLAGS* register is set based on the difference of the two operands of the *CMP* instruction. The operands are subtracted and the *FLAGS* are set based on the result, but the result is *not* stored anywhere. If you need the result use the *SUB* instead of the *CMP* instruction.

For unsigned integers, there are two flags (bits in the *FLAGS* register) that are important: the zero (ZF) and carry (CF) flags. The zero flag is set (1) if the resulting difference would be zero. The carry flag is used as a borrow flag for subtraction. Consider a comparison like:

\[
\text{cmp \hspace{1em} vleft, vright}
\]

The difference of \( vleft - vright \) is computed and the flags are set accordingly. If the difference of the of *CMP* is zero, \( vleft = vright \), then ZF is set (\( i.e. 1 \)) and the CF is unset (\( i.e. 0 \)). If \( vleft > vright \), then ZF is unset and CF is unset (no borrow). If \( vleft < vright \), then ZF is unset and CF is set (borrow).

For signed integers, there are three flags that are important: the zero (ZF) flag, the overflow (OF) flag and the sign (SF) flag. The overflow flag is set if the result of an operation overflows (or underflows). The sign flag is set if the result of an operation is negative. If \( vleft = vright \), the ZF is set (just as for unsigned integers). If \( vleft > vright \), ZF is unset and SF = OF. If \( vleft < vright \), ZF is unset and SF ≠ OF.

Do not forget that other instructions can also change the *FLAGS* register, not just *CMP*.

### 2.2.2 Branch instructions

Branch instructions can transfer execution to arbitrary points of a program. In other words, they act like a *goto*. There are two types of branches: unconditional and conditional. An unconditional branch is just like a *goto*, it always makes the branch. A conditional branch may or may not make the branch depending on the flags in the *FLAGS* register. If a conditional branch does not make the branch, control passes to the next instruction.

The *JMP* (short for *jump*) instruction makes unconditional branches. Its single argument is usually a *code label* to the instruction to branch to. The assembler or linker will replace the label with correct address of the instruction. This is another one of the tedious operations that the assembler does to make the programmer’s life easier. It is important to realize that the statement immediately after the *JMP* instruction will never be executed unless another instruction branches to it!

There are several variations of the jump instruction:

**SHORT** This jump is very limited in range. It can only move up or down 128 bytes in memory. The advantage of this type is that it uses less
memory than the others. It uses a single signed byte to store the displacement of the jump. The displacement is how many bytes to move ahead or behind. (The displacement is added to EIP). To specify a short jump, use the SHORT keyword immediately before the label in the JMP instruction.

**NEAR** This jump is the default type for both unconditional and conditional branches, it can be used to jump to any location in a segment. Actually, the 80386 supports two types of near jumps. One uses two bytes for the displacement. This allows one to move up or down roughly 32,000 bytes. The other type uses four bytes for the displacement, which of course allows one to move to any location in the code segment. The four byte type is the default in 386 protected mode. The two byte type can be specified by putting the WORD keyword before the label in the JMP instruction.

**FAR** This jump allows control to move to another code segment. This is a very rare thing to do in 386 protected mode.

Valid code labels follow the same rules as data labels. Code labels are defined by placing them in the code segment in front of the statement they label. A colon is placed at the end of the label at its point of definition. The colon is not part of the name.

There are many different conditional branch instructions. They also take a code label as their single operand. The simplest ones just look at a single flag in the FLAGS register to determine whether to branch or not. See Table 2.3 for a list of these instructions. (PF is the parity flag which indicates the odd or evenness of the number of bits set in the lower 8-bits of the result.)

The following pseudo-code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>branches only if ZF is set</td>
</tr>
<tr>
<td>JNZ</td>
<td>branches only if ZF is unset</td>
</tr>
<tr>
<td>JO</td>
<td>branches only if OF is set</td>
</tr>
<tr>
<td>JNO</td>
<td>branches only if OF is unset</td>
</tr>
<tr>
<td>JS</td>
<td>branches only if SF is set</td>
</tr>
<tr>
<td>JNS</td>
<td>branches only if SF is unset</td>
</tr>
<tr>
<td>JC</td>
<td>branches only if CF is set</td>
</tr>
<tr>
<td>JNC</td>
<td>branches only if CF is unset</td>
</tr>
<tr>
<td>JP</td>
<td>branches only if PF is set</td>
</tr>
<tr>
<td>JNP</td>
<td>branches only if PF is unset</td>
</tr>
</tbody>
</table>

Table 2.3: Simple Conditional Branches
Chapter 2. Basic Assembly Language

if ( EAX == 0 )
    EBX = 1;
else
    EBX = 2;

could be written in assembly as:

    1  cmp   eax, 0 ; set flags (ZF set if eax - 0 = 0)
    2  jz    thenblock ; if ZF set branch to thenblock
    3  mov   ebx, 2 ; ELSE part of IF
    4  jmp   next ; jump over THEN part of IF
    5  thenblock:
    6  mov   ebx, 1 ; THEN part of IF
    7  next:

Other comparisons are not so easy using the conditional branches in Table 2.3. To illustrate, consider the following pseudo-code:

if ( EAX >= 5 )
    EBX = 1;
else
    EBX = 2;

If EAX is greater than or equal to five, the ZF may be set or unset and SF will equal OF. Here is assembly code that tests for these conditions (assuming that EAX is signed):

    1  cmp   eax, 5 ; set flags (ZF set if eax - 0 = 0)
    2  js    signon ; if SF = 1 goto signon
    3  jo    elseblock ; if OF = 1 and SF = 0 goto elseblock
    4  jmp   thenblock ; if SF = 0 and OF = 0 goto thenblock
    5  signon:
    6  jo    thenblock ; goto thenblock if SF = 1 and OF = 1
    7  elseblock:
    8  mov   ebx, 2
    9  jmp   next
   10  thenblock:
   11  mov   ebx, 1
   12  next:

The above code is very awkward. Fortunately, the 80x86 provides additional branch instructions to make these type of tests much easier. There are signed and unsigned versions of each. Table 2.4 shows these instructions. The equal and not equal branches (JE and JNE) are the same for both signed and unsigned integers. (In fact, JE and JNE are really identical
2.2. CONTROL STRUCTURES

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE branches if vleft = vright</td>
<td>JE branches if vleft = vright</td>
</tr>
<tr>
<td>JNE branches if vleft ≠ vright</td>
<td>JNE branches if vleft ≠ vright</td>
</tr>
<tr>
<td>JL, JNGE branches if vleft &lt; vright</td>
<td>JB, JNAE branches if vleft &lt; vright</td>
</tr>
<tr>
<td>JLE, JNG branches if vleft ≤ vright</td>
<td>JBE, JNA branches if vleft ≤ vright</td>
</tr>
<tr>
<td>JG, JNLE branches if vleft &gt; vright</td>
<td>JA, JNBE branches if vleft &gt; vright</td>
</tr>
<tr>
<td>JGE, JNL branches if vleft ≥ vright</td>
<td>JAE, JNB branches if vleft ≥ vright</td>
</tr>
</tbody>
</table>

Table 2.4: Signed and Unsigned Comparison Instructions

to JZ and JNZ, respectively.) Each of the other branch instructions have two synonyms. For example, look at JL (jump less than) and JNGE (jump not greater than or equal to). These are the same instruction because:

\[ x < y \iff \text{not}(x \geq y) \]

The unsigned branches use A for above and B for below instead of L and G.

Using these new branch instructions, the pseudo-code above can be translated to assembly much easier.

```assembly
1    cmp   eax, 5
2    jge  thenblock
3    mov   ebx, 2
4    jmp   next
5    thenblock:
6    mov   ebx, 1
7    next:
```

2.2.3 The loop instructions

The 80x86 provides several instructions designed to implement for-like loops. Each of these instructions takes a code label as its single operand.

**LOOP** Decrement ECX, if ECX ≠ 0, branches to label

**LOOPE, LOOPZ** Decrement ECX (FLAGS register is not modified), if ECX ≠ 0 and ZF = 1, branches

**LOOPNE, LOOPNZ** Decrement ECX (FLAGS unchanged), if ECX ≠ 0 and ZF = 0, branches

The last two loop instructions are useful for sequential search loops. The following pseudo-code:
CHAPTER 2. BASIC ASSEMBLY LANGUAGE

sum = 0;
for ( i=10; i > 0; i-- )
    sum += i;

could be translated into assembly as:

1. mov eax, 0 ; eax is sum
2. mov ecx, 10 ; ecx is i
3. loop_start:
4.    add eax, ecx
5.    loop loop_start

2.3 Translating Standard Control Structures

This section looks at how the standard control structures of high level languages can be implemented in assembly language.

2.3.1 If statements

The following pseudo-code:

if ( condition )
    then_block;
else
    else_block;

could be implemented as:

1. ; code to set FLAGS
2. jxx else_block ; select xx so that branches if condition false
3. ; code for then block
4. jmp endif
5. else_block:
6. ; code for else block
7. endif:

If there is no else, then the else_block branch can be replaced by a branch to endif.

1. ; code to set FLAGS
2. jxx endif ; select xx so that branches if condition false
3. ; code for then block
4. endif:
2.3.2 While loops

The *while* loop is a top tested loop:

```plaintext
while( condition ){
    body of loop;
}
```

This could be translated into:

```plaintext
1 while:
2     ; code to set FLAGS based on condition
3     jxx endwhile ; select xx so that branches if false
4     ; body of loop
5     jmp while
6 endwhile:
```

2.3.3 Do while loops

The *do while* loop is a bottom tested loop:

```plaintext
do {
    body of loop;
} while( condition );
```

This could be translated into:

```plaintext
1 do:
2     ; body of loop
3     ; code to set FLAGS based on condition
4     jxx do ; select xx so that branches if true
```

2.4 Example: Finding Prime Numbers

This section looks at a program that finds prime numbers. Recall that prime numbers are evenly divisible by only 1 and themselves. There is no formula for doing this. The basic method this program uses is to find the factors of all odd numbers below a given limit. If no factor can be found for an odd number, it is prime. Figure 2.3 shows the basic algorithm written in C.

Here's the assembly version:

3.2 is the only even prime number.
unsigned guess;  /* current guess for prime  */
unsigned factor;  /* possible factor of guess  */
unsigned limit;  /* find primes up to this value */

printf(“Find primes up to: ”);
scanf(“%u”, &limit);
prompt(“2
”);  /* treat first two primes as */
prompt(“3
”);  /* special case */
guess = 5;  /* initial guess */
while ( guess <= limit ) {
    /* look for a factor of guess */
    factor = 3;
    while ( factor*factor < guess &&
            guess % factor != 0 )
        factor += 2;
    if ( guess % factor != 0 )
        prompt(“%d
”, guess);
    guess += 2;  /* only look at odd numbers */
}

Figure 2.3:

--prime.asm---------------------------------------------------

%include "asm_io.inc"

segment .data
Message  db  "Find primes up to: ", 0

segment .bss
Limit    resd  1 ; find primes up to this limit
Guess    resd  1 ; the current guess for prime

segment .text
_global _asm_main
_asm_main:
    enter 0,0 ; setup routine
    pusha

    mov  eax, Message
call print_string
call read_int ; scanf("%u", & limit );
mov  [Limit], eax

--prime.asm---------------------------------------------------
2.4. **EXAMPLE: FINDING PRIME NUMBERS**

```
20    mov    eax, 2          ; printf("2\n");
21    call   print_int
22    call   print_nl
23    mov    eax, 3          ; printf("3\n");
24    call   print_int
25    call   print_nl

26    mov    dword [Guess], 5      ; Guess = 5;
27    while_limit:             ; while ( Guess <= Limit )
28    mov    eax,[Guess]
29    cmp    eax, [Limit]
30    jnbe   end_while_limit    ; use jnbe since numbers are unsigned
31
32    mov    ebx, 3            ; ebx is factor = 3;
33    while_factor:
34    mov    eax,ebx
35    mul    eax              ; edx:eax = eax*eax
36    jo     end_while_factor ; if answer won't fit in eax alone
37    cmp    eax, [Guess]
38    jnb    end_while_factor ; if !(factor*factor < guess)
39    mov    eax,[Guess]
40    mov    edx,0
41    div    ebx              ; edx = edx:eax % ebx
42    cmp    edx, 0
43    je     end_while_factor ; if !(guess % factor != 0)
44
45    add    ebx,2            ; factor += 2;
46    jmp    while_factor
47    end_while_factor:
48    je     end_if            ; if !(guess % factor != 0)
49    mov    eax,[Guess]
50    call   print_int
51    call   print_nl
52    end_if:
53    add    dword [Guess], 2   ; guess += 2
54    jmp    end_while_limit
55    end_while_limit:
56
57    popa
58    mov    eax, 0           ; return back to C
59    leave
60    ret
```

_____________________________ prime.asm ______________________________
Chapter 3

Bit Operations

3.1 Shift Operations

Assembly language allows the programmer to manipulate the individual bits of data. One common bit operation is called a *shift*. A shift operation moves the position of the bits of some data. Shifts can be either toward the left (i.e. toward the most significant bits) or toward the right (the least significant bits).

3.1.1 Logical shifts

A logical shift is the simplest type of shift. It shifts in a very straightforward manner. Figure 3.1 shows an example of a shifted single byte number.

<table>
<thead>
<tr>
<th>Original</th>
<th>Left shifted</th>
<th>Right shifted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1 0 1 0</td>
<td>1 1 0 1 0 1 0 0</td>
<td>0 1 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

Figure 3.1: Logical shifts

Note that new, incoming bits are always zero. The *SHL* and *SHR* instructions are used to perform logical left and right shifts respectively. These instructions allow one to shift by any number of positions. The number of positions to shift can either be a constant or can be stored in the CL register. The last bit shifted out of the data is stored in the carry flag. Here are some code examples:

```assembly
1 mov ax, 0C123H
2 shl ax, 1 ; shift 1 bit to left, ax = 8246H, CF = 1
3 shr ax, 1 ; shift 1 bit to right, ax = 4123H, CF = 0
4 shr ax, 1 ; shift 1 bit to right, ax = 2091H, CF = 1
5 mov ax, 0C123H
```
CHAPTER 3. BIT OPERATIONS

3.1.2 Use of shifts

Fast multiplication and division are the most common uses of a shift operations. Recall that in the decimal system, multiplication and division by a power of ten are simple, just shift digits. The same is true for powers of two in binary. For example, to double the binary number 1011 (or 11 in decimal), shift once to the left to get 10110 (or 22). The quotient of a division by a power of two is the result of a right shift. To divide by just 2, use a single right shift; to divide by 4 ($2^2$), shift right 2 places; to divide by 8 ($2^3$), shift 3 places to the right, etc. Shift instructions are very basic and are much faster than the corresponding MUL and DIV instructions!

Actually, logical shifts can be used to multiply and divide unsigned values. They do not work in general for signed values. Consider the 2-byte value FFFF (signed $-1$). If it is logically right shifted once, the result is 7FFF which is +32,767! Another type of shift can be used for signed values.

3.1.3 Arithmetic shifts

These shifts are designed to allow signed numbers to be quickly multiplied and divided by powers of 2. They insure that the sign bit is treated correctly.

**SAL** Shift Arithmetic Left - This instruction is just a synonym for SHL. It is assembled into the exactly the same machine code as SHL. As long as the sign bit is not changed by the shift, the result will be correct.

**SAR** Shift Arithmetic Right - This is a new instruction that does not shift the sign bit (*i.e.* the msb) of its operand. The other bits are shifted as normal except that the new bits that enter from the left are copies of the sign bit (that is, if the sign bit is 1, the new bits are also 1). Thus, if a byte is shifted with this instruction, only the lower 7 bits are shifted. As for the other shifts, the last bit shifted out is stored in the carry flag.
3.1. **SHIFT OPERATIONS**

3.1.4 **Rotate shifts**

The rotate shift instructions work like logical shifts except that bits lost off one end of the data are shifted in on the other side. Thus, the data is treated as if it is a circular structure. The two simplest rotate instructions are **ROL** and **ROR** which make left and right rotations, respectively. Just as for the other shifts, these shifts leave a copy of the last bit shifted around in the carry flag.

1. `mov ax, 0C123H`
2. `rol ax, 1 ; ax = 8247H, CF = 1`
3. `rol ax, 1 ; ax = 048FH, CF = 1`
4. `rol ax, 1 ; ax = 091EH, CF = 0`
5. `ror ax, 2 ; ax = 8247H, CF = 1`
6. `ror ax, 1 ; ax = C123H, CF = 1`

There are two additional rotate instructions that shift the bits in the data and the carry flag named **RCL** and **RCR**. For example, if the **AX** register is rotated with these instructions, the 17-bits made up of **AX** and the carry flag are rotated.

1. `mov ax, 0C123H`
2. `clc ; clear the carry flag (CF = 0)`
3. `rcl ax, 1 ; ax = 8246H, CF = 1`
4. `rcl ax, 1 ; ax = 048DH, CF = 1`
5. `rcl ax, 1 ; ax = 091BH, CF = 0`
6. `rcr ax, 2 ; ax = 8246H, CF = 1`
7. `rcr ax, 1 ; ax = C123H, CF = 0`

3.1.5 **Simple application**

Here is a code snippet that counts the number of bits that are “on” (i.e. 1) in the EAX register.

1. `mov bl, 0 ; bl will contain the count of ON bits`
2. `mov ecx, 32 ; ecx is the loop counter`
3. `count_loop:`
4. `shl eax, 1 ; shift bit into carry flag`
5. `jnc skip_inc ; if CF == 0, goto skip_inc`
6. `inc bl`
7. `skip_inc:`
8. `loop count_loop`
3.2 Boolean Bitwise Operations

There are four common boolean operators: AND, OR, XOR and NOT. A truth table shows the result of each operation for each possible value of its operands.

### 3.2.1 The AND operation

The result of the AND of two bits is only 1 if both bits are 1, else the result is 0 as the truth table in Table 3.1 shows.

Processors support these operations as instructions that act independently on all the bits of data in parallel. For example, if the contents of AL and BL are ANDed together, the basic AND operation is applied to each of the 8 pairs of corresponding bits in the two registers as Figure 3.2 shows. Below is a code example:

```
1 mov ax, 0C123H
2 and ax, 82F6H ; ax = 8022H
```

### 3.2.2 The OR operation

The inclusive OR of 2 bits is 0 only if both bits are 0, else the result is 1 as the truth table in Table 3.2 shows. Below is a code example:

```
1 mov ax, 0C123H
2 or ax, 0E831H ; ax = E933H
```
3.2. BOOLEAN BITWISE OPERATIONS

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X OR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: The OR operation

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X XOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.3: The XOR operation

3.2.3 The XOR operation

The exclusive OR of 2 bits is 0 if and only if both bits are equal, else the result is 1 as the truth table in Table 3.3 shows. Below is a code example:

```
1       mov   ax, 0C123H
2       xor   ax, 0E831H  ; ax = 2912H
```

3.2.4 The NOT operation

The NOT operation is a unary operation (i.e. it acts on one operand, not two like binary operations such as AND). The NOT of a bit is the opposite value of the bit as the truth table in Table 3.4 shows. Below is a code example:

```
1       mov   ax, 0C123H
2       not   ax          ; ax = 3EDCH
```

Note that the NOT finds the one's complement. Unlike the other bitwise operations, the NOT instruction does not change any of the bits in the FLAGS register.

3.2.5 The TEST instruction

The TEST instruction performs an AND operation, but does not store the result. It only sets the FLAGS register based on what the result would be (much like how the CMP instruction performs a subtraction but only sets FLAGS). For example, if the result would be zero, ZF would be set.
### Not Operation

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\text{NOT } X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: The NOT operation

- **Turn on bit $i$**  
  *OR* the number with $2^i$ (which is the binary number with just bit $i$ on)

- **Turn off bit $i$**  
  *AND* the number with the binary number with only bit $i$ off. This operand is often called a *mask*

- **Complement bit $i$**  
  *XOR* the number with $2^i$

### Uses of Bit Operations

Bit operations are very useful for manipulating individual bits of data without modifying the other bits. Table 3.5 shows three common uses of these operations. Below is some example code, implementing these ideas.

```assembly
1 mov ax, 0C123H
2 or  ax, 8      ; turn on bit 3, ax = C12BH
3 and ax, 0FFDFH ; turn off bit 5, ax = C10BH
4 xor ax, 8000H ; invert bit 31, ax = 410BH
5 or  ax, 0F00H ; turn on nibble, ax = 4F0BH
6 and ax, 0FFF0H ; turn off nibble, ax = 4F00H
7 xor ax, 0F00FH ; invert nibbles, ax = BF0FH
8 xor ax, 0FFFFH ; 1's complement, ax = 40F0H
```

The *AND* operation can also be used to find the remainder of a division by a power of two. To find the remainder of a division by $2^i$, *AND* the number with a mask equal to $2^i - 1$. This mask will contain ones from bit 0 up to bit $i - 1$. It is just these bits that contain the remainder. The result of the *AND* will keep these bits and zero out the others. Next is a snippet of code that finds the quotient and remainder of the division of 100 by 16.

```assembly
1 mov eax, 100 ; 100 = 64H
2 mov ebx, 0000000FH ; mask = 16 - 1 = 15 or F
3 and ebx, eax ; ebx = remainder = 4
```

Using the CL register it is possible to modify arbitrary bits of data. Next is an example that sets (turns on) an arbitrary bit in EAX. The number of the bit to set is stored in BH.

```assembly
1 mov eax, 0C123H
2 or  ax, 8
3 and ax, 0FFDFH
4 xor ax, 8000H
5 and ax, 0FFF0H
6 xor ax, 0F00FH
7 xor ax, 0FFFFH
```
3.3. AVOIDING CONDITIONAL BRANCHES

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov bl, 0</td>
<td>bl will contain the count of ON bits</td>
</tr>
<tr>
<td>2</td>
<td>mov ecx, 32</td>
<td>ecx is the loop counter</td>
</tr>
<tr>
<td>3</td>
<td>count_loop:</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shl eax, 1</td>
<td>shift bit into carry flag</td>
</tr>
<tr>
<td>5</td>
<td>adc bl, 0</td>
<td>add just the carry flag to bl</td>
</tr>
<tr>
<td>6</td>
<td>loop count_loop</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Counting bits with ADC

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov cl, bh</td>
<td>first build the number to OR with</td>
</tr>
<tr>
<td>2</td>
<td>mov ebx, 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shl ebx, cl</td>
<td>shift left cl times</td>
</tr>
<tr>
<td>4</td>
<td>or eax, ebx</td>
<td>turn on bit</td>
</tr>
</tbody>
</table>

Turning a bit off is just a little harder.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov cl, bh</td>
<td>first build the number to AND with</td>
</tr>
<tr>
<td>2</td>
<td>mov ebx, 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shl ebx, cl</td>
<td>shift left cl times</td>
</tr>
<tr>
<td>4</td>
<td>not ebx</td>
<td>invert bits</td>
</tr>
<tr>
<td>5</td>
<td>and eax, ebx</td>
<td>turn off bit</td>
</tr>
</tbody>
</table>

Code to complement an arbitrary bit is left as an exercise for the reader.

It is not uncommon to see the following puzzling instruction in a 80x86 program:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xor eax, eax</td>
<td>eax = 0</td>
</tr>
</tbody>
</table>

A number XOR’ed with itself always results in zero. This instruction is used because its machine code is smaller than the corresponding MOV instruction.

3.3 Avoiding Conditional Branches

Modern processors use very sophisticated techniques to execute code as quickly as possible. One common technique is known as speculative execution. This technique uses the parallel processing capabilities of the CPU to execute multiple instructions at once. Conditional branches present a problem with this idea. The processor, in general, does not know whether the branch will be taken or not. If it is taken, a different set of instructions will be executed than if it is not taken. Processors try to predict whether the branch will be taken. If the prediction is wrong, the processor has wasted its time executing the wrong code.
One way to avoid this problem is to avoid using conditional branches when possible. The sample code in [3.1.5] provides a simple example of where one could do this. In the previous example, the “on” bits of the EAX register are counted. It uses a branch to skip the INC instruction. Figure 3.3 shows how the branch can be removed by using the ADC instruction to add the carry flag directly.

The SETxx instructions provide a way to remove branches in certain cases. These instructions set the value of a byte register or memory location to zero or one based on the state of the FLAGS register. The characters after SET are the same characters used for conditional branches. If the corresponding condition of the SETxx is true, the result stored is a one, if false a zero is stored. For example,

```plaintext
setz al ; AL = 1 if Z flag is set, else 0
```

Using these instructions, one can develop some clever techniques that calculate values without branches.

For example, consider the problem of finding the maximum of two values. The standard approach to solving this problem would be to use a CMP and use a conditional branch to act on which value was larger. The example program below shows how the maximum can be found without any branches.

```plaintext
; file: max.asm
%include "asm_io.inc"
segment .data

message1 db "Enter a number: ",0
message2 db "Enter another number: ", 0
message3 db "The larger number is: ", 0

segment .bss

input1 resd 1 ; first number entered

segment .text
global _asm_main

_asm_main:
enter 0,0 ; setup routine
pusha

mov eax, message1 ; print out first message
call print_string

mov eax, message2 ; print out second message
call print_string

call read_int ; input first number
```
### 3.3. AVOIDING CONDITIONAL BRANCHES

```assembly
mov [input1], eax

mov eax, message2 ; print out second message
call print_string

mov eax, message2 ; print out second message
call print_string

xor ebx, ebx ; ebx = 0
cmp eax, [input1] ; compare second and first number
setg bl ; ebx = (input2 > input1) ? 1 : 0
neg ebx ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
mov ecx, ebx ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
and ecx, eax ; ecx = (input2 > input1) ? input2 : 0
not ebx ; ebx = (input2 > input1) ? 0 : 0xFFFFFFFF
and ebx, [input1] ; ebx = (input2 > input1) ? 0 : input1
or ecx, ebx ; ecx = (input2 > input1) ? input2 : input1

mov eax, message3 ; print out result
call print_string
mov eax, ecx
call print_int
call print_nl

popa
mov eax, 0 ; return back to C
leave
ret
```

The trick is to create a bit mask that can be used to select the correct value for the maximum. The `SETG` instruction in line 30 sets BL to 1 if the second input is the maximum or 0 otherwise. This is not quite the bit mask desired. To create the required bit mask, line 31 uses the `NEG` instruction on the entire EBX register. (Note that EBX was zeroed out earlier.) If EBX is 0, this does nothing; however, if EBX is 1, the result is the two’s complement representation of -1 or 0xFFFFFFFF. This is just the bit mask required. The remaining code uses this bit mask to select the correct input as the maximum.

An alternative trick is to use the `DEC` statement. In the above code, if the `NEG` is replaced with a `DEC`, again the result will either be 0 or 0xFFFFFFFF. However, the values are reversed than when using the `NEG` instruction.
3.4 Manipulating bits in C

3.4.1 The bitwise operators of C

Unlike some high-level languages, C does provide operators for bitwise operations. The AND operation is represented by the binary \& operator.\(^1\) The OR operation is represented by the binary | operator. The XOR operation is represented by the binary ^ operator. And the NOT operation is represented by the unary ~ operator.

The shift operations are performed by C’s << and >> binary operators. The << operator performs left shifts and the >> operator performs right shifts. These operators take two operands. The left operand is the value to shift and the right operand is the number of bits to shift by. If the value to shift is an unsigned type, a logical shift is made. If the value is a signed type (like int), then an arithmetic shift is used. Below is some example C code using these operators:

```c
1 short int s;  /* assume that short int is 16-bit */
2 short unsigned u;
3 s = −1;        /* s = 0xFFFF (2’s complement) */
4 u = 100;       /* u = 0x0064 */
5 u = u | 0x0100; /* u = 0x0164 */
6 s = s & 0xFFF0; /* s = 0xFFF0 */
7 s = s ^ u;     /* s = 0xFE94 */
8 u = u << 3;    /* u = 0x0B20 (logical shift ) */
9 s = s >> 2;    /* s = 0xFFA5 (arithmetic shift ) */
```

3.4.2 Using bitwise operators in C

The bitwise operators are used in C for the same purposes as they are used in assembly language. They allow one to manipulate individual bits of data and can be used for fast multiplication and division. In fact, a smart C compiler will use a shift for a multiplication like, \(x \ast = 2\), automatically.

Many operating system API\(^2\)'s (such as POSIX\(^3\) and Win32) contain functions which use operands that have data encoded as bits. For example, POSIX systems maintain file permissions for three different types of users: user (a better name would be owner), group and others. Each type of user can be granted permission to read, write and/or execute a file. To change the permissions of a file requires the C programmer to manipulate individual bits. POSIX defines several macros to help (see Table 3.6). The

---
\(^1\)This operator is different from the binary && and unary & operators!
\(^2\)Application Programming Interface
\(^3\)stands for Portable Operating System Interface for Computer Environments. A standard developed by the IEEE based on UNIX.
3.5. BIG AND LITTLE ENDIAN REPRESENTATIONS

<table>
<thead>
<tr>
<th>Macro</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>user can read</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>user can write</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>user can execute</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>group can read</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>group can write</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>group can execute</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>others can read</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>others can write</td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>others can execute</td>
</tr>
</tbody>
</table>

Table 3.6: POSIX File Permission Macros

The chmod function can be used to set the permissions of a file. This function takes two parameters, a string with the name of the file to act on and an integer with the appropriate bits set for the desired permissions. For example, the code below sets the permissions to allow the owner of the file to read and write to it, users in the group to read the file and others have no access.

```plaintext
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP);
```

The POSIX stat function can be used to find out the current permission bits for the file. Used with the chmod function, it is possible to modify some of the permissions without changing others. Here is an example that removes write access to others and adds read access to the owner of the file. The other permissions are not altered.

```plaintext
struct stat file_stats; /* struct used by stat() */
stat("foo", & file_stats); /* read file info. */
    file_stats.st_mode holds permission bits */
chmod("foo", ( file_stats.st_mode & ~S_IWOTH) | S_IRUSR);
```

3.5 Big and Little Endian Representations

Chapter 1 introduced the concept of big and little endian representations of multibyte data. However, the author has found that this subject confuses many people. This section covers the topic in more detail.

The reader will recall that endianness refers to the order that the individual bytes (not bits) of a multibyte data element is stored in memory. Big endian is the most straightforward method. It stores the most significant byte first, then the next significant byte and so on. In other words the big bits are stored first. Little endian stores the bytes in the opposite

---

4 Actually a parameter of type mode_t which is a typedef to an integral type.
order (least significant first). The x86 family of processors use little endian representation.

As an example, consider the double word representing 12345678₁₆. In big endian representation, the bytes would be stored as 12 34 56 78. In little endian representation, the bytes would be stored as 78 56 34 12.

The reader is probably asking himself right now, why any sane chip designer would use little endian representation? Were the engineers at Intel sadists for inflicting this confusing representations on multitudes of programmers? It would seem that the CPU has to do extra work to store the bytes backward in memory like this (and to unreverse them when read back in to memory). The answer is that the CPU does not do any extra work to write and read memory using little endian format. One has to realize that the CPU is composed of many electronic circuits that simply work on bit values. The bits (and bytes) are not in any necessary order in the CPU.

Consider the 2-byte AX register. It can be decomposed into the single byte registers: AH and AL. There are circuits in the CPU that maintain the values of AH and AL. Circuits are not in any order in a CPU. That is, the circuits for AH are not before or after the circuits for AL. A mov instruction that copies the value of AX to memory copies the value of AL then AH. This is not any harder for the CPU to do than storing AH first.

The same argument applies to the individual bits in a byte. They are not really in any order in the circuits of the CPU (or memory for that matter). However, since individual bits can not be addressed in the CPU or memory, there is no way to know (or care about) what order they seem to be kept internally by the CPU.

The C code in Figure 3.4 shows how the endianness of a CPU can be determined. The p pointer treats the word variable as a two element character array. Thus, p[0] evaluates to the first byte of word in memory which depends on the endianness of the CPU.

```c
unsigned short word = 0x1234; /* assumes sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
    printf("Big Endian Machine\n");
else
    printf(" Little Endian Machine\n");
```

Figure 3.4: How to Determine Endianness
3.5. BIG AND LITTLE ENDIAN REPRESENTATIONS

unsigned invert_endian ( unsigned x )
{
    unsigned invert;
    const unsigned char * xp = (const unsigned char *) &x;
    unsigned char * ip = (unsigned char *) & invert;

    ip[0] = xp[3]; /* reverse the individual bytes */
    ip[1] = xp[2];
    ip[2] = xp[1];
    ip[3] = xp[0];

    return invert; /* return the bytes reversed */
}

Figure 3.5: invert_endian Function

3.5.1 When to Care About Little and Big Endian

For typical programming, the endianness of the CPU is not significant. The most common time that it is important is when binary data is transferred between different computer systems. This is usually either using some type of physical data media (such as a disk) or a network. Since ASCII data is single byte, endianness is not an issue for it.

All internal TCP/IP headers store integers in big endian format (called network byte order). TCP/IP libraries provide C functions for dealing with endianness issues in a portable way. For example, the htonl() function converts a double word (or long integer) from host to network format. The ntohl() function performs the opposite transformation. For a big endian system, the two functions just return their input unchanged. This allows one to write network programs that will compile and run correctly on any system irrespective of its endianness. For more information, about endianness and network programming see W. Richard Steven’s excellent book, UNIX Network Programming.

Figure 3.5 shows a C function that inverts the endianness of a double word. The 486 processor introduced a new machine instruction named BSWAP that reverses the bytes of any 32-bit register. For example,

bswap edx ; swap bytes of edx

The instruction can not be used on 16-bit registers. However, the XCHG

---

With the advent of multi-byte character sets, like UNICODE, endianness is important for even text data. UNICODE supports either endianness and has a mechanism for specifying which endianness is being used to represent the data.

---

\(^5\)Actually, reversing the endianness of an integer simply reverses the bytes; thus, converting from big to little or little to big is the same operation. So both of these functions do the same thing.
int count_bits ( unsigned int data )
{
    int cnt = 0;

    while ( data != 0 ) {
        data = data & (data - 1);
        cnt++;
    }
    return cnt;
}

Figure 3.6: Bit Counting: Method One

instruction can be used to swap the bytes of the 16-bit registers that can be decomposed into 8-bit registers. For example:

xchg ah,al ; swap bytes of ax

3.6 Counting Bits

Earlier a straightforward technique was given for counting the number of bits that are “on” in a double word. This section looks at other less direct methods of doing this as an exercise using the bit operations discussed in this chapter.

3.6.1 Method one

The first method is very simple, but not obvious. Figure 3.6 shows the code.

How does this method work? In every iteration of the loop, one bit is turned off in data. When all the bits are off (i.e. when data is zero), the loop stops. The number of iterations required to make data zero is equal to the number of bits in the original value of data.

Line 6 is where a bit of data is turned off. How does this work? Consider the general form of the binary representation of data and the rightmost 1 in this representation. By definition, every bit after this 1 must be zero. Now, what will be the binary representation of data - 1? The bits to the left of the rightmost 1 will be the same as for data, but at the point of the rightmost 1 the bits will be the complement of the original bits of data. For example:

data = xxxxxx10000
data - 1 = xxxxxx01111
3.6. COUNTING BITS

```c
static unsigned char byte_bit_count [256]; /* lookup table */

void initialize_count_bits ()
{
    int cnt, i, data;

    for ( i = 0; i < 256; i++ )
    {
        cnt = 0;
        data = i;
        while ( data != 0 )
        {
            data = data & (data - 1);
            cnt++;
        }
        byte_bit_count [i] = cnt;
    }
}

int count_bits ( unsigned int data )
{
    const unsigned char * byte = ( unsigned char *) & data;
    return byte_bit_count [byte[0]] + byte_bit_count [byte[1]] +
    byte_bit_count [byte[2]] + byte_bit_count [byte [3]];
}
```

Figure 3.7: Method Two

where the x’s are the same for both numbers. When `data` is `AND`ed with `data - 1`, the result will zero the rightmost 1 in `data` and leave all the other bits unchanged.

3.6.2 Method two

A lookup table can also be used to count the bits of an arbitrary double word. The straightforward approach would be to precompute the number of bits for each double word and store this in an array. However, there are two related problems with this approach. There are roughly 4 billion double word values! This means that the array will be very big and that initializing it will also be very time consuming. (In fact, unless one is going to actually use the array more than 4 billion times, more time will be taken to initialize the array than it would require to just compute the bit counts using method one!)
A more realistic method would precompute the bit counts for all possible byte values and store these into an array. Then the double word can be split up into four byte values. The bit counts of these four byte values are looked up from the array and summed to find the bit count of the original double word. Figure 3.7 shows the code implementing this approach.

The `initialize_count_bits` function must be called before the first call to the `count_bits` function. This function initializes the global `byte_bit_count` array. The `count_bits` function looks at the `data` variable not as a double word, but as an array of four bytes. The `dword` pointer acts as a pointer to this four byte array. Thus, `dword[0]` is one of the bytes in `data` (either the least significant or the most significant byte depending on if the hardware is little or big endian, respectively.) Of course, one could use a construction like:

```
(data >> 24) & 0x000000FF
```

to find the most significant byte value and similar ones for the other bytes; however, these constructions will be slower than an array reference.

One last point, a for loop could easily be used to compute the sum on lines 22 and 23. But, a for loop would include the overhead of initializing a loop index, comparing the index after each iteration and incrementing the index. Computing the sum as the explicit sum of four values will be faster. In fact, a smart compiler would convert the for loop version to the explicit sum. This process of reducing or eliminating loop iterations is a compiler optimization technique known as loop unrolling.

### 3.6.3 Method three

There is yet another clever method of counting the bits that are on in data. This method literally adds the one’s and zero’s of the data together. This sum must equal the number of one’s in the data. For example, consider counting the one’s in a byte stored in a variable named `data`. The first step is to perform the following operation:

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

What does this do? The hex constant 0x55 is 01010101 in binary. In the first operand of the addition, `data` is AND’ed with this, bits at the odd bit positions are pulled out. The second operand `((data >> 1) & 0x55)` first moves all the bits at the even positions to an odd position and uses the same mask to pull out these same bits. Now, the first operand contains the odd bits and the second operand the even bits of `data`. When these two operands are added together, the even and odd bits of `data` are added together. For example, if `data` is 10110011₂, then:
3.6. COUNTING BITS

```c
int count_bits(unsigned int x)
{
    static unsigned int mask[] = {
        0x55555555,
        0x33333333,
        0x0F0F0F0F,
        0x00FF00FF,
        0x0000FFFF
    };

    int i;
    int shift;  /* number of positions to shift to right */

    for ( i=0, shift =1; i < 5; i++, shift *= 2 )
        x = (x & mask[i]) + ( (x >> shift) & mask[i] );

    return x;
}
```

Figure 3.8: Method 3

The addition on the right shows the actual bits added together. The bits of the byte are divided into four 2-bit fields to show that actually there are four independent additions being performed. Since the most these sums can be is two, there is no possibility that the sum will overflow its field and corrupt one of the other field’s sums.

Of course, the total number of bits have not been computed yet. However, the same technique that was used above can be used to compute the total in a series of similar steps. The next step would be:

\[ \text{data} = (\text{data} \& 0x33) + ((\text{data} \gg 2) \& 0x33); \]

Continuing the above example (remember that \text{data} now is 01100010_2):

\[
\begin{array}{c}
\text{data} \& 01010101_2 \\
+ (\text{data} \gg 1) \& 01010101_2 \\
\end{array}
\begin{array}{cccc}
00 & 01 & 00 & 01 \\
01 & 10 & 00 & 10 \\
\end{array}
\]

Now there are two 4-bit fields to that are independently added.

The next step is to add these two bit sums together to form the final result:

\[ \text{data} = (\text{data} \& 0x0F) + ((\text{data} \gg 4) \& 0x0F); \]

Using the example above (with \text{data} equal to 00110010_2):

\[
\begin{array}{c}
\text{data} \& 00000011_2 \\
+ (\text{data} \gg 2) \& 00000011_2 \\
\end{array}
\begin{array}{cccc}
0010 & 0010 \\
0001 & 0000 \\
0011 & 0010 \\
\end{array}
\]

Now there are four 1-bit fields to that are independently added.
Now data is 5 which is the correct result. Figure 3.8 shows an implementation of this method that counts the bits in a double word. It uses a for loop to compute the sum. It would be faster to unroll the loop; however, the loop makes it clearer how the method generalizes to different sizes of data.
Chapter 4

Subprograms

This chapter looks at using subprograms to make modular programs and to interface with high level languages (like C). Functions and procedures are high level language examples of subprograms.

The code that calls a subprogram and the subprogram itself must agree on how data will be passed between them. These rules on how data will be passed are called calling conventions. A large part of this chapter will deal with the standard C calling conventions that can be used to interface assembly subprograms with C programs. This (and other conventions) often pass the addresses of data (i.e. pointers) to allow the subprogram to access the data in memory.

4.1 Indirect Addressing

Indirect addressing allows registers to act like pointer variables. To indicate that a register is to be used indirectly as a pointer, it is enclosed in square brackets ([ ]). For example:

1. mov ax, [Data] ; normal direct memory addressing of a word
2. mov ebx, Data ; ebx = & Data
3. mov ax, [ebx] ; ax = *ebx

Because AX holds a word, line 3 reads a word starting at the address stored in EBX. If AX was replaced with AL, only a single byte would be read. It is important to realize that registers do not have types like variables do in C. What EBX is assumed to point to is completely determined by what instructions are used. Furthermore, even the fact that EBX is a pointer is completely determined by the what instructions are used. If EBX is used incorrectly, often there will be no assembler error; however, the program will not work correctly. This is one of the many reasons that assembly programming is more error prone than high level programming.
All the 32-bit general purpose (EAX, EBX, ECX, EDX) and index (ESI, EDI) registers can be used for indirect addressing. In general, the 16-bit and 8-bit registers can not be.

### 4.2 Simple Subprogram Example

A subprogram is an independent unit of code that can be used from different parts of a program. In other words, a subprogram is like a function in C. A jump can be used to invoke the subprogram, but returning presents a problem. If the subprogram is to be used by different parts of the program, it must return back to the section of code that invoked it. Thus, the jump back from the subprogram can not be hard coded to a label. The code below shows how this could be done using the indirect form of the JMP instruction. This form of the instruction uses the value of a register to determine where to jump to (thus, the register acts much like a function pointer in C.) Here is the first program from chapter 1 rewritten to use a subprogram.

---

```assembly
// file: sub1.asm

; Subprogram example program
%include "asm_io.inc"

segment .data
prompt1 db "Enter a number: ", 0 ; don't forget null terminator
prompt2 db "Enter another number: ", 0
outmsg1 db "You entered ", 0
outmsg2 db " and ", 0
outmsg3 db " the sum of these is ", 0

segment .bss
input1 resd 1
input2 resd 1

segment .text
\_asm\_main:
\_enter \_0, \_0 ; setup routine
\_pusha
\n\mov \_eax, \_prompt1 ; print out prompt
\_call \_print\_string
\n\mov \_ebx, \_input1 ; store address of \_input1 into \_ebx
```

---
4.2. SIMPLE SUBPROGRAM EXAMPLE

```assembly
26    mov   ecx, ret1   ; store return address into ecx
27    jmp   short get_int   ; read integer
28  ret1:
29    mov   eax, prompt2   ; print out prompt
30    call  print_string
31
32    mov   ebx, input2
33    mov   ecx, $ + 7   ; ecx = this address + 7
34    jmp   short get_int
35
36    mov   eax, [input1]   ; eax = dword at input1
37    add   eax, [input2]   ; eax += dword at input2
38    mov   ebx, eax   ; ebx = eax
39
40    mov   eax, outmsg1
41    call  print_string   ; print out first message
42    mov   eax, [input1]
43    call  print_int   ; print out input1
44    mov   eax, outmsg2
45    call  print_string   ; print out second message
46    mov   eax, [input2]
47    call  print_int   ; print out input2
48    mov   eax, outmsg3
49    call  print_string   ; print out third message
50    mov   eax, ebx
51    call  print_int   ; print out sum (ebx)
52    call  print_nl   ; print new-line
53
54    popa
55    mov   eax, 0   ; return back to C
56    leave
57    ret
58
59    ; subprogram get_int
60    ; Parameters:
61    ; ebx - address of dword to store integer into
62    ; ecx - address of instruction to return to
63    ; Notes:
64    ; value of eax is destroyed
65    get_int:
66    call  read_int
67    mov   [ebx], eax   ; store input into memory
68    jmp   ecx   ; jump back to caller
```

sub1.asm
The get_int subprogram uses a simple, register-based calling convention. It expects the EBX register to hold the address of the DWORD to store the number input into and the ECX register to hold the code address of the instruction to jump back to. In lines 25 to 28, the ret1 label is used to compute this return address. In lines 32 to 34, the $ operator is used to compute the return address. The $ operator returns the current address for the line it appears on. The expression $ + 7 computes the address of the MOV instruction on line 36.

Both of these return address computations are awkward. The first method requires a label to be defined for each subprogram call. The second method does not require a label, but does require careful thought. If a near jump was used instead of a short jump, the number to add to $ would not be 7! Fortunately, there is a much simpler way to invoke subprograms. This method uses the stack.

4.3 The Stack

Many CPU’s have built-in support for a stack. A stack is a Last-In First-Out (LIFO) list. The stack is an area of memory that is organized in this fashion. The PUSH instruction adds data to the stack and the POP instruction removes data. The data removed is always the last data added (that is why it is called a last-in first-out list).

The SS segment register specifies the segment that contains the stack (usually this is the same segment data is stored into). The ESP register contains the address of the data that would be removed from the stack. This data is said to be at the top of the stack. Data can only be added in double word units. That is, one can not push a single byte on the stack.

The PUSH instruction inserts a double word\(^1\) on the stack by subtracting 4 from ESP and then stores the double word at [ESP]. The POP instruction reads the double word at [ESP] and then adds 4 to ESP. The code below demonstrates how these instructions work and assumes that ESP is initially 1000H.

```
1 push    dword 1  ; 1 stored at 0FFCh, ESP = 0FFCh
2 push    dword 2  ; 2 stored at 0FF8h, ESP = 0FF8h
3 push    dword 3  ; 3 stored at 0FF4h, ESP = 0FF4h
4 pop     eax      ; EAX = 3, ESP = 0FF8h
5 pop     ebx      ; EBX = 2, ESP = 0FFCh
6 pop     ecx      ; ECX = 1, ESP = 1000h
```

\(^1\)Actually words can be pushed too, but in 32-bit protected mode, it is better to work with only double words on the stack.
4.4. THE CALL AND RET INSTRUCTIONS

The stack can be used as a convenient place to store data temporarily. It is also used for making subprogram calls, passing parameters and local variables.

The 80x86 also provides a **PUSHA** instruction that pushes the values of EAX, EBX, ECX, EDX, ESI, EDI and EBP registers (not in this order). The **POPA** instruction can be used to pop them all back off.

### 4.4 The CALL and RET Instructions

The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy. The **CALL** instruction makes an unconditional jump to a subprogram and *pushes* the address of the next instruction on the stack. The **RET** instruction *pops off* an address and jumps to that address. When using these instructions, it is very important that one manage the stack correctly so that the right number is popped off by the **RET** instruction!

The previous program can be rewritten to use these new instructions by changing lines 25 to 34 to be:

```
mov ebx, input1
call get_int
mov ebx, input2
call get_int
```

and change the subprogram `get_int` to:

```
get_int:
call read_int
mov [ebx], eax
ret
```

There are several advantages to **CALL** and **RET**:

- It is simpler!
- It allows subprograms calls to be nested easily. Notice that `get_int` calls `read_int`. This call pushes another address on the stack. At the end of `read_int`'s code is a **RET** that pops off the return address and jumps back to `get_int`'s code. Then when `get_int`'s **RET** is executed, it pops off the return address that jumps back to `asm_main`. This works correctly because of the LIFO property of the stack.
CHAPTER 4. SUBPROGRAMS

Remember it is very important to pop off all data that is pushed on the stack. For example, consider the following:

```
get_int:
    call read_int
    mov [ebx], eax
    push eax
    ret ; pops off EAX value, not return address!!
```

This code would not return correctly!

4.5 Calling Conventions

When a subprogram is invoked, the calling code and the subprogram (the callee) must agree on how to pass data between them. High-level languages have standard ways to pass data known as calling conventions. For high-level code to interface with assembly language, the assembly language code must use the same conventions as the high-level language. The calling conventions can differ from compiler to compiler or may vary depending on how the code is compiled (e.g. if optimizations are on or not). One universal convention is that the code will be invoked with a CALL instruction and return via a RET.

All PC C compilers support one calling convention that will be described in the rest of this chapter in stages. These conventions allow one to create subprograms that are reentrant. A reentrant subprogram may be called at any point of a program safely (even inside the subprogram itself).

4.5.1 Passing parameters on the stack

Parameters to a subprogram may be passed on the stack. They are pushed onto the stack before the CALL instruction. Just as in C, if the parameter is to be changed by the subprogram, the address of the data must be passed, not the value. If the parameter’s size is less than a double word, it must be converted to a double word before being pushed.

The parameters on the stack are not popped off by the subprogram, instead they are accessed from the stack itself. Why?

- Since they have to be pushed on the stack before the CALL instruction, the return address would have to be popped off first (and then pushed back on again).
- Often the parameters will have to be used in several places in the subprogram. Usually, they can not be kept in a register for the entire subprogram and would have to be stored in memory. Leaving them
4.5. CALLING CONVENTIONS

Consider a subprogram that is passed a single parameter on the stack. When the subprogram is invoked, the stack looks like Figure 4.1. The parameter can be accessed using indirect addressing (\([ESP+4]\)\(^2\)).

If the stack is also used inside the subprogram to store data, the number needed to be added to ESP will change. For example, Figure 4.2 shows what the stack looks like if a DWORD is pushed the stack. Now the parameter is at \(ESP + 8\) not \(ESP + 4\). Thus, it can be very error prone to use ESP when referencing parameters. To solve this problem, the 80386 supplies another register to use: EBP. This register’s only purpose is to reference data on the stack. The C calling convention mandates that a subprogram first save the value of EBP on the stack and then set EBP to be equal to ESP. This allows ESP to change as data is pushed or popped off the stack without modifying EBP. At the end of the subprogram, the original value of EBP must be restored (this is why it is saved at the start of the subprogram.) Figure 4.3 shows the general form of a subprogram that follows these conventions.

Lines 2 and 3 in Figure 4.3 make up the general prologue of a subprogram. Lines 5 and 6 make up the epilogue. Figure 4.4 shows what the stack looks like immediately after the prologue. Now the parameter can be access with \([EBP + 8]\) at any place in the subprogram without worrying about what else has been pushed onto the stack by the subprogram.

After the subprogram is over, the parameters that were pushed on the stack must be removed. The C calling convention specifies that the caller code must do this. Other conventions are different. For example, the Pascal calling convention specifies that the subprogram must remove the parame-

\(^2\)It is legal to add a constant to a register when using indirect addressing. More complicated expressions are possible too. This topic is covered in the next chapter.

When using indirect addressing, the 80x86 processor accesses different segments depending on what registers are used in the indirect addressing expression. ESP (and EBP) use the stack segment while EAX, EBX, ECX and EDX use the data segment. However, this is usually unimportant for most protected mode programs, because for them the data and stack segments are the same.
subprogram_label:
  push ebp          ; save original EBP value on stack
  mov ebp, esp      ; new EBP = ESP
  ; subprogram code
  pop ebp           ; restore original EBP value
  ret

Figure 4.3: General subprogram form

<table>
<thead>
<tr>
<th>ESP + 8</th>
<th>EBP + 8</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP + 4</td>
<td>EBP + 4</td>
<td>Return address</td>
</tr>
<tr>
<td>ESP</td>
<td>EBP</td>
<td>saved EBP</td>
</tr>
</tbody>
</table>

Figure 4.4:

ters. (There is another form of the RET instruction that makes this easy to do.) Some C compilers support this convention too. The pascal keyword is used in the prototype and definition of the function to tell the compiler to use this convention. In fact, the stdcall convention that the MS Windows API C functions use also works this way. What is the advantage of this way? It is a little more efficient than the C convention. Why do all C functions not use this convention, then? In general, C allows a function to have varying number of arguments (e.g., the printf and scanf functions). For these types of functions, the operation to remove the parameters from the stack will vary from one call of the function to the next. The C convention allows the instructions to perform this operation to be easily varied from one call to the next. The Pascal and stdcall convention makes this operation very difficult. Thus, the Pascal convention (like the Pascal language) does not allow this type of function. MS Windows can use this convention since none of its API functions take varying numbers of arguments.

Figure 4.5 shows how a subprogram using the C calling convention would be called. Line 3 removes the parameter from the stack by directly manipulating the stack pointer. A POP instruction could be used to do this also, but would require the useless result to be stored in a register. Actually, for this particular case, many compilers would use a POP ECX instruction to remove the parameter. The compiler would use a POP instead of an ADD because the ADD requires more bytes for the instruction. However, the POP also changes ECX’s value! Next is another example program with two subprograms that use the C calling conventions discussed above. Line 54 (and other lines) shows that multiple data and text segments may be declared in a single
4.5. CALLING CONVENTIONS

```
push dword 1    ; pass 1 as parameter
call fun
add esp, 4     ; remove parameter from stack
```

Figure 4.5: Sample subprogram call

source file. They will be combined into single data and text segments in the linking process. Splitting up the data and code into separate segments allow the data that a subprogram uses to be defined close by the code of the subprogram.

```
%include "asm_io.inc"

segment .data
sum dd 0

segment .bss
input resd 1

; pseudo-code algorithm
; i = 1;
; sum = 0;
; while( get_int(i, &input), input != 0 ) {
; sum += input;
; i++;
; }
; print_sum(num);

segment .text
    global _asm_main
_asm_main:
    enter 0,0    ; setup routine
    pusha

mov edx, 1    ; edx is 'i' in pseudo-code
while_loop:
    push edx    ; save i on stack
    push dword input    ; push address on input on stack
    call get_int
    add esp, 8     ; remove i and &input from stack
```
CHAPTER 4. SUBPROGRAMS

74

30

mov eax, [input]
31
cmp eax, 0
32
je end_while
33
34
add [sum], eax ; sum += input
35
36
inc edx
37
38
jmp short while_loop
39
40
end_while:
41
push dword [sum] ; push value of sum onto stack
42
call print_sum
43
pop ecx ; remove [sum] from stack
44
45
popa
46
leave
47
ret
48
49 ; subprogram get_int
50 ; Parameters (in order pushed on stack)
51 ; number of input (at [ebp + 12])
52 ; address of word to store input into (at [ebp + 8])
53 ; Notes:
54 ; values of eax and ebx are destroyed
55 segment .data
56 prompt db ") Enter an integer number (0 to quit): ", 0
57
58 segment .text
59 get_int:
60
push ebp
61
mov ebp, esp
62
63
mov eax, [ebp + 12]
64
call print_int
65
66
mov eax, prompt
67
call print_string
68
69
call read_int
70
mov ebx, [ebp + 8]
71
mov [ebx], eax ; store input into memory
4.5. CALLING CONVENTIONS

pop    ebp
ret                      ; jump back to caller

; subprogram print_sum
; prints out the sum
; Parameter:
;  sum to print out (at [ebp+8])
; Note: destroys value of eax

segment .data
result  db      "The sum is ", 0

segment .text
print_sum:
push    ebp
mov     ebp, esp
mov     eax, result
call    print_string
mov     eax, [ebp+8]
call    print_int
call    print_nl
pop     ebp
ret

4.5.2 Local variables on the stack

The stack can be used as a convenient location for local variables. This is exactly where C stores normal (or automatic in C lingo) variables. Using the stack for variables is important if one wishes subprograms to be reentrant. A reentrant subprogram will work if it is invoked at any place, including the subprogram itself. In other words, reentrant subprograms can be invoked recursively. Using the stack for variables also saves memory. Data not stored on the stack is using memory from the beginning of the program until the end of the program (C calls these types of variables global or static). Data stored on the stack only use memory when the subprogram they are defined for is active.

Local variables are stored right after the saved EBP value in the stack. They are allocated by subtracting the number of bytes required from ESP
in the prologue of the subprogram. Figure 4.6 shows the new subprogram skeleton. The EBP register is used to access local variables. Consider the C function in Figure 4.7. Figure 4.8 shows how the equivalent subprogram could be written in assembly.

Figure 4.9 shows what the stack looks like after the prologue of the program in Figure 4.8. This section of the stack that contains the parameters, return information and local variable storage is called a stack frame. Every invocation of a C function creates a new stack frame on the stack.

The prologue and epilogue of a subprogram can be simplified by using two special instructions that are designed specifically for this purpose. The ENTER instruction performs the prologue code and the LEAVE performs the epilogue. The ENTER instruction takes two immediate operands. For the C calling convention, the second operand is always 0. The first operand is the number bytes needed by local variables. The LEAVE instruction has no operands. Figure 4.10 shows how these instructions are used. Note that the program skeleton (Figure 4.7) also uses ENTER and LEAVE.

Despite the fact that ENTER and LEAVE simplify the prologue and epilogue they are not used very often. Why? Because they are slower than the equivalent simpler instructions! This is an example of when one can not assume that a one instruction sequence is faster than a multiple instruction one.
4.6 Multi-Module Programs

A multi-module program is one composed of more than one object file. All the programs presented here have been multi-module programs. They consisted of the C driver object file and the assembly object file (plus the C library object files). Recall that the linker combines the object files into a single executable program. The linker must match up references made to each label in one module (i.e. object file) to its definition in another module. In order for module A to use a label defined in module B, the extern directive must be used. After the extern directive comes a comma delimited list of labels. The directive tells the assembler to treat these labels as external to the module. That is, these are labels that can be used in this module, but are defined in another. The asm_io.inc file defines the read_int, etc. routines as external.

In assembly, labels can not be accessed externally by default. If a label
Figure 4.9:

```
subprogram_label:
    enter LOCAL_BYTES, 0 ; = # bytes needed by locals
    ; subprogram code
    leave
    ret
```

Figure 4.10: General subprogram form with local variables using ENTER and LEAVE

can be accessed from other modules than the one it is defined in, it must be declared global in its module. The global directive does this. Line 13 of the skeleton program listing in Figure 1.7 shows the _asm_main label being defined as global. Without this declaration, there would be a linker error. Why? Because the C code would not be able to refer to the internal _asm_main label.

Next is the code for the previous example, rewritten to use two modules. The two subprograms (get_int and print_sum) are in a separate source file than the _asm_main routine.
4.6. MULTI-MODULE PROGRAMS

```
15 mov edx, 1 ; edx is 'i' in pseudo-code
16 while_loop:
17 push edx ; save i on stack
18 push dword input ; push address on input on stack
19 call get_int
20 add esp, 8 ; remove i and &input from stack
21 mov eax, [input]
22 cmp eax, 0
23 je end_while
24 add [sum], eax ; sum += input
25 inc edx
26 jmp short while_loop
27 end_while:
28 push dword [sum] ; push value of sum onto stack
29 call print_sum
30 pop ecx ; remove [sum] from stack
31 popa
32 leave
33 ret
```

```
1 %include "asm_io.inc"
2 segment .data
3 prompt db ") Enter an integer number (0 to quit): ", 0
4 segment .text
5 global get_int, print_sum
6 get_int:
7 enter 0,0
8 mov eax, [ebp + 12]
9 call print_int
10 mov eax, prompt
11 call print_string
```
CHAPTER 4. SUBPROGRAMS

```assembly
chapter 4
17 call read_int
18 mov ebx, [ebp + 8]
19 mov [ebx], eax ; store input into memory
20
21 leave
22 ret ; jump back to caller

23 segment .data
24 result db "The sum is ", 0
25
26 segment .text
27 print_sum:
28 enter 0,0
29
30 mov eax, result
31 call print_string
32
33 mov eax, [ebp+8]
34 call print_int
35 call print_nl
36
37 leave
38 ret
```

The previous example only has global code labels; however, global data labels work exactly the same way.

4.7 Interfacing Assembly with C

Today, very few programs are written completely in assembly. Compilers are very good at converting high level code into efficient machine code. Since it is much easier to write code in a high level language, it is more popular. In addition, high level code is much more portable than assembly!

When assembly is used, it is often only used for small parts of the code. This can be done in two ways: calling assembly subroutines from C or inline assembly. Inline assembly allows the programmer to place assembly statements directly into C code. This can be very convenient; however, there are disadvantages to inline assembly. The assembly code must be written in the format the compiler uses. No compiler at the moment supports NASM’s format. Different compilers require different formats. Borland and Microsoft require MASM format. DJGPP and Linux’s gcc require GAS format. The

---

\(^3\)GAS is the assembler that all GNU compiler’s use. It uses the AT&T syntax which
4.7. INTERFACING ASSEMBLY WITH C

<table>
<thead>
<tr>
<th>segment .data</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x dd 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>format db &quot;x = %d\n&quot;, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>segment .text</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>push dword [x] ; push x’s value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>push dword format ; push address of format string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call _printf ; note underscore!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add esp, 8 ; remove parameters from stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.11: Call to printf

The technique of calling an assembly subroutine is much more standardized on the PC.

Assembly routines are usually used with C for the following reasons:

- Direct access is needed to hardware features of the computer that are difficult or impossible to access from C.

- The routine must be as fast as possible and the programmer can hand optimize the code better than the compiler can.

The last reason is not as valid as it once was. Compiler technology has improved over the years and compilers can often generate very efficient code (especially if compiler optimizations are turned on). The disadvantages of assembly routines are: reduced portability and readability.

Most of the C calling conventions have already been specified. However, there are a few additional features that need to be described.

4.7.1 Saving registers

First, C assumes that a subroutine maintains the values of the following registers: EBX, ESI, EDI, EBP, CS, DS, SS, ES. This does not mean that the subroutine can not change them internally. Instead, it means that if it does change their values, it must restore their original values before the subroutine returns. The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables. Usually the stack is used to save the original values of these registers.

---

The register keyword can be used in a C variable declaration to suggest to the compiler that it use a register for this variable instead of a memory location. These are known as register variables. Modern compilers do this automatically without requiring any suggestions.
EBP + 12 | value of x
EBP + 8 | address of format string
EBP + 4 | Return address
EBP | saved EBP

Figure 4.12: Stack inside printf

4.7.2 Labels of functions

Most C compilers prepend a single underscore (_) character at the beginning of the names of functions and global/static variables. For example, a function named f will be assigned the label _f. Thus, if this is to be an assembly routine, it must be labelled _f, not f. The Linux gcc compiler does not prepend any character. Under Linux ELF executables, one simply would use the label f for the C function f. However, DJGPP’s gcc does prepend an underscore. Note that in the assembly skeleton program (Figure 1.7), the label for the main routine is _asm_main.

4.7.3 Passing parameters

Under the C calling convention, the arguments of a function are pushed on the stack in the reverse order that they appear in the function call.

Consider the following C statement: printf("x = %d\n", x); Figure 4.11 shows how this would be compiled (shown in the equivalent NASM format). Figure 4.12 shows what the stack looks like after the prologue inside the printf function. The printf function is one of the C library functions that can take any number of arguments. The rules of the C calling conventions were specifically written to allow these types of functions. Since the address of the format string is pushed last, its location on the stack will always be at EBP + 8 no matter how many parameters are passed to the function. The printf code can then look at the format string to determine how many parameters should have been passed and look for them on the stack.

Of course, if a mistake is made, printf(”x = %d\n”), the printf code will still print out the double word value at [EBP + 12]. However, this will not be x’s value!

4.7.4 Calculating addresses of local variables

Finding the address of a label defined in the data or bss segments is simple. Basically, the linker does this. However, calculating the address of a local variable (or parameter) on the stack is not as straightforward. However, this is a very common need when calling subroutines. Consider the case of passing the address of a variable (let’s call it x) to a function.

It is not necessary to use assembly to process an arbitrary number of arguments in C. The stdarg.h header file defines macros that can be used to process them portably. See any good C book for details.
(let’s call it `foo`). If `x` is located at `EBP − 8` on the stack, one cannot just use:

```
    mov    eax, ebp - 8
```

Why? The value that `MOV` stores into `EAX` must be computed by the assembler (that is, it must in the end be a constant). However, there is an instruction that does the desired calculation. It is called `LEA` (for *Load Effective Address*). The following would calculate the address of `x` and store it into `EAX`:

```
    lea    eax, [ebp - 8]
```

Now `EAX` holds the address of `x` and could be pushed on the stack when calling function `foo`. Do not be confused, it looks like this instruction is reading the data at `[EBP-8]`; however, this is not true. The `LEA` instruction *never* reads memory! It only computes the address that would be read by another instruction and stores this address in its first register operand. Since it does not actually read any memory, no memory size designation (*e.g.* `dword`) is needed or allowed.

### 4.7.5 Returning values

Non-void C functions return back a value. The C calling conventions specify how this is done. Return values are passed via registers. All integral types (`char`, `int`, `enum`, *etc.*) are returned in the `EAX` register. If they are smaller than 32-bits, they are extended to 32-bits when stored in `EAX`. (How they are extended depends on if they are signed or unsigned types.) 64-bit values are returned in the `EDX:EAX` register pair. Pointer values are also stored in `EAX`. Floating point values are stored in the `ST0` register of the math coprocessor. (This register is discussed in the floating point chapter.)

### 4.7.6 Other calling conventions

The rules above describe the standard C calling convention that is supported by all 80x86 C compilers. Often compilers support other calling conventions as well. When interfacing with assembly language it is very important to know what calling convention the compiler is using when it calls your function. Usually, the default is to use the standard calling convention; however, this is not always the case. Compilers that use multiple conventions often have command line switches that can be used to change

---

4The Watcom C compiler is an example of one that does *not* use the standard convention by default. See the example source code file for Watcom for details.
the default convention. They also provide extensions to the C syntax to explicitly assign calling conventions to individual functions. However, these extensions are not standardized and may vary from one compiler to another.

The GCC compiler allows different calling conventions. The convention of a function can be explicitly declared by using the __attribute__ extension. For example, to declare a void function that uses the standard calling convention named f that takes a single int parameter, use the following syntax for its prototype:

```c
void f( int ) __attribute__((cdecl));
```

GCC also supports the standard call calling convention. The function above could be declared to use this convention by replacing the cdecl with stdcall. The difference in stdcall and cdecl is that stdcall requires the subroutine to remove the parameters from the stack (as the Pascal calling convention does). Thus, the stdcall convention can only be used with functions that take a fixed number of arguments (i.e. ones not like printf and scanf).

GCC also supports an additional attribute called regparm that tells the compiler to use registers to pass up to 3 integer arguments to a function instead of using the stack. This is a common type of optimization that many compilers support.

Borland and Microsoft use a common syntax to declare calling conventions. They add the cdecl and stdcall keywords to C. These keywords act as function modifiers and appear immediately before the function name in a prototype. For example, the function f above would be defined as follows for Borland and Microsoft:

```c
void __cdecl f( int );
```

There are advantages and disadvantages to each of the calling conventions. The main advantages of the cdecl convention is that it is simple and very flexible. It can be used for any type of C function and C compiler. Using other conventions can limit the portability of the subroutine. Its main disadvantage is that it can be slower than some of the others and use more memory (since every invocation of the function requires code to remove the parameters on the stack).

The advantages of the stdcall convention is that it uses less memory than cdecl. No stack cleanup is required after the CALL instruction. Its main disadvantage is that it can not be used with functions that have variable numbers of arguments.

The advantage of using a convention that uses registers to pass integer parameters is speed. The main disadvantage is that the convention is more complex. Some parameters may be in registers and others on the stack.
4.7. INTERFACING ASSEMBLY WITH C

4.7.7 Examples

Next is an example that shows how an assembly routine can be interfaced to a C program. (Note that this program does not use the assembly skeleton program (Figure 1.7) or the driver.c module.)

```c
#include <stdio.h>

/* prototype for assembly routine */
void calc_sum( int, int * ) __attribute__((cdecl));

int main( void )
{
    int n, sum;

    printf("Sum integers up to: ");
    scanf("%d", &n);
    calc_sum(n, &sum);
    printf("Sum is %d\n", sum);
    return 0;
}
```

```asm
; subroutine _calc_sum
; finds the sum of the integers 1 through n
; Parameters:
; n - what to sum up to (at [ebp + 8])
; sump - pointer to int to store sum into (at [ebp + 12])
; pseudo C code:
; void calc_sum( int n, int * sump )
{
    int i, sum = 0;
    for( i=1; i <= n; i++ )
        sum += i;
    *sum = sum;
}

segment .text
    global _calc_sum
```

CHAPTER 4. SUBPROGRAMS

Sum integers up to: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
+16 BFFFFB80 080499EC
+12 BFFFFB7C BFFFFB80
+8 BFFFFB78 0000000A
+4 BFFFFB74 08048501
+0 BFFFFB70 BFFFFB88
-4 BFFFFB6C 00000000
-8 BFFFFB68 4010648C
Sum is 55

Figure 4.13: Sample run of sub5 program

; local variable:
;  sum at [ebp-4]
_calc_sum:
    enter 4,0 ; make room for sum on stack
    push ebx ; IMPORTANT!
    mov dword [ebp-4],0 ; sum = 0
    dump_stack 1, 2, 4 ; print out stack from ebp-8 to ebp+16
    mov ecx, 1 ; ecx is i in pseudocode
    for_loop:
        cmp ecx, [ebp+8] ; cmp i and n
        jnle end_for ; if not i <= n, quit
        add [ebp-4], ecx ; sum += i
        inc ecx
        jmp short for_loop
    end_for:
    mov ebx, [ebp+12] ; ebx = sump
    mov eax, [ebp-4] ; eax = sum
    mov [ebx], eax
    pop ebx ; restore ebx
    leave
    ret

sub5.asm
Why is line 22 of sub5.asm so important? Because the C calling convention requires the value of EBX to be unmodified by the function call. If this is not done, it is very likely that the program will not work correctly.

Line 25 demonstrates how the dump_stack macro works. Recall that the first parameter is just a numeric label, and the second and third parameters determine how many double words to display below and above EBP respectively. Figure 4.13 shows an example run of the program. For this dump, one can see that the address of the dword to store the sum is BFFFFB80 (at EBP + 12); the number to sum up to is 0000000A (at EBP + 8); the return address for the routine is 08048501 (at EBP + 4); the saved EBP value is BFFFFFB88 (at EBP); the value of the local variable is 0 at (EBP - 4); and finally the saved EBX value is 4010648C (at EBP - 8).

The calc_sum function could be rewritten to return the sum as its return value instead of using a pointer parameter. Since the sum is an integral value, the sum should be left in the EAX register. Line 11 of the main5.c file would be changed to:

\[
\text{sum = calc\_sum(n);}\
\]

Also, the prototype of calc\_sum would need be altered. Below is the modified assembly code:

```
__________ sub6.asm __________________________
1 ; subroutine _calc_sum
2 ; finds the sum of the integers 1 through n
3 ; Parameters:
4 ; n - what to sum up to (at [ebp + 8])
5 ; Return value:
6 ; value of sum
7 ; pseudo C code:
8 ; int calc_sum( int n )
9 ; {
10 ; int i, sum = 0;
11 ; for( i=1; i <= n; i++ )
12 ; sum += i;
13 ; return sum;
14 ; }
15 segment .text
16 ; global _calc_sum
17 ;
18 ; local variable:
19 ; sum at [ebp-4]
20 _calc_sum:
21 enter 4,0 ; make room for sum on stack
```
CHAPTER 4. SUBPROGRAMS

segment .data
format db "%d", 0

segment .text
...
    lea eax, [ebp-16]
    push eax
    push dword format
    call _scanf
    add esp, 8
...

Figure 4.14: Calling `scanf` from assembly

    mov dword [ebp-4], 0 ; sum = 0
    mov ecx, 1 ; ecx is i in pseudocode
for_loop:
    cmp ecx, [ebp+8] ; cmp i and n
    jnle end_for ; if not i <= n, quit
    add [ebp-4], ecx ; sum += i
    inc ecx
    jmp short for_loop
end_for:
    mov eax, [ebp-4] ; eax = sum

    leave
    ret

sub6.asm

4.7.8 Calling C functions from assembly

One great advantage of interfacing C and assembly is that allows assembly code to access the large C library and user-written functions. For example, what if one wanted to call the `scanf` function to read in an integer from the keyboard? Figure 4.14 shows code to do this. One very important point to remember is that `scanf` follows the C calling standard to the letter. This means that it preserves the values of the EBX, ESI and EDI registers; however, the EAX, ECX and EDX registers may be modified! In fact, EAX will definitely be changed, as it will contain the return value of the `scanf`
call. For other examples of using interfacing with C, look at the code in \texttt{asm\_io.asm} which was used to create \texttt{asm\_io.obj}.

### 4.8 Reentrant and Recursive Subprograms

A reentrant subprogram must satisfy the following properties:

- It must not modify any code instructions. In a high level language this would be difficult, but in assembly it is not hard for a program to try to modify its own code. For example:

  \begin{verbatim}
  mov word [cs:$+7], 5 ; copy 5 into the word 7 bytes ahead
  add ax, 2 ; previous statement changes 2 to 5!
  \end{verbatim}

  This code would work in real mode, but in protected mode operating systems the code segment is marked as read only. When the first line above executes, the program will be aborted on these systems. This type of programming is bad for many reasons. It is confusing, hard to maintain and does not allow code sharing (see below).

- It must not modify global data (such as data in the \texttt{data} and the \texttt{bss} segments). All variables are stored on the stack.

There are several advantages to writing reentrant code.

- A reentrant subprogram can be called recursively.

- A reentrant program can be shared by multiple processes. On many multi-tasking operating systems, if there are multiple instances of a program running, only \textit{one} copy of the code is in memory. Shared libraries and DLL's (Dynamic Link Libraries) use this idea as well.

- Reentrant subprograms work much better in multi-threaded programs. Windows 9x/NT and most UNIX-like operating systems (Solaris, Linux, etc.) support multi-threaded programs.

#### 4.8.1 Recursive subprograms

These types of subprograms call themselves. The recursion can be either direct or indirect. Direct recursion occurs when a subprogram, say \texttt{foo}, calls itself inside \texttt{foo}'s body. Indirect recursion occurs when a subprogram is not called by itself directly, but by another subprogram it calls. For example, subprogram \texttt{foo} could call \texttt{bar} and \texttt{bar} could call \texttt{foo}.

---

\footnote{A multi-threaded program has multiple threads of execution. That is, the program itself is multi-tasked.}
CHAPTER 4. SUBPROGRAMS

Figure 4.15: Recursive factorial function

Recursive subprograms must have a termination condition. When this condition is true, no more recursive calls are made. If a recursive routine does not have a termination condition or the condition never becomes true, the recursion will never end (much like an infinite loop).

Figure 4.15 shows a function that calculates factorials recursively. It could be called from C with:

```c
x = fact(3); /* find 3! */
```

Figure 4.16 shows what the stack looks like at its deepest point for the above function call.

Figures 4.17 and 4.18 show another more complicated recursive example in C and assembly, respectively. What is the output is for $f(3)$? Note that the ENTER instruction creates a new $i$ on the stack for each recursive call. Thus, each recursive instance of $f$ has its own independent variable $i$. Defining $i$ as a double word in the data segment would not work the same.
### 4.8. REENTRANT AND RECURSIVE SUBPROGRAMS

Figure 4.16: Stack frames for factorial function

<table>
<thead>
<tr>
<th>n=3 frame</th>
<th>n(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Saved EBP</td>
</tr>
<tr>
<td>n=2 frame</td>
<td>n(2)</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Saved EBP</td>
</tr>
<tr>
<td>n=1 frame</td>
<td>n(1)</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
<tr>
<td></td>
<td>Saved EBP</td>
</tr>
</tbody>
</table>

```c
void f(int x)
{
    int i;
    for (i = 0; i < x; i++) {
        printf("%d\n", i);
        f(i);
    }
}
```

Figure 4.17: Another example (C version)

#### 4.8.2 Review of C variable storage types

C provides several types of variable storage.

**global** These variables are defined outside of any function and are stored at fixed memory locations (in the *data* or *bss* segments) and exist from the beginning of the program until the end. By default, they can be accessed from any function in the program; however, if they are declared as **static**, only the functions in the same module can access them (*i.e.* in assembly terms, the label is internal, not external).

**static** These are *local* variables of a function that are declared **static**. (Unfortunately, C uses the keyword **static** for two different purposes!) These variables are also stored at fixed memory locations (in *data* or *bss*), but can only be directly accessed in the functions they are defined in.
%define i ebp-4
%define x ebp+8 ; useful macros
segment .data
format db "%d", 10, 0 ; 10 = '\n'
segment .text
    global _f
    extern _printf
_f:
    enter 4,0 ; allocate room on stack for i
    mov dword [i], 0 ; i = 0
lp:
    mov eax, [i] ; is i < x?
cmp eax, [x]
jnl quit
    push eax ; call printf
    push format
call _printf
    add esp, 8
    push dword [i] ; call f
call _f
    pop eax
    inc dword [i] ; i++
jmp short lp
quit:
    leave
    ret

Figure 4.18: Another example (assembly version)
automatic This is the default type for a C variable defined inside a function. This variables are allocated on the stack when the function they are defined in is invoked and are deallocated when the function returns. Thus, they do not have fixed memory locations.

register This keyword asks the compiler to use a register for the data in this variable. This is just a request. The compiler does not have to honor it. If the address of the variable is used anywhere in the program it will not be honored (since registers do not have addresses). Also, only simple integral types can be register values. Structured types can not be; they would not fit in a register! C compilers will often automatically make normal automatic variables into register variables without any hint from the programmer.

volatile This keyword tells the compiler that the value of the variable may change any moment. This means that the compiler can not make any assumptions about when the variable is modified. Often a compiler might store the value of a variable in a register temporarily and use the register in place of the variable in a section of code. It can not do these types of optimizations with volatile variables. A common example of a volatile variable would be one could be altered by two threads of a multi-threaded program. Consider the following code:

```
1  x = 10;
2  y = 20;
3  z = x;
```

If x could be altered by another thread, it is possible that the other thread changes x between lines 1 and 3 so that z would not be 10. However, if the x was not declared volatile, the compiler might assume that x is unchanged and set z to 10.

Another use of volatile is to keep the compiler from using a register for a variable.
Chapter 5

Arrays

5.1 Introduction

An array is a contiguous block of list of data in memory. Each element of the list must be the same type and use exactly the same number of bytes of memory for storage. Because of these properties, arrays allow efficient access of the data by its position (or index) in the array. The address of any element can be computed by knowing three facts:

- The address of the first element of the array.
- The number of bytes in each element
- The index of the element

It is convenient to consider the index of the first element of the array to be zero (just as in C). It is possible to use other values for the first index, but it complicates the computations.

5.1.1 Defining arrays

Defining arrays in the data and bss segments

To define an initialized array in the data segment, use the normal db, dw, etc. directives. NASM also provides a useful directive named TIMES that can be used to repeat a statement many times without having to duplicate the statements by hand. Figure 5.1 shows several examples of these.

To define an uninitialized array in the bss segment, use the resb, resw, etc. directives. Remember that these directives have an operand that specifies how many units of memory to reserve. Figure 5.1 also shows examples of these types of definitions.
Defining arrays as local variables on the stack

There is no direct way to define a local array variable on the stack. As before, one computes the total bytes required by all local variables, including arrays, and subtracts this from ESP (either directly or using the ENTER instruction). For example, if a function needed a character variable, two double word integers and a 50 element word array, one would need \(1 + 2 \times 4 + 50 \times 2 = 109\) bytes. However, the number subtracted from ESP should be a multiple of four (112 in this case) to keep ESP on a double word boundary. One could arrange the variables inside this 109 bytes in several ways. Figure 5.2 shows two possible ways. The unused part of the first ordering is there to keep the double words on double word boundaries to speed up memory accesses.

5.1.2 Accessing elements of arrays

There is no \([\ ]\) operator in assembly language as in C. To access an element of an array, its address must be computed. Consider the following two array definitions:

```
array1 db 5, 4, 3, 2, 1 ; array of bytes
array2 dw 5, 4, 3, 2, 1 ; array of words
```

Here are some examples using this arrays:
In line 5, element 1 of the word array is referenced, not element 2. Why? Words are two byte units, so to move to the next element of a word array, one must move two bytes ahead, not one. Line 7 will read one byte from the first element and one from the second. In C, the compiler looks at the type of a pointer in determining how many bytes to move in an expression that uses pointer arithmetic so that the programmer does not have to. However, in assembly, it is up to the programmer to take the size of array elements in account when moving from element to element.

Figure 5.3 shows a code snippet that adds all the elements of array1 in the previous example code. In line 7, AX is added to DX. Why not AL? First, the two operands of the ADD instruction must be the same size. Secondly, it would be easy to add up bytes and get a sum that was too big to fit into a byte. By using DX, sums up to 65,535 are allowed. However, it is important to realize that AH is being added also. This is why AH is set to zero in line 3.

Figures 5.4 and 5.5 show two alternative ways to calculate the sum. The lines in italics replace lines 6 and 7 of Figure 5.3.

Setting AH to zero is implicitly assuming that AL is an unsigned number. If it is signed, the appropriate action would be to insert a CBW instruction between lines 6 and 7.
5.1.3 More advanced indirect addressing

Not surprisingly, indirect addressing is often used with arrays. The most general form of an indirect memory reference is:

\[
[ \text{base reg} + \text{factor} \times \text{index reg} + \text{constant}] 
\]

where:

- **base reg** is one of the registers EAX, EBX, ECX, EDX, EBP, ESP, ESI orEDI.

- **factor** is either 1, 2, 4 or 8. (If 1, factor is omitted.)

- **index reg** is one of the registers EAX, EBX, ECX, EDX, EBP, ESI, EDI. (Note that ESP is not in list.)
5.1. INTRODUCTION

constant is a 32-bit constant. The constant can be a label (or a label expression).

5.1.4 Example

Here is an example that uses an array and passes it to a function. It uses the array1c.c program (listed below) as a driver, not the driver.c program.

```asm
mov ebx, array1 ; ebx = address of array1
mov dx, 0 ; dx will hold sum
mov ecx, 5
lp:
    add dl, [ebx] ; dl += *ebx
    adc dh, 0 ; dh += carry flag + 0
    inc ebx ; bx++
    loop lp
```

Figure 5.5: Summing elements of an array (Version 3)
; initialize array to 100, 99, 98, 97, ...

    mov    ecx, ARRAY_SIZE
    mov    ebx, array

    init_loop:
        mov    [ebx], ecx
        add    ebx, 4
        loop   init_loop

    push    dword FirstMsg
    call    _puts
    pop     ecx

    push    dword 10
    push    dword array
    call    _print_array
    add esp, 8

; prompt user for element index

    Prompt_loop:
        push    dword Prompt
        call    _printf
        pop     ecx

        lea     eax, [ebp-4] ; eax = address of local dword
        push    eax
        push    dword InputFormat
        call    _scanf
        add esp, 8
        cmp eax, 1 ; eax = return value of scanf
        je      InputOK

    call    _dump_line ; dump rest of line and start over
    jmp     Prompt_loop ; if input invalid

    InputOK:
        mov    esi, [ebp-4]
        push    dword [array + 4*esi]
        push    esi
        push    dword SecondMsg
        call    _printf
        add esp, 12
5.1. INTRODUCTION

push    dword ThirdMsg       ; print out elements 20-29
call    _puts
pop     ecx

push    dword 10
push    dword array + 20*4    ; address of array[20]
call    _print_array
add     esp, 8

pop     esi
pop     ebx
mov     eax, 0                ; return back to C
leave
ret

; routine _print_array
; C-callable routine that prints out elements of a double word array as
; signed integers.
; C prototype:
; void print_array( const int * a, int n);
; Parameters:
;    a - pointer to array to print out (at ebp+8 on stack)
;    n - number of integers to print out (at ebp+12 on stack)

segment .data
OutputFormat  db "%-5d  %5d", NEW_LINE, 0

segment .text
_global    _print_array
_print_array:
    enter  0,0
    push   esi
    push   ebx

    xor     esi, esi            ; esi = 0
    mov     ecx, [ebp+12]        ; ecx = n
    mov     ebx, [ebp+8]         ; ebx = address of array

    print_loop:
    push    ecx                  ; printf might change ecx!
push dword [ebx + 4*esi] ; push array[esi]
push esi
push dword OutputFormat
call _printf
add esp, 12 ; remove parameters (leave ecx!)
ing esi
pop ecx
loop print_loop
pop ebx
pop esi
leave
ret

#include <stdio.h>

int asm_main( void );
void dump_line( void );

int main()
{
    int ret_status ;
    ret_status = asm_main();
    return ret_status ;
}

/*
 * function dump_line
 * dumps all chars left in current line from input buffer
 */
void dump_line()
{
    int ch;
    
    while( (ch = getchar()) != EOF & & ch != '\n')
    /* null body*/ ;
}
The LEA instruction revisited

The LEA instruction can be used for other purposes than just calculating addresses. A fairly common one is for fast computations. Consider the following:

\[ \text{lea ebx, [4\times\text{eax} + \text{eax}] } \]

This effectively stores the value of \( 5 \times \text{EAX} \) into EBX. Using LEA to do this is both easier and faster than using MUL. However, one must realize that the expression inside the square brackets must be a legal indirect address. Thus, for example, this instruction can not be used to multiple by 6 quickly.

5.1.5 Multidimensional Arrays

Multidimensional arrays are not really very different than the plain one dimensional arrays already discussed. In fact, they are represented in memory as just that, a plain one dimensional array.

Two Dimensional Arrays

Not surprisingly, the simplest multidimensional array is a two dimensional one. A two dimensional array is often displayed as a grid of elements. Each element is identified by a pair of indices. By convention, the first index is identified with the row of the element and the second index the column.

Consider an array with three rows and two columns defined as:

\[ \text{int a[3][2];} \]

The C compiler would reserve room for a 6 (= 2 \( \times \) 3) integer array and map the elements as follows:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[2][0]</td>
<td>a[2][1]</td>
</tr>
</tbody>
</table>

What the table attempts to show is that the element referenced as \( \text{a[0][0]} \) is stored at the beginning of the 6 element one dimensional array. Element \( \text{a[0][1]} \) is stored in the next position (index 1) and so on. Each row of the two dimensional array is stored contiguously in memory. The last element of a row is followed by the first element of the next row. This is known as the rowwise representation of the array and is how a C/C++ compiler would represent the array.

How does the compiler determine where \( \text{a[i][j]} \) appears in the rowwise representation? A simple formula will compute the index from \( i \) and \( j \). The formula in this case is \( 2i + j \). It’s not too hard to see how this formula is derived. Each row is two elements long; so, the first element of row \( i \) is at position \( 2i \). Then the position of column \( j \) is found by adding \( j \) to \( 2i \).
CHAPTER 5. ARRAYS

This analysis also shows how the formula is generalized to an array with $N$ columns: $N \times i + j$. Notice that the formula does not depend on the number of rows.

As an example, let us see how gcc compiles the following code (using the array $a$ defined above):

\[
x = a[i][j];
\]

Figure 5.6 shows the assembly this is translated into. Thus, the compiler essentially converts the code to:

\[
x = \ast (\&a[0][0] + 2 \ast i + j);
\]

and in fact, the programmer could write this way with the same result.

There is nothing magical about the choice of the rowwise representation of the array. A columnwise representation would work just as well:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>$a[0][0]$</td>
<td>$a[1][0]$</td>
<td>$a[2][0]$</td>
<td>$a[0][1]$</td>
<td>$a[1][1]$</td>
<td>$a[2][1]$</td>
</tr>
</tbody>
</table>

In the columnwise representation, each column is stored contiguously. Element $[i][j]$ is stored at position $i + 3j$. Other languages (FORTRAN, for example) use the columnwise representation. This is important when interfacing code with multiple languages.

Dimensions Above Two

For dimensions above two, the same basic idea is applied. Consider a three dimensional array:

```c
int b [4][3][2];
```

This array would be stored like it was four two dimensional arrays each of size $[3][2]$ consecutively in memory. The table below shows how it starts out:
The formula for computing the position of $b[i][j][k]$ is $6i + 2j + k$. The $6$ is determined by the size of the $[3][2]$ arrays. In general, for an array dimensioned as $a[L][M][N]$ the position of element $a[i][j][k]$ will be $M \times N \times i + N \times j + k$. Notice again that the first dimension ($L$) does not appear in the formula.

For higher dimensions, the same process is generalized. For an $n$ dimensional array of dimensions $D_1$ to $D_n$, the position of element denoted by the indices $i_1$ to $i_n$ is given by the formula:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

or for the über math geek, it can be written more succinctly as:

$$\sum_{j=1}^{n} \left( \prod_{k=j+1}^{n} D_k \right) i_j$$

The first dimension, $D_1$, does not appear in the formula.

For the columnwise representation, the general formula would be:

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

or in über math geek notation:

$$\sum_{j=1}^{n} \left( \prod_{k=1}^{j-1} D_k \right) i_j$$

In this case, it is the last dimension, $D_n$, that does not appear in the formula.

**Passing Multidimensional Arrays as Parameters in C**

The rowwise representation of multidimensional arrays has a direct effect in C programming. For one dimensional arrays, the size of the array is not required to compute where any specific element is located in memory. This is not true for multidimensional arrays. To access the elements of these arrays, the compiler must know all but the first dimension. This becomes apparent when considering the prototype of a function that takes a multidimensional array as a parameter. The following will not compile:

```c
void f( int a [][] ); /* no dimension information */
```
However, the following does compile:

```c
void f( int a [ ][2] );
```

Any two dimensional array with two columns can be passed to this function. The first dimension is not required.

Do not be confused by a function with this prototype:

```c
void f( int * a [ ];
```

This defines a single dimensional array of integer pointers (which incidently can be used to create an array of arrays that acts much like a two dimensional array).

For higher dimensional arrays, all but the first dimension of the array must be specified for parameters. For example, a four dimensional array parameter might be passed like:

```c
void f( int a [ ][4][3][2] );
```

### 5.2 Array/String Instructions

The 80x86 family of processors provide several instructions that are designed to work with arrays. These instructions are called *string instructions*. They use the index registers (ESI and EDI) to perform an operation and then to automatically increment or decrement one or both of the index registers. The *direction flag* (DF) in the FLAGS register determines where the index registers are incremented or decremented. There are two instructions that modify the direction flag:

- **CLD** clears the direction flag. In this state, the index registers are incremented.
- **STD** sets the direction flag. In this state, the index registers are decremented.

A very common mistake in 80x86 programming is to forget to explicitly put the direction flag in the correct state. This often leads to code that works most of the time (when the direction flag happens to be in the desired state), but does not work all the time.

#### 5.2.1 Reading and writing memory

The simplest string instructions either read or write memory or both. They may read or write a byte, word or double word at a time. Figure 5.7

---

2 A size can be specified here, but it is ignored by the compiler.
5.2. ARRAY/STRING INSTRUCTIONS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ESI = ESI ± 1</td>
<td></td>
<td>EDI = EDI ± 1</td>
</tr>
<tr>
<td></td>
<td>ESI = ESI ± 2</td>
<td></td>
<td>EDI = EDI ± 2</td>
</tr>
<tr>
<td></td>
<td>ESI = ESI ± 4</td>
<td></td>
<td>EDI = EDI ± 4</td>
</tr>
</tbody>
</table>

Figure 5.7: Reading and writing string instructions

```plaintext
segment .data
array1 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
segment .bss
array2 resd 10
```

```plaintext
segment .text
  cld ; don’t forget this!
  mov esi, array1
  mov edi, array2
  mov ecx, 10

lp:
  lodsd
  stosd
  loop lp
```

Figure 5.8: Load and store example

shows these instructions with a short pseudo-code description of what they do. There are several points to notice here. First, ESI is used for reading and EDI for writing. It is easy to remember this if one remembers that SI stands for Source Index and DI stands for Destination Index. Next, notice that the register that holds the data is fixed (either AL, AX or EAX). Finally, note that the storing instructions use ES to determine the segment to write to, not DS. In protected mode programming this is not usually a problem, since there is only one data segment and ES should be automatically initialized to reference it (just as DS is). However, in real mode programming, it is very important for the programmer to initialize ES to the correct segment selector value. Figure 5.8 shows an example use of these instructions that

---

3 Another complication is that one can not copy the value of the DS register into the ES register directly using a single MOV instruction. Instead, the value of DS must be copied to a general purpose register (like AX) and then copied from that register to ES using two
copies an array into another.

The combination of a **LODSx** and **STOSx** instruction (as in lines 13 and 14 of Figure 5.8) is very common. In fact, this combination can be performed by a single **MOV$Sx** string instruction. Figure 5.9 describes the operations that these instructions perform. Lines 13 and 14 of Figure 5.8 could be replaced with a single **MOVSD** instruction with the same effect. The only difference would be that the EAX register would not be used at all in the loop.

5.2.2 The REP instruction prefix

The 80x86 family provides a special instruction prefix\(^4\) called **REP** that can be used with the above string instructions. This prefix tells the CPU to repeat the next string instruction a specified number of times. The ECX

---

\(^4\)A instruction prefix is not an instruction, it is a special byte that is placed before a string instruction that modifies the instructions behavior. Other prefixes are also used to override segment defaults of memory accesses
5.2. ARRAY/STRING INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>ESI Change</th>
<th>EDI Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPSB</td>
<td>compares byte [DS:ESI] and byte [ES:EDI]</td>
<td>ESI = ESI ± 1</td>
<td>EDI = EDI ± 1</td>
</tr>
<tr>
<td>CMPSW</td>
<td>compares word [DS:ESI] and word [ES:EDI]</td>
<td>ESI = ESI ± 2</td>
<td>EDI = EDI ± 2</td>
</tr>
<tr>
<td>CMPSD</td>
<td>compares dword [DS:ESI] and dword [ES:EDI]</td>
<td>ESI = ESI ± 4</td>
<td>EDI = EDI ± 4</td>
</tr>
<tr>
<td>SCASB</td>
<td>compares AL and [ES:EDI]</td>
<td>EDI ± 1</td>
<td></td>
</tr>
<tr>
<td>SCASW</td>
<td>compares AX and [ES:EDI]</td>
<td>EDI ± 2</td>
<td></td>
</tr>
<tr>
<td>SCASD</td>
<td>compares EAX and [ES:EDI]</td>
<td>EDI ± 4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.11: Comparison string instructions

register is used to count the iterations (just as for the LOOP instruction). Using the REP prefix, the loop in Figure 5.8 (lines 12 to 15) could be replaced with a single line:

```
rep movsd
```

Figure 5.10 shows another example that zeroes out the contents of an array.

5.2.3 Comparison string instructions

Figure 5.11 shows several new string instructions that can be used to compare memory with other memory or a register. They are useful for comparing or searching arrays. They set the FLAGS register just like the CMP instruction. The CMPSx instructions compare corresponding memory locations and the SCASx scan memory locations for a specific value.

Figure 5.12 shows a short code snippet that searches for the number 12 in a double word array. The SCASD instruction on line 10 always adds 4 to EDI, even if the value searched for is found. Thus, if one wishes to find the address of the 12 found in the array, it is necessary to subtract 4 from EDI (as line 16 does).

5.2.4 The REPx instruction prefixes

There are several other REP-like instruction prefixes that can be used with the comparison string instructions. Figure 5.13 shows the two new
segment .bss
array resd 100

segment .text
cld
mov edi, array ; pointer to start of array
mov ecx, 100 ; number of elements
mov eax, 12 ; number to scan for
lp:
    scasd
    je found
    loop lp
; code to perform if not found
    jmp onward
found:
    sub edi, 4 ; edi now points to 12 in array
; code to perform if found
onward:

Figure 5.12: Search example

<table>
<thead>
<tr>
<th>REPE, REPZ</th>
<th>repeats instruction while Z flag is set or at most ECX times</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPNE, REPNZ</td>
<td>repeats instruction while Z flag is cleared or at most ECX times</td>
</tr>
</tbody>
</table>

Figure 5.13: REPx instruction prefixes

prefixes and describes their operation. REPE and REPZ are just synonyms for the same prefix (as are REPNE and REPNZ). If the repeated comparison string instruction stops because of the result of the comparison, the index register or registers are still incremented and ECX decremented; however, the FLAGS register still holds the state that terminated the repetition. Thus, it is possible to use the Z flag to determine if the repeated comparisons stopped because of a comparison or ECX becoming zero.

Figure 5.14 shows an example code snippet that determines if two blocks of memory are equal. The JE on line 7 of the example checks to see the result of the previous instruction. If the repeated comparison stopped because it found two unequal bytes, the Z flag will still be cleared and no branch is made; however, if the comparisons stopped because ECX became zero, the Z flag will still be set and the code branches to the equal label.
5.2. ARRAY/STRING INSTRUCTIONS

5.2.5 Example

This section contains an assembly source file with several functions that implement array operations using string instructions. Many of the functions duplicate familiar C library functions.

```assembly
global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy

segment .text
    ; function _asm_copy
    ; copies blocks of memory
    ; C prototype
    ; void asm_copy( void * dest, const void * src, unsigned sz);
    ; parameters:
    ; dest - pointer to buffer to copy to
    ; src - pointer to buffer to copy from
    ; sz - number of bytes to copy
    ; next, some helpful symbols are defined

%define dest [ebp+8]
%define src [ebp+12]
%define sz [ebp+16]
_asm_copy:
    enter 0, 0
    push esi
```

Figure 5.14: Comparing memory blocks
21 push edi
22
23 mov esi, src ; esi = address of buffer to copy from
24 mov edi, dest ; edi = address of buffer to copy to
25 mov ecx, sz ; ecx = number of bytes to copy
26
27 cld ; clear direction flag
28 rep movsb ; execute movsb ECX times
29
30 pop edi
31 pop esi
32 leave
33 ret
34
35
36 ; function _asm_find
37 ; searches memory for a given byte
38 ; void * asm_find( const void * src, char target, unsigned sz);
39 ; parameters:
40 ; src - pointer to buffer to search
41 ; target - byte value to search for
42 ; sz - number of bytes in buffer
43 ; return value:
44 ; if target is found, pointer to first occurrence of target in buffer
45 ; is returned
46 ; else
47 ; NULL is returned
48 ; NOTE: target is a byte value, but is pushed on stack as a dword value.
49 ; The byte value is stored in the lower 8-bits.
50 ;
51 %define src [ebp+8]
52 %define target [ebp+12]
53 %define sz [ebp+16]
54
55 _asm_find:
56 enter 0,0
57 push edi
58
59 mov eax, target ; al has value to search for
60 mov edi, src
61 mov ecx, sz
62 cld
5.2. ARRAY/STRING INSTRUCTIONS

63         repne  scasb          ; scan until ECX == 0 or [ES:EDI] == AL
65         je     found_it       ; if zero flag set, then found value
66         mov   eax, 0          ; if not found, return NULL pointer
67         jmp   short quit
69     found_it:
70         mov   eax, edi
71         dec   eax            ; if found return (DI - 1)
72     quit:
73         pop   edi
74         leave
75         ret

77         ; function _asm_strlen
78         ; returns the size of a string
79         ; unsigned asm_strlen( const char * );
80         ; parameter:
81         ;   src - pointer to string
82         ; return value:
83         ;   number of chars in string (not counting, ending 0) (in EAX)
84         
85         %define src [ebp + 8]
86     _asm_strlen:
87           enter  0,0
88           push  edi
89           mov   edi, src        ; edi = pointer to string
90           mov   ecx, OFFFFFFFFFh ; use largest possible ECX
91           xor   al,al           ; al = 0
92           cld
93           repnz  scasb         ; scan for terminating 0
94           mov   eax, OFFFFFFFFFh
95           sub   eax, ecx       ; length = OFFFFFFFFFh - ecx

CHAPTER 5. ARRAYS

105        pop    edi
106       leave
107       ret
108
109     ; function _asm_strcpy
110     ; copies a string
111     ; void asm_strcpy( char * dest, const char * src);
112     ; parameters:
113     ; dest - pointer to string to copy to
114     ; src - pointer to string to copy from
115     ;
116     %define dest [ebp + 8]
117     %define src [ebp + 12]
118
119 _asm_strcpy:
120       enter 0,0
121       push esi
122       push edi
123
124       mov edi, dest
125       mov esi, src
126       cld
127
128 cpy_loop:
129       lodsb ; load AL & inc si
130       stosb ; store AL & inc di
131       or    al, al ; set condition flags
132       jnz   cpy_loop ; if not past terminating 0, continue
133
134       pop    edi
135       pop    esi
136       leave
137       ret

#include <stdio.h>
#define STR_SIZE 30
/* prototypes */

void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
void * asm_find( const void *,
                 char target, unsigned ) __attribute__((cdecl));
5.2. ARRAY/STRING INSTRUCTIONS

```c
unsigned asm_strlen( const char * ) __attribute__((cdecl));
void asm_strcpy( char *, const char *) __attribute__((cdecl));

int main()
{
    char st1[STR_SIZE] = "test string";
    char st2[STR_SIZE];
    char * st;
    char ch;

    asm_copy(st2, st1, STR_SIZE); /* copy all 30 chars of string */
    printf("%s\n", st2);

    printf("Enter a char : "); /* look for byte in string */
    scanf("%c%[^\n]%" , &ch);
    st = asm_find(st2, ch, STR_SIZE);
    if ( st )
        printf("Found it: %s\n", st);
    else
        printf("Not found\n");

    st1[0] = 0;
    printf("Enter string : ");
    scanf("%s", st1);
    printf("len = %u\n", asm_strlen(st1));

    asm_strcpy( st2, st1); /* copy meaningful data in string */
    printf("%s\n", st2);

    return 0;
}
```

memex.c
Chapter 6

Floating Point

6.1 Floating Point Representation

6.1.1 Non-integral binary numbers

When number systems were discussed in the first chapter, only integer
values were discussed. Obviously, it must be possible to represent non-
integral numbers in other bases as well as decimal. In decimal, digits to the
right of the decimal point have associated negative powers of ten:

\[0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}\]

Not surprisingly, binary numbers work similarly:

\[0.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625\]

This idea can be combined with the integer methods of Chapter 1 to convert
a general number:

\[110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375\]

Converting from decimal to binary is not very difficult either. In general,
divide the decimal number into two parts: integer and fraction. Convert the
integer part to binary using the methods from Chapter 1. The fractional
part is converted using the method described below.

Consider a binary fraction with the bits labeled \(a, b, c, \ldots\). The number
in binary then looks like:

\[0.abcdef \ldots\]

Multiply the number by two. The binary representation of the new number
will be:

\[a.bcdef \ldots\]
\[
\begin{align*}
0.5625 \times 2 &= 1.125 & \text{first bit} &= 1 \\
0.125 \times 2 &= 0.25 & \text{second bit} &= 0 \\
0.25 \times 2 &= 0.5 & \text{third bit} &= 0 \\
0.5 \times 2 &= 1.0 & \text{fourth bit} &= 1
\end{align*}
\]

Figure 6.1: Converting 0.5625 to binary

\[
\begin{align*}
0.85 \times 2 &= 1.7 \\
0.7 \times 2 &= 1.4 \\
0.4 \times 2 &= 0.8 \\
0.8 \times 2 &= 1.6 \\
0.6 \times 2 &= 1.2 \\
0.2 \times 2 &= 0.4 \\
0.4 \times 2 &= 0.8 \\
0.8 \times 2 &= 1.6
\end{align*}
\]

Figure 6.2: Converting 0.85 to binary

Note that the first bit is now in the one’s place. Replace the \( a \) with 0 to get:

\[
0.bcdef \ldots
\]

and multiply by two again to get:

\[
b.cdef \ldots
\]

Now the second bit (\( b \)) is in the one’s position. This procedure can be repeated until as many bits are needed are found. Figure 6.1 shows a real example that converts 0.5625 to binary. The method stops when a fractional part of zero is reached.

As another example, consider converting 23.85 to binary. It is easy to convert the integral part (23 = 101112), but what about the fractional part (0.85)? Figure 6.2 shows the beginning of this calculation. If one looks at
the numbers carefully, an infinite loop is found! This means that 0.85 is a repeating binary (as opposed to a repeating decimal in base $10^1$). There is a pattern to the numbers in the calculation. Looking at the pattern, one can see that $0.85 = 0.1101101102$. Thus, $23.85 = 10111.11011011102$.

One important consequence of the above calculation is that 23.85 cannot be represented exactly in binary using a finite number of bits. (Just as $\frac{1}{3}$ cannot be represented in decimal with a finite number of digits.) As this chapter shows, float and double variables in C are stored in binary. Thus, values like 23.85 can not be stored exactly into these variables. Only an approximation of 23.85 can be stored.

To simplify the hardware, floating point numbers are stored in a consistent format. This format uses scientific notation (but in binary, using powers of two, not ten). For example, 23.85 or $10111.11011001100110\ldots$ would be stored as:

$$1.01111011001100110\ldots \times 2^{100}$$

(where the exponent (100) is in binary). A normalized floating point number has the form:

$$1.sssssssssssssss \times 2^{eeeeee}$$

where $1.sssssssssssss$ is the significand and $eeeeee$ is the exponent.

### 6.1.2 IEEE floating point representation

The IEEE (Institute of Electrical and Electronic Engineers) is an international organization that has designed specific binary formats for storing floating point numbers. This format is used on most (but not all!) computers made today. Often it is supported by the hardware of the computer itself. For example, Intel’s numeric (or math) coprocessors (which are built into all its CPU’s since the Pentium) use it. The IEEE defines two different formats with different precisions: single and double precision. Single precision is used by float variables in C and double precision is used by double variables.

Intel’s math coprocessor also uses a third, higher precision called extended precision. In fact, all data in the coprocessor itself is in this precision. When it is stored in memory from the coprocessor it is converted to either single or double precision automatically. Extended precision uses a slightly different general format than the IEEE float and double formats and so will not be discussed here.

---

1. It should not be so surprising that a number might repeat in one base, but not another. Think about $\frac{1}{3}$, it repeats in decimal, but in ternary (base 3) it would be $0.1_3$.

2. Some compiler’s (such as Borland) long double type uses this extended precision. However, other compilers use double precision for both double and long double. (This is allowed by ANSI C.)
IEEE single precision

Single precision floating point uses 32 bits to encode the number. It is usually accurate to 7 significant decimal digits. Floating point numbers are stored in a much more complicated format than integers. Figure 6.3 shows the basic format of a IEEE single precision number. There are several quirks to the format. Floating point numbers do not use the two’s complement representation for negative numbers. They use a signed magnitude representation. Bit 31 determines the sign of the number as shown.

The binary exponent is not stored directly. Instead, the sum of the exponent and 7F is stored from bit 23 to 30. This biased exponent is always non-negative.

The fraction part assumes a normalized significand (in the form 1.ssssssssss). Since the first bit is always an one, the leading one is not stored! This allows the storage of an additional bit at the end and so increases the precision slightly. This idea is known as the hidden one representation.

How would 23.85 be stored? First, it is positive so the sign bit is 0. Next the true exponent is 4, so the biased exponent is 7F + 4 = 83. Finally, the fraction is 0111101100110011001100 (remember the leading one is hidden). Putting this all together (to help clarify the different sections of the floating point format, the sign bit and the fraction have been underlined and the bits have been grouped into 4-bit nibbles):

\[
\begin{array}{ccccc}
0 & 100 & 0001 & 1011 & 11101100110011001100 \\
\end{array}
\]

This is not exactly 23.85 (since it is a repeating binary). If one converts the above back to decimal, one finds that it is approximately 23.849998474. This number is very close to 23.85, but it is not exact. Actually, in C, 23.85 would not be represented exactly as above. Since the left-most bit that was truncated from the exact representation is 1, the last bit is rounded up to 1. So 23.85 would be represented as 41 BE CC CD in hex using single precision. Converting this to decimal results in 23.850000381 which is a slightly better approximation of 23.85.
6.1. FLOATING POINT REPRESENTATION

- $e = 0$ and $f = 0$ denotes the number zero (which can not be normalized) Note that there is a +0 and -0.
- $e = 0$ and $f \neq 0$ denotes a denormalized number. These are discussed in the next section.
- $e = FF$ and $f = 0$ denotes infinity ($\infty$). There are both positive and negative infinities.
- $e = FF$ and $f \neq 0$ denotes an undefined result, known as NaN (Not a Number).

Table 6.1: Special values of $f$ and $e$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: IEEE double precision

How would -23.85 be represented? Just change the sign bit: C1 BE CC CD. Do not take the two’s complement!

Certain combinations of $e$ and $f$ have special meanings for IEEE floats. Table 6.1 describes these special values. An infinity is produced by an overflow or by division by zero. An undefined result is produced by an invalid operation such as trying to find the square root of a negative number, adding two infinities, etc.

Normalized single precision numbers can range in magnitude from $1.0 \times 2^{-126}$ ($\approx 1.1755 \times 10^{-38}$) to $1.11111 \ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{38}$).

Denormalized numbers

Denormalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below $1.0 \times 2^{-126}$). For example, consider the number $1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$). In the given normalized form, the exponent is too small. However, it can be represented in the unnormalized form: $0.01001_2 \times 2^{-127}$. To store this number, the biased exponent is set to 0 (see Table 6.1) and the fraction is the complete significand of the number written as a product with $2^{-127}$ (i.e. all bits are stored including the one to the left of the decimal point). The representation of $1.001 \times 2^{-129}$ is then:

0 000 0000 0001 0010 0000 0000 0000 0000 0000
IEEE double precision

IEEE double precision uses 64 bits to represent numbers and is usually accurate to about 15 significant decimal digits. As Figure 6.4 shows, the basic format is very similar to single precision. More bits are used for the biased exponent (11) and the fraction (52) than for single precision.

The larger range for the biased exponent has two consequences. The first is that it is calculated as the sum of the true exponent and 3FF (1023) (not 7F as for single precision). Secondly, a large range of true exponents (and thus a larger range of magnitudes) is allowed. Double precision magnitudes can range from approximately $10^{-308}$ to $10^{308}$.

It is the larger field of the fraction that is responsible for the increase in the number of significant digits for double values.

As an example, consider 23.85 again. The biased exponent will be $4 + 3FF = 403$ in hex. Thus, the double representation would be:

```
0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010
```
or 40 37 D9 99 99 99 99 9A in hex. If one converts this back to decimal, one finds 23.8500000000000014 (there are 12 zeros!) which is a much better approximation of 23.85.

The double precision has the same special values as single precision.

Denormalized numbers are also very similar. The only main difference is that double denormalized numbers use $2^{-1023}$ instead of $2^{-127}$.

6.2 Floating Point Arithmetic

Floating point arithmetic on a computer is different than in continuous mathematics. In mathematics, all numbers can be considered exact. As shown in the previous section, on a computer many numbers can not be represented exactly with a finite number of bits. All calculations are performed with limited precision. In the examples of this section, numbers with an 8-bit significand will be used for simplicity.

6.2.1 Addition

To add two floating point numbers, the exponents must be equal. If they are not already equal, then they must be made equal by shifting the significand of the number with the smaller exponent. For example, consider $10.375 + 6.34375 = 16.71875$ or in binary:

```
1.0100110 \times 2^3  \\
+ 1.1001011 \times 2^2
```

\footnote{The only difference is that for the infinity and undefined values, the biased exponent is 7FF not FF.}
6.2. FLOATING POINT ARITHMETIC

These two numbers do not have the same exponent so shift the significand to make the exponents the same and then add:

\[
\begin{align*}
1.0100110 \times 2^3 \\
+ 0.1100110 \times 2^3 \\
\hline
10.0001110 \times 2^4
\end{align*}
\]

Note that the shifting of 1.1001011 \times 2^2 drops off the trailing one and after rounding results in 0.1100110 \times 2^3. The result of the addition, 10.0001110 \times 2^4 (or 1.00001100 \times 2^4) is equal to 10000.110_2 or 16.75. This is not equal to the exact answer (16.71875)! It is only an approximation due to the round off errors of the addition process.

It is important to realize that floating point arithmetic on a computer (or calculator) is always an approximation. The laws of mathematics do not always work with floating point numbers on a computer. Mathematics assumes infinite precision which no computer can match. For example, mathematics teaches that \((a + b) - b = a\); however, this may not hold true exactly on a computer!

6.2.2 Subtraction

Subtraction works very similarly and has the same problems as addition. As an example, consider 16.75 – 15.9375 = 0.8125:

\[
\begin{align*}
1.0000110 \times 2^4 \\
- 1.1111111 \times 2^3 \\
\hline
0.0000110 \times 2^4
\end{align*}
\]

Shifting 1.1111111 \times 2^3 gives (rounding up) 1.0000000 \times 2^4

\[
\begin{align*}
1.0000110 \times 2^4 \\
- 1.0000000 \times 2^4 \\
\hline
0.0000110 \times 2^4
\end{align*}
\]

0.0000110 \times 2^4 = 0.11_2 = 0.75 which is not exactly correct.

6.2.3 Multiplication and division

For multiplication, the significands are multiplied and the exponents are added. Consider 10.375 \times 2.5 = 25.9375:

\[
\begin{align*}
1.0100110 \times 2^3 \\
\times 1.0100000 \times 2^1 \\
\hline
10100110 \\
+ 10100110 \\
\hline
1.10011111000000 \times 2^4
\end{align*}
\]
Of course, the real result would be rounded to 8-bits to give:

\[ 1.1010000 \times 2^4 = 11010.000_2 = 26 \]

Division is more complicated, but has similar problems with round off errors.

### 6.2.4 Ramifications for programming

The main point of this section is that floating point calculations are not exact. The programmer needs to be aware of this. A common mistake that programmers make with floating point numbers is to compare them assuming that a calculation is exact. For example, consider a function named \( f(x) \) that makes a complex calculation and a program is trying to find the function’s root. One might be tempted to use the following statement to check to see if \( x \) is a root:

```c
if ( f(x) == 0.0 )
```

But, what if \( f(x) \) returns \( 1 \times 10^{-30} \)? This very likely means that \( x \) is a very good approximation of a true root; however, the equality will be false. There may not be any IEEE floating point value of \( x \) that returns exactly zero, due to round off errors in \( f(x) \).

A much better method would be to use:

```c
if ( fabs(f(x)) < EPS )
```

where EPS is a macro defined to be a very small positive value (like \( 1 \times 10^{-10} \)). This is true whenever \( f(x) \) is very close to zero. In general, to compare a floating point value (say \( x \)) to another (\( y \)) use:

```c
if ( fabs(x - y)/fabs(y) < EPS )
```

### 6.3 The Numeric Coprocessor

#### 6.3.1 Hardware

The earliest Intel processors had no hardware support for floating point operations. This does not mean that they could not perform float operations. It just means that they had to be performed by procedures composed of many non-floating point instructions. For these early systems, Intel did provide an additional chip called a **math coprocessor**. A math coprocessor has machine instructions that perform many floating point operations much faster than using a software procedure (on early processors, at least 10 times faster than using a software procedure).

---

^4A root of a function is a value \( x \) such that \( f(x) = 0 \)
6.3. THE NUMERIC COPROCESSOR

faster!). The coprocessor for the 8086/8088 was called the 8087. For the 80286, there was a 80287 and for the 80386, a 80387. The 80486DX processor integrated the math coprocessor into the 80486 itself. Since the Pentium, all generations of 80x86 processors have a built-in math coprocessor; however, it is still programmed as if it was a separate unit. Even earlier systems without a coprocessor can install software that emulates a math coprocessor. These emulator packages are automatically activated when a program executes a coprocessor instruction and run a software procedure that produces the same result as the coprocessor would have (though much slower, of course).

The numeric coprocessor has eight floating point registers. Each register holds 80 bits of data. Floating point numbers are always stored as 80-bit extended precision numbers in these registers. The registers are named ST0, ST1, ST2, ... ST7. The floating point registers are used differently than the integer registers of the main CPU. The floating point registers are organized as a stack. Recall that a stack is a Last-In First-Out (LIFO) list. ST0 always refers to the value at the top of the stack. All new numbers are added to the top of the stack. Existing numbers are pushed down on the stack to make room for the new number.

There is also a status register in the numeric coprocessor. It has several flags. Only the 4 flags used for comparisons will be covered: C0, C1, C2 and C3. The use of these is discussed later.

6.3.2 Instructions

To make it easy to distinguish the normal CPU instructions from coprocessor ones, all the coprocessor mnemonics start with an F.

Loading and storing

There are several instructions that load data onto the top of the coprocessor register stack:

- `FLD source` loads a floating point number from memory onto the top of the stack. The source may be a single, double or extended precision number or a coprocessor register.
- `FILD source` reads an integer from memory, converts it to floating point and stores the result on top of the stack. The source may be either a word, double word or quad word.
- `FLD1` stores a one on the top of the stack.
- `FLDZ` stores a zero on the top of the stack.

There are also several instructions that store data from the stack into memory. Some of these instructions also `pop (i.e. remove)` the number from

5However, the 80486SX did not have an integrated coprocessor. There was a separate 80487SX chip for these machines.
the stack as it stores it.

**FST dest**
stores the top of the stack (ST0) into memory. The destination may either be a single or double precision number or a coprocessor register.

**FSTP dest**
stores the top of the stack into memory just as FST; however, after the number is stored, its value is popped from the stack. The destination may either a single, double or extended precision number or a coprocessor register.

**FIST dest**
stores the value of the top of the stack converted to an integer into memory. The destination may either a word or a double word. The stack itself is unchanged. How the floating point number is converted to an integer depends on some bits in the coprocessor's control word. This is a special (non-floating point) word register that controls how the coprocessor works. By default, the control word is initialized so that it rounds to the nearest integer when it converts to integer. However, the FSTCW (Store Control Word) and FLDCW (Load Control Word) instructions can be used to change this behavior.

**FISTP dest**
Same as FIST except for two things. The top of the stack is popped and the destination may also be a quad word.

There are two other instructions that can move or remove data on the stack itself.

**FXCH STn**
exchanges the values in ST0 and STn on the stack (where n is register number from 1 to 7).

**FFREE STn**
frees up a register on the stack by marking the register as unused or empty.

**Addition and subtraction**

Each of the addition instructions compute the sum of ST0 and another operand. The result is always stored in a coprocessor register.

**FADD src**
ST0 += src. The src may be any coprocessor register or a single or double precision number in memory.

**FADD dest, ST0**
dest += ST0. The dest may be any coprocessor register.

**FADDP dest** or **FADDP dest, ST0**
dest += ST0 then pop stack. The dest may be any coprocessor register.

**FIADD src**
ST0 += (float) src. Adds an integer to ST0. The src must be a word or double word in memory.

There are twice as many subtraction instructions than addition because the order of the operands is important for subtraction (i.e. \( a + b = b + a \), but \( a - b \neq b - a \)). For each instruction, there is an alternate one that subtracts in the reverse order. These reverse instructions all end in either
R or RP. Figure 6.5 shows a short code snippet that adds up the elements of an array of doubles. On lines 10 and 13, one must specify the size of the memory operand. Otherwise the assembler would not know whether the memory operand was a float (dword) or a double (qword).

<table>
<thead>
<tr>
<th>Segment .bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>array resq SIZE</td>
</tr>
<tr>
<td>sum resq 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment .text</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ecx, SIZE</td>
</tr>
<tr>
<td>mov esi, array</td>
</tr>
<tr>
<td>fldz ; ST0 = 0</td>
</tr>
<tr>
<td>lp:</td>
</tr>
<tr>
<td>fadd qword [esi] ; ST0 += *(esi)</td>
</tr>
<tr>
<td>add esi, 8 ; move to next double</td>
</tr>
<tr>
<td>loop lp</td>
</tr>
<tr>
<td>fstp qword sum ; store result into sum</td>
</tr>
</tbody>
</table>

**Figure 6.5: Array sum example**

FSUB src ST0 -= src. The src may be any coprocessor register or a single or double precision number in memory.

FSUBR src ST0 = src - ST0. The src may be any coprocessor register or a single or double precision number in memory.

FSUB dest, ST0 dest -= ST0. The dest may be any coprocessor register.

FSUBR dest, ST0 dest = ST0 - dest. The dest may be any coprocessor register.

FSUBP dest or dest -= ST0 then pop stack. The dest may be any coprocessor register.

FSUBRP dest or dest = ST0 - dest then pop stack. The dest may be any coprocessor register.

FISUB src ST0 -= (float) src. Subtracts an integer from ST0. The src must be a word or double word in memory.

FISUBR src ST0 = (float) src - ST0. Subtracts ST0 from an integer. The src must be a word or double word in memory.
CHAPTER 6. FLOATING POINT

Multiplication and division

The multiplication instructions are completely analogous to the addition instructions.

\[
\text{FMUL } src \quad \text{ST0} *= src. \text{ The src may be any coprocessor register or a single or double precision number in memory.}
\]

\[
\text{FMUL dest, ST0} \quad \text{dest} *= \text{ST0}. \text{ The dest may be any coprocessor register.}
\]

\[
\text{FMULP dest or FMULP dest, ST0} \quad \text{dest} *= \text{ST0} \text{ then pop stack. The dest may be any coprocessor register.}
\]

\[
\text{FIMUL src} \quad \text{ST0} *= (\text{float}) src. \text{ Multiplies an integer to ST0. The src must be a word or double word in memory.}
\]

Not surprisingly, the division instructions are analogous to the subtraction instructions. Division by zero results in an infinity.

\[
\text{FDIV src} \quad \text{ST0} /= src. \text{ The src may be any coprocessor register or a single or double precision number in memory.}
\]

\[
\text{FDIVR src} \quad \text{ST0} = src / \text{ST0}. \text{ The src may be any coprocessor register or a single or double precision number in memory.}
\]

\[
\text{FDIV dest, ST0} \quad \text{dest} /= \text{ST0}. \text{ The dest may be any coprocessor register.}
\]

\[
\text{FDIVR dest, ST0} \quad \text{dest} = \text{ST0} / \text{dest}. \text{ The dest may be any coprocessor register.}
\]

\[
\text{FDIVP dest or FDIVP dest, ST0} \quad \text{dest} /= \text{ST0} \text{ then pop stack. The dest may be any coprocessor register.}
\]

\[
\text{FDIVRP dest or FDIVRP dest, ST0} \quad \text{dest} = \text{ST0} / \text{dest} \text{ then pop stack. The dest may be any coprocessor register.}
\]

\[
\text{FIDIV src} \quad \text{ST0} /= (\text{float}) src. \text{ Divides ST0 by an integer. The src must be a word or double word in memory.}
\]

\[
\text{FIDIVR src} \quad \text{ST0} = (\text{float}) src / \text{ST0}. \text{ Divides an integer by ST0. The src must be a word or double word in memory.}
\]

Comparisons

The coprocessor also performs comparisons of floating point numbers. The FCOM family of instructions does this operation.
6.3. THE NUMERIC COPROCESSOR

Figure 6.6: Comparison example

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>; if ( x &gt; y )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>fld qword [x] ; ST0 = x</td>
<td>The instruction loads the value of x into ST0.</td>
</tr>
<tr>
<td>4</td>
<td>fcomp qword [y] ; compare ST0 and y</td>
<td>The instruction compares the value of ST0 with that at memory location [y].</td>
</tr>
<tr>
<td>5</td>
<td>fstsw ax ; move C bits into FLAGS</td>
<td>The instruction moves the C bits from ST0 into the AX register.</td>
</tr>
<tr>
<td>6</td>
<td>sahf</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>jna else_part ; if x not above y, goto else_part</td>
<td>The jump instruction transfers control to the else_part.</td>
</tr>
<tr>
<td>8</td>
<td>then_part;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>; code for then part</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>jmp end_if</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>else_part;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>; code for else part</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>end_if:</td>
<td></td>
</tr>
</tbody>
</table>

FCOM src compares ST0 and src. The src can be a coprocessor register or a float or double in memory.

FCOMP src compares ST0 and src, then pops stack. The src can be a coprocessor register or a float or double in memory.

FCOMPP compares ST0 and ST1, then pops stack twice.

FICOM src compares ST0 and (float) src. The src can be a word or dword integer in memory.

FICOMP src compares ST0 and (float)src, then pops stack. The src can be a word or dword integer in memory.

FTST compares ST0 and 0.

These instructions change the C₀, C₁, C₂ and C₃ bits of the coprocessor status register. Unfortunately, it is not possible for the CPU to access these bits directly. The conditional branch instructions use the FLAGS register, not the coprocessor status register. However, it is relatively simple to transfer the bits of the status word into the corresponding bits of the FLAGS register using some new instructions:

FSTSW dest Stores the coprocessor status word into either a word in memory or the AX register.

SAHF Stores the AH register into the FLAGS register.

LAHF Loads the AH register with the bits of the FLAGS register.

Figure 6.6 shows a short example code snippet. Lines 5 and 6 transfer the C₀, C₁, C₂ and C₃ bits of the coprocessor status word into the FLAGS register. The bits are transferred so that they are analogous to the result of a comparison of two unsigned integers. This is why line 7 uses a JNA instruction.
The Pentium Pro (and later processors (Pentium II and III)) support two new comparison operators that directly modify the CPU’s FLAGS register.

- **FCOMI src** compares ST0 and src. The src must be a coprocessor register.
- **FCOMIP src** compares ST0 and src, then pops stack. The src must be a coprocessor register.

Figure 6.7 shows an example subroutine that finds the maximum of two doubles using the FCOMIP instruction. Do not confuse these instructions with the integer comparison functions (FICOM and FICOMP).

### Miscellaneous instructions

This section covers some other miscellaneous instructions that the coprocessor provides.

- **FCHS ST0** = - ST0 Changes the sign of ST0
- **FABS ST0** = |ST0| Takes the absolute value of ST0
- **FSQRT ST0** = \( \sqrt{ST0} \) Takes the square root of ST0
- **FScale ST0** = \( ST0 \times 2^{\lfloor ST1 \rfloor} \) multiples ST0 by a power of 2 quickly. ST1 is not removed from the coprocessor stack. Figure 6.8 shows an example of how to use this instruction.

### 6.3.3 Examples

### 6.3.4 Quadratic formula

The first example shows how the quadratic formula can be encoded in assembly. Recall that the quadratic formula computes the solutions to the quadratic equation:

\[ ax^2 + bx + c = 0 \]

The formula itself gives two solutions for \( x \): \( x_1 \) and \( x_2 \).

\[ x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

The expression inside the square root \( (b^2 - 4ac) \) is called the *discriminant*. Its value is useful in determining which of the following three possibilities are true for the solutions.

1. There is only one real degenerate solution. \( b^2 - 4ac = 0 \)
2. There are two real solutions. \( b^2 - 4ac > 0 \)
3. There are two complex solutions. \( b^2 - 4ac < 0 \)

Here is a small C program that uses the assembly subroutine:
```c
#include <stdio.h>

int quadratic ( double, double, double, double *, double *);

int main()
{
    double a, b, c, root1, root2;

    printf ("Enter a, b, c:");
    scanf("%lf %lf %lf", &a, &b, &c);
    if ( quadratic ( a, b, c, &root1, &root2) )
        printf ("roots: %.10g %.10g \n", root1, root2);
    else
        printf ("No real roots\n");
    return 0;
}
```

Here is the assembly routine:

```assembly
; function quadratic
; finds solutions to the quadratic equation:
; a*x^2 + b*x + c = 0
; C prototype:
; int quadratic( double a, double b, double c,
; double * root1, double *root2 )
; Parameters:
; a, b, c - coefficients of powers of quadratic equation (see above)
; root1 - pointer to double to store first root in
; root2 - pointer to double to store second root in
; Return value:
; returns 1 if real roots found, else 0

%define a qword [ebp+8]
%define b qword [ebp+16]
%define c qword [ebp+24]
%define root1 dword [ebp+32]
%define root2 dword [ebp+36]
%define disc qword [ebp-8]
```
%define one_over_2a qword [ebp-16]

segment .data
MinusFour dw -4

segment .text
  global _quadratic
  _quadratic:
    push ebp
    mov ebp, esp
    sub esp, 16 ; allocate 2 doubles (disc & one_over_2a)
    push ebx ; must save original ebx

    fild word [MinusFour]; stack -4
    fld a ; stack: a, -4
    fld c ; stack: c, a, -4
    fmulp st1 ; stack: a*c, -4
    fmulp st1 ; stack: -4*a*c
    fld b
    fld b ; stack: b, b, -4*a*c
    fmulp st1 ; stack: b*b, -4*a*c
    faddp st1 ; stack: b*b - 4*a*c
    ftst ; test with 0
    fstsw ax
    sahf
    jb no_real_solutions ; if disc < 0, no real solutions
    fsqrt ; stack: sqrt(b*b - 4*a*c)
    fstp disc ; store and pop stack
    fld1 ; stack: 1.0
    fld a ; stack: a, 1.0
    fscale ; stack: a * 2^-1.0 = 2*a, 1
    fdivp st1 ; stack: 1/(2*a)
    fst one_over_2a ; stack: 1/(2*a)
    fld b ; stack: b, 1/(2*a)
    fld disc ; stack: disc, b, 1/(2*a)
    fsubrp st1 ; stack: disc - b, 1/(2*a)
    fmulp st1 ; stack: (-b + disc)/(2*a)
    mov ebx, root1
    fstp qword [ebx] ; store in *root1
    fld b ; stack: b
    fld disc ; stack: disc, b
    fchs ; stack: -disc, b
6.3. THE NUMERIC COPROCESSOR

6.3.5 Reading array from file

In this example, an assembly routine reads doubles from a file. Here is a short C test program:

```c
/*
 * This program tests the 32-bit read_doubles() assembly procedure.
 * It reads the doubles from stdin. (Use redirection to read from file.)
 */
#include <stdio.h>
extern int read_doubles( FILE *, double *, int );
#define MAX 100

int main()
{
    int i, n;
    double a[MAX];

    n = read_doubles(stdin, a, MAX);

    for (i=0; i < n; i++)
        printf("%3d %g\n", i, a[i]);
    return 0;
}
```
Here is the assembly routine

```assembly
section .data
format db "%lf", 0 ; format for fscanf()

section .text
global _read_doubles
extern _fscanf

%define SIZEOF_DOUBLE 8
%define FP dword [ebp + 8]
%define ARRAYP dword [ebp + 12]
%define ARRAY_SIZE dword [ebp + 16]
%define TEMP_DOUBLE [ebp - 8]

; ; function _read_doubles
; C prototype:
;   int read_doubles( FILE * fp, double * arrayp, int array_size );
; This function reads doubles from a text file into an array, until
; EOF or array is full.
; Parameters:
;   fp - FILE pointer to read from (must be open for input)
;   arrayp - pointer to double array to read into
;   array_size - number of elements in array
; Return value:
;   number of doubles stored into array (in EAX)

_read_doubles:
    push ebp
    mov ebp,esp
    sub esp, SIZEOF_DOUBLE ; define one double on stack

    push esi ; save esi
    mov esi, ARRAYP ; esi = ARRAYP
    xor edx, edx ; edx = array index (initially 0)

    while_loop:
        cmp edx, ARRAY_SIZE ; is edx < ARRAY_SIZE?
        jnl short quit ; if not, quit loop
```
6.3. THE NUMERICAL CO PROCESSOR

6.3.6 Finding primes

This final example looks at finding prime numbers again. This implementation is more efficient than the previous one. It stores the primes it has found in an array and only divides by the previous primes it has found instead of every odd number to find new primes.
One other difference is that it computes the square root of the guess for the next prime to determine at what point it can stop searching for factors. It alters the coprocessor control word so that when it stores the square root as an integer, it truncates instead of rounding. This is controlled by bits 10 and 11 of the control word. These bits are called the RC (Rounding Control) bits. If they are both 0 (the default), the coprocessor rounds when converting to integer. If they are both 1, the coprocessor truncates integer conversions. Notice that the routine is careful to save the original control word and restore it before it returns.

Here is the C driver program:

```
#include <stdio.h>
#include <stdlib.h>

/*
 * function find_primes
 * finds the indicated number of primes
 * Parameters:
 * a — array to hold primes
 * n — how many primes to find
 */
extern void find_primes ( int *a , unsigned n );

int main()
{
  int status ;
  unsigned i ;
  unsigned max ;
  int * a ;

  printf ( "How many primes do you wish to find? " );
  scanf ( "%u" , &max );
  a = calloc ( sizeof(int) , max );

  if ( a ) {
    find_primes ( a , max );
    /* print out the last 20 primes found */
    for ( i = ( max > 20 ) ? max - 20 : 0 ; i < max ; i++ )
      printf ( "%3d %d\n" , i+1 , a[i]);
```
6.3. THE NUMERIC COPROCESSOR

```c
free(a);
status = 0;
}
else {
    fprintf(stderr,"Can not create array of %u ints\n", max);
    status = 1;
}
return status;
}
```

Here is the assembly routine:

```assembly
segment .text
    global _find_primes
;
; function find_primes
; finds the indicated number of primes
; Parameters:
;    array - array to hold primes
;    n_find - how many primes to find
; C Prototype:
;extern void find_primes( int * array, unsigned n_find )
;
%define array ebp + 8
%define n_find ebp + 12
%define n ebp - 4    ; number of primes found so far
%define isqrt ebp - 8 ; floor of sqrt of guess
%define orig_cntl_wd ebp - 10 ; original control word
%define new_cntl_wd ebp - 12 ; new control word

_find_primes:
    enter 12,0    ; make room for local variables

    push ebx      ; save possible register variables
    push esi

    fstcw word [orig_cntl_wd]    ; get current control word
    mov ax, [orig_cntl_wd]
```
or  ax, 0C00h ; set rounding bits to 11 (truncate)
mov  [new_cntl_wd], ax
fldcw  word [new_cntl_wd]

mov  esi, [array] ; esi points to array
mov  dword [esi], 2 ; array[0] = 2
mov  dword [esi + 4], 3 ; array[1] = 3
mov  ebx, 5 ; ebx = guess = 5
mov  dword [n], 2 ; n = 2

; This outer loop finds a new prime each iteration, which it adds to the
; end of the array. Unlike the earlier prime finding program, this function
; does not determine primeness by dividing by all odd numbers. It only
; divides by the prime numbers that it has already found. (That’s why they
; are stored in the array.)

while_limit:
  mov  eax, [n]
  cmp  eax, [n_find] ; while ( n < n_find )
  jnb  short quit_limit

  mov  ecx, 1 ; ecx is used as array index
  push ebx ; store guess on stack
  fild  dword [esp] ; load guess onto coprocessor stack
  pop  ebx ; get guess off stack
  fsqrt ; find sqrt(guess)
  fistp  dword [isqrt] ; isqrt = floor(sqrt(guess))

while_factor:
  mov  eax, dword [esi + 4*ecx] ; eax = array[ecx]
  cmp  eax, [isqrt] ; while ( isqrt < array[ecx]
  jnbe  short quit_factor_prime
  mov  eax, ebx
  xor  edx, edx
  div  dword [esi + 4*ecx]
  or  edx, edx ; && guess % array[ecx] != 0 )
  jz  short quit_factor_not_prime
  inc  ecx ; try next prime
6.3. THE NUMERIC COPROCESSOR

69  jmp short while_factor
70
71 ;
72 ; found a new prime!
73 ;
74 quit_factor_prime:
75    mov eax, [n]
76    mov dword [esi + 4*eax], ebx ; add guess to end of array
77    inc eax
78    mov [n], eax ; inc n
79
80 quit_factor_not_prime:
81    add ebx, 2 ; try next odd number
82    jmp short while_limit
83
84 quit_limit:
85
86    fldcw word [orig_cntl_wd] ; restore control word
87    pop esi ; restore register variables
88    pop ebx
89
90 leave
91 ret

---------------------------------------- prime2.asm ----------------------------------------
global _dmax

segment .text
; function _dmax
; returns the larger of its two double arguments
; C prototype
; double dmax( double d1, double d2 )
; Parameters:
; d1 - first double
; d2 - second double
; Return value:
; larger of d1 and d2 (in ST0)
%define d1 ebp+8
%define d2 ebp+16
_dmax:
    enter 0, 0
    fld qword [d2]
    fld qword [d1] ; ST0 = d1, ST1 = d2
    fcomip st1 ; ST0 = d2
    jna short d2_bigger
    fcomp st0 ; pop d2 from stack
    fld qword [d1] ; ST0 = d1
    jmp short exit
    d2_bigger: ; if d2 is max, nothing to do
    exit:
    leave
    ret

Figure 6.7: FCOMIP example
6.3. THE NUMERIC COPROCESSOR

segment .data
  x    dq  2.75 ; converted to double format
  five dw  5

segment .text
  fild  dword [five] ; ST0 = 5
  fld qword [x] ; ST0 = 2.75, ST1 = 5
  fscale ; ST0 = 2.75 * 32, ST1 = 5

Figure 6.8: FSERVICE example
Chapter 7

Structures and C++

7.1 Structures

7.1.1 Introduction

Structures are used in C to group together related data into a composite variable. This technique has several advantages:

1. It clarifies the code by showing that the data defined in the structure are intimately related.

2. It simplifies passing the data to functions. Instead of passing multiple variables separately, they can be passed as a single unit.

3. It increases the locality\(^1\) of the code.

From the assembly standpoint, a structure can be considered as an array with elements of *varying* size. The elements of real arrays are always the same size and type. This property is what allows one to calculate the address of any element by knowing the starting address of the array, the size of the elements and the desired element’s index.

A structure’s elements do not have to be the same size (and usually are not). Because of this each element of a structure must be explicitly specified and is given a *tag* (or name) instead of a numerical index.

In assembly, the element of a structure will be accessed in a similar way as an element of an array. To access an element, one must know the starting address of the structure and the *relative offset* of that element from the beginning of the structure. However, unlike an array where this offset can be calculated by the index of the element, the element of a structure is assigned an offset by the compiler.

\(^1\)See the virtual memory management section of any Operating System text book for discussion of this term.
For example, consider the following structure:

```
struct S {
    short int x; /* 2-byte integer */
    int y; /* 4-byte integer */
    double z; /* 8-byte float */
};
```

Figure 7.1 shows how a variable of type S might look in the computer’s memory. The ANSI C standard states that the elements of a structure are arranged in the memory in the same order as they are defined in the `struct` definition. It also states that the first element is at the very beginning of the structure (i.e., offset zero). It also defines another useful macro in the `stddef.h` header file named `offsetof()`. This macro computes and returns the offset of any element of a structure. The macro takes two parameters, the first is the name of the type of the structure, the second is the name of the element to find the offset of. Thus, the result of `offsetof(S, y)` would be 2 from Figure 7.1.
7.1. **STRUCTURES**

```c
struct S {
    short int x; /* 2-byte integer */
    int y;      /* 4-byte integer */
    double z;  /* 8-byte float */
} __attribute__((packed));
```

Figure 7.3: Packed struct using *gcc*

### 7.1.2 Memory alignment

If one uses the `offsetof` macro to find the offset of `y` using the *gcc* compiler, they will find that it returns 4, not 2! Why? Because *gcc* (and many other compilers) align variables on double word boundaries by default. In 32-bit protected mode, the CPU reads memory faster if the data starts at a double word boundary. Figure 7.2 shows how the `S` structure really looks using *gcc*. The compiler inserts two unused bytes into the structure to align `y` (and `z`) on a double word boundary. This shows why it is a good idea to use `offsetof` to compute the offsets instead of calculating them oneself when using structures defined in C.

Of course, if the structure is only used in assembly, the programmer can determine the offsets himself. However, if one is interfacing C and assembly, it is very important that both the assembly code and the C code agree on the offsets of the elements of the structure! One complication is that different C compilers may give different offsets to the elements. For example, as we have seen, the *gcc* compiler creates an `S` structure that looks like Figure 7.2; however, Borland’s compiler would create a structure that looks like Figure 7.1. C compilers provide ways to specify the alignment used for data. However, the ANSI C standard does not specify how this will be done and thus, different compilers do it differently.

The *gcc* compiler has a flexible and complicated method of specifying the alignment. The compiler allows one to specify the alignment of any type using a special syntax. For example, the following line:

```c
typedef short int unaligned_int __attribute__((aligned (1)));
```

defines a new type named `unaligned_int` that is aligned on byte boundaries. (Yes, all the parenthesis after `__attribute__` are required!) The 1 in the `aligned` parameter can be replaced with other powers of two to specify other alignments. (2 for word alignment, 4 for double word alignment, etc.) If the `y` element of the structure was changed to be an `unaligned_int` type, *gcc* would put `y` at offset 2. However, `z` would still be at offset 8 since doubles are also double word aligned by default. The definition of `z`’s type would have to be changed as well for it to put at offset 6.
The `gcc` compiler also allows one to pack a structure. This tells the compiler to use the minimum possible space for the structure. Figure 7.3 shows how `S` could be rewritten this way. This form of `S` would use the minimum bytes possible, 14 bytes.

Microsoft’s and Borland’s compilers both support the same method of specifying alignment using a `#pragma` directive.

```c
#pragma pack(1)
```

The directive above tells the compiler to pack elements of structures on byte boundaries (i.e., with no extra padding). The one can be replaced with two, four, eight or sixteen to specify alignment on word, double word, quad word and paragraph boundaries, respectively. The directive stays in effect until overridden by another directive. This can cause problems since these directives are often used in header files. If the header file is included before other header files with structures, these structures may be laid out differently than they would by default. This can lead to a very hard to find error. Different modules of a program might lay out the elements of the structures in different places!

There is a way to avoid this problem. Microsoft and Borland support a way to save the current alignment state and restore it later. Figure 7.4 shows how this would be done.

### 7.1.3 Bit Fields

Bit fields allow one to specify members of a struct that only use a specified number of bits. The size of bits does not have to be a multiple of eight. A bit field member is defined like an `unsigned int` or `int` member with a colon and bit size appended to it. Figure 7.5 shows an example. This defines a 32-bit variable that is decomposed in the following parts:
7.1. **STRUCTURES**

```c
struct S {
    unsigned f1 : 3;  /* 3-bit field */
    unsigned f2 : 10; /* 10-bit field */
    unsigned f3 : 11; /* 11-bit field */
    unsigned f4 : 8;  /* 8-bit field */
};
```

Figure 7.5: Bit Field Example

<table>
<thead>
<tr>
<th>Byte \ Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Operation Code (08h)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Logical Unit #</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msb of LBA</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>middle of Logical Block Address</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lsb of Logical Block Address</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Transfer Length</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Control</td>
</tr>
</tbody>
</table>

Figure 7.6: SCSI Read Command Format

<table>
<thead>
<tr>
<th>8 bits</th>
<th>11 bits</th>
<th>10 bits</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>f4</td>
<td>f3</td>
<td>f2</td>
<td>f1</td>
</tr>
</tbody>
</table>

The first bitfield is assigned to the least significant bits of its double word.

However, the format is not so simple if one looks at how the bits are actually stored in memory. The difficulty occurs when bitfields span byte boundaries. Because the bytes on a little endian processor will be reversed in memory. For example, the S struct bitfields will look like this in memory:

<table>
<thead>
<tr>
<th>5 bits</th>
<th>3 bits</th>
<th>3 bits</th>
<th>5 bits</th>
<th>8 bits</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2l</td>
<td>f1</td>
<td>f3l</td>
<td>f2m</td>
<td>f3m</td>
<td>f4</td>
</tr>
</tbody>
</table>

The $f2l$ label refers to the last five bits (i.e., the five least significant bits) of the $f2$ bit field. The $f2m$ label refers to the five most significant bits of $f2$. The double vertical lines show the byte boundaries. If one reverses all the bytes, the pieces of the $f2$ and $f3$ fields will be reunited in the correct place.

The physical memory layout is not usually important unless the data is being transferred in or out of the program (which is actually quite common with bit fields). It is common for hardware devices interfaces to use odd number of bits that bitfields could be useful to represent.

---

2Actually, the ANSI/ISO C standard gives the compiler some flexibility in exactly how the bits are laid out. However, common C compilers (gcc, Microsoft and Borland) will lay the fields out like this.
One example is SCSI[3]. A direct read command for a SCSI device is specified by sending a six byte message to the device in the format specified in Figure 7.6. The difficulty representing this using bitfields is the logical block address which spans 3 different bytes of the command. From Figure 7.6, one sees that the data is stored in big endian format. Figure 7.7 shows a definition that attempts to work with all compilers. The first two lines define a macro that is true if the code is compiled with the Microsoft or Borland compilers. The potentially confusing parts are lines 11 to 14. First one might wonder why the lba_mid and lba_lsb fields are defined separately and not as a single 16-bit field? The reason is that the data is in big endian order. A 16-bit field would be stored in little endian order by the compiler. Next, the lba_msb and logical_unit fields appear to be reversed; however,

[3] Small Computer Systems Interface, an industry standard for hard disks, etc.
7.1. STRUCTURES

<table>
<thead>
<tr>
<th>8 bits</th>
<th>8 bits</th>
<th>8 bits</th>
<th>8 bits</th>
<th>3 bits</th>
<th>5 bits</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>transfer_length</td>
<td>lba_lsb</td>
<td>lba_mid</td>
<td>logical_unit</td>
<td>lba_msb</td>
<td>opcode</td>
</tr>
</tbody>
</table>

Figure 7.8: Mapping of SCSI read cmd fields

```c
struct SCSI_read_cmd {
    unsigned char opcode;
    unsigned char lba_msb : 5;
    unsigned char logical_unit : 3;
    unsigned char lba_mid; /* middle bits */
    unsigned char lba_lsb;
    unsigned char transfer_length;
    unsigned char control;
};
```

Figure 7.9: Alternate SCSI Read Command Format Structure

this is not the case. They have to be put in this order. Figure 7.8 shows how the fields are mapped as a 48-bit entity. (The byte boundaries are again denoted by the double lines.) When this is stored in memory in little endian order, the bits are arranged in the desired format (Figure 7.6).

To complicate matters more, the definition for the SCSI_read_cmd does not quite work correctly for Microsoft C. If the sizeof(SCSI_read_cmd) expression is evaulutated, Microsoft C will return 8, not 6! This is because the Microsoft compiler uses the type of the bitfield in determining how to map the bits. Since all the bit fields are defined as unsigned types, the compiler pads two bytes at the end of the structure to make it an integral number of double words. This can be remedied by making all the fields unsigned short instead. Now, the Microsoft compiler does not need to add any pad bytes since six bytes is an integral number of two-byte words. The other compilers also work correctly with this change. Figure 7.9 shows yet another definition that works for all three compilers. It avoids all but two of the bit fields by using unsigned char.

The reader should not be discouraged if he found the previous discussion confusing. It is confusing! The author often finds it less confusing to avoid bit fields altogether and use bit operations to examine and modify the bits.

---

Mixing different types of bit fields leads to very confusing behavior! The reader is invited to experiment.
manually.

### 7.1.4 Using structures in assembly

As discussed above, accessing a structure in assembly is very much like accessing an array. For a simple example, consider how one would write an assembly routine that would zero out the y element of an S structure. Assuming the prototype of the routine would be:

```c
void zero_y(S *s_p);
```

the assembly routine would be:

```assembly
%define y_offset 4
_zero_y:
  enter 0,0
  mov eax, [ebp + 8] ; get s_p (struct pointer) from stack
  mov dword [eax + y_offset], 0
  leave
  ret
```

C allows one to pass a structure by value to a function; however, this is almost always a bad idea. When passed by value, the entire data in the structure must be copied to the stack and then retrieved by the routine. It is much more efficient to pass a pointer to a structure instead.

C also allows a structure type to be used as the return value of a function. Obviously a structure cannot be returned in the EAX register. Different compilers handle this situation differently. A common solution that compilers use is to internally rewrite the function as one that takes a structure pointer as a parameter. The pointer is used to put the return value into a structure defined outside of the routine called.

Most assemblers (including NASM) have built-in support for defining structures in your assembly code. Consult your documentation for details.

### 7.2 Assembly and C++

The C++ programming language is an extension of the C language. Many of the basic rules of interfacing C and assembly language also apply to C++. However, some rules need to be modified. Also, some of the extensions of C++ are easier to understand with a knowledge of assembly language. This section assumes a basic knowledge of C++.
7.2. ASSEMBLY AND C++

7.2.1 Overloading and Name Mangling

C++ allows different functions (and class member functions) with the same name to be defined. When more than one function share the same name, the functions are said to be overloading. If two functions are defined with the same name in C, the linker will produce an error because it will find two definitions for the same symbol in the object files it is linking. For example, consider the code in Figure 7.10. The equivalent assembly code would define two labels named \_f which will obviously be an error.

C++ uses the same linking process as C, but avoids this error by performing name mangling or modifying the symbol used to label the function. In a way, C already uses name mangling, too. It adds an underscore to the name of the C function when creating the label for the function. However, C will mangle the name of both functions in Figure 7.10 the same way and produce an error. C++ uses a more sophisticated mangling process that produces two different labels for the functions. For example, the first function in Figure 7.10 would be assigned by DJGPP the label \_f\_Fi and the second function, \_f\_Fd. This avoids any linker errors.

Unfortunately, there is no standard for how to manage names in C++ and different compilers mangle names differently. For example, Borland C++ would use the labels @f$qi and @f$qd for the two functions in Figure 7.10. However, the rules are not completely arbitrary. The mangled name encodes the signature of the function. The signature of a function is defined by the order and the type of its parameters. Notice that the function that takes a single int argument has an i at the end of its mangled name (for both DJGPP and Borland) and that the one that takes a double argument has a d at the end of its mangled name. If there was a function named f with the prototype:

```c
#include <stdio.h>

void f( int x )
{
  printf(\"%d\n\", x);
}

void f( double x )
{
  printf(\"%g\n\", x);
}
```

Figure 7.10: Two f() functions
void f ( int x, int y, double z);

DJGPP would mangle its name to be _f_Fiid and Borland to @f$qiid.

The return type of the function is not part of a function’s signature and is not encoded in its mangled name. This fact explains a rule of overloading in C++. Only functions whose signatures are unique may be overloaded. As one can see, if two functions with the same name and signature are defined in C++, they will produce the same mangled name and will create a linker error. By default, all C++ functions are name mangled, even ones that are not overloaded. When it is compiling a file, the compiler has no way of knowing whether a particular function is overloading or not, so it mangles all names. In fact, it also mangles the names of global variables by encoding the type of the variable in a similar way as function signatures. Thus, if one defines a global variable in one file as a certain type and then tries to use it in another file as the wrong type, a linker error will be produced. This characteristic of C++ is known as typesafe linking. It also exposes another type of error, inconsistent prototypes. This occurs when the definition of a function in one module does not agree with the prototype used by another module. In C, this can be a very difficult problem to debug. C does not catch this error. The program will compile and link, but will have undefined behavior as the calling code will be pushing different types on the stack than the function expects. In C++, it will produce a linker error.

When the C++ compiler is parsing a function call, it looks for a matching function by looking at the types of the arguments passed to the function. If it finds a match, it then creates a CALL to the correct function using the compiler’s name mangling rules.

Since different compilers use different name mangling rules, C++ code compiled by different compilers may not be able to be linked together. This fact is important when considering using a precompiled C++ library! If one wishes to write a function in assembly that will be used with C++ code, she must know the name mangling rules for the C++ compiler to be used (or use the technique explained below).

The astute student may question whether the code in Figure 7.10 will work as expected. Since C++ name mangles all functions, then the printf function will be mangled and the compiler will not produce a CALL to the label printf. This is a valid concern! If the prototype for printf was simply placed at the top of the file, this would happen. The prototype is:

```
int printf ( const char *, ...);
```

DJGPP would mangle this to be _printf_FPCce. (The F is for function, P

---

5The match does not have to be an exact match, the compiler will consider matches made by casting the arguments. The rules for this process are beyond the scope of this book. Consult a C++ book for details.
for pointer, \texttt{C} for \texttt{const}, \texttt{c} for \texttt{char} and \texttt{e} for ellipsis.) This would not call the regular C library’s \texttt{printf} function! Of course, there must be a way for C++ code to call C code. This is very important because there is a lot of useful old C code around. In addition to allowing one to call legacy C code, C++ also allows one to call assembly code using the normal C mangling conventions.

C++ extends the \texttt{extern} keyword to allow it to specify that the function or global variable it modifies uses the normal C conventions. In C++ terminology, the function or global variable uses \textit{C linkage}. For example, to declare \texttt{printf} to have C linkage, use the prototype:

\begin{verbatim}
extern "C" int printf ( const char *, ... );
\end{verbatim}

This instructs the compiler not to use the C++ name mangling rules on this function, but instead to use the C rules. However, by doing this, the \texttt{printf} function may not be overloaded. This provides the easiest way to interface C++ and assembly, define the function to use C linkage and then use the C calling convention.

For convenience, C++ also allows the linkage of a block of functions and global variables to be defined. The block is denoted by the usual curly braces.

\begin{verbatim}
extern "C" {
    /* C linkage global variables and function prototypes */
}
\end{verbatim}

If one examines the ANSI C header files that come with C/C++ compilers today, they will find the following near the top of each header file:

\begin{verbatim}
#ifdef __cplusplus
extern "C" {
#endif

/* C linkage global variables and function prototypes */

#ifdef __cplusplus
}
#endif
\end{verbatim}

And a similar construction near the bottom containing a closing curly brace. C++ compilers define the \_\_cplusplus macro (with two leading underscores). The snippet above encloses the entire header file within an \texttt{extern "C"} block if the header file is compiled as C++, but does nothing if compiled as C (since a C compiler would give a syntax error for \texttt{extern "C"}). This same technique can be used by any programmer to create a header file for assembly routines that can be used with either C or C++.

\section{References}

\textit{References} are another new feature of C++. They allow one to pass parameters to functions without explicitly using pointers. For example, consider the code in Figure \ref{fig:references}. Actually, reference parameters are pretty
simple, they really are just pointers. The compiler just hides this from
the programmer (just as Pascal compilers implement var parameters as
pointers). When the compiler generates assembly for the function call on
line 7, it passes the address of y. If one was writing function f in assembly,
they would act as if the prototype was:

```
void f( int *xp);
```

References are just a convenience that are especially useful for opera-
tor overloading. This is another feature of C++ that allows one to define
meanings for common operators on structure or class types. For example, a
common use is to define the plus (+) operator to concatenate string objects.
Thus, if a and b were strings, a + b would return the concatenation of the
strings a and b. C++ would actually call a function to do this (in fact, these
expression could be rewritten in function notation as operator +(a,b)).
For efficiency, one would like to pass the address of the string objects in-
stead of passing them by value. Without references, this could be done as
operator +(a, &b), but this would require one to write in operator syntax
as &a + &b. This would be very awkward and confusing. However, by using
references, one can write it as a + b, which looks very natural.

### 7.2.3 Inline functions

Inline functions are yet another feature of C++ that are meant to replace the error-prone, preprocessor-based macros that take parameters. Recall from C, that writing a macro that squares a number might look like:

```
#define square(n) (n) * (n)
```

Of course, they might want to declare the function with C linkage to avoid name mangling as discussed in Section 7.2.2.

C compilers often support this feature as an extension of ANSI C.
7.2. ASSEMBLY AND C++

```cpp
inline int inline_f ( int x )
{
    return x * x;
}

int f ( int x )
{
    return x * x;
}

int main()
{
    int y, x = 5;
    y = f(x);
    y = inline_f (x);
    return 0;
}
```

Figure 7.12: Inlining example

```cpp
#define SQR(x) ((x)*(x))
```

Because the preprocessor does not understand C and does simple substitutions, the parenthesis are required to compute the correct answer in most cases. However, even this version will not give the correct answer for SQR(x++).

Macros are used because they eliminate the overhead of making a function call for a simple function. As the chapter on subprograms demonstrated, performing a function call involves several steps. For a very simple function, the time it takes to make the function call may be more than the time to actually perform the operations in the function! Inline functions are a much more friendly way to write code that looks like a normal function, but that does not CALL a common block of code. Instead, calls to inline functions are replaced by code that performs the function. C++ allows a function to be made inline by placing the keyword inline in front of the function definition. For example, consider the functions declared in Figure 7.12. The call to function f on line 10 does a normal function call (in assembly, assuming x is at address ebp-8 and y is at ebp-4):

```assembly
push    dword [ebp-8]
call    _f
pop     ecx
mov     [ebp-4], eax
```

However, the call to function inline_f on line 11 would look like:
In this case, there are two advantages to inlining. First, the inline function is faster. No parameters are pushed on the stack, no stack frame is created and then destroyed, no branch is made. Secondly, the inline function call uses less code! This last point is true for this example, but does not hold true in all cases.

The main disadvantage of inlining is that inline code is not linked and so the code of an inline function must be available to all files that use it. The previous example assembly code shows this. The call of the non-inline function only requires knowledge of the parameters, the return value type, calling convention and the name of the label for the function. All this information is available from the prototype of the function. However, using the inline function requires knowledge of all the code of the function. This means that if any part of an inline function is changed, all source files that use the function must be recompiled. Recall that for non-inline functions, if the prototype does not change, often the files that use the function need not be recompiled. For all these reasons, the code for inline functions are usually placed in header files. This practice is contrary to the normal hard and fast rule in C that executable code statements are never placed in header files.

7.2.4 Classes

A C++ class describes a type of object. An object has both data members and function members. In other words, it’s a struct with data and functions associated with it. Consider the simple class defined in Figure 7.13. A variable of Simple type would look just like a normal C struct with a single int member. The functions are not stored in memory assigned to the structure. However, member functions are different from other functions. They are passed a hidden parameter. This parameter is a pointer to the object that the member function is acting on.

For example, consider the set data method of the Simple class of Figure 7.13. If it was written in C, it would look like a function that was explicitly passed a pointer to the object being acted on as the code in Figure 7.14 shows. The -S switch on the DJGPP compiler (and the gcc and Borland compilers as well) tells the compiler to produce an assembly file containing the equivalent assembly language for the code produced. For DJGPP and gcc the assembly file ends in an .s extension and unfortu-

---

```
mov  eax, [ebp-8]
imul eax, eax
mov  [ebp-4], eax
```
7.2. ASSEMBLY AND C++

```cpp
class Simple {
public:
    Simple();       // default constructor
    ~Simple();      // destructor
    int get_data() const; // member functions
    void set_data(int);
private:
    int data;       // member data
};

Simple::Simple()
{ data = 0; }

Simple::~Simple()
{ /* null body */ }

int Simple::get_data() const
{ return data; }

void Simple::set_data(int x)
{ data = x; }
```

Figure 7.13: A simple C++ class

Chapter 7 nately uses AT&T assembly language syntax which is quite different from NASM and MASM syntaxes\footnote{The gcc compiler system includes its own assembler called gas. The gas assembler uses AT&T syntax and thus the compiler outputs the code in the format for gas. There are several pages on the web that discuss the differences in INTEL and AT&T formats. There is also a free program named a2i (http://www.multimania.com/placr/a2i.html), that converts AT&T format to NASM format.} (Borland and MS compilers generate a file with a .asm extension using MASM syntax.) Figure 7.15 shows the output of DJGPP converted to NASM syntax and with comments added to clarify the purpose of the statements. On the very first line, note that the `set_data` method is assigned a mangled label that encodes the name of the method, the name of the class and the parameters. The name of the class is encoded because other classes might have a method named `set_data` and the two methods must be assigned different labels. The parameters are encoded so that the class can overload the `set_data` method to take other parameters just as normal C++ functions. However, just as before, different compilers will encode this information differently in the mangled label.

Next on lines 2 and 3, the familiar function prologue appears. On line 5,
void set_data( Simple * object, int x )
{
    object->data = x;
}

Figure 7.14: C Version of Simple::set_data()

push ebp
mov ebp, esp

mov eax, [ebp + 8] ; eax = pointer to object (this)

mov edx, [ebp + 12] ; edx = integer parameter
mov [eax], edx ; data is at offset 0

leave
ret

Figure 7.15: Compiler output of Simple::set_data( int )

the first parameter on the stack is stored into EAX. This is not the x parameter! Instead it is the hidden parameter that points to the object being acted on. Line 6 stores the x parameter into EDX and line 7 stores EDX into the double word that EAX points to. This is the data member of the Simple object being acted on, which being the only data in the class, is stored at offset 0 in the Simple structure.

Example

This section uses the ideas of the chapter to create a C++ class that represents an unsigned integer of arbitrary size. Since the integer can be any size, it will be stored in an array of unsigned integers (double words). It can be made any size by using dynamical allocation. The double words are stored in reverse order (i.e. the least significant double word is at index 0). Figure 7.16 shows the definition of the Big_int class. The size of a Big_int is measured by the size of the unsigned array that is used to store

---

10 As usual, nothing is hidden in the assembly code!
11 Why? Because addition operations will then always start processing at the beginning of the array and move forward.
12 See the code example source for the complete code for this example. The text will only refer to some of the code.
class Big_int {
public:
    /*
    * Parameters:
    * size      — size of integer expressed as number of
    * normal unsigned int's
    * initial_value — initial value of Big_int as a normal unsigned int
    */
    explicit Big_int ( size_t size,
        unsigned initial_value = 0);
    /*
    * Parameters:
    * size      — size of integer expressed as number of
    * normal unsigned int’s
    * initial_value — initial value of Big_int as a string holding
    * hexadecimal representation of value.
    */
    Big_int ( size_t size,
        const char * initial_value );

    Big_int ( const Big_int & big_int_to_copy );
    ~Big_int ();

    // returns size of Big_int ( in terms of unsigned int's)
    size_t size () const;

    const Big_int & operator = ( const Big_int & big_int_to_copy );
    friend Big_int operator + ( const Big_int & op1,
        const Big_int & op2 );
    friend Big_int operator - ( const Big_int & op1,
        const Big_int & op2);
    friend bool operator == ( const Big_int & op1,
        const Big_int & op2 );
    friend bool operator < ( const Big_int & op1,
        const Big_int & op2);
    friend ostream & operator << ( ostream & os,
        const Big_int & op );

private:
    size_t size_;       // size of unsigned array
    unsigned * number_;  // pointer to unsigned array holding value
};

Figure 7.16: Definition of Big_int class
// prototypes for assembly routines

extern "C" {
    int add_big_ints ( Big_int & res,
                const Big_int & op1,
                const Big_int & op2);
    int sub_big_ints ( Big_int & res,
                const Big_int & op1,
                const Big_int & op2);
}

inline Big_int operator + ( const Big_int & op1, const Big_int & op2) {
    Big_int result (op1.size ());
    int res = add_big_ints ( result, op1, op2);
    if (res == 1)
        throw Big_int::Overflow();
    if (res == 2)
        throw Big_int::Size_mismatch();
    return result;
}

inline Big_int operator - ( const Big_int & op1, const Big_int & op2) {
    Big_int result (op1.size ());
    int res = sub_big_ints ( result, op1, op2);
    if (res == 1)
        throw Big_int::Overflow();
    if (res == 2)
        throw Big_int::Size_mismatch();
    return result;
}

Figure 7.17: Big_int Class Arithmetic Code
its data. The `size_` data member of the class is assigned offset zero and the `number_` member is assigned offset 4.

To simplify these example, only object instances with the same size arrays can be added to or subtracted from each other.

The class has three constructors: the first (line 9) initializes the class instance by using a normal unsigned integer; the second (line 18) initializes the instance by using a string that contains a hexadecimal value. The third constructor (line 21) is the `copy constructor`.

This discussion focuses on how the addition and subtraction operators work since this is where the assembly language is used. Figure 7.17 shows the relevant parts of the header file for these operators. They show how the operators are set up to call the assembly routines. Since different compilers use radically different mangling rules for operator functions, inline operator functions are used to set up calls to C linkage assembly routines. This makes it relatively easy to port to different compilers and is just as fast as direct calls. This technique also eliminates the need to throw an exception from assembly!

Why is assembly used at all here? Recall that to perform multiple precision arithmetic, the carry must be moved from one dword to be added to the next significant dword. C++ (and C) do not allow the programmer to access the CPU’s carry flag. Performing the addition could only be done by having C++ independently recalculate the carry flag and conditionally add it to the next dword. It is much more efficient to write the code in assembly where the carry flag can be accessed and using the `ADC` instruction which automatically adds the carry flag in makes a lot of sense.

For brevity, only the `add_big_ints` assembly routine will be discussed here. Below is the code for this routine (from `big_math.asm`):

```assembly
segment .text
global add_big_ints, sub_big_ints
%define size_offset 0
%define number_offset 4

%define EXIT_OK 0
%define EXIT_OVERFLOW 1
%define EXIT_SIZE_MISMATCH 2

; Parameters for both add and sub routines
%define res ebp+8
%define op1 ebp+12
%define op2 ebp+16
```
add_big_ints:
    push ebp
    mov ebp, esp
    push ebx
    push esi
    push edi

    ; first set up esi to point to op1
    ; edi to point to op2
    ; ebx to point to res
    mov esi, [op1]
    mov edi, [op2]
    mov ebx, [res]

    ; make sure that all 3 Big_int’s have the same size
    mov eax, [esi + size_offset]
    cmp eax, [edi + size_offset]
    jne sizes_not_equal ; op1.size_ != op2.size_
    cmp eax, [ebx + size_offset]
    jne sizes_not_equal ; op1.size_ != res.size_

    mov ecx, eax ; ecx = size of Big_int’s

    ; now, set registers to point to their respective arrays
    ; esi = op1.number_
    ; edi = op2.number_
    ; ebx = res.number_
    mov ebx, [ebx + number_offset]
    mov esi, [esi + number_offset]
    mov edi, [edi + number_offset]

    clc ; clear carry flag
    xor edx, edx ; edx = 0

    ; addition loop
    add_loop:
    mov eax, [edi+4*edx]
    adc eax, [esi+4*edx]
    mov [ebx + 4*edx], eax
    inc edx ; does not alter carry flag
7.2. ASSEMBLY AND C++

```assembly
loop    add_loop
jc      overflow
ok_done:
xor    eax, eax      ; return value = EXIT_OK
jmp    done
overflow:
mov    eax, EXIT_OVERFLOW
jmp    done
sizes_not_equal:
mov    eax, EXIT_SIZE_MISMATCH
done:
    pop    edi
    pop    esi
    pop    ebx
    leave
    ret
```

Hopefully, most of this code should be straightforward to the reader by now. Lines 25 to 27 store pointers to the **Big int** objects passed to the function into registers. Remember that references really are just pointers. Lines 31 to 35 check to make sure that the sizes of the three objects’s arrays are the same. (Note that the offset of **size** is added to the pointer to access the data member.) Lines 44 to 46 adjust the registers to point to the array used by the respective objects instead of the objects themselves. (Again, the offset of the **number** member is added to the object pointer.)

The loop in lines 52 to 57 adds the integers stored in the arrays together by adding the least significant dword first, then the next least significant dwords, etc. The addition must be done in this sequence for extended precision arithmetic (see Section 2.1.5). Line 59 checks for overflow, on overflow the carry flag will be set by the last addition of the most significant dword. Since the dwords in the array are stored in little endian order, the loop starts at the beginning of the array and moves forward toward the end.

Figure 7.18 shows a short example using the **Big int** class. Note that **Big int** constants must be declared explicitly as on line 16. This is necessary for two reasons. First, there is no conversion constructor that will convert an unsigned int to a **Big int**. Secondly, only **Big int**’s of the same size can be added. This makes conversion problematic since it would be difficult to know what size to convert to. A more sophisticated implementation of the class would allow any size to be added to any other size. The author did not wish to over complicate this example by implementing this here. (However, the reader is encouraged to do this.)
#include "big_int.hpp"
#include <iostream>
using namespace std;

int main()
{
    try {
        Big_int b(5,"8000000000000a00b");
        Big_int a(5,"8000000000010230");
        Big_int c = a + b;
        cout << a << " + " << b << " = " << c << endl;
        for (int i=0; i < 2; i++) {
            c = c + a;
            cout << "c = " << c << endl;
        }
        cout << "c−1 = " << c − Big_int(5,1) << endl;
        Big_int d(5, "12345678");
        cout << "d = " << d << endl;
        cout << "c == d " << (c == d) << endl;
        cout << "c > d " << (c > d) << endl;
    }
    catch( const char * str ) {
        cerr << "Caught: " << str << endl;
    }
    catch( Big_int :: Overflow ) {
        cerr << "Overflow" << endl;
    }
    catch( Big_int :: Size_mismatch ) {
        cerr << "Size mismatch" << endl;
    }
    return 0;
}

Figure 7.18: Simple Use of Big_int
#include <cstddef>
#include <iostream>
using namespace std;

class A {
public:
    void __cdecl m() { cout << "A::m()" << endl; }
    int ad;
};

class B : public A {
public:
    void __cdecl m() { cout << "B::m()" << endl; }
    int bd;
};

void f(A * p)
{
    p->ad = 5;
    p->m();
}

int main()
{
    A a;
    B b;
    cout << "Size of a: " << sizeof(a) << " Offset of ad: " << offsetof(A, ad) << endl;
    cout << "Size of b: " << sizeof(b) << " Offset of ad: " << offsetof(B, ad) << " Offset of bd: " << offsetof(B, bd) << endl;
    f(&a);
    f(&b);
    return 0;
}

Figure 7.19: Simple Inheritance
7.2.5 Inheritance and Polymorphism

Inheritance allows one class to inherit the data and methods of another. For example, consider the code in Figure 7.19. It shows two classes, A and B, where class B inherits from A. The output of the program is:

```
Size of a: 4 Offset of ad: 0
Size of b: 8 Offset of ad: 0 Offset of bd: 4
A::m()
A::m()
```

Notice that the ad data members of both classes (B inherits it from A) are at the same offset. This is important since the f function may be passed a pointer to either an A object or any object of a type derived (i.e. inherited from) A. Figure 7.20 shows the (edited) asm code for the function (generated by gcc).

Note that in the output that A’s m method was called for both the a and b objects. From the assembly, one can see that the call to A::m() is hard-coded into the function. For true object-oriented programming, the method called should depend on what type of object is passed to the function. This is known as polymorphism. C++ turns this feature off by default. One uses the virtual keyword to enable it. Figure 7.21 shows how the two classes would be changed. None of the other code needs to be changed. Polymorphism can be implemented many ways. Unfortunately, gcc’s implementation is in transition at the time of this writing and is becoming significantly more complicated than its initial implementation. In the interest of simplifying this discussion, the author will only cover the implementation of polymorphism which the Windows based Microsoft and Borland compilers use. This
implementation has not changed in many years and probably will not change in the foreseeable future.

With these changes, the output of the program changes:

Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()  B::m()

Now the second call to \texttt{f} calls the \texttt{B::m()} method because it is passed a \texttt{B} object. This is not the only change however. The size of an \texttt{A} is now 8 (and \texttt{B} is 12). Also, the offset of \texttt{ad} is 4, not 0. What is at offset 0? The answer to these questions are related to how polymorphism is implemented.

A C++ class that has any virtual methods is given an extra hidden field that is a pointer to an array of method pointers\footnote{For classes without virtual methods C++ compilers always make the class compatible with a normal C struct with the same data members.}. This table is often called the \textit{vtable}. For the \texttt{A} and \texttt{B} classes this pointer is stored at offset 0. The Windows compilers always put this pointer at the beginning of the class at the top of the inheritance tree. Looking at the assembly code (Figure 7.22) generated for function \texttt{f} (from Figure 7.19) for the virtual method version of the program, one can see that the call to method \texttt{m} is not to a label. Line 9 finds the address of the vtable from the object. The address of the object is pushed on the stack in line 11. Line 12 calls the virtual method by branching to the first address in the vtable\footnote{Of course, this value is already in the \texttt{ECX} register. It was put there in line 8 and line 10 could be removed and the next line changed to push \texttt{ECX}. The code is not very efficient because it was generated without compiler optimizations turned on.}. This call does not use a label, it branches to the code address pointed to by \texttt{EDX}. This type of call is an
example of *late binding*. Late binding delays the decision of which method to call until the code is running. This allows the code to call the appropriate method for the object. The normal case (Figure 7.20) hard-codes a call to a certain method and is called *early binding* (since here the method is bound early, at compile time).

The attentive reader will be wondering why the class methods in Figure 7.21 are explicitly declared to use the C calling convention by using the _cdecl keyword. By default, Microsoft uses a different calling convention for C++ class methods than the standard C convention. It passes the pointer to the object acted on by the method in the ECX register instead of using the stack. The stack is still used for the other explicit parameters of the method. The _cdecl modifier tells it to use the standard C calling convention. Borland C++ uses the C calling convention by default.

Next let’s look at a slightly more complicated example (Figure 7.23). In it, the classes A and B each have two methods: m1 and m2. Remember that since class B does not define its own m2 method, it inherits the A class’s method. Figure 7.24 shows how the b object appears in memory. Figure 7.25 shows the output of the program. First, look at the address of the vtable for each object. The two B objects’s addresses are the same and thus, they share the same vtable. A vtable is a property of the class not an object (like a static data member). Next, look at the addresses in the vtables. From looking at assembly output, one can determine that the m1 method pointer
class A {
public:
    virtual void _cdecl m1() { cout << "A::m1()" << endl; }
    virtual void _cdecl m2() { cout << "A::m2()" << endl; }
    int ad;
};

class B : public A {  // B inherits A's m2()
public:
    virtual void _cdecl m1() { cout << "B::m1()" << endl; }
    int bd;
};
/*
prints the vtable of given object */
void print_vtable ( A * pa )
{
    // p sees pa as an array of dwords
    unsigned * p = reinterpret_cast<unsigned *>(pa);
    // vt sees vtable as an array of pointers
    void ** vt = reinterpret_cast<void **>(p[0]);
    cout << hex << "vtable address = " << vt << endl;
    for ( int i=0; i < 2; i++ )
        cout << "dword " << i << " : " << vt[i] << endl;

    // call virtual functions in EXTREMELY non-portable way!
    void (*m1func_pointer)(A *);  // function pointer variable
    m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
    m1func_pointer(pa);  // call method m1 via function pointer

    void (*m2func_pointer)(A *);  // function pointer variable
    m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
    m2func_pointer(pa);  // call method m2 via function pointer
}

int main()
{
    A a;  B b1;  B b2;
    cout << "a: " << endl; print_vtable (&a);
    cout << "b1: " << endl; print_vtable (&b);
    cout << "b2: " << endl; print_vtable (&b2);
    return 0;
}

Figure 7.23: More complicated example
Figure 7.24: Internal representation of `b1`

Figure 7.25: Output of program in Figure 7.23
is at offset 0 (or dword 0) and \texttt{m2} is at offset 4 (dword 1). The \texttt{m2} method pointers are the same for the \texttt{A} and \texttt{B} class vtables because class \texttt{B} inherits the \texttt{m2} method from the \texttt{A} class.

Lines 25 to 32 show how one could call a virtual function by reading its address out of the vtable for the object. The method address is stored into a C-type function pointer with an explicit \texttt{this} pointer. From the output in Figure 7.25, one can see that it does work. However, please do not write code like this! This is only used to illustrate how the virtual methods use the vtable.

There are some practical lessons to learn from this. One important fact is that one would have to be very careful when reading and writing class variables to a binary file. One can not just use a binary read or write on the entire object as this would read or write out the vtable pointer to the file! This is a pointer to where the vtable resides in the program’s memory and will vary from program to program. This same problem can occur in C with structs, but in C, structs only have pointers in them if the programmer explicitly puts them in. There are no obvious pointers defined in either the \texttt{A} or \texttt{B} classes.

Again, it is important to realize that different compilers implement virtual methods differently. In Windows, COM (Component Object Model) class objects use vtables to implement COM interfaces. Only compilers that implement virtual method vtables as Microsoft does can create COM classes. This is why Borland uses the same implementation as Microsoft and one of the reasons why \texttt{gcc} can not be used to create COM classes.

The code for the virtual method looks exactly like a non-virtual one. Only the code that calls it is different. If the compiler can be absolutely sure of what virtual method will be called, it can ignore the vtable and call the method directly (\textit{e.g.}, use early binding).

7.2.6 Other C++ features

The workings of other C++ features (\textit{e.g.}, RunTime Type Information, exception handling and multiple inheritance) are beyond the scope of this text. If the reader wishes to go further, a good starting point is \textit{The Annotated C++ Reference Manual} by Ellis and Stroustrup and \textit{The Design and Evolution of C++} by Stroustrup.

\textsuperscript{15}Remember this code only works with the MS and Borland compilers, not \texttt{gcc}.

\textsuperscript{16}COM classes also use the \texttt{stdcall} calling convention, not the standard C one.
Appendix A

80x86 Instructions

A.1 Non-floating Point Instructions

This section lists and describes the actions and formats of the non-floating point instructions of the Intel 80x86 CPU family.

The formats use the following abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>general register</td>
</tr>
<tr>
<td>R8</td>
<td>8-bit register</td>
</tr>
<tr>
<td>R16</td>
<td>16-bit register</td>
</tr>
<tr>
<td>R32</td>
<td>32-bit register</td>
</tr>
<tr>
<td>SR</td>
<td>segment register</td>
</tr>
<tr>
<td>M</td>
<td>memory</td>
</tr>
<tr>
<td>M8</td>
<td>byte</td>
</tr>
<tr>
<td>M16</td>
<td>word</td>
</tr>
<tr>
<td>M32</td>
<td>double word</td>
</tr>
<tr>
<td>I</td>
<td>immediate value</td>
</tr>
</tbody>
</table>

These can be combined for the multiple operand instructions. For example, the format $R, R$ means that the instruction takes two register operands. Many of the two operand instructions allow the same operands. The abbreviation $O2$ is used to represent these operands: $R, R, R, R, M, R, I, M, R, M, I$. If a 8-bit register or memory can be used for an operand, the abbreviation, $R/M8$ is used.

The table also shows how various bits of the FLAGS register are affected by each instruction. If the column is blank, the corresponding bit is not affected at all. If the bit is always changed to a particular value, a 1 or 0 is shown in the column. If the bit is changed to a value that depends on the operands of the instruction, a $C$ is placed in the column. Finally, if the bit is modified in some undefined way a $?$ appears in the column. Because the
only instructions that change the direction flag are `CLD` and `STD`, it is not listed under the `FLAGS` columns.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Formats</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>O2</td>
<td>O</td>
</tr>
<tr>
<td>ADD</td>
<td>Add Integers</td>
<td>O2</td>
<td>C</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td>O2</td>
<td>C</td>
</tr>
<tr>
<td>BSWAP</td>
<td>Byte Swap</td>
<td>R32</td>
<td>C</td>
</tr>
<tr>
<td>CALL</td>
<td>Call Routine</td>
<td>R M I</td>
<td>C</td>
</tr>
<tr>
<td>CBW</td>
<td>Convert Byte to Word</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CDQ</td>
<td>Convert Dword to Qword</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear Carry</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear Direction Flag</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CMC</td>
<td>Complement Carry</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare Integers</td>
<td>O2</td>
<td>C</td>
</tr>
<tr>
<td>CMPSB</td>
<td>Compare Bytes</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CMPSW</td>
<td>Compare Words</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CMPSD</td>
<td>Compare Dwords</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CWD</td>
<td>Convert Word to Dword into DX:AX</td>
<td>R16,R/M16,</td>
<td>C</td>
</tr>
<tr>
<td>CWDE</td>
<td>Convert Word to Dword into EAX</td>
<td>R16,R/M16,</td>
<td>C</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement Integer</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>DIV</td>
<td>Unsigned Divide</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>ENTER</td>
<td>Make stack frame</td>
<td>I,0</td>
<td>C</td>
</tr>
<tr>
<td>IDIV</td>
<td>Signed Divide</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>IMUL</td>
<td>Signed Multiply</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>INC</td>
<td>Increment Integer</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>INT</td>
<td>Generate Interrupt</td>
<td>I</td>
<td>C</td>
</tr>
<tr>
<td>JA</td>
<td>Jump Above</td>
<td>I</td>
<td>C</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump Above or Equal</td>
<td>I</td>
<td>C</td>
</tr>
<tr>
<td>JB</td>
<td>Jump Below</td>
<td>I</td>
<td>C</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump Below or Equal</td>
<td>I</td>
<td>C</td>
</tr>
<tr>
<td>JC</td>
<td>Jump Carry</td>
<td>I</td>
<td>C</td>
</tr>
</tbody>
</table>
### A.1. NON-FLOATING POINT INSTRUCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Formats</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCXZ</td>
<td>Jump if CX = 0</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JE</td>
<td>Jump Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>Jump Greater</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JGE</td>
<td>Jump Greater or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JL</td>
<td>Jump Less</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JLE</td>
<td>Jump Less or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>Unconditional Jump</td>
<td>R M I</td>
<td></td>
</tr>
<tr>
<td>JNA</td>
<td>Jump Not Above</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump Not Above or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNB</td>
<td>Jump Not Below</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump Not Below or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>Jump No Carry</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNE</td>
<td>Jump Not Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNG</td>
<td>Jump Not Greater</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump Not Greater or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNL</td>
<td>Jump Not Less</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump Not Less or Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNO</td>
<td>Jump No Overflow</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNS</td>
<td>Jump No Sign</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump Not Zero</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JO</td>
<td>Jump Overflow</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JPE</td>
<td>Jump Parity Even</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JPO</td>
<td>Jump Parity Odd</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>Jump Sign</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>JZ</td>
<td>Jump Zero</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>LAHF</td>
<td>Load FLAGS into AH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEA</td>
<td>Load Effective Address</td>
<td>R32,M</td>
<td></td>
</tr>
<tr>
<td>LEAVE</td>
<td>Leave Stack Frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LODSB</td>
<td>Load Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LODSW</td>
<td>Load Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LODSD</td>
<td>Load Dword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>Loop</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>LOOPE/LOOPZ</td>
<td>Loop If Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>LOOPNE/LOOPNZ</td>
<td>Loop If Not Equal</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Formats</td>
<td>Flags</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------</td>
<td>-------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Data</td>
<td>O2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR,R/M16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M16,SR</td>
<td></td>
</tr>
<tr>
<td>MOVSB</td>
<td>Move Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSW</td>
<td>Move Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSD</td>
<td>Move Dword</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSX</td>
<td>Move Signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R16,R/M8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R32,R/M8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R32,R/M16</td>
<td></td>
</tr>
<tr>
<td>MOVZX</td>
<td>Move Unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R16,R/M8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R32,R/M8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R32,R/M16</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Unsigned Multiply</td>
<td>R M</td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
<td>R M</td>
<td>C</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td></td>
<td>C C C C C C</td>
</tr>
<tr>
<td>NOT</td>
<td>1’s Complement</td>
<td>R M</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>Bitwise OR</td>
<td>O2</td>
<td>0</td>
</tr>
<tr>
<td>POP</td>
<td>Pop From Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPA</td>
<td>Pop All</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>Pop FLAGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>Push to Stack</td>
<td>R/M16</td>
<td></td>
</tr>
<tr>
<td>PUSHA</td>
<td>Push All</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push FLAGS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td>Rotate Left with Carry</td>
<td>R/M,I</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M,CL</td>
<td></td>
</tr>
<tr>
<td>RCR</td>
<td>Rotate Right with Carry</td>
<td>R/M,I</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M,CL</td>
<td></td>
</tr>
<tr>
<td>REP</td>
<td>Repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>Repeat If Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>Repeat If Not Equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate Left</td>
<td>R/M,I</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M,CL</td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate Right</td>
<td>R/M,I</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M,CL</td>
<td></td>
</tr>
<tr>
<td>SAHF</td>
<td>Copies AH into FLAGS</td>
<td></td>
<td>C C C C C C</td>
</tr>
</tbody>
</table>
## A.1. NON-FLOATING POINT INSTRUCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Formats</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAL</td>
<td>Shifts to Left</td>
<td>R/M,I</td>
<td>O S Z A P C</td>
</tr>
<tr>
<td>SBB</td>
<td>Subtract with Borrow</td>
<td>O2</td>
<td>C C C C C C</td>
</tr>
<tr>
<td>SCASB</td>
<td>Scan for Byte</td>
<td>R/M, CL</td>
<td>C C C C C C</td>
</tr>
<tr>
<td>SCASW</td>
<td>Scan for Word</td>
<td></td>
<td>C C C C C C</td>
</tr>
<tr>
<td>SCASD</td>
<td>Scan for Dword</td>
<td></td>
<td>C C C C C C</td>
</tr>
<tr>
<td>SETA</td>
<td>Set Above</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETAE</td>
<td>Set Above or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETB</td>
<td>Set Below</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETBE</td>
<td>Set Below or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETC</td>
<td>Set Carry</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETE</td>
<td>Set Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETG</td>
<td>Set Greater</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETGE</td>
<td>Set Greater or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETL</td>
<td>Set Less</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETLE</td>
<td>Set Less or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNA</td>
<td>Set Not Above</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNAE</td>
<td>Set Not Above or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNB</td>
<td>Set Not Below</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNBE</td>
<td>Set Not Below or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNC</td>
<td>Set No Carry</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNE</td>
<td>Set Not Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNG</td>
<td>Set Not Greater</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNGE</td>
<td>Set Not Greater or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNL</td>
<td>Set Not Less</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNLE</td>
<td>Set Not Less or Equal</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNO</td>
<td>Set No Overflow</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNS</td>
<td>Set No Sign</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETNZ</td>
<td>Set Not Zero</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETO</td>
<td>Set Overflow</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETPE</td>
<td>Set Parity Even</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETPO</td>
<td>Set Parity Odd</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td>Set Sign</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SETZ</td>
<td>Set Zero</td>
<td>R/M8</td>
<td></td>
</tr>
<tr>
<td>SAR</td>
<td>Arithmetic Shift to Right</td>
<td>R/M,I, CL</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Formats</td>
<td>O</td>
</tr>
<tr>
<td>------</td>
<td>------------------------</td>
<td>------------------</td>
<td>---</td>
</tr>
<tr>
<td>SHR</td>
<td>Logical Shift to Right</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M, CL</td>
<td></td>
</tr>
<tr>
<td>SHL</td>
<td>Logical Shift to Left</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M, CL</td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td>Set Carry</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td>Set Direction Flag</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>STOSB</td>
<td>Store Bbyte</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>STOSW</td>
<td>Store Word</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>STOSD</td>
<td>Store Dword</td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>O2</td>
<td>C</td>
</tr>
<tr>
<td>TEST</td>
<td>Logical Compare</td>
<td>R/M, R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R/M, I</td>
<td></td>
</tr>
<tr>
<td>XCHG</td>
<td>Exchange</td>
<td>R/M, R</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R, R/M</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>Bitwise XOR</td>
<td>O2</td>
<td></td>
</tr>
</tbody>
</table>
A.2 Floating Point Instructions

In this section, many of the 80x86 math coprocessor instructions are described. The description section briefly describes the operation of the instruction. To save space, information about whether the instruction pops the stack is not given in the description.

The format column shows what type of operands can be used with each instruction. The following abbreviations are used:

<table>
<thead>
<tr>
<th>STn</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STn</td>
<td>A coprocessor register</td>
</tr>
<tr>
<td>F</td>
<td>Single precision number in memory</td>
</tr>
<tr>
<td>D</td>
<td>Double precision number in memory</td>
</tr>
<tr>
<td>E</td>
<td>Extended precision number in memory</td>
</tr>
<tr>
<td>I16</td>
<td>Integer word in memory</td>
</tr>
<tr>
<td>I32</td>
<td>Integer double word in memory</td>
</tr>
<tr>
<td>I64</td>
<td>Integer quad word in memory</td>
</tr>
</tbody>
</table>

Instructions requiring a Pentium Pro or better are marked with an asterisk (*).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS</td>
<td>ST0 =</td>
<td>ST0</td>
</tr>
<tr>
<td>FADD src</td>
<td>ST0 += src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FADD dest, ST0</td>
<td>dest += ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FADD dest[,ST0]</td>
<td>dest += ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FCHS</td>
<td>ST0 = −ST0</td>
<td></td>
</tr>
<tr>
<td>FCOM src</td>
<td>Compare ST0 and src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FCOM src</td>
<td>Compare ST0 and src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FCOMPP src</td>
<td>Compares ST0 and ST1</td>
<td></td>
</tr>
<tr>
<td>FCOMIP src</td>
<td>Compares into FLAGS</td>
<td>STn</td>
</tr>
<tr>
<td>FDIV src</td>
<td>ST0 /= src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FDIV dest, ST0</td>
<td>dest /= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FDIVP dest[,ST0]</td>
<td>dest /= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FDIVR src</td>
<td>ST0 = src/ST0</td>
<td>STn F D</td>
</tr>
<tr>
<td>FDIVR dest, ST0</td>
<td>dest = ST0/dest</td>
<td>STn</td>
</tr>
<tr>
<td>FDIVRP dest[,ST0]</td>
<td>dest = ST0/dest</td>
<td>STn</td>
</tr>
<tr>
<td>FFREE dest</td>
<td>Marks as empty</td>
<td>STn</td>
</tr>
<tr>
<td>FADD src</td>
<td>ST0 += src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FICOM src</td>
<td>Compare ST0 and src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FICOMPP src</td>
<td>Compare ST0 and src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FIDIV src</td>
<td>ST0 /= src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FIDIVR src</td>
<td>ST0 = src/ST0</td>
<td>I16 I32</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Format</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>FILD src</td>
<td>Push src on Stack</td>
<td>I16 I32 I64</td>
</tr>
<tr>
<td>FIMUL src</td>
<td>ST0 *= src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FINIT</td>
<td>Initialize Coprocessor</td>
<td></td>
</tr>
<tr>
<td>FIST dest</td>
<td>Store ST0</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FISTP dest</td>
<td>Store ST0</td>
<td>I16 I32 I64</td>
</tr>
<tr>
<td>FISUB src</td>
<td>ST0 -= src</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FISUBR src</td>
<td>ST0 = src - ST0</td>
<td>I16 I32</td>
</tr>
<tr>
<td>FLD src</td>
<td>Push src on Stack</td>
<td>STn F D E</td>
</tr>
<tr>
<td>FLD1</td>
<td>Push 1.0 on Stack</td>
<td></td>
</tr>
<tr>
<td>FLDCW src</td>
<td>Load Control Word Register</td>
<td>I16</td>
</tr>
<tr>
<td>FLDPI</td>
<td>Push π on Stack</td>
<td></td>
</tr>
<tr>
<td>FLDZ</td>
<td>Push 0.0 on Stack</td>
<td></td>
</tr>
<tr>
<td>FMUL src</td>
<td>ST0 *= src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FMUL dest, ST0</td>
<td>dest *= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FMULP dest [,ST0]</td>
<td>dest *= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FRNDINT</td>
<td>Round ST0</td>
<td></td>
</tr>
<tr>
<td>FSQRT</td>
<td>ST0 = √ST0</td>
<td></td>
</tr>
<tr>
<td>FST dest</td>
<td>Store ST0</td>
<td>STn F D</td>
</tr>
<tr>
<td>FSTP dest</td>
<td>Store ST0</td>
<td>STn F D E</td>
</tr>
<tr>
<td>FSTCW dest</td>
<td>Store Control Word Register</td>
<td>I16</td>
</tr>
<tr>
<td>FSTSW dest</td>
<td>Store Status Word Register</td>
<td>I16 AX</td>
</tr>
<tr>
<td>FSUB src</td>
<td>ST0 -= src</td>
<td>STn F D</td>
</tr>
<tr>
<td>FSUB dest, ST0</td>
<td>dest -= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FSUBP dest [,ST0]</td>
<td>dest -= ST0</td>
<td>STn</td>
</tr>
<tr>
<td>FSUBR src</td>
<td>ST0 = src-ST0</td>
<td>STn F D</td>
</tr>
<tr>
<td>FSUBR dest, ST0</td>
<td>dest = ST0-dest</td>
<td>STn</td>
</tr>
<tr>
<td>FSUBP dest [,ST0]</td>
<td>dest = ST0-dest</td>
<td>STn</td>
</tr>
<tr>
<td>FTST</td>
<td>Compare ST0 with 0.0</td>
<td></td>
</tr>
<tr>
<td>FXCH dest</td>
<td>Exchange ST0 and dest</td>
<td>STn</td>
</tr>
</tbody>
</table>
Index

ADC, 37, 54
ADD, 13, 36
AND, 50
array1.asm, 99–102
arrays, 95–115
  accessing, 96–102
  defining, 95–96
    local variable, 96
    static, 95
  multidimensional, 103–106
    parameters, 105–106
    two dimensional, 103–104
assembler, 11
assembly language, 11, 12

binary, 1, 2
  addition, 2
bit operations
  AND, 50
  assembly, 52, 53
  C, 56, 57
  NOT, 51
  OR, 50
  shifts, 47, 50
    arithmetic shifts, 48
    logical shifts, 47, 48
    rotates, 49
  XOR, 51
branch prediction, 53
bss segment, 21
BSWAP, 59
BYTE, 16
byte, 4
C driver, 19

C++, 150–171
  Big.int example, 158–163
  classes, 156–171
  copy constructor, 161
  early binding, 168
  extern "C", 153
  inheritance, 166–171
  inline functions, 154–156
  late binding, 168
  member functions, see methods
  name mangling, 151–153
  polymorphism, 166–171
  references, 153–154
  typesafe linking, 152
  virtual, 166
  vtable, 167–171
CALL, 69–70
calling convention, 65, 70–76, 83
  _cdecl, 84
  __stdcall, 84
C, 21, 71, 80–84
  labels, 82
  parameters, 82
  registers, 81
  return values, 83
Pascal, 71
  register, 84
  standard call, 84
  stdcall, 72, 84, 171
CBW, 31
CDQ, 31
CLC, 31
INDEX

CLD, 106
CLK, 5
CMP, 37–38
CMPSB, 109
CMPSD, 109
CMPSW, 109
code segment, 21
COM, 171
comment, 12
compiler, 5, 11
Borland, 22, 23
DJGPP, 22, 23
gcc, 22
_attribute__, 84, 145, 148, 149
Microsoft, 22
_pragmas pack, 146, 148, 149
Watcom, 83
conditional branch, 39–41
counting bits, 60–64
method one, 60–61
method two, 61–62
CPU, 5–7
80x86, 6
CWD, 31
CWDE, 31
data segment, 21
data, 14–15
decimal, 1
directive, 13, 15
%define, 14
D.X, 14, 95
data, 14, 15
D.D, 15
D.Q, 15
equ, 13
extern, 77
global, 21, 78, 80
RESX, 14, 95
TIMES, 15, 95
DIV, 34, 48
do while loop, 43
DWORD, 16
endianess, 24–25, 57, 60
invert_endian, 59
FABS, 130
FADD, 126
FADDP, 126
FCHS, 130
FCOM, 129
FCOM1, 130
FCOMIP, 130, 140
FCOMP, 129
FCOMP, 129
FDIV, 128
FDIVP, 128
FDIVR, 128
FDIVRP, 128
FFREE, 126
FIADD, 126
FICOM, 129
FICOMP, 129
FIDIV, 128
FIDIVR, 128
FLD, 125
FLDCW, 126
FLD1, 125
floating point, 117–139
arithmetic, 122, 124
denormalized, 121
double precision, 122
hidden one, 120
IEEE, 119, 122
single precision, 120, 121
floating point coprocessor, 124, 139
addition and subtraction, 126–127
comparisons, 128–130
hardware, 124–125
loading and storing data, 125–126
multiplication and division, 128
FMUL, 128
FMULP, 128
FScale, 130, 141
FSQRT, 130
FST, 126
FSTCW, 126
FSTP, 126
FSTSW, 129
FSUB, 127
FSUBP, 127
FSUBR, 127
FSUBRP, 127
FTST, 129
FXCH, 126
gas, 157
hexadecimal, 3–4
I/O, 16–18
asm, io library, 16–18
dump, math, 18
dump, mem, 17
dump, regs, 17
dump, stack, 17
print, char, 17
print, int, 17
print, num, 17
print, string, 17
read, char, 17
read, int, 17
IDIV, 34
if statement, 42
immediate, 12
IMUL, 33, 34
INC, 13
indirect addressing, 65–66
arrays, 98–102
integer, 27–38
comparisons, 37–38
division, 34
extended precision, 36–37
multiplication, 33–34
representation, 27–33
one’s complement, 28
signed magnitude, 27
two’s complement, 28–30
sign bit, 27–30
sign extension, 30–33
signed, 27–30, 38
unsigned, 27–38
interfacing with C, 80–89
interrupt, 10
JC, 39
JE, 41
JG, 41
JGE, 41
JL, 41
JLE, 41
JNC, 39
JNE, 41
JNG, 41
JNGE, 41
JNL, 41
JNLE, 41
JNO, 39
JNP, 39
JNS, 39
JNZ, 39
JO, 39
JP, 39
JS, 39
JZ, 39
label, 14–16
LAHF, 129
lea, 83, 103
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>linking</td>
<td>23</td>
</tr>
<tr>
<td>listing file</td>
<td>23–24</td>
</tr>
<tr>
<td>locality</td>
<td>143</td>
</tr>
<tr>
<td>LODSB</td>
<td>107</td>
</tr>
<tr>
<td>LODSD</td>
<td>107</td>
</tr>
<tr>
<td>LODSW</td>
<td>107</td>
</tr>
<tr>
<td>LOOP</td>
<td>41</td>
</tr>
<tr>
<td>LOOPE</td>
<td>41</td>
</tr>
<tr>
<td>LOOPNE</td>
<td>41</td>
</tr>
<tr>
<td>LOOPNZ</td>
<td>41</td>
</tr>
<tr>
<td>LOOPZ</td>
<td>41</td>
</tr>
<tr>
<td>machine language</td>
<td>5, 11</td>
</tr>
<tr>
<td>MASM</td>
<td>12</td>
</tr>
<tr>
<td>math.asm</td>
<td>35–36</td>
</tr>
<tr>
<td>memory</td>
<td>4–5</td>
</tr>
<tr>
<td>pages</td>
<td>10</td>
</tr>
<tr>
<td>segments</td>
<td>9–10</td>
</tr>
<tr>
<td>virtual</td>
<td>9–10</td>
</tr>
<tr>
<td>memory.asm</td>
<td>111–115</td>
</tr>
<tr>
<td>memory:segments</td>
<td>9</td>
</tr>
<tr>
<td>methods</td>
<td>156</td>
</tr>
<tr>
<td>mnemonic</td>
<td>11</td>
</tr>
<tr>
<td>MOV</td>
<td>12</td>
</tr>
<tr>
<td>MOVSIB</td>
<td>108</td>
</tr>
<tr>
<td>MOVSID</td>
<td>108</td>
</tr>
<tr>
<td>MOVSIV</td>
<td>108</td>
</tr>
<tr>
<td>MOVSX</td>
<td>31</td>
</tr>
<tr>
<td>MOVZX</td>
<td>31</td>
</tr>
<tr>
<td>MUL</td>
<td>33–34</td>
</tr>
<tr>
<td>QWORD</td>
<td>16</td>
</tr>
<tr>
<td>quad.asm</td>
<td>130–133</td>
</tr>
<tr>
<td>RCL</td>
<td>49</td>
</tr>
<tr>
<td>RCR</td>
<td>49</td>
</tr>
<tr>
<td>read.asm</td>
<td>133–135</td>
</tr>
<tr>
<td>real mode</td>
<td>8–9</td>
</tr>
<tr>
<td>recursion</td>
<td>89–90</td>
</tr>
<tr>
<td>register</td>
<td>7–8</td>
</tr>
<tr>
<td>32-bit</td>
<td>8</td>
</tr>
<tr>
<td>base pointer</td>
<td>7–8</td>
</tr>
<tr>
<td>EDI</td>
<td>107</td>
</tr>
<tr>
<td>EAX</td>
<td>31</td>
</tr>
<tr>
<td>EDX</td>
<td>33–34</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>8</td>
</tr>
<tr>
<td>EIP</td>
<td>8</td>
</tr>
<tr>
<td>ESI</td>
<td>107</td>
</tr>
<tr>
<td>FLAGS</td>
<td>7–8</td>
</tr>
<tr>
<td>CF</td>
<td>38</td>
</tr>
<tr>
<td>DF</td>
<td>106</td>
</tr>
<tr>
<td>OF</td>
<td>38</td>
</tr>
<tr>
<td>PF</td>
<td>39</td>
</tr>
<tr>
<td>SF</td>
<td>38</td>
</tr>
<tr>
<td>ZF</td>
<td>38</td>
</tr>
<tr>
<td>index</td>
<td>7</td>
</tr>
<tr>
<td>IP</td>
<td>7</td>
</tr>
<tr>
<td>segment</td>
<td>7–8</td>
</tr>
<tr>
<td>stack pointer</td>
<td>7–8</td>
</tr>
<tr>
<td>REP</td>
<td>108–109</td>
</tr>
<tr>
<td>REPE</td>
<td>110</td>
</tr>
<tr>
<td>REPNE</td>
<td>110</td>
</tr>
<tr>
<td>REPNZ, <em>see</em> REPNE</td>
<td></td>
</tr>
<tr>
<td>REPZ, <em>see</em> REPE</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>69–70</td>
</tr>
<tr>
<td>ROL</td>
<td>49</td>
</tr>
<tr>
<td>ROR</td>
<td>49</td>
</tr>
<tr>
<td>SAHF</td>
<td>129</td>
</tr>
<tr>
<td>SAL</td>
<td>48</td>
</tr>
<tr>
<td>SAR</td>
<td>48</td>
</tr>
<tr>
<td>SBB</td>
<td>37</td>
</tr>
<tr>
<td>SCASB</td>
<td>109</td>
</tr>
</tbody>
</table>
SCASD, 109
SCASW, 109
SCSI, 147–149
SETxx, 54
SETG, 55
SHL, 47
SHR, 47
skeleton file, 25
speculative execution, 53
stack, 68–76
local variables, 75–76, 82–83
parameters, 70–73
startup code, 23
STD, 106
storage types
automatic, 91
global, 91
register, 93
static, 91
volatile, 93
STOSB, 107
STOSD, 107
string instructions, 106–115
structures, 143–150
alignment, 145–146
bit fields, 146–150
offsetof(), 144
SUB, 13, 36
subprogram, 66–93
calling, 69–76
reentrant, 89
subroutine, see subprogram
TASM, 12
TCP/IP, 59
TEST, 51
text segment, see code segment
two’s complement, 28–30
arithmetic, 33–37
TWORD, 16
UNICODE, 59
while loop, 43