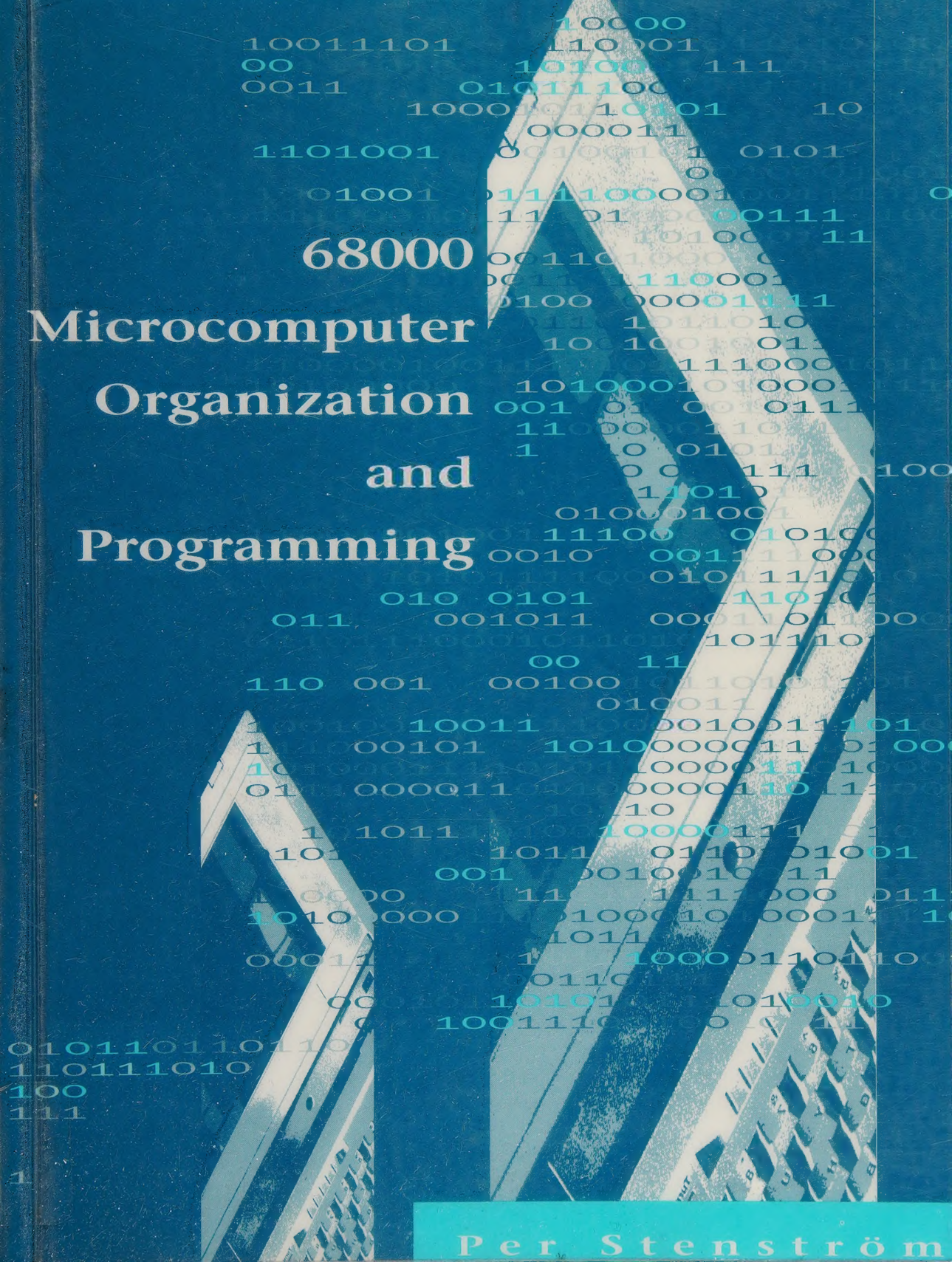
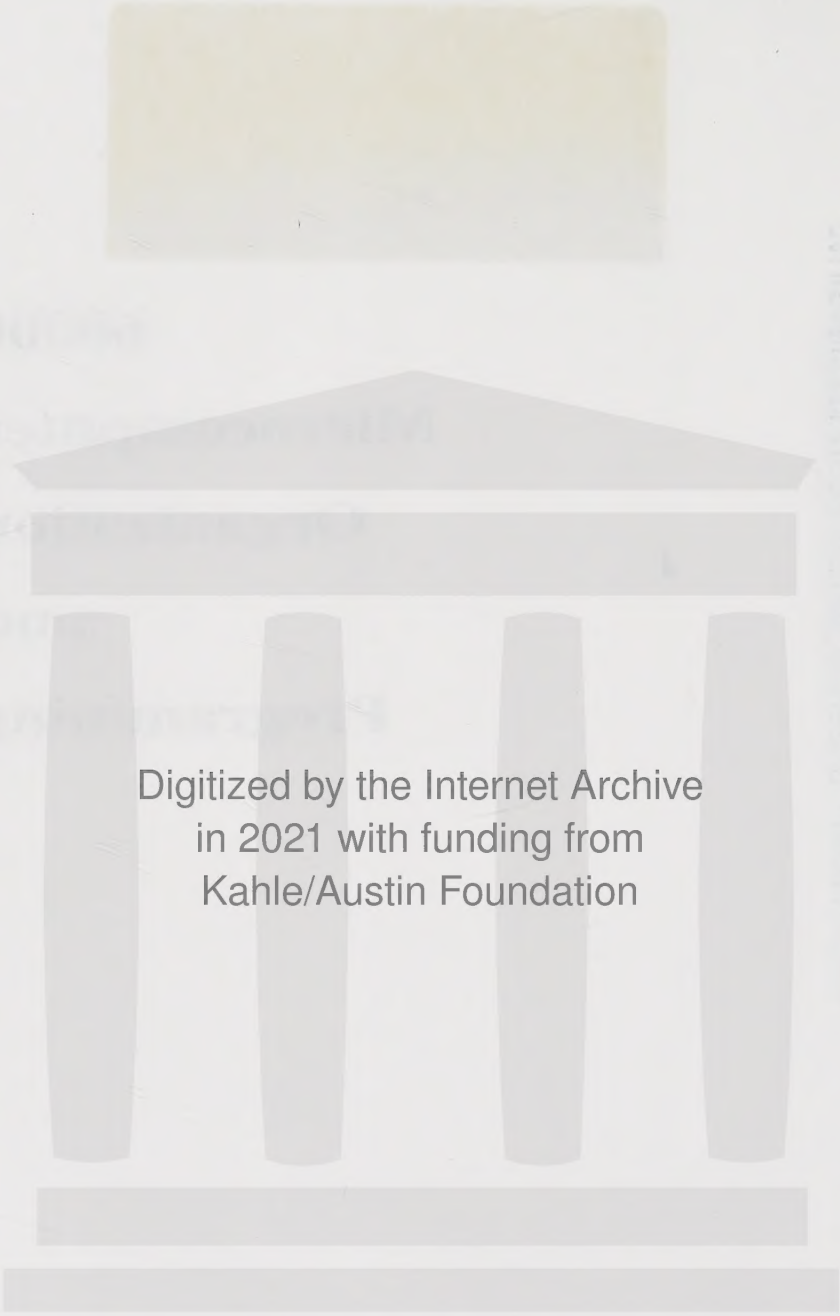
# 68000 Microcomputer Organization and Programming



Per Stenström

# 68000 Microcomputer Organization and Programming



# 68000 Microcomputer Organization and Programming

Per Stenström

Prentice Hall
New York London Toronto Sydney Tokyo Singapore

First published 1992 by
Prentice Hall International (UK) Ltd
Campus 400, Maylands Avenue
Hemel Hempstead
Hertfordshire, HP2 7EZ
A division of
Simon & Schuster International Group



Printed and bound in Great Britain by
Dotesios Ltd, Trowbridge, Wiltshire

Library of Congress Cataloging-in-Publication Data

Stenström, Per.
68000 Microcomputer organization and programming / Per Stenström.
p. cm.
Includes index.
ISBN 0–13–584855–5
1. Motorola 68000 (Microprocessor) 2. Computer organization.
3. Microcomputers—Programming. I. Title.
QA76.8.M67S73 1992
004.165—dc20 92-27338
CIP

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0–13–584855–5 (pbk)

1 2 3 4 5 96 95 94 93 92

To Carina

# Contents

# Preface

This book is an introduction to microcomputer system organization and assembly language programming and in particular for the Motorola 68000 (M68000). Besides presenting the basic concepts of microcomputer systems and instruction set models, it also presents techniques that facilitate the use of a microcomputer system as a component in system control applications.

There are two important objectives of this book. First, it provides an introduction to computer system organization by presenting the functional components of a naked computer system that is stripped of all the layers of software that it is usually clothed in. Second, it provides an introduction to assembly language programming by presenting the most important concepts of instruction set models. These objectives are met in the following way.

The important concepts of microcomputer systems are introduced step by step by using a series of successively refined models. All details that are not relevant for the time being are hidden so as to let the reader concentrate on one issue at a time. I have found that this is important in order not to drown in all the details that often are associated with complex systems such as microcomputers.

The instruction set of the M68000 is also introduced step by step, starting with a subset of all the available registers and such instructions and addressing modes that are relevant for the model at hand. The model of the M68000 is then expanded step by step to cover more instructions and more addressing modes, leading to more functionality.

The book gives a pure functional presentation of M68000-based microcomputer systems and does not include implementation issues, thus making it suitable to use the book for first-year or second-year students in the electrical or computer engineering curriculum. In fact, the only prerequisite needed is some experience of programming in a high-level language. I have used the material in the book successfully for first-year students in the electrical and computer engineering curriculum at Lund University for several years.

Control systems is an important area in which microcomputer systems play an

important role. Microcomputers are often used as components in a system consisting of a large number of communicating microcomputer systems. In order to emphasize this important application area, I have included a chapter on real-time control which illustrates how one can use a microcomputer system to support concurrently executing processes. We will learn in some detail how to design schedulers for time-sharing and real-time operating systems.

The outline of the book is as follows: Chapters 1 and 2 provide sufficient prerequisites in number systems and elementary computer arithmetic. In Chapter 3 and 4, I give a detailed presentation of the functional components and instruction set models for microcomputer systems in general, and for the M68000 in particular. At the conclusion of Chapter 4 I have introduced most of the instructions so that we will be able to design assembly language programs. The theme of Chapter 5 is to provide the reader with guidelines on how to design correct assembly language programs. This is achieved by applying a commonly-used technique known as step-wise refinement. I advocate the use of a high-level notation to specify the problem before it is coded in assembly. A Pascal-like notation is used throughout to express algorithms and solutions. In Chapters 6 and 7, we look in more detail into how the computer communicates with the outside world. Various schemes for synchronization of program execution with external events, such as polling and interrupt, are presented. We also look more closely into how a computer supports subroutines by introducing the stack. In Chapter 7, we concentrate on the issue of how two computerized devices can communicate. We note that this can be done by using programmable interfaces that can be set up to meet the communication requirements. Finally, in Chapter 8, we will see that a computer system can efficiently be used to execute several programs in a time-shared fashion. An important issue in this context is the management of real-time and I/O. I will show how a simple time-sharing and a real-time scheduler can be designed to meet this goal.

The textbook contains several worked examples to highlight the basic ideas. In addition, it also contains a large number of exercises. The appendices contain solutions to all these exercises, a summary of most instructions for the M68000, and an ASCII table.

This book is a result of teaching undergraduate students on the subject for more than ten years. Experience has been gathered by many people. I am indebted to my colleagues, past and present, for having contributed to many fundamental ideas behind this book, especially Lars Philipson and Lennart Ohlsson. A special thanks goes to Mats Cedervall for having reviewed the chapters about number systems and computer arithmetic. Finally, I am indebted to a large number of students from whom I've received many constructive ideas. Thank you all!

Lund, May 1992

*Per Stenström*

Chapter 1

# Number Systems and Symbol Representation in Computers

## 1.1 Number systems

When we deal with numbers we often mean the decimal number system consisting of the ten digits, i.e. $\{0, \ldots, 9\}$. The historical reason for this is that we have ten fingers, which enabled people in the old days to use their fingers as calculators. In a sense, the reason why we use the decimal number system is arbitrary. We could use another base or *radix* other than 10 just as well.

Computers use the *binary number system*. The reason for this is that they are built from digital devices which are based on two distinct voltages, namely 'high' and 'low' or '1' and '0'. This makes it extremely convenient to perform all computations based on the binary number system with radix 2, or a power of 2.

A drawback with the binary number system is that numbers, in general, tend to be very long. For instance, the decimal number 65.537 needs 17 binary digits, abbreviated *bits*, to be represented in the binary number system. In order to express binary numbers in a more concise form, the *hexadecimal number system* has been widely used.

The radix of the hexadecimal number system is 16 ($= 2^4$). It thus comprises 16 digits; $0, 1, 2, \ldots, 9$,A,B,C,D,E,F, where A,B,...,F denote the decimal numbers $10, 11, \ldots, 15$. The reasons why 16 is a convenient radix are (i) it is a power of 2 (as we will see this makes it extremely easy to convert binary numbers into hexadecimal ones and vice versa), (ii) it is sufficiently large to enable us to express fairly large numbers concisely, and (iii) it has a reasonable number of digits; for example, if 32 was used as a base, it would result in 32 digits. After this motivating introduction to number systems, we will look at number systems more formally.

A number system is characterized by a radix $r$, $r > 1$, and a set of $r$ digits or symbols given by the set $D = \{d_0, d_1, \ldots, d_{r-1}\}$. Any integer $N$ can be represented in this number system by a finite sequence of digits as

$$N_r = b_{n-1}b_{n-2}\ldots b_1 b_0 \tag{1.1}$$

**Table 1.1** Representation of the 16 smallest non-negative integers in the decimal, binary, and hexadecimal number systems.

| Decimal | Binary | Hexadecimal | Decimal | Binary | Hexadecimal |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 8 | 1000 | 8 |
| 1 | 1 | 1 | 9 | 1001 | 9 |
| 2 | 10 | 2 | 10 | 1010 | A |
| 3 | 11 | 3 | 11 | 1011 | B |
| 4 | 100 | 4 | 12 | 1100 | C |
| 5 | 101 | 5 | 13 | 1101 | D |
| 6 | 110 | 6 | 14 | 1110 | E |
| 7 | 111 | 7 | 15 | 1111 | F |

where each $b_i \in D$. The length of the number is $n$. Note that we are precise in expressing which number system we use by the subscript $r$. In daily life, often we implicitly assume the decimal number system and can then omit $r$. Throughout the book, in case there could be any confusion as to which number system we mean, we will be careful to tag explicitly the number with its radix.

It is important to distinguish between the *representation* and *numeric value* of an integer. For example $10_2$ and $2_{10}$ have the same numeric value but differ in their representations. The first number is represented in the binary number system and the second one is represented in the decimal number system. The numeric value is an abstraction that is independent of number representation. We could just as well have represented the numeric values of the numbers by two fingers or two oranges.

The *numeric value* $V(N_r)$ of an integer $N_r$ which comprises $n$ digits is computed as

$$V(N_r) = \sum_{i=n-1}^{0} b_i r^i \tag{1.2}$$

It is natural to represent the numeric value in the decimal number system, which is why we will never talk about a numeric value in a particular number system. Consequently, this formula will turn out to be useful when we convert a number in any number system into a decimal number, which will be done in the next section.

We end this section by presenting the 16 smallest non-negative integers in the three number systems we have discussed. They are found in Table 1.1.

## 1.2 Conversion between number systems

The first case we consider is a method to convert a number in a number system with radix $r$ into a decimal number. This is easily accomplished by applying the formula according to Equation 1.2.

Let us give an example: Convert the binary number $101011_2$ into decimal representation. The length of the binary number is 6 which is why we get

$$101011_2 = \sum_{i=5}^{0} b_i 2^i = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43_{10}$$

The leftmost bit is called the *most significant bit* (or *msb* for short) and the rightmost bit is called the *least significant bit* (or *lsb* for short).

In the second example, we want to convert $3F_{16}$ into decimal representation. The length of the hexadecimal number is 2 and thus

$$3F_{16} = \sum_{i=1}^{0} b_i 16^i = 3 \times 16^1 + F_{16} \times 16^0 = 3 \times 16^1 + 15_{10} \times 1 = 63_{10}$$

The second method we present applies to conversions of a decimal number to a number system with an arbitrary radix $r$. It is based on *Euclid's theorem* and can be found in almost every textbook on discrete mathematics.

We want to convert $N_{10}$ represented in the decimal number system into $N_r = b_{m-1}b_{m-2}\ldots b_1 b_0$ represented in a number system with radix $r$, where $N_r$ comprises $m$ digits. The relation between $N_{10}$ and $N_r$ is given by $N_{10} = b_{m-1}r^{m-1} + b_{m-2}r^{m-2} + \cdots + b_1 r + b_0$ according to Equation 1.2.

Euclid's theorem says that $N_{10} = Ar + b_0$, where $b_0 < r$. Thus, $b_0$ can be obtained as the remainder from the division $N_{10}/r$. The remainder from the division of $A$ by $B$ is denoted $A(\text{mod } B)$, and thus

$$b_0 = N_{10}(\text{mod } r)$$

But since $b_1 = \lfloor N_{10}/r \rfloor (\text{mod } r)$, where $\lfloor N_{10}/r \rfloor$ is the integer part of the division $N_{10}/r$, we can obtain all digits of $N_r$. This is done by repeatedly applying Euclid's theorem to the integer part of the division, which is expressed in the following algorithm:

**Step 1**: Let $X = N_{10}$ and $i = 0$.

**Step 2**: $b_i = X(\text{mod } r)$.

**Step 3**: Let $X = \lfloor X/r \rfloor$ and $i = i + 1$.

**Step 4**: Repeat Step 2 and 3 until $i = m$.

Now, it may happen that the result of the integer division is zero, before $i = m$. We could stop at this point, because further application of Step 2 will yield $b_i = 0$. These digits result in leading zeros and are not significant. For instance, $12_{10}$ and $012_{10}$ both have the same numeric value.

It is now time to look at some examples. Convert $43_{10}$ into an 8-bit binary number ($m = 8$).

$$
\begin{array}{lrrl}
b_0 = & & 43_{10} (\text{mod } 2) = & 1 \\
b_1 = & \lfloor 43_{10}/2 \rfloor (\text{mod } 2) = & 21_{10} (\text{mod } 2) = & 1 \\
b_2 = & \lfloor 21_{10}/2 \rfloor (\text{mod } 2) = & 10_{10} (\text{mod } 2) = & 0 \\
b_3 = & \lfloor 10_{10}/2 \rfloor (\text{mod } 2) = & 5 (\text{mod } 2) = & 1 \\
b_4 = & \lfloor 5/2 \rfloor (\text{mod } 2) = & 2 (\text{mod } 2) = & 0 \\
b_5 = & \lfloor 2/2 \rfloor (\text{mod } 2) = & 1 (\text{mod } 2) = & 1 \\
b_6 = & \lfloor 1/2 \rfloor (\text{mod } 2) = & 0 (\text{mod } 2) = & 0 \\
b_7 = & \lfloor 0/2 \rfloor (\text{mod } 2) = & 0 (\text{mod } 2) = & 0
\end{array}
$$

Thus, $43_{10} = 00101011_2$. As mentioned, we could stop after we calculated $b_6$, because the result of the integer division is zero. Let us convert $63_{10}$ into a hexadecimal number with 3 digits.

$$
\begin{array}{lrrl}
b_0 = & 63_{10} (\text{mod } 16) = & 15_{10} = & \text{F} \\
b_1 = & \lfloor 63_{10}/16 \rfloor (\text{mod } 16) = & 3 (\text{mod } 16) = & 3 \\
b_2 = & \lfloor 3/16 \rfloor (\text{mod } 16) = & 0 (\text{mod } 16) = & 0
\end{array}
$$

Thus, $63_{10} = 3\text{F}_{16}$ which is what we expected since we made the opposite conversion earlier in this section.

Now we turn our attention to the conversion between binary and hexadecimal numbers. As we pointed out in the beginning of this chapter, the convenience of hexadecimal numbers stems from the fact that they are easily converted into binary numbers. This is because each hexadecimal digit can be represented by exactly four bits (binary digits). For instance, $0010_2 = 2_{16}$ and $1011_2 = \text{B}_{16}$ (see Table 1.1). This means that we can convert any hexadecimal number by converting each individual hexadecimal digit in that number. Convert $\text{AB3}_{16}$ into a binary number: Since $\text{A}_{16} = 1010_2$, $\text{B}_{16} = 1011_2$, and $3_{16} = 0011_2$ expressed as 4-position binary numbers we obtain $\text{AB3}_{16} = 101010110011_2$. If instead we convert $123_{16}$ into a binary number we obtain $000100100011_2 = 100100011_2$. In this example, we obtained leading zeros which, of course, are not significant. Therefore we can omit them.

Hopefully, we have now an idea of how to make the opposite conversion, that is, how to convert a binary number into a hexadecimal. If the length of the binary number is a multiple of four, it is straightforward. Then, we simply decompose the binary number in groups of four bits each, and use Table 1.1 to convert each individual group into one hexadecimal number as in the following example: Convert $110101101001_2$ into a hexadecimal number. We first decompose the binary number in three four-tuples as 1101|0110|1001. Using Table 1.1, we then obtain $\text{D69}_{16}$, because $1101_2 = \text{D}_{16}$, $0110 = 6_{16}$, and $1001_2 = 9_{16}$. But how do we convert $1101011010_2$ into a hexadecimal number? The problem here is that the binary number comprises ten bits which is not a multiple of four. However, we can always add as many leading zeros as we need, since this will not change the numeric value of the number. That is, $1101011010_2 = 0011|0101|1010 = 35\text{A}_{16}$.

We can now summarize the methods we have employed in this section to convert between different number systems as follows:

- Conversion of hexadecimal and binary numbers into decimal numbers is performed by using Equation 1.2.
- Conversion of decimal numbers into binary or hexadecimal numbers is performed by using Euclid's theorem.
- Conversion of hexadecimal numbers into binary numbers or vice versa is performed by converting each hexadecimal digit or binary four-tuple individually.

### Exercises

**1.1** Convert $101010_2$ into a decimal number.

**1.2** Convert $8FF_{16}$ into a decimal number.

**1.3** Convert $2337_{10}$ into a 4-digit hexadecimal number.

**1.4** Convert $29_{10}$ into a 6-bit binary number.

**1.5** Convert $9A8_{16}$ into a binary number.

**1.6** Convert $10011100101110_2$ into a hexadecimal number.

## 1.3 Symbol representation in a computer

In this section, we are concerned with the problem of how to represent different kinds of information such as unsigned and signed integers, e.g. $43_{10}$, $-115_{10}$, and alpha-numeric characters such as in the string 'Hello World'.

Most information coding schemes for computers use *binary codes*. However, the coding scheme differs; e.g. integers and characters are coded differently. It is therefore important to know that the meaning of a sequence of bits is determined not only by the sequence itself, but also by the coding scheme that has been used.

In the previous section, we showed one coding scheme, namely how non-negative integers can be represented by binary numbers. In this section, we will demonstrate other coding schemes that allow us to represent negative integers and characters. In Chapter 4, we will present yet another coding scheme, namely how the instructions of a computer, the Motorola 68000, are coded.

### 1.3.1 Length of a sequence of bits

In a computer information entities, such as numbers, are represented by a fixed number of bits. This is referred to as the *word length* of the computer. The word

**Table 1.2** Different word lengths used throughout the text.

| Name | Number of bits |
|---|---|
| Bit | 1 |
| Nibble | 4 |
| Byte | 8 |
| Word | 16 |
| Long word | 32 |

length differs from computer to computer. Some early microprocessors such as the Intel 8080 and the Zilog Z80 used 8-bit words which are commonly referred to as a *byte*. Early minicomputers such as the DEC PDP-11 used 16-bit words. Contemporary microcomputers often use 32-bit words and mainframes often use 64 bits. Throughout the book, we will refer to 4, 8, 16, and 32 bits word length as nibble, byte, word, and long word. They are summarized in Table 1.2.

Note that it is only nibble and byte that are terms commonly accepted to denote 4 and 8 bits, respectively. The meaning of word and long word may differ from manufacturer to manufacturer.

Why is the word length so important? The reason is that it determines the range of numbers we can express. To show this, we first consider 4-bit unsigned integers.

### 1.3.2 Unsigned integers

In order to represent unsigned integers, computers use the binary number representation presented in the previous section. This means that we can simply use Equation 1.2 to convert them into decimal numbers. If the word length is a nibble (4 bits), we can express all decimal numbers in the range $[0, 15]$.

In general, if we have n-bit binary numbers interpreted as unsigned integers, we can express all decimal numbers in the range

$$R_{10} = [0, 2^n - 1] \tag{1.3}$$

because each bit can take one of two values so the number of codes are $2^n$. We say that $R_{10}$ is the decimal *range* of the binary numbers.

The range not only limits the possibility of expressing any integer, we must also carefully consider its effects on arithmetic operations, as we will see in the next chapter.

An important point is that we can obtain a range with any size, given a sufficient word length. However, we can only express non-negative integers. We will therefore now present a coding scheme that is commonly used by computers to represent negative and non-negative integers in a range determined by the word length.

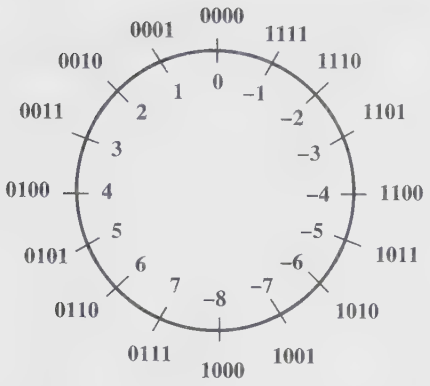



**Figure 1.1** Coding scheme for 4-bit two's complement numbers.

### 1.3.3 Signed integers

We shall now consider how to represent integers in an arbitrary range. What we would like to achieve is to be able to express the same number of negative as non-negative integers. Given n-bit binary numbers, we would like the range to be

$$R_{10} = [-2^{n-1}, 2^{n-1} - 1] \tag{1.4}$$

This way, we have obtained the same number of non-negative as negative integers, namely $2^{n-1}$. We will present a coding scheme which turns out to be convenient to make conversions between its decimal number counterpart, and, as will be demonstrated in the next chapter, especially superior in dealing with integer arithmetic. The coding scheme is called *two's complement representation* and will be defined below.

We present the coding scheme for nibbles in Figure 1.1. We show the coding scheme by means of a circle. This will turn out to be convenient when we reason about arithmetic in the next chapter. Inside the circle, we show the decimal numbers that can be represented, and outside the circle we show the corresponding two's complement representation of these numbers.

There are some interesting observations that can be made about the coding scheme: (i) the negative integers are coded with a leading '1' and non-negative integers with a leading '0', and (ii) the non-negative integers are coded in the same way we used before.

The strange thing with the coding scheme is that one might ask why we code the negative integers in this way. We will not, however, answer this question until the next chapter. What we would like to do now is to demonstrate a method of how to code the negative integers. Given a decimal number $N_{10}$ in the range $R_{10}$ as defined in Equation 1.4, the method is as follows:

- If $N_{10} \geq 0$, convert it into an n-bit binary number.
- If $N_{10} < 0$, convert $(2^n + N_{10})(\text{mod } 2^n)$ into an $n$-bit binary number.

We need to comment on the last point. If we code $-7_{10}$ as a 4-bit two's complement number, we code $(2^4 - 7)_{10} = 9_{10} = 1001_2$. This may seem awkward. The reason is that the coding scheme should allow us to add arbitrary integers. For instance, given two numbers $A$ and $B$, where $B = -A$, then $A + B\ (= 0)$ is coded as $(2^n + A - A)(\text{mod } 2^n) = 2^n\ (\text{mod } 2^n) = 0$, but more about this in the next chapter.

Convert $4_{10}$ into an 8-bit two's complement binary number: Since 4 is non-negative, we simply convert it to a binary number. Thus, $4_{10} = 00000100_2$. Now, to convert $-4_{10}$ into an 8-bit two's complement number: We first obtain $2^8 - 4 = 252_{10}$. This is then coded using Euclid's theorem as $11111100_2$. Thus, $-4_{10} = 11111100_2$ in 8-bit two's complement representation.

## Exercises

**1.7** What range can be obtained using 8 bits to express unsigned integers?

**1.8** What range can be obtained using 6 bits to express unsigned integers?

**1.9** What range can be obtained using 8 bits to express an equal number of negative and non-negative integers using two's complement representation?

**1.10** What range can be obtained using 6 bits to express an equal number of negative and non-negative integers using two's complement representation?

**1.11** Convert $7_{10}$ into an 8-bit two's complement binary number.

**1.12** Convert $-7_{10}$ into an 8-bit two's complement binary number.

**1.13** Interpret the binary string 1001 in the range $[0, 15]$ and as a 4-bit two's complement number.

### 1.3.4 Representation of alpha-numeric characters

It is not sufficient for the computer to deal with numbers only. It must also be able to deal with text. One obvious example is to manage large databases of bibliographical information. In this case the computer can be used to aid people in

**Table 1.3** The ASCII coding scheme for alpha-numeric characters.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [ | k | { |
| C | FF | FS | , | < | L | \ | l | \| |
| D | CR | GS | - | = | M | ] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | _ | o | DEL |

searching for literature. In order to do this, it must be able to treat characters in a way that makes it possible to perform computations. For instance, when sorting a number of words into lexicographical order, it must be able to compare the letter 'A' with 'B' and conclude that 'B' is greater than 'A', abstractly speaking. Since the same information is to be used by computers from different manufacturers, it is also important to have a standard coding scheme for characters.

Characters are not only letters. They are basically all symbols that can be typed on an ordinary typewriter such as comma ',', exclamation mark '!', as well as the digit '8'. These are collectively called *alpha-numeric characters.*

In order to meet these requirements, almost all computers agree upon a standard coding scheme for alpha-numeric characters called ASCII (American Standard Code for Information Interchange). Each character is represented by a 7-bit code. We show these codes in hexadecimal representation in Table 1.3.

Each character is associated with a 7-bit code. In Table 1.3, this code is represented as 2 digit hexadecimal numbers. The first digit is retrieved from the column and the second one is retrieved from the row. For instance, the letter 'A' is coded as $41_{16} = 1000001_2$. The ASCII code for the digit '5' is $35_{16}$ and so forth.

Besides the visible characters, there are some 'invisible' control characters such as 'carriage return' (CR) coded as $0D_{16}$, 'line feed' (LF), and 'space' (SP). In addition, there are some special characters used for information control such as 'end of text' (ETX) etc.

### Exercises

**1.14** What is the ASCII code of the letter 'F'?

**1.15** What is the ASCII code of the character '+'?

**1.16** What is the ASCII code of the letter 'a'?

**1.17** What is the sequence of ASCII codes that encodes the string 'HELLO'?

**1.18** What string has the sequence of ASCII codes '36 38 30 30 30'? (All codes are in hexadecimal representation)

## 1.4 Summary and concluding remarks

In this chapter, we have presented three coding schemes based on sequences of binary digits that enable us to represent integers and characters in a computer.

Coding schemes are based on binary codes because computers interpret binary coded information. We began, therefore, by demonstrating methods of how to make conversions between binary and decimal numbers. Since binary numbers tend to be long compared to their decimal number counterparts, the hexadecimal number system was introduced. It enables us to (i) express binary numbers in a more concise form and (ii) simplify conversions to the binary number system.

Unsigned integers are simply coded as binary numbers. In order to extend the representation to cover negative integers as well, we introduced the two's complement representation and a method to convert decimal integers into this representation. In the next chapter, where we deal with arithmetic, we will reveal the motivation as to why the two's complement representation is used.

In order to represent characters, we presented a standard coding scheme called ASCII and explained how strings of text are coded in this coding scheme.

An important conclusion of this chapter is that a sequence of bits is interpreted differently depending on the coding scheme used. For instance, $0001010_2$ is interpreted as $10_{10}$ if it is coded as a 7-bit two's complement number, or it could be interpreted as 'line feed' if it is coded according to ASCII.

Chapter 2

# Elementary Computer Arithmetic and Logic

## 2.1 Unsigned integer arithmetic

In the previous chapter, we noted that given a word length of $n$ bits, we can express unsigned integers in the range $[0, 2^n - 1]$. We show this range for nibbles by means of a circle in Figure 2.1. When we add two numbers, say 5+2, we can illustrate this by moving 5 steps counter-clockwise and then 2 steps further yielding the correct result 7. However, what happens if we add 9+7? When we have moved 7 steps ahead of 9, we get the result zero. The reason for this is that we have exceeded the range limit $2^4 - 1 = 15$.

We shall now explain formally how addition is performed on unsigned numbers with a specific length $n$. Given two n-bit numbers $A$ and $B$, the result of the addition is

$$(A + B) (\text{mod } 2^n)$$

This means that the result is correct if and only if $A + B < 2^n$. Otherwise, we say that the arithmetic operation has led to *overflow*.

Computers must detect overflow in one way or another. We shall now explain how this is done. Computers do not actually use circles to perform additions or subtractions. In fact, the electronic devices that perform arithmetic operations work in the same way we learn in elementary school, namely, according to the well-known 'pen-and-pencil method'. Given two n-bit binary numbers $A$ and $B$, where $A = a_{n-1}a_{n-2} \ldots a_1a_0$ and $B = b_{n-1}b_{n-2} \ldots b_1b_0$, the sum is also given by an n-bit number $S = s_{n-1}s_{n-2} \ldots s_1s_0$.

Let us give an example to demonstrate the method. Suppose that $A = 1010_2 = 10_{10}$ and $B = 0011_2 = 3_{10}$

| $C$ | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|
| $A$ | | 1 | 0 | 1 | 0 |
| $B$ | + | 0 | 0 | 1 | 1 |
| $S$ | | 1 | 1 | 0 | 1 |

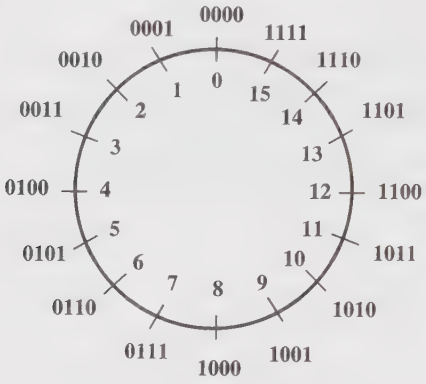



**Figure 2.1** The range of unsigned integers for nibbles.

These numbers are added by adding the bits in each column, taking into account the carry from the previous position. Formally, the sum is obtained according to

$$s_i = (a_i + b_i + c_i)(\text{mod } 2) \tag{2.1}$$

where the carry-bit is computed as

$$c_{i+1} = \lfloor (a_i + b_i + c_i)/2 \rfloor \tag{2.2}$$

and $c_0 = 0$. Note that $c_{i+1}$ is computed as the integer part of the division of the sum with the radix (2). This is the rule we learn in elementary school when calculating the carry.

From the example, we note that the leftmost carry-bit, $c_n = 0$. The result of the addition yielded the correct value $1101_2 = 13_{10}$. Now look at the addition of $A = 1010_2 = 10_{10}$ and $B = 0111_2 = 7_{10}$.

| $C$ | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|
| $A$ | | 1 | 0 | 1 | 0 |
| $B$ | + | 0 | 1 | 1 | 1 |
| $S$ | | 0 | 0 | 0 | 1 |

In this case we get overflow. However, the addition resulted in $c_n = 1$. These examples suggest a method to detect overflow.

For addition of n-bit binary numbers, X, in the range $[0, 2^n - 1]$, overflow is detected as the most significant (leftmost) carry-bit, $c_n = 1$.

### Exercises

**2.1** Add the two 4-bit, unsigned numbers $A = 0111_2$ and $B = 0001_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.2** Add the two 5-bit, unsigned numbers $A = 00100_2$ and $B = 11110_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.3** Add the two 6-bit, unsigned numbers $A = 011000_2$ and $B = 000001_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.4** Add the two 8-bit, unsigned numbers $A = 10000000_2$ and $B = 10000000_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

## 2.2 Two's complement arithmetic

Two's complement representation not only makes it possible to express negative integers; it also suggests a method to perform subtraction. This stems from the fact that the subtraction $A - B = A + (-B)$. We can simply subtract $B$ from $A$ by adding $B$'s negative equivalence. Given a number $A$, we call $A$'s negative equivalence the *inverse* of $A$ and denote it $\tilde{A}$.

In the previous chapter, we learned how to code an integer according to the two's complement scheme. This method also suggests a way to derive the inverse $\tilde{A}$ of any integer $A$.

Given two n-bit two's complement numbers $A$ and $B$. The subtraction $A - B(\text{mod } 2^n) = A + 2^n - B(\text{mod } 2^n) = A + \tilde{B}(\text{mod } 2^n)$, where $\tilde{B} = 2^n - B$. Consider the following example: Perform the subtraction $A - B$, where A = $0100_2 = 4_{10}$, and B = $0010_2 = 2_{10}$ expressed as 4-bit two's complement numbers. We first determine the inverse of $B$, $\tilde{B} = (2^4 - 2)_{10} = 14_{10} = 1110_2$. We then perform the addition (i.e. according to the pen-and-pencil method) $A + \tilde{B} = 0100_2 + 1110_2 = 0010_2 = 2_{10}$.

There is a convenient method to determine the inverse of an arbitrary two's complement number which we present without any proof.

**Step 1**: Replace all zeros by ones and vice versa.
**Step 2**: Add one to the remaining number.

For example, find the inverse of $B = 0010$: Step 1 yields 1101. Adding one to this number yields the result $\tilde{B} = 1110$ (compare with $\tilde{B}$ in the above example).

We have now learned how to make addition and subtraction with two's complement binary numbers. The problem of exceeding the range still exists, and we have to devise a test as to whether the result is correct or not. Let us consider the following example: We want to add two numbers $A = 0100_2 = 4_{10}$ and $B = 0101_2 = 5_{10}$ which are 4-bit two's complement numbers. Recall that the range is $[-8, 7]$. Since $A + B = 0100_2 + 0101_2 = 1001_2 = -7_{10}$, the result is not correct. The reason is that the sum exceeds the upper range limit 7. Likewise, if we add two negative numbers, for instance, $A = 1100_2 = -4_{10}$ and $B = 1011_2 = -5_{10}$, we obtain $0111_2 = 7_{10}$.

> **Condition: (Overflow)** The addition of two two's complement numbers results in overflow iff
>
> > Both numbers have the **same** sign (either negative or non-negative), **and**
> > the sign of the sum is **opposite** to the numbers added.

Applying this condition to the previous example, we can immediately see that the addition results in overflow by just examining the sign of the terms and the sum. Since both terms have the same sign (negative) and the sum is of the opposite sign (non-negative), both subconditions are satisfied. For a computer, this test is easily implemented because the sign of a number is specified by the most significant bit.

### Exercises

**2.5** Add the two 4-bit, two's complement numbers $A = 0111_2$ and $B = 0001_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.6** Add the two 5-bit, two's complement numbers $A = 00100_2$ and $B = 11110_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.7** Add the two 6-bit, two's complement numbers $A = 011000_2$ and $B = 000001_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

**2.8** Add the two 8-bit, two's complement numbers $A = 10000000_2$ and $B = 10000000_2$. Determine the decimal values of $A$, $B$, and the sum and whether the addition resulted in overflow.

## 2.3 Logical operations

We have mostly been concerned with how computers perform arithmetic. There are

other kinds of computations that must be carried out as well. An important class of computation that we need to be familiar with is referred to as *logical operations.*

Logical operations enable the computer to decide whether a statement like 'it's raining and you are outside' is true or false. This statement is true provided that the statements 'it's raining' AND 'you are outside' both are true. We could consider the statement as a composition of logical variables: Let $X$ denote 'it's raining' and $Y$ denote 'you are outside' then we can rewrite the statement as a logical function: $f(X,Y) = X \wedge Y$ where $\wedge$ denotes AND. The nice thing about this is that $X$ and $Y$ can take only one of two values, namely 'true' or 'false'. This gives us the splendid idea of representing them as binary variables with '1' denoting 'true' and '0' denoting 'false'. Doing this, we can define $f(X,Y)$ with a *truth table*:

AND

| $X$ | $Y$ | $X \wedge Y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

This function is referred to as the *logical* AND operation. Note that both X AND Y must be '1' in order for the operation to yield the result '1'. Another useful logical operation is the following one: $f(X) = X'$. It is called the *logical inverse*, or NOT, of X; when $X$ is true $X'$ is false, or if we use a truth table:

NOT

| $X$ | $X'$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

In order for the computer to perform logical reasoning, we need other logical operations as well. Consider the example 'it's raining or it's snowing'. In this example, we need the *inclusive-or*, also denoted OR, operation to deal with it. We can rewrite the statement as a logical function according to: $f(X,Y) = X \vee Y$, where $\vee$ denotes OR. We get the following truth table:

OR

| $X$ | $Y$ | $X \vee Y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

The logical OR operation is called *inclusive-or* because it is true if either or both subconditions is true.

In the following logical expression, we want to distinguish the case when both subconditions are true: 'either it's raining or it's snowing'. This statement says that either subcondition is true, but not both. The logical operation 'either ... or' is denoted *exclusive-or* , or EOR, and has the following truth table:

EOR

| $X$ | $Y$ | $X \oplus Y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

As we have seen, computers deal with binary strings of various lengths. Sometimes it is useful to perform logical operations on strings of bits in order to compare two strings. Therefore, the logical operations must also be defined for a string of bits. Given two n-bit binary strings $A$ and $B$, where $A = a_{n-1}a_{n-2}\ldots a_1a_0$ and $B = b_{n-1}b_{n-2}\ldots b_1b_0$, the logical AND between these strings, that is $C = A \wedge B$, provides a string C of $n$ bits, where $c_i = a_i \wedge b_i$, $i = 0, 1, \ldots, n-1$. We illustrate this operation by an example. Suppose that $A = 1010$ and $B = 0011$, then

| | | | | | |
|---|---|---|---|---|---|
| A | | 1 | 0 | 1 | 0 |
| B | $\wedge$ | 0 | 0 | 1 | 1 |
| C | | 0 | 0 | 1 | 0 |

OR and EOR on strings are defined analogously, namely, the operation on individual bits is performed positionally. With the same bit strings as in the previous example, the OR operation yields

| | | | | | |
|---|---|---|---|---|---|
| A | | 1 | 0 | 1 | 0 |
| B | $\vee$ | 0 | 0 | 1 | 1 |
| C | | 1 | 0 | 1 | 1 |

and the EOR yields

| | | | | | |
|---|---|---|---|---|---|
| A | | 1 | 0 | 1 | 0 |
| B | $\oplus$ | 0 | 0 | 1 | 1 |
| C | | 1 | 0 | 0 | 1 |

Note, in the truth-table above, that $c_i = 1$ iff $a_i \neq b_i$. This observation can be used to find out in which positions two binary strings differ.

The operations AND, OR, and EOR as defined above are examples of *binary operations* because they take two operands. The NOT operation only takes one operand and is called a *unary operation.*

### Exercises

**2.9** Perform the logical AND operation on the 4-bit strings $A = 1111$ and $B = 0010$.

**2.10** Perform the logical AND operation on the 4-bit strings $A = 1111$ and $B = 0011$. What can you say about the result as compared to B?

**2.11** Perform the OR operation (inclusive or) on the 4-bit strings $A = 0000$ and $B = 1010$. What can you say about the result as compared to B?

**2.12** Perform the EOR operation (exclusive or) on the 4-bit strings $A = 0101$ and $B = 0101$.

**2.13** Perform the EOR operation on the 4-bit strings $A = 1111$ and $B = 0000$. Comment on the result as compared to the previous exercise.

## 2.4 Summary and concluding remarks

In this chapter, we learned how to perform elementary arithmetic and logical operations on binary numbers.

An important issue that was raised when performing arithmetic operations was how to detect overflow, namely, when the operation gives an erroneous result. We presented simple techniques that are used by computers to test for overflow.

We also provided a deeper insight into the advantage of the two's complement coding scheme in dealing with arithmetic. It provided us with a means to implement subtraction by adding the inverse of the number. For computers, this leads to a simpler machine design in that the same electronic devices can be used for both subtraction and addition. Finally, we presented some elementary logical operations. In the subsequent chapters, we will show that these primitive operations are essential for the support of high-level language constructs.

Chapter 3

# Computer System Model

In order to understand the operation of a computer system, one can use descriptions on several levels of abstractions. The intention behind this text is to understand the operation of a computer on the instruction set level. This abstraction is usually referred to as the *architecture* or *programming model* of the computer. Although this eliminates the need of explaining details about the electronic design of the computer, this abstraction level is still too complex.

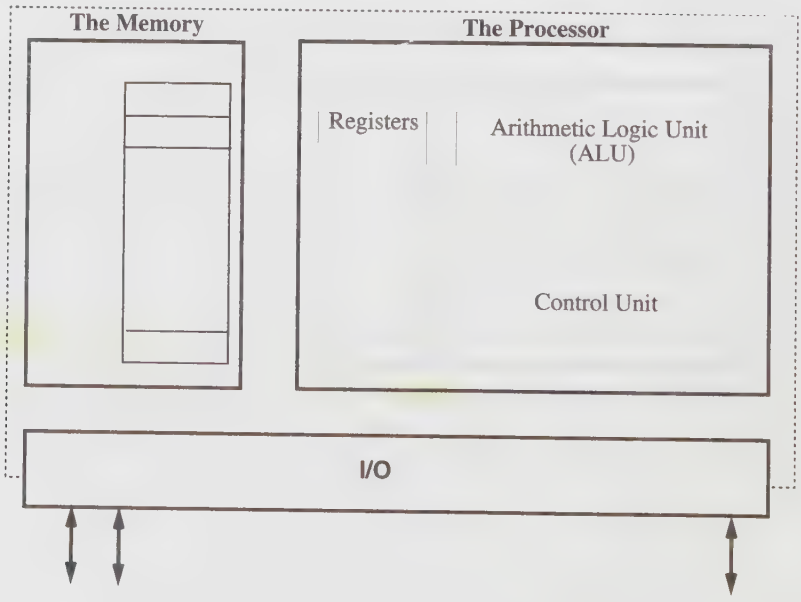



**Figure 3.1** The functional units of a computer.

A computer has three main components (Figure 3.1): A Microprocessor (or processor for short), a Memory, and an Input/Output System (I/O). The processor

consists of three main parts: the Arithmetic Logic Unit (ALU) where all arithmetic and logical operations take place, the Registers where data is temporarily stored, and the Control Unit which interprets the instructions contained in the program. The processor is an active component that synchronizes all actions in a computer. We will later describe its functionality in more detail.

We have chosen the approach of 'information hiding' to explain the operation of a computer system. The Motorola 68000 (M68000) is but one example of a processor, although a widely used one. Choosing a concrete example enables us to exercise practically the concepts of machine language programming. However, it is important to know that some of its features are not general. We will therefore concentrate on those features that can be found in most computer systems rather than specifics about the M68000.

The memory contains the program (a sequence of instructions) and the data to be processed. The memory content can be read as well as modified by the processor. The I/O-units are essential in order for the computer to communicate with the outside world. A computer without I/O-units could be thought of as a person without the abilities to listen and talk; how good a problem solver this person may be is irrelevant as long as his/her thoughts cannot be communicated to the outside world.

We will start by looking at a model of the memory sufficient for the rest of the text, and then present the fundamental operation of the processor, the *instruction cycle*, namely, that of fetching the next instruction in memory and then executing it.

## 3.1 Memory model

The memory contains the program and temporary data. It is divided into a number of storage units, called memory cells, and each memory cell can store a number of bits. The size of a memory cell, i.e. the number of bits it contains, specifies the least amount of information that can be accessed by the processor. In order to retrieve the information contained in a memory cell, each memory cell is identified by a unique number called an *address*. We will assume that a memory cell contains 8 bits (a byte).

The processor can basically perform two operations on the memory. It can retrieve the information contained in one or a number of consecutively numbered memory cells by *reading* their contents, or it can change the information in them by *writing* to the memory. For both of these operations, the processor must tell the memory which memory cell(s) it wants to access by sending two parameters to the memory; the address of the first memory cell and the length (byte, word, or long word) of the information to be accessed.

Figure 3.2 shows a model of a memory containing $B = 2^A$ bytes. Since a memory address is an unsigned number in the range $[0, 2^A - 1]$, the address can be expressed

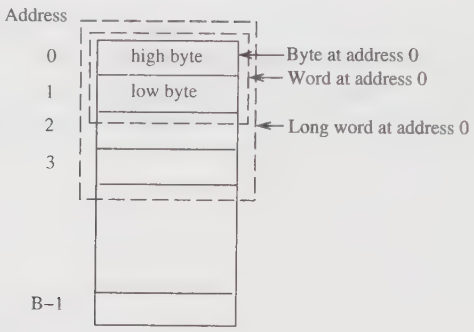



**Figure 3.2** The memory model.

by $A$ bits. Since the processor can read and write entities of different sizes, there is a need for a rule of how to store words and long words. In Figure 3.2, we show this rule for the M68000. A word at address $i$ is stored with the most significant byte (high byte) in the memory cell at address $i$ , and the least significant byte at address $i + 1$. Likewise, a long word (containing four bytes) at address $i$ is stored with the most significant byte at address $i$ and the least significant byte at address $i + 3$.

Formally, a read operation contains the following parameters

READ(Size, Address)

where Size $\in$ { Byte (B), Word (W), Long word (L)}, and Address $\in \{0, i, 2i, \ldots, B - i\}$. The addresses that are allowed depend on the Size; if Size is Byte then $i = 1$ and if Size is Word or Long word then $i = 2$. For instance, the possible word addresses are $0, 2, \ldots, B - 2$.

A write operation contains the following parameters

WRITE( Size, Address, Data)

where Size and Address are defined above and Data is the information that replaces the contents of the memory cells defined by Address and Size.

The maximum size of the memory that is contained is determined by the number of address bits provided by the processor. For the M68000, which we consider in this text, the size of the memory that can be attached is $2^{24}$ bytes. This is a huge amount of memory. By the same reason that we use entities like 'kilo metre' (abbreviated km) to denote 1000 metres, we use Kb (Kilo byte) to denote 1024 ($= 2^{10}$) bytes, and Mb (Mega byte) to denote $2^{20}$ bytes. We will now start to look at the fundamental operation of a computer.

## 3.2 The instruction cycle

We now turn our attention to the processor. A computer program consists of a

```
                    | 8000  MOVE.L  #1,D0  ; i := 1
                    | 8004  CMPI.L  #N,D0  ; if i > N
for  i:= 1 to N do  | 8008  BHI     $8014  ; then goto 8014
     j:= j+1;       | 800A  ADDI.L  #1,j   ; j := j+1
                    | 800E  ADDI.L  #1,D0  ; i:= i+1
                    | 8012  BRA     $8004  ; goto 8004
                    | 8014
```

**Figure 3.3** An example of a Pascal program to the left and its translation into M68000 machine language to the right.

sequence of instructions that exactly specifies what computation the computer is required to perform. Computer programs are mostly written by using a *high-level language* (HLL) such as Pascal, C, and Fortran. A program written in a HLL cannot be interpreted directly by the computer. It must first be translated into a *machine language program*, which is performed by a program sometimes called a compiler. The machine language consists of a set of binary coded instructions. Unlike a HLL, the machine language is specific to a particular computer. The elements of the machine language, that is the computer instructions, can only perform elementary tasks as compared to the more powerful HLL statements.

In order to get an idea of the properties of a machine language, we consider the example program in Pascal and its M68000 machine language equivalence according to Figure 3.3. The Pascal program appears to the left and the corresponding M68000 machine language program, in symbolic form, appears to the right. The machine language program is stored in memory at consecutive addresses which are shown to the left. Several important observations can be made: First, the HLL program consists of two statements, whereas the machine language program consists of more than twice this number of instructions. In general, a HLL program is translated into more than five times as many instructions as the number of statements. Second, the operation that is performed by each individual M68000 instruction is very simple. For example, the first instruction (`MOVE.L #1,D0`) assigns the value 1 to one of the registers (small memory in the processor), namely register D0, and the subsequent instruction compares D0 with N. If D0 contains a number that is greater than N, the execution will continue at address `8014`. Otherwise, the statement `j:=j+1` is carried out and `i` is incremented before the execution continues at address `8004` by the instruction `BRA $8004`.

As has been pointed out previously, the machine language program is binary coded and stored in the memory. In the example above, we presented the machine language program in a more readable form for human beings, the so called *assembly language* notation. In the next chapter we will present the entire set of machine instructions for the M68000 and show how these are coded in the memory. At this point we shall only make clear that there is usually a **one-to-one correspon-**

**dence** between an assembly language instruction and the binary coded machine instruction. This enables us to explain a few things of how the computer executes a machine language program without bothering about coding details.

Instructions in a machine language program are stored at consecutive addresses in the memory. Each instruction occupies at least one word and, depending on the complexity of the instruction, up to five words. Basically, the processor fetches the next instruction from the memory. It then executes it. In order to keep track of the address of the next instruction to be fetched, the processor has a special-purpose storage element, called *program counter* (PC), that stores the address of the next instruction to be executed. Most of the time, the next instruction to be executed is the one that appears at the next higher address. However, as in the example above, there are exceptions to this rule. Special instructions, such as the branch-instructions, assign a new value to the program counter so as to change the control flow of the executed program. Consequently, the basic task of the processor is to perform the following elementary cycle over and over again:

**Step 1**: Fetch the instruction at the memory address specified by PC.
**Step 2**: Update PC.
**Step 3**: Execute the instruction.

This cycle is referred to as the *instruction cycle*. The instruction cycle constitutes our first model of the functionality of the processor. In the next chapter we will refine this model, when we present the instruction set of the M68000.

The I/O system enables the processor to transfer information between the memory and the outside world. The I/O system is a general term for all devices that can convey such information such as terminals, printers, and disks. We will look more closely into this in the subsequent chapters.

## 3.3 Concepts of computer instructions

The instructions in the example program of Figure 3.3 typically perform an operation on a number of *operands* and store the result in a storage device, which could be the memory or a register in the processor. In general, an instruction contains information about three things: which operation to be performed, where the operands are, and where to put the result. In order to express these basic tasks in a concise form, we will extensively use the following notation:

$$A \rightarrow B$$

Place the value of $A$ in location $B$

A location is specified by either the register name, in case it is a register location, or the address of the memory cell in case it is a memory location. Consequently, $B$ denotes where to put the result. The operation itself is specified by the expression

$A$. In order to express the **content** of location $L$, we will use the notation $(L)$. For example: $(L)+3 \rightarrow L$ means: 'The sum of the content of location $L$ and the constant number 3 is placed in location $L$'. Note that this is what the assignment statement `L:=L+3` would do in Pascal. Thus, parentheses are used to express the *content* of a location and the arrow denotes assignment.

The operands are either stored in the memory, in the internal registers of the processor, or explicitly contained in the instruction. In the instruction '`MOVE.L #1,D0`' from Figure 3.3, one of the operands is explicitly contained in the instruction, namely '1', whereas the other is stored in register D0. In the instruction '`ADDI.L #1,`$j$', $j$ corresponds to a memory address, that is, the result is to be stored at the memory location with address $j$. These examples demonstrate a few of all the possible ways to point out the location of an operand. The concept of doing this is called *addressing modes*. So far, we have introduced the following three addressing modes:

- *Register direct addressing.* The name of the register designates the operand location. Example: (D0) $\rightarrow$ D1 copies the content of register D0 to register D1.
- *Absolute addressing.* The address of the memory designates the operand location. Example: (1) $\rightarrow$ 2 copies the content of the memory cell at address 1 to the memory cell at address 2.
- *Immediate addressing.* The operand is explicitly contained in the instruction. Example: 1 $\rightarrow$ 2 copies the constant value 1 to the memory cell at address 2.

The assembly language lets the programmer write symbolic names for the operations and their operands. A program written in an assembly language can be translated automatically by another program, called an *assembler*. Below, we provide an example of a line of assembly code:

```
LOOP  ADD.B  #$12,D0  ; Add 12 (hex) to D0
```

The line consists of four different fields. The first field ('`LOOP`') is optional and, when used, gives a unique name, called a *label*, to this particular instruction. The second field ('`ADD.B`') contains the symbol for the machine operation and the third field ('`#$12,D0`') specifies its operands. The fourth field ('`; Add...`') is used for an optional comment, and the semicolon indicates the beginning of the comment.

Special characters are used to denote number representation and immediate addressing. '#' in the instruction tells the assembler that the operand is a number. In order to enable the programmer to express numbers in different number systems, there must be a way to point this out in the assembly language program. We will use the notation found in Table 3.1 which, as a matter of fact, is the notation chosen by Motorola. The default representation is decimal, that is, if a

**Table 3.1** Assembly language notation to express constant values in different number systems.

| Number system | Notation |
|---|---|
| Decimal (default) | |
| Binary | % |
| Hexadecimal | $ |

constant value is not preceded by any of the characters in Table 3.1, the number is decimal. In the example above, `ADD.B #$12,D0` the constant number 12 in hexadecimal representation ($ precedes the number) is added to the content of D0. In the instruction

```
ADD.B  #%10101010,D0
```

the binary constant $10101010_2$ is added to the content of D0 because the constant is preceded by %.

## Exercises

**3.1** A computer can address 4 Mb memory. How many address bits are required to address each memory location?

**3.2** A memory is 4 Mb. How many long words does it contain?

**3.3** At what address does the most significant byte of the long word at address $10_{16}$ reside?

**3.4** At what address does the least significant byte of the long word at address $20_{16}$ reside?

**3.5** An instruction occupies 2 words and is stored at address $8000_{16}$. What does the program counter contain when the next instruction is to be fetched?

**3.6** An instruction does the following; $(42_{16}) \vee 55_{16} \rightarrow 45_{16}$. Suppose that $(42_{16}) = AA_{16}$. What does the memory cell at address $45_{16}$ contain after execution?

**3.7** An instruction does the following; $(42_{16}) \wedge 55_{16} \rightarrow 45_{16}$. Suppose that $(42_{16}) = AA_{16}$. What does the memory cell at address $45_{16}$ contain after execution?

**3.8** An instruction does the following; $(42_{16}) + 55_{16} \rightarrow 45_{16}$. Suppose that $(42_{16}) = 55_{16}$. What does the memory cell at address $45_{16}$ contain after execution?

---

## 3.4 Summary and concluding remarks

In this section, we introduced a simple model of a computer system including the Microprocessor (processor for short), the Memory, and the I/O-system. The memory stores the program and the data to be processed. The memory is organized as a vector of memory cells, where each memory cell is identified by a number called an address.

In order for the processor to execute instructions and process data stored in the memory, the memory can be accessed by reading the content of a memory cell or modifying its content by a write operation. In order for the processor to access several consecutively stored memory cells, it can specify the number of memory cells by the Size attribute (byte, word, or long word).

The processor keeps track of the next instruction to execute by a storage element called program counter. The processor performs the conceptually simple task of fetching an instruction, updating the program counter, and then executing the instruction. This repetitive task is referred to as the instruction cycle.

High-level languages are used to specify the computation intended by the programmer. However, a program written in a HLL cannot be interpreted by a computer and must be translated into machine language instructions, which is performed by the compiler.

The instructions of a processor are binary coded and almost impossible for a human being to understand. In order to simplify the task of machine language programming, an assembly language is associated with each processor type. The assembly language instructions are semantically close to the structure of the binary coded machine language. In fact, there is usually a one-to-one correspondence between the assembly language and the machine language instructions. The assembly language not only makes it conceptually attractive to understand the operation of a computer, it also constitutes a method to write programs and take advantage of the resources of the computer in an efficient way. Because of the one-to-one correspondence, the task of translating an assembly language program to machine instructions is simple. The program that performs this task is referred to as an assembler.

Chapter 4

# Instruction Set Model

This chapter aims to give a thorough understanding of the instruction set model of the processor. We follow the approach of 'information hiding' by first looking at a small subset of the instruction set and explaining only those parts of the processor that are relevant for this subset. We then successively refine the instruction set model by introducing more instructions and more functionality of the processor. When we finish our presentation in Chapter 6, we have introduced all concepts and functional units depicted in Figure 4.1, which shows the major parts of a computer system based on the microprocessor Motorola 68000.

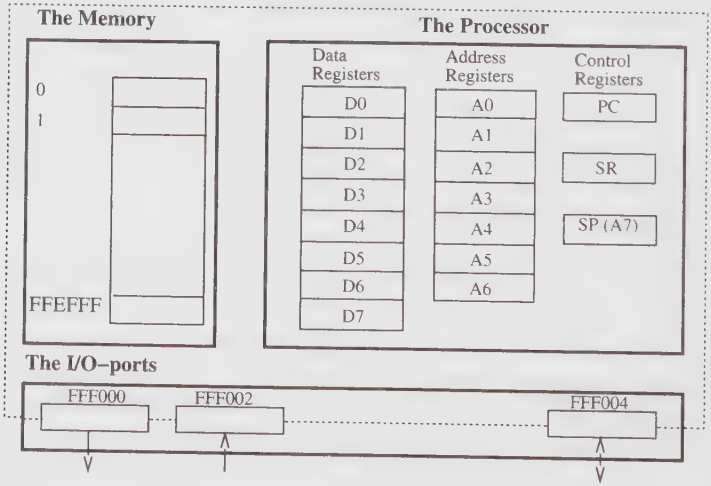



**Figure 4.1** A model of a computer system based on M68000.

## 4.1 The data register model

The Motorola 68000 (M68000) uses 24 address bits. It can thus address a memory containing at most $2^{24} = 16{,}777{,}216$ memory cells of 8 bits each. In Figure 4.2, we show the memory. Note that memory addresses are hexadecimal (address $\mathrm{FFFFFF}_{16} = 16,777,215_{10}$). We will always use hexadecimal representation for addresses.

Since a lot of computations are performed on the same data over and over again and since the memory access time is long compared to the processing speed of the processor, the processor contains a small set of high-speed memory cells denoted *registers*. Some of the registers are general purpose while other have dedicated functions. In the first model, we will introduce the general purpose registers, called *data registers*.

The processor contains eight data registers, denoted D0 to D7. The data registers are used as temporary storage for all arithmetic and logical operations. Our first model consists of the program counter (PC), the data registers, and the memory as shown in Figure 4.2. In Table 4.1, we list some instructions for arithmetic and logical operations between operands stored in memory locations and/or data registers relevant for the instruction set model so far.

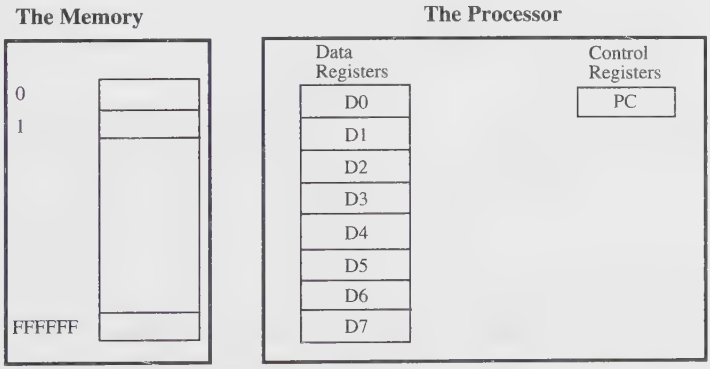



**Figure 4.2** The data register model of the M68000.

Each data register consists of 32 bits. It can thus store long words. Sometimes, however, computations need only to deal with bytes or words. Therefore, for all instructions listed in Table 4.1, there is an option to specify the size of the data to be manipulated. This is denoted by the suffix S appended to each operation word of the instruction. To denote a byte, S is replaced by B. Likewise, W and L represent word and long word manipulation, respectively.

The instructions listed in Table 4.1 can be used to load data (`MOVE`), add or subtract data (`ADD`, `SUB` (Subtract)) or perform logical operations such as logical AND (`AND`), inclusive-or (`OR`), or exclusive-or (`EOR`). Besides these binary operations (two operands), there are three unary operations (one operand) (`CLR`, `NEG`, and

NOT) which assigns zero, negates, and performs the logical inverse of an operand, respectively.

In the previous chapter we introduced three methods to refer to an operand (addressing modes): (data) register direct, absolute addressing, and immediate addressing. For the instructions in Table 4.1, the first two addressing modes can be used by replacing '$a$' by the name of a data register, or an absolute address. The immediate addressing mode is obtained by using '#' in front of a number.

**Table 4.1** Assembly instructions relevant for the data register model. S specifies the operand size (B, W, or L), $a$ denotes an absolute address or a data register name, # before a numeric value ($n$) designates a constant (immediate addressing). D$i$ denotes any of the data registers D0 to D7.

| Name | | Operation | |
|---|---|---|---|
| MOVE.S | $a_1$, $a_2$ | $(a_1)$ | $\rightarrow a_2$ |
| MOVE.S | $\#n$,$a$ | $n$ | $\rightarrow a$ |
| ADD.S | $a$,D$i$ | $(\mathrm{D}i) + (a)$ | $\rightarrow \mathrm{D}i$ |
| ADD.S | D$i$,$a$ | $(\mathrm{D}i) + (a)$ | $\rightarrow a$ |
| ADDI.S | $\#n$,D$i$ | $(\mathrm{D}i) + n$ | $\rightarrow \mathrm{D}i$ |
| SUB.S | $a$,D$i$ | $(\mathrm{D}i) - (a)$ | $\rightarrow \mathrm{D}i$ |
| AND.S | $a$,D$i$ | $(\mathrm{D}i) \wedge (a)$ | $\rightarrow \mathrm{D}i$ |
| OR.S | $a$,D$i$ | $(\mathrm{D}i) \vee (a)$ | $\rightarrow \mathrm{D}i$ |
| EOR.S | D$i$,$a$ | $(\mathrm{D}i) \oplus (a)$ | $\rightarrow a$ |
| CLR.S | $a$ | $0$ | $\rightarrow a$ |
| NEG.S | $a$ | $0 - (a)$ | $\rightarrow a$ |
| NOT.S | $a$ | $(a)'$ | $\rightarrow a$ |

The first issue to be discussed is the impact of the operand size on the execution of an instruction. Suppose that we want to perform the operation: (2) $\rightarrow$ D0 (place the content of memory cell 2 in register D0). This is carried out by the instruction:

```
MOVE.B  $2,D0
```

Note that the memory address is hexadecimal. We use the notation from the previous chapter '$' to express hexadecimal values.

Since D0 can store a long word, a natural question is to which part of D0 the content of memory cell 2 is copied. The answer is that it is copied to the least significant bits of D0. Consequently, bits 7 through 0 will contain the same value as memory cell 2. All other bits of D0 remain unaffected as shown in Figure 4.3. Likewise, if the same operation with operand size word (W) is performed, that is,

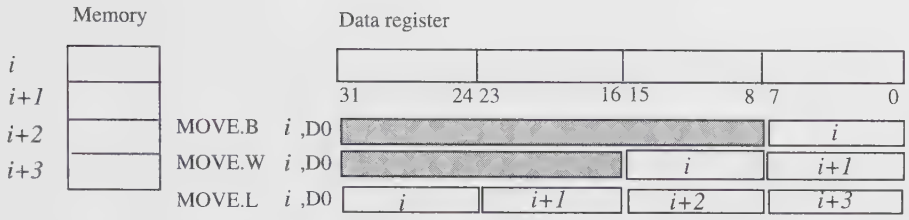



**Figure 4.3** The impact of the operand size on the result of a MOVE operation between memory and a data register.

```
MOVE.W  $2,D0
```

the content of memory cell 2 is copied to bits 15 through 8 and the content of memory cell 3 is copied to bits 7 through 0 (recall from the previous chapter how words and long words are stored at memory). Finally, if we use a long word (L), such as in

```
MOVE.L  $2,D0
```

the content of memory cell 2 is copied to bits 31 through 24, the content of memory cell 3 to bits 23 through 16, the content of memory cell 4 to bits 15 through 8, and finally, the content of memory cell 5 to bits 7 through 0 in D0. We summarize the impact of the operand size on the operation in Figure 4.3. The shaded parts of the register are not affected by the instruction.

The second issue to be discussed is the use of addressing modes. Almost all instructions listed in Table 4.1 have one thing in common, namely, they need two operands, where the second operand (the rightmost) specifies the location where the result is put. We will refer to the first operand as the *source operand*, and the second operand as the *destination operand*.

In Table 4.1, *a* denotes either the address of a memory location or the name of a data register. For instance, the operation: (D0) $\rightarrow$ D1 is performed by the instruction

```
MOVE.L  D0,D1
```

If we want to copy a constant into a data register or a memory location, that is, using immediate addressing, we put '#' before the constant as in the following example

```
MOVE.L  #$25,D1
```

where the constant $00000025_{16}$ is copied into D1 (long word operation).

The ADD-instruction performs addition according to the methods we presented in Chapter 2. From Table 4.1, we note that there are three versions of the ADD-instruction. They differ in the use of addressing modes while the first two versions either use a data register as a source or destination operand, the third version (ADDI) employs immediate addressing on the source operand. Although not listed in Table 4.1, the same addressing-mode combinations are allowed for all other arithmetic and logical instructions such as SUB, AND, and OR. For example, if the source operand is a constant, we use SUBI, ANDI, ORI, and EORI. For EOR, however, the source operand must always be a data register.

The last three instructions in Table 4.1 (CLR, NEG, and NOT) take only one operand, that is, the source and destination operand, both are the same. CLR copies zero to the operand, NEG converts an operand into its negative equivalence, and NOT performs the logical inverse of the operand.

In order to perform a specific computation, more than one instruction is usually needed. For example, if we wish to perform: (1) + (2) $\rightarrow$ 3, the following instructions are needed:

```
MOVE.B  $1,D0
ADD.B   $2,D0
MOVE.B  D0,$3
```

## Exercises

**4.1** Suppose that (D0) = $12345678_{16}$ and the contents of memory cells 0 through 3 are (0) = $87_{16}$, (1) = $65_{16}$, (2) = $43_{16}$, (3) = $21_{16}$. What is the content of D0 after execution of (a) `MOVE.B $0,D0` (b) `MOVE.W $0,D0` (c) `MOVE.L $0,D0`?

**4.2** Suppose that (D0) = $01010101_{16}$ and the contents of memory cells 0 through 3 are (0) = $87_{16}$, (1) = $65_{16}$, (2) = $43_{16}$, (3) = $21_{16}$. What is the content of D0 after execution of (a) `ADD.B $0,D0` (b) `ADD.W $0,D0` (c) `ADD.L $0,D0`?

**4.3** Suppose that (D0) = $AAAAAAAA_{16}$ and the contents of memory cells 0 through 3 are (0) = $55_{16}$, (1) = $AA_{16}$, (2) = $55_{16}$, (3) = $AA_{16}$. What is the content of D0 after execution of (a) `AND.B $0,D0` (b) `AND.W $0,D0` (c) `AND.L $0,D0`?

**4.4** What sequence of instructions performs the operation $(21F_{16}) + 25_{10} \rightarrow 2FA_{16}$, assuming 8-bit operands?

**4.5** What sequence of instructions performs the operation $(1234_{16}) - 25_{10} \rightarrow 25_{16}$, assuming 8-bit operands?

**4.6** What sequence of instructions performs the operation $(3) \vee 4 \rightarrow 3$, assuming 8-bit operands?

---

In the example programs above, we have used memory locations as variables, much like variables in high-level languages. In fact, variable names in high-level languages are translated into absolute memory addresses by the compiler. In order for this to happen, most high-level languages require that the programmer declares all the variables used by the program. The compiler then uses the declarations to substitute all symbolic variable names with absolute addresses. A nice thing about this is that one can use variable names that reflect the use of them.

Assemblers allow the programmer to use symbolic names for memory locations as long as they are declared. In the following, we will use symbolic names for memory locations. However, for the moment, we will assume that they are declared in the program somehow. We will return to how symbolic names are declared in Chapter 5.

Given that LOC is a symbolic name of a memory cell, the following instruction performs the operation (LOC) + 1 $\rightarrow$ LOC on 8-bit operands

```
ADDI.B  #1,LOC
```

We now present a somewhat more complex task. The following sequence of instructions performs the task: (VAR1) + (VAR2) $\rightarrow$ VAR3

```
MOVE.B  VAR1,VAR3
MOVE.B  VAR2,D0
ADD.B   D0,VAR3
```

Note that the program above is not a unique solution to the problem. The following sequence of instructions also performs the same task:

```
MOVE.B  VAR2,VAR3
MOVE.B  VAR1,D0
ADD.B   D0,VAR3
```

It is important to note that there are often several solutions to the same problem. In Appendix A, we will provide only one solution to the exercises although other solutions exist.

### Exercises

All operands are assumed to occupy 8 bits if nothing else is said.

**4.7** What sequence of instructions performs the operations: $(FFF_{16})+(ABC_{16}) \rightarrow ABC_{16}$ and $(FFF_{16})+(DEF_{16}) \rightarrow DEF_{16}$?

**4.8** Write a sequence of instructions that performs the operation: $(ROW) + (COL) + 1 \rightarrow MAT$?

**4.9** Write a sequence of instructions that performs the operation: $22_{10} + (LOC) \rightarrow LOC$?

**4.10** Write a sequence of instructions that performs the operation: $NUM-(VAR) \rightarrow VAR$?

**4.11** Write a sequence of instructions that performs the operations:
$1 + (NUM1) \rightarrow NUM1$
$2 + (NUM2) \rightarrow NUM2$
$3 + (NUM3) \rightarrow NUM3$?

## 4.2 Program flow control

In the example programs we have met so far, instructions are executed in the order they appear in the program. This order is maintained by the program counter (PC); PC is incremented when an instruction has been fetched so as to fetch the next instruction in the program sequence.

It is obvious that a computer would be rather useless if there were no way to alter the execution order. For example, one of the strengths of high-level languages is to express fairly complex computations concisely in terms of loops such as for-loops in Pascal. Another feature in high-level languages is the alternative execution order provided by conditionals such as if-statements.

**Table 4.2** Some compare and program control instructions.

| Name | | Operation |
|---|---|---|
| CMP.S | $a$,D$i$ | (D$i$) − ($a$) |
| CMPI.S | #$n$,$a$ | ($a$) − $n$ |
| BRA | *label* | branch to instruction at address *label* |
| BEQ | *label* | **if** (Z)=1 **then** BRA *label* |
| BNE | *label* | **if** (Z)=0 **then** BRA *label* |
| STOP | #$n$ | the execution stops after this instruction. |

In order for loops and if-statements in a high-level language program to be translated into a machine-language program, a mechanism must be provided by the processor to alter execution order. The *branch-instruction* is the most primitive instruction to accomplish this task The branch-instruction takes a symbolic memory address, called a *label*, as its operand. Its effect is to make that instruction the next one to be executed. The processor performs this task by simply loading the program counter with the label. The label is the address from which the next instruction is to be fetched.

In the following (rather useless) program, a branch is performed to label NEXT.

```
        MOVE.B  #0,D0   ; Execute this one and
        ADDI.B  #1,D0   ; this one
        BRA     NEXT    ; Branch to label NEXT
        ...             ; Skip these instructions
NEXT    ADDI.B  #1,D0   ; Continue here
        SUBI.B  #1,D0   ; and then here
```

In the example program above, a branch is always taken independent of the result of the execution. Such branch-instructions are denoted *unconditional branches* to reflect that they are always taken. In order to support high-level language features for execution alteration such as for-loops and if-statements, a branch should be taken only when a certain logical condition is satisfied. Such branch-instructions are referred to as *conditional*. There are a large number of logical conditions to test. To simplify the discussion, we will first look at conditional branch-instructions that take a branch depending on whether the result of an arithmetic or logical operation is zero. To perform this test, the processor controls a special flag (a 1-bit register), called the Z-flag (Z for Zero). This flag is set to one by the processor when the result of an operation is zero and reset to zero otherwise.

In Table 4.2, we show two conditional branch-instructions and two compare-instructions that compare two operands. The comparison is performed by subtracting the first operand from the second one. Unlike the subtract-instruction, however, the result is not stored; the only effect this instruction has is that it affects the Z-flag. This can be used to perform conditional branches as we will demonstrate in the next example program.

Consider the following computation: $N+(N-1)+\ldots+3+2+1 \rightarrow$ D0. The following program performs this task, assuming $N$ is greater than zero and that the sum never exceeds $255_{10}$:

```
        MOVE.B  #0,D0
        MOVE.B  #N,D1   ; D1 is loaded with N
LOOP    ADD.B   D1,D0   ; Add D1 to D0
        SUBI.B  #1,D1   ; Z-flag is affected
        BNE     LOOP    ; Branch to LOOP if (Z)=0
```

Note that the subtract instruction serves two purposes: (i) D1 is decremented to contain the next number to be added to D0 and (ii) the Z-flag is affected so that the loop is exited when D1 contains zero. The above program illustrates how a repetition-statement such as a for-loop can be implemented by an assembly language program. In the next example, we consider the implementation of the following if-statement: **if** (A)=(B) **then** 0 $\rightarrow$ A **else** 0 $\rightarrow$ B.

```
        MOVE.B  A,D0
        CMP.B   B,D0    ; if (A) = (B)
        BNE     ELSE
        MOVE.B  #0,A    ; then 0 -> A
        BRA     DONE
ELSE    MOVE.B  #0,B    ; else 0 -> B
DONE
```

In the example above, we have used the compare-instruction (`CMP.B B,D0`) to check whether (A) = (B). Note that this instruction only affects the Z-flag depending on the result of the subtraction (D0) − (B).

An important feature we have forgotten is a means to stop the execution of a program. One could ask what will happen when the last instruction has been executed in the example programs demonstrated so far. According to what we now know about execution order, the next instruction to be executed is the one that appears on the next consecutive address in memory. In order to stop the execution, one can use the instruction `STOP`. For instance, if we want the execution to cease when the instruction at label `DONE` is executed in the last example, we write

```
        MOVE.B  A,D0
        CMP.B   B,D0
        BNE     ELSE
        MOVE.B  #0,A
        BRA     DONE
ELSE    MOVE.B  #0,B
DONE    STOP    #$2700
```

The operand to the STOP instruction affects the internal state of the processor. For now, we will simply assume that $2700 is appropriate. In Chapter 6, we will explain this further.

### Exercises

All operands are assumed to occupy 8 bits if not stated otherwise.

**4.12** What sequence of instructions performs the computation: (NUM) + (NUM) $\rightarrow$ NUM?

**4.13** What sequence of instructions performs the computation: (NUM) + (NUM) + ... + (NUM) $\rightarrow$ NUM (8 times)?

**4.14** What sequence of instructions performs the multiplication: (M1) $*$ (M2) $\rightarrow$ P?

**4.15** Write a sequence of instructions that performs the operation: **if** ((A) = 1) **or** ((A) = 2) **then** (A) $\rightarrow$ B **else** (B) $\rightarrow$ A.

## 4.3 Arithmetic and condition codes

In Chapter 2, we learned how the computer performs arithmetic operations such as addition and subtraction. A fundamental issue was to devise a test as to whether arithmetic operations result in overflow. Most computers perform this test automatically. The result of the test is available in a dedicated control register denoted the *condition code register* (CCR). This register can be read by move instructions. More importantly, there are conditional branch-instructions that change the control flow depending on how the bits in this register are set. Besides overflow, the CCR also indicates other properties of the result.

In the M68000, the condition code register comprises five flags that are stored in the five least significant bits of the *status register* (SR) as shown in Figure 4.4.

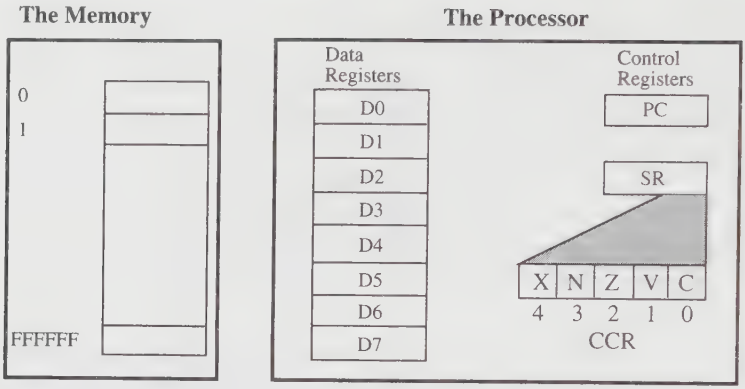



**Figure 4.4** The status register (SR) and the condition code register (CCR) of the M68000.

The status register contains 16 bits. We will examine the other bits later. In this section, we will be concerned with how the different *flags* in the condition code register are affected by arithmetic and logical operations, and how we can use conditional branch instructions to change the control flow when the result of the operation has special properties such as overflow, zero, or negative. The condition code register (bit 0 through 4 in the status register) contains five flags denoted X, N, Z, V, and C as shown in Figure 4.4.

In Table 4.3, we show all conditional branch-instructions and their branch conditions. We have already introduced the Z(ero)-flag, which is set when the result of an arithmetic or logical operation is zero. We have also introduced the conditional branch-instruction `BEQ` *label* that takes a branch provided that the Z-flag is set ((Z)=1) and `BNE` *label* that takes a branch provided that the Z-flag is cleared ((Z)=0). In Table 4.3, all branch conditions are presented as logical expressions: for instance, the logical expression $[(N) \wedge (V)'] \vee [(N)' \wedge (V)]$ is true, and the branch is taken, provided that: [(N)=1 AND (V) = 0] OR [(N)=0 AND (V) = 1].

### 4.3.1 Conditions for unsigned integer arithmetic

In Chapter 2, we noted that the overflow test differs depending on the representation of numbers. Given n-bit numbers, we could represent all numbers in the range $[0, 2^n - 1]$, that is, unsigned number representation.

Addition of two numbers in the unsigned representation results in overflow if and only if the carry-bit from the most significant position is one. The C(arry)-flag in the CCR reflects this. It can thus be used to test for overflow of unsigned number arithmetic as in the following example:

**Table 4.3** Conditional branch-instructions and the their branch conditions.

| Name | | Branch condition |
|---|---|---|
| BEQ | *label* | (Z) |
| BNE | *label* | (Z)' |
| BCS | *label* | (C) |
| BCC | *label* | (C)' |
| BHI | *label* | (C)' $\wedge$ (Z)' |
| BLS | *label* | (C) $\vee$ (Z) |
| BMI | *label* | (N) |
| BPL | *label* | (N)' |
| BVS | *label* | (V) |
| BVC | *label* | (V)' |
| BGT | *label* | (Z)' $\wedge$ [[(N) $\wedge$ (V)] $\vee$ [(N)' $\wedge$ (V)']] |
| BGE | *label* | [(N) $\wedge$ (V)] $\vee$ [(N)' $\wedge$ (V)'] |
| BLT | *label* | [(N) $\wedge$ (V)'] $\vee$ [(N)' $\wedge$ (V)] |
| BLE | *label* | (Z) $\vee$ [(N) $\wedge$ (V)'] $\vee$ [(N)' $\wedge$ (V)] |

```
            ADD.L   A,D0
            BCS     OVERFLOW
            ...
OVERFLOW    ...
            ...
```

Now consider the subtraction of two n-bit numbers $A - B$ that both belong to the range $[0, 2^n - 1]$. This subtraction results in overflow if and only if $B > A$. When subtracting two unsigned numbers (using the SUB-instruction), the C-flag is set when overflow occurs.

There are two useful conditional branch-instructions that in conjunction with the compare-instruction CMP.S A,D0 can test the relation between (A) and (D0). These are BHI and BLS. Consider the test 'branch if (D0) > (A)'. This test is equivalent to 'branch if (D0)−(A) > 0'. According to what has been said about overflow for unsigned numbers under subtraction, the branch should be taken if and only if (C)= 0 and (Z)= 0 which expressed as a logical expression is (C)' $\wedge$ (Z)'. This is exactly the test performed by the BHI-instruction (see Table 4.3). Thus,

```
        CMP.L   A,D0
        BHI     GREAT
        ...
GREAT   ...
```

implements the test. If we replace `BHI` by `BLS`, we test the opposite relation.

### 4.3.2 Conditions for signed integer arithmetic

We shall now turn our attention to two's complement number arithmetic. The N(egative)-flag reflects the most significant bit of the result of an operation, and consequently, the sign of the result when arithmetic operations are performed on two's complement numbers. In the following example program, execution continues at label `MINUS` if the result of `ADD.L A,D0`, that is, the content of D0, is negative:

```
        ADD.L   A,D0
        BMI     MINUS
        ...
MINUS   ...
        ...
```

The condition for overflow when two two's complement numbers are added is that the signs of the operands are the same and opposite to the sign of the result. This test is performed by the processor and the result is obtained from the V-flag (oVerflow-flag) in the CCR. For instance, in the following example program, execution continues at label `OVERFLOW` if addition of two two's complement numbers results in overflow.

```
        ADD.L   A,D0
        BVS     OVERFLOW
        ...
OVERFLOW ...
        ...
```

Besides taking a branch when the result is zero, we must also be able to take a branch based on the other relational operators $< 0, \leq 0, \geq 0$, and $> 0$. For two's complement arithmetic, the corresponding conditional branch-instructions are `BLT`, `BLE`, `BGE`, and `BGT`. Note that the names reflect their operation: `BLT` is an abbreviation of 'Branch Less Than zero'.

These branch-instructions can be exclusively used for testing the result of two's complement number arithmetic, which their branch conditions reveal. For instance, the branch condition for the `BGE`-instruction is [(N)=1 AND (V)=1] OR [(N)=0 AND (V)=0]. Assume that two positive two's complement numbers $A$ and $B$ are added and that the addition results in overflow. Then (V)=1 and the sign of the result is opposite to the operands, that is, (N)=1. It means that although the addition resulted in overflow, the test will be correct.

### 4.3.3 Extending the range beyond long words

Besides using the C-flag as a test for overflow for unsigned number arithmetic, it can also be used to extend arithmetic operations to 64-bit numbers. In this case, we must use two long words to store an operand. For instance, let us assume that two 64-bit numbers are to be added. The first number is stored with its most significant long word in register D1 and the least significant long word in D0. The second number is stored in D3 (most significant long word) and D2 (least significant long word). The addition of these numbers can be performed by first adding the least significant long words. We then add the most significant long words and the carry-bit (if any) from the addition of the least significant long words.

While the carry-bit is affected by (almost) all instructions, there is another flag, the X-flag (eXtended-flag), which is set according to the same rules as the C-flag. But, unlike the C-flag, it is only affected by arithmetic instructions such as `ADD` and `SUB` and shift instructions (to be presented in the next section). In the example below, we implement 64-bit addition by making use of the instruction `ADDX.L` D$i$, D$j$ which performs (D$i$) + (D$j$) + (X) $\rightarrow$ D$j$.

```
ADD.L   D0,D2
ADDX.L  D1,D3
```

Likewise, there is a corresponding subtraction instruction, `SUBX.L` D$i$,D$j$ which performs (D$j$)−(D$i$)−(X) $\rightarrow$ D$j$. Consequently, the following sequence of instructions performs 64-bit subtraction.

```
SUB.L   D0,D2
SUBX.L  D1,D3
```

For both these examples, the result is available in registers D3 (most significant long word) and D2 (least significant long word). Note that the scheme presented can be extended to operands of any size.

**Table 4.4** The general conditions for setting the flags in the CCR.

| Flag | Condition |
|---|---|
| Z | Set when the result is zero. |
| C and X | Set when carry/borrow is generated. |
| N | Set when the result is negative. |
| V | Set when a two's complement operation results in overflow. |

We end this section by pointing out some general guidelines for the use of the condition codes. In general, **all instructions affect the condition code register**. It is therefore important to be aware of how they are affected. In this section, we have pointed out their general behavior which we summarize in Table 4.4. It should be noted, however, that there are exceptions. Therefore, it is important for the programmer to examine how each individual instruction affects the condition codes. In Appendix B, we show how each instruction affects the condition code register.

## Exercises

**4.16** Assuming 32-bit unsigned integers A and B, write a sequence of instructions that implements the following if-statement
**if** (A) > (B) **then** 0 → A **else** 0 → B.

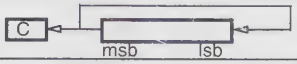
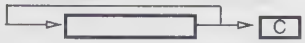
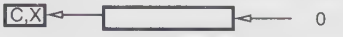
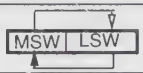
**4.17** Assuming 32-bit unsigned integers A and B, write a sequence of instructions that implements the following if-statement
**if** (A) <= (B) **then** 0 → A **else** 0 → B.

**4.18** Assuming 32-bit signed integers A and B, write a sequence of instructions that implements the following if-statement
**if** (A) > (B) **then** 0 → A **else** 0 → B.

**4.19** Assuming 32-bit signed integers A and B, write a sequence of instructions that implements the following if-statement
**if** (A) <= (B) **then** 0 → A **else** 0 → B.

**4.20** Write a sequence of instructions that performs 128-bit addition: (A) + (B) → B. A is stored in D0, D1, D2, and D3 with the least significant 32 bits in D3 and B is stored in D4, D5, D6, and D7 with the least significant 32 bits in D7.

**Table 4.5** Shift instructions for the M68000.

| | |
|---|---|
| ROL.S Di,Dj<br>ROL.S #n,Di<br>ROL a |  |
| ROR.S Di,Dj<br>ROR.S #n,Di<br>ROR a |  |
| LSL.S Di,Dj<br>LSL.S #n,Di<br>LSL a |  |
| LSR.S Di,Dj<br>LSR.S #n,Di<br>LSR a |  |
| ASL.S Di,Dj<br>ASL.S #n,Di<br>ASL a |  |
| ASR.S Di,Dj<br>ASR.S #n,Di<br>ASR a |  |
| SWAP Di |  |

**4.21** Write a sequence of instructions that performs 128-bit subtraction (B)−(A) → B. A is stored in D0, D1, D2, and D3 with the least significant 32 bits in D3 and B is stored in D4, D5, D6, and D7 with the least significant 32 bits in D7.

---

## 4.4 Shift instructions

We are now able to write simple assembly language programs that perform arithmetic and logical operations on operands stored in memory locations or data registers. The move-instructions we have introduced help us to copy data between locations. The least amount of data to be copied is a byte. It is sometimes useful, however, to move bits within a location. This is exactly what the shift instructions perform.

In Table 4.5, we show the shift instructions. These instructions have in common that they move bits either to the left or to the right within the destination operand. The number of steps, called the *shift count*, is specified by the source operand which can be a data register or a constant using immediate addressing ($n$). The destination operand (the operand to be affected) is a data register. The range of allowed shift counts differs; for immediate addressing, the shift count is in the range $[1,8]$ and if a data register is used, the shift count is in the range $[1,63]$. There is also a third variation of the shift instructions using a memory location ($a$ in Table 4.5). This instruction shifts the operand in memory beginning at location $a$ one step. The implicit operand size in the last case is always a word.

In order to specify exactly what the shift-instructions perform, let us denote the operand $X = x_{n-1}x_{n-2}\ldots x_0$. The rotate-instructions shift the operand so that the bits that are shifted out in either end will appear in the other. Given a shift count of $d$, the left-rotate instruction (`ROL`) changes the operand so that $x_i = x_j$, where $j = i - d \pmod{n}$ (recall the meaning of the mod-operation from Chapter 1), and the right-rotate instruction (`ROR`) changes the operand so that $x_i = x_j$, where $j = i+d \pmod{n}$, for $i = 0, 1, \ldots, n-1$. Note that the C-flag is assigned the value of $x_{n-d}$ for the left-rotate and $x_{d-1}$ for the right-rotate instruction, respectively (see Table 4.5). For instance, if (D0) = $0F00_{16}$, the result of the execution of `ROL.W #5,D0` is (D0) =$E001_{16}$ and the C-flag is set. Note that the four ones stored in bits 8 11 are moved 5 steps to the left which means that bits 15 13 and 0 are set after execution.

While the rotate-instructions establish a connection between the most and least significant bits, the other shift-instructions simply move bits out of the most or least significant end of the operand. The logical shift-instructions (`LSL` and `LSR`) replace the empty bits in the least significant bit-field (`LSL`) or the most significant bit-field (`LSR`) by zeros. The C- and X-flags are assigned the last bit that was shifted out. For example, suppose that the least significant 16 bits of D0 are $0001000000000000_2$, then the instruction `LSL.W #4,D0` sets the C- and X-flags because bit 12 is set.

The motivation behind the operation of arithmetic shift-instructions lies in the fact that they can facilitate multiplication and integer division by a number which is a power of two. For instance, multiplication by two can be performed by a left-shift operation: $2_{10} \times 2_{10} = 0010_2 \times 0010_2 = 0100_2$. Likewise, division by 2 can be implemented by a right-shift operation: $4_{10}/2_{10} = 0100_2/0010_2 = 0010_2$. In order to correctly handle the sign bit for two's complement numbers, it becomes essential to fill the most significant bit positions with the sign bits when performing a right-shift: $-4_{10}/2_{10} = 1100_2/0010_2 = 1110_2$. This process is called *sign extension* and is automatically performed by the arithmetic-right-shift instruction `ASR`.

The arithmetic-shift-instructions (`ASL` and `ASR`) resemble the operation of the logical shift-instructions. The difference between the arithmetic and logical left-shift-instructions is that the arithmetic-left-shift instruction (`ASL`) also affects the V-flag; the V-flag is set iff the most significant bit (which is the sign bit) is changed during the shift operation. The arithmetic right-shift operation (`ASR`) copies the sign bit in each step.

The last instruction in Table 4.5 (`SWAP`) takes a data register as an operand and swaps the most significant word (MSW) with the least significant word (LSW).

## Exercises

---

**4.22** What instruction rotates the content of D0 five steps to the right? Assume a 32-bit operand.

**4.23** What instruction rotates the content of D0 two steps to the left? Assume an 8-bit operand.

**4.24** What instruction rotates the word stored in memory at location NUM one step to the right?

**4.25** What shift instruction can be used to implement integer division by two of a memory word stored at location NUM?

**4.26** What shift instruction can be used to implement multiplication by two of a memory word stored at location NUM?

---

## 4.5 Indirect addressing

We have now introduced a set of instructions so that we can construct simple programs that can perform computations using repetition by means of unconditional and conditional branch-instructions. However, we have yet only seen a few methods of how to refer to operands; either the operand explicitly is contained in the instruction (immediate addressing), or the location of the operand explicitly is contained in the instruction (absolute or register direct addressing). These methods are examples of addressing modes. As we will see in the next example, these addressing modes are not sufficient.

```
SUM := 0;
for i:=0 to 4 do
    SUM := SUM + VEC[ i ];
```

The above program computes the sum of the vector elements contained in vector `VEC`. A naive attempt to construct a machine language program to perform this task would be the following one, assuming that the first vector element is stored at address `VEC` and that each vector element occupies one byte:

```
MOVE.B  #0,D0
ADD.B   VEC,D0
ADD.B   VEC+1,D0
ADD.B   VEC+2,D0
ADD.B   VEC+3,D0
ADD.B   VEC+4,D0
MOVE.B  D0,SUM
```

One realizes that if the number of vector elements is large, the corresponding program using absolute addressing becomes prohibitively large. We would like to

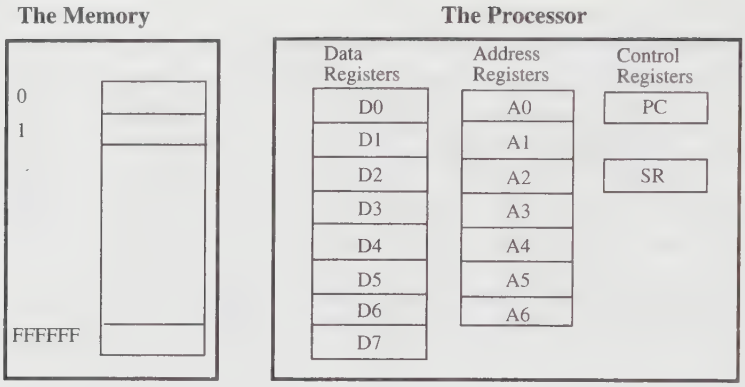



**Figure 4.5** The address register model of the M68000.

**Table 4.6** Assembly language instructions relevant for the address register model. S specifies the operand size, $a$ denotes an absolute address or a register name, # before a numeric value ($n$) designates a constant (immediate addressing). A$i$ denotes any of the address registers A0 to A6.

| Name | | Operation | |
|---|---|---|---|
| `MOVEA.S` | $a$, A$i$ | $(a)$ | $\rightarrow$ A$i$ |
| `MOVEA.S` | #$n$, A$i$ | $n$ | $\rightarrow$ A$i$ |
| `ADDA.S` | $a$, A$i$ | $(\text{A}i) + (a)$ | $\rightarrow$ A$i$ |
| `SUBA.S` | $a$, A$i$ | $(\text{A}i) - (a)$ | $\rightarrow$ A$i$ |
| `CMPA.S` | $a$, A$i$ | $(\text{A}i) - (a)$ | |

code the program in a loop so that the address of the vector changes in each loop iteration. We will now look at some advanced addressing modes to accomplish this.

The M68000 contains seven dedicated registers, called *address registers*, denoted A0 to A6, that are used to store operand addresses. In Figure 4.5, we introduce these registers and in Table 4.6, we present some instructions that manipulate the contents of address registers. Only Word (W) and Long Word (L) are valid Size attributes for these instructions. Note that the source operands denoted $a$ in the instructions in Tables 4.1, 4.2, and 4.5 in general can be replaced by an address register.

Given the address registers, we are now able to code the above program using a loop.

```
        MOVEA.L  #VEC,A0   ; Address of VEC[ 0 ] to A0
        MOVE.B   #0,D0     ; SUM := 0
        MOVE.B   #5,D1     ; for i:=0 to 4 do
LOOP    ADD.B    (A0),D0   ; SUM := SUM + VEC[ i ]
        ADDA.L   #1,A0
        SUBI.B   #1,D1
        BNE      LOOP
        MOVE.B   D0,SUM
```

We make the following important observations on the use of A0. First, A0 is initialized to contain the address of the first vector element using the instruction `MOVEA.L #VEC,A0`. Note that we load the address of `VEC[0]` as a 32-bit constant into A0. This is important, since addresses consist of 24 bits. Consequently, using a 16 bit operand size would not suffice. Second, we use the instruction `ADD.B (A0),D0` to add a vector element to D0 (the temporary sum is stored in D0) by using the content of A0 as the address of the operand. This is a new addressing mode denoted *address register indirect addressing*. Using our notation, we can express the operation performed by `ADD.B (A0),D0` as ((A0)) + (D0) $\rightarrow$ D0. Note that the content of the location whose address is contained in A0 is expressed as ((A0)). Third, in order to point to the next vector element, we must add one to the content of A0, which is done by the instruction `ADDA.L #1,A0`. If each vector element consisted of a word, we would have incremented A0 by 2 and if each vector element consisted of a long word, we would have incremented A0 by 4.

In fact, incrementing or decrementing the content of an address register in conjunction with the use of indirect addressing is so common that the designers of the M68000 have combined the increment and decrement operation with the indirect addressing mode as in the example

```
MOVE.B  (A0)+,D0
```

which performs the same operation as

```
MOVE.B  (A0),D0
ADDA.L  #1,A0
```

This addressing mode is called *address register indirect with postincrement*. The amount by which the address register is incremented is determined by the operand size. In the example above, the operand size is Byte and the address register is incremented by 1. If we would have used the instruction `MOVE.W (A0)+,D0`, the address register would have been incremented by 2, because each operand occupies

one word. Finally, if the operand size would have been a long word, the address register would have been incremented by 4. We now show the same program again using the optimization provided by this addressing mode.

```
        MOVEA.L  #VEC,A0
        MOVE.B   #0,D0
        MOVE.B   #5,D1
LOOP    ADD.B    (A0)+,D0
        SUBI.B   #1,D1
        BNE      LOOP
        MOVE.B   D0,SUM
```

It is also possible to traverse the vector in the reverse order so that the next vector element to be accessed is the one that appears at the next lower address:

```
SUBA.L  #1,A0
MOVE.B  (A0),D0
```

There is a shorthand form for this computation that does the same, which is illustrated by

```
MOVE.B  -(A0),D0
```

This addressing mode is called *address register indirect with predecrement*. As with the postincrement addressing mode, the amount by which the address register is decremented is determined by the operand size. Note that the address register is decremented **before** the operand is accessed. One would think that we could place the minus sign after (A0) in order to decrement the address register after the operand is accessed. This is not possible and the reason is that the designers did not prioritize this possibility. Likewise, it is not possible to place the plus sign in front of (A0) when the increment addressing mode is used.

There are other useful addressing modes which we will describe before we close this section. Consider the following Pascal program:

```
SUM := 0;
for i:=0 to N-1 do
    SUM := SUM + VEC[ i + N ] - VEC[ i ];
```

This program calculates the accumulated difference between two subvectors of size $N$ stored in VEC. In each iteration, the difference of the two vector elements that

appear $N$ elements apart is calculated. The following program demonstrates the use of an efficient addressing mode to refer to the vector elements.

```
        MOVEA.L  #VEC,A0
        MOVE.B   #0,D0
        MOVE.B   #N,D2
LOOP    MOVE.B   N(A0),D1   ; VEC[ i + N ] -> D1
        SUB.B    (A0)+,D1   ; (D1) - VEC[ i ] -> D1
        ADD.B    D1,D0      ; (D0) + (D1) -> D0
        SUBI.B   #1,D2
        BNE      LOOP
        MOVE.B   D0,SUM
```

In order to access `VEC[ i+N ]`, we have used yet another addressing mode called *address register indirect with displacement*. The instruction `MOVE.B N(A0),D1` performs $(N + (A0)) \rightarrow D1$. The displacement `N` is a constant value to be added to the address register before the operand is accessed. It is stored as a 16-bit two's complement number which means that the displacement can be in the range $[-32768, 32767]$.

Instead of adding a constant value to the address register, it may sometimes be convenient to add the content of another register which we call the *index register*. This is useful if we want to perform the following computation:

```
SUM := 0;
for i:=0 step 8 to K do
   SUM := SUM + VEC[ i ];
```

This program can be implemented by

```
        MOVEA.L  #VEC,A0
        MOVE.L   #0,D0          ; SUM := 0
        MOVE.B   #0,D1          ; i := 0
LOOP    ADD.B    0(A0,D1),D0    ; SUM := SUM + VEC[ i ]
        ADDI.B   #8,D1          ; i := i + 8
        CMPI.B   #K,D1
        BLE      LOOP
```

The instruction `ADD.B 0(A0,D1),D0` uses register D1 as an index register to access the operand at location 0 + (D1) + (A0). This addressing mode is called *address register indirect with index*. The instruction `ADD.B 0(A0,D1),D0` performs $(0 + (A0) + (D1)) \rightarrow D0$. Any of the data or address registers can be used as index registers. However, the displacement (in this case 0) is an 8-bit two's complement number (range $[-128, 127]$) when used in conjunction with index registers.

Instead of using an address register, it may be convenient to refer to an operand relative to the location of the next instruction to be executed. This location is pointed out by the program counter (PC). There are two addressing modes which use the program counter; the *program counter indirect with displacement* and the *program counter indirect with index* addressing mode. We exemplify the first addressing mode with the instruction

```
MOVE.B  20(PC),D1
```

which copies the content at location 20 + (PC) to D1. The following instruction

```
MOVE.B  20(PC,D0),D1
```

copies the content at location 20+(PC)+(D0) to D1. We summarize all addressing modes we have described in Table 4.7. We also provide examples of their use by means of the generic move instruction `MOVE.B` *a*,`D0`.

In Tables 4.1, 4.2, 4.5, and 4.6 we have introduced a number of instructions that are relevant for the model of the 68000 depicted in Figure 4.5. All addressing modes found in Table 4.7 can generally be applied to designate the source and destination operands by replacing *a* in the instruction tables we have presented by any of the addressing modes for the M68000. Unfortunately, however, there is no simple rule for which addressing modes are applicable to a specific instruction. Consequently, therefore, one should consult the detailed information about the available addressing modes for each instruction. This information is provided in Appendix B.

Note the correspondence between the assembly language syntax and the notation. In general, the rule for the assembly syntax is to point out the address (or register name) for the operand. For instance, the address of an operand pointed to by indirect addressing is $(Ai)$, that is, the content of register $Ai$ which is exactly what the notation says.

## Exercises

**4.27** Write a program that performs the following operation, using indirect addressing with postincrement and assuming 16-bit operands: $(100_{16}) + (102_{16}) + (104_{16}) \rightarrow 106_{16}$.

**4.28** Generalize the previous program to compute: $(100_{16}) + (102_{16}) + \ldots + (100_{16} + 2N) \rightarrow 100_{16} + 2N + 2$, where $1 \leq N \leq 255_{10}$.

**4.29** Rewrite the same program so that it uses the address register indirect with displacement addressing mode.

**Table 4.7** The addressing modes for the M68000.

| Addressing mode | Example | Notation | |
|---|---|---|---|
| Data register direct | `MOVE.B D1,D0` | (D1) | → D0 |
| Address register direct | `MOVE.L A0,D0` | (A0) | → D0 |
| Absolute | `MOVE.B 1,D0` | (1) | → D0 |
| Immediate | `MOVE.B #1,D0` | 1 | → D0 |
| Indirect | `MOVE.B (A0),D0` | ((A0)) | → D0 |
| Indirect with postincrement | `MOVE.B (A0)+,D0` | ((A0)) | → D0; |
| | | (A0) + 1 | → A0 |
| Indirect with predecrement | `MOVE.B -(A0),D0` | (A0) − 1 | → A0; |
| | | ((A0)) | → D0 |
| Address register indirect with displacement | `MOVE.B 10(A0),D0` | ((A0) + 10) | → D0 |
| Address register indirect with index | `MOVE.B 10(A0,D1),D0` | ((A0) + (D1) + 10) | → D0 |
| Program counter indirect with displacement | `MOVE.B 10(PC),D0` | ((PC) + 10) | → D0 |
| Program counter indirect with index | `MOVE.B 10(PC,D1),D0` | ((PC) + (D1) + 10) | → D0 |

**4.30** Rewrite the same program so that it uses the address register indirect with index addressing mode.

## 4.6 Subroutines

We have seen how to capture repeating sequences of operations by using conditional branches to construct loops. But there are other kinds of common patterns for which the loop concept is not sufficient. Consider the following computation:

$2^{(A)} + 2^{(B)} + 2^{(C)} \rightarrow$ RESULT. This could be accomplished by the following sequence of instructions, ignoring the overflow that might occur:

```
MOVE.L  A,D1
MOVE.L  #1,D0
ASL.L   D1,D0       ; Shift (A) times
MOVE.L  D0,RESULT

MOVE.L  B,D1
MOVE.L  #1,D0
ASL.L   D1,D0       ; Shift (B) times
ADD.L   D0,RESULT

MOVE.L  C,D1
MOVE.L  #1,D0
ASL.L   D1,D0       ; Shift (C) times
ADD.L   D0,RESULT
```

The code above contains three sequences of instructions that are exactly identical, namely, the instruction sequence that computes two to the power of an operand contained in register D1. What we would like to do is to write this code segment only once and somehow make a reference to it so that it can be 'called' from various places in the program. There are many important reasons why we should not repeat the code of this computation. First, in order to reuse the same code over and over again and second, it occupies less space in memory. What we want to achieve is a way to support what in high-level languages are referred to as a *subroutine* or a *procedure*, that is, a piece of code that can be called at various places in the program. There are two fundamental mechanisms needed to support subroutines; a subroutine call and a subroutine return mechanism.

The calling and returning mechanism on the M68000 is supported by the instructions `BSR` and `RTS`, see Table 4.8. The `BSR` instruction is like a `BRA` instruction except that the processor 'remembers' where the call was made. The `RTS` instruction is like a `BRA` instruction except that its argument is implicitly the address of the instruction following the corresponding `BSR` instruction. For now, we will ignore how these mechanisms are implemented in the processor. We will return to this issue in Chapter 6.

By using the subroutine call and return instructions, we can rewrite the previous program as follows:

```
MOVE.L  A,D1
BSR     POW2
MOVE.L  D0,RESULT
```

**Table 4.8** The call and return instructions.

| Name | Operation |
|---|---|
| BSR *label* | BRA *label*<br>remember return address |
| RTS | BRA *return address* |

```
        MOVE.L  B,D1
        BSR     POW2
        ADD.L   D0,RESULT
        MOVE.L  C,D1
        BSR     POW2
        ADD.L   D0,RESULT

POW2    MOVE.L  #1,D0
        ASL.L   D1,D0
        RTS
```

## Exercises

**4.31** Write a subroutine that determines which of the two 32-bit unsigned integers in D0 and D1 that are largest. D0 is assigned the largest of these numbers.

**4.32** Use the subroutine in the previous exercise to write a program that determines the largest of all vector elements contained in vector VEC[$i$] , $i = 0, 1, 2, \ldots, N-1$, where $N$ all vector elements are 32-bit unsigned numbers.

**4.33** Write a subroutine that computes the integer division $(D0)/2^{(D1)} \rightarrow D0$, where D0 contains a 32-bit operand. Hint: Use an arithmetic shift-instruction.

**4.34** Use the subroutine in the previous exercise to compute VEC[0]+VEC[1]$/2^1 + \ldots$VEC[N]$/2^N$. All vector elements contain 32 bits and the result should be available in register D2.

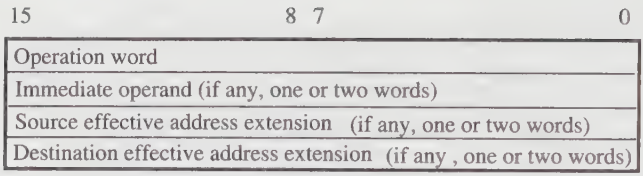



**Figure 4.6** Instruction format for M68000.

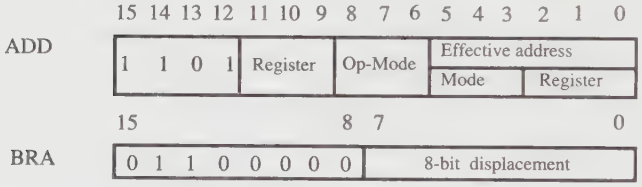



**Figure 4.7** Operation word of the `ADD` and `BRA` instructions.

## 4.7 Instruction format and coding

In this chapter, we have been concerned with the operation of the processor at the instruction set level. In order to simplify the discussion, we used the assembly language notation to describe the semantics of the instruction set of the M68000. This is not the true picture, however, when it comes to the questions on how the machine language program is stored in memory, and how the processor fetches the next instruction to be executed.

It was mentioned in Chapter 3 that the assembly language instructions provide a one-to-one correspondence to the binary-coded machine language instructions. Therefore, the process of translating an assembly language program into a sequence of machine instructions is conceptually simple. In this section, we will look at how the M68000 instructions are coded to get an idea of how a machine language program, in general, is coded.

Each M68000 instruction makes up one to five consecutive words in memory. The first word, the *operation word*, specifies the instruction and the addressing modes to be used. The additional four words (if any), which are stored at the next higher addresses, keep information about immediate operands, and/or absolute addresses as shown in Figure 4.6.

We shall look more closely at the instruction coding scheme by considering the `ADD` and `BRA` instructions in detail. These instructions will give a general idea of the kind of information stored in the operation word and in the subsequent words that accompany the instruction. The operation words of the `ADD` and `BRA` instructions are shown in Figure 4.7. Let us first look at the `ADD` instruction.

Bits 15–12 in the operation word of the `ADD` instruction is always 1101. This is the operation code of the instruction. The additional information in the operation

**Table 4.9** The op-mode field of the ADD instruction. Bit 8 specifies whether the data register is a source or destination operand and bits 7 and 6 specify the operand size.

| 8 | 7 | 6 | Description |
|---|---|---|---|
| 0 | – | – | Data register is destination operand |
| 1 | – | – | Data register is source operand |
| – | 0 | 0 | Byte operation |
| – | 0 | 1 | Word operation |
| – | 1 | 0 | Long word operation |

**Table 4.10** The effective address field of the ADD instruction. † These addressing modes are allowed only when specifying source operands.

| Mode | Register | Description |
|---|---|---|
| 000 | Data reg. number | Data register direct† |
| 111 | 100 | Immediate† |
| 111 | 001 | Absolute |
| 001 | Addr. reg. number | Address register direct† |
| 010 | Addr. reg. number | Address register indirect |
| 011 | Addr. reg. number | Address register indirect with postincrement |
| 100 | Addr. reg number | Address register indirect with predecrement |
| 101 | Addr. reg. number | Address register indirect with displacement |
| 110 | Addr. reg. number | Address register indirect with index |
| 111 | 010 | Program counter indirect with displacement† |
| 111 | 011 | Program counter indirect with index† |

word specifies the variations of the ADD instruction allowed. As can be seen from Table 4.1, one of the operands in the ADD instruction is always a data register. Bits 11 9 designates one of the eight data registers. For instance 000 designates D0 and 101 designates D5. From Table 4.1, we also note that there are two forms of the ADD instruction, namely, ADD.S *a*,D*i* and ADD.S D*i*,*a*. The first form uses the data register as a destination operand while the second one uses the data register as a source operand. This choice and the Size attribute are encoded by bits 8–6, according to Table 4.9.

Bits 5 0 specify the addressing mode of the operand denoted *a* in Table 4.1 according to Table 4.10. It is referred to as the effective address.

Let us look at some examples:

| Instruction | 15–12 | 11–9 | 8–6 | 5–3 | 2–0 | Hex. |
|---|---|---|---|---|---|---|
| `ADD.W A3,D5` | 1101 | 101 | 001 | 001 | 011 | DA4B |
| `ADD.B D0,D1` | 1101 | 001 | 000 | 000 | 000 | D200 |
| `ADD.L D2,(A1)` | 1101 | 010 | 110 | 010 | 001 | D591 |
| `ADD.W D5,-(A4)` | 1101 | 101 | 101 | 100 | 100 | DB64 |

The instructions coded above have in common that they only need the operation word to exactly specify what is to be performed. Some of the addressing modes need additional information. For instance, if we use immediate or absolute addressing, the constant and the address must accompany the instruction which is done in one or a sequence of words that are stored at the next higher addresses. Below we present some more examples:

| Instruction | 15–12 | 11–9 | 8–6 | 5–3 | 2–0 | Hex. |
|---|---|---|---|---|---|---|
| `ADD.W #1,D5` | 1101 | 101 | 001 | 111 | 100 | DA7C |
| | | | | | | 0001 |
| `ADD.L #$12345678,D1` | 1101 | 001 | 010 | 111 | 100 | D2BC |
| | | | | | | 1234 |
| | | | | | | 5678 |
| `ADD.L $F234,D2` | 1101 | 010 | 010 | 111 | 001 | D4B9 |
| | | | | | | 0000 |
| | | | | | | F234 |

Note that the constant value to be added in the instruction `ADD.L #$12345678,D1` is a long word. It is stored with the 16 most significant bits first and the 16 least significant bits last. This instruction comprises three words. It is also noticeable that the instruction `ADD.L $F234,D2` stores its absolute address as a long word with the most significant bits (all zeros) in the word immediately following the instruction and the least significant bits in the next word.

The address and program counter indirect addressing modes which use displacement and/or index registers also need some additional information. This information is kept in the word immediately following the operation word (at the next higher address).

We shall now turn our attention to how the `BRA` instruction is coded. The operation word appears in Figure 4.7. The branch-instruction takes a label as an operand. The label is simply a memory address at which the next instruction is to be fetched. Since the execution time of an instruction depends on the number of words that the instruction makes up, a primary objective for the designers of a computer is to minimize the number of words that an instruction occupies. Although it could have been possible to code the branch-instruction as an operation word accompanied by a long word which denotes the address of the next instruction, a fundamental property steered the designers to store the branch address in another way.

Branch-instructions are mostly used to implement repetitive high-level language constructs such as for- and while-loops in Pascal. Since such loops do not contain

many statements (it is unlikely that a for-loop covers several pages in a program listing), the number of memory cells that store the loop is a fairly small number. Therefore, another approach of storing the branch address is to store a constant, the *displacement* to be added to the current value of the program counter, as in the program below:

```
8000   ADD.B   D0,D1      D200
8002   ADD.B   D1,D1      D201
8004   ADD.B   D2,D1      D202
8006   BRA     $8000      60F8
8008   ADD.B   D3,D1      D203
```

In this example, we show the machine code of each instruction to the right. The branch-instruction at address $8006_{16}$ is coded $60F8_{16}$ according to Figure 4.7 and we shall now explain why. When the branch instruction is executed, the program counter has been updated to contain the address of the next instruction to be executed, that is, $(PC) = 8008_{16}$. Consequently, $-8$ should be added to PC in order to take a branch to address $8000_{16}$. The operation word of the `BRA` instruction contains an 8-bit displacement which designates a number in the range $[-128, 127]$ (8-bit two's complement representation). Consequently, the machine code of the instruction `BRA $8000` in the example above is $60F8_{16}$, since $F8_{16} = -8_{10}$ in 8-bit two's complement representation.

The conditional branch-instructions found in Table 4.3 and the subroutine call instruction `BSR` use similar coding schemes. A natural question that arises is how to translate a branch-instruction that needs a displacement larger than the range permits. M68000 allows displacements in the range $[-32768, 32767]$ by using an extra word after the operation word. However, the question remains what to do if this displacement is not sufficient. To solve this problem, there is an unconditional branch-instruction `JMP address` that takes a long word as the branch address. Likewise, there is an alternative branch-subroutine instruction denoted `JSR`, that has the same function as `BSR` and can be used if the displacement is not sufficient. We show these instructions in Table 4.11.

**Table 4.11** Alternative branch instructions if the displacement provided by `BRA` and `BSR` is not sufficient.

| Name | | Operation |
|---|---|---|
| JMP | *address* | BRA *address* |
| JSR | *address* | BSR *address* |

### Exercises

**4.35** Determine the machine code in hexadecimal representation of the following instructions: (a) `ADD.L D5,D6` (b) `ADD.B #%10110110,D3` (c) `ADD.W $53254,D2`

**4.36** Determine the machine code in hexadecimal representation of the following instructions: (a) `ADD.L D7,(A2)` (b) `ADD.B (A3)+,D2` (c) `ADD.W A2,D2`

**4.37** Determine the machine code in hexadecimal representation of the following instructions, assuming that they are stored at address $1000_{16}$ in memory. (a) `BRA $1004` (b) `BRA $FFE` (c) `BRA $FF0`

## 4.8 Summary and concluding remarks

In this chapter, we have looked in detail at the instruction set model of a processor. This chapter provided an insight into the kind of instructions and their semantics provided by the M68000. By illustrating the concepts of instruction set models by a concrete example, we can practically exercise on existing computers. A drawback of this, however, is the necessity of burdening the reader with a fair amount of details. Therefore, the intention behind this summary is to provide the reader with those principles of instruction set models, that are applicable to most computer systems.

A processor usually contains a small number of registers to act as high-speed memory cells. These are used to carry out a computation on operands by reusing and accessing them efficiently. The instructions we have seen carry the following vital information: the operation, the operand locations, and the size of the operands.

The operations we have seen are either binary in nature, requiring two operands (source and destination operands) or unary requiring only one operand. Examples of binary operations provided by most computers are addition, subtraction, logical operations such as logical AND, OR (inclusive-or), and EOR (exclusive-or). Other kinds of operations we have seen are branch, shift, and subroutine-call instructions.

An important feature that we examined was the use of condition codes to control the program flow depending on the result of an arithmetic or logical operation. This is supported by a class of branch-instructions called conditional branch-instructions.

The operand locations are specified by the so-called addressing modes of the computer. The most widely used addressing modes are register direct, immediate,

absolute, and register indirect addressing. We presented some additional addressing modes that are provided by the M68000. We also noted that the operand size can be controlled by the instruction in the operand size attribute.

Finally, we looked at how assembly instructions are translated into machine language instructions by two example instructions, namely, the `ADD` and `BRA` instructions. An important observation is that a primary objective is to code the instructions in a concise form so as to reduce the number of words to be fetched when the processor fetches the next instruction. An example of this is the coding scheme for the `BRA` instruction that stores a displacement to be added to the current value of PC instead of storing the branch address explicitly.

Chapter 5

# Assembly Language Programming

In the previous chapter, we presented the syntax and semantics of the most commonly used assembly language instructions for a computer in general, and for M68000 in particular. We designed small programs from well-defined descriptions. When it comes to solving larger problems, however, it is almost never the case that the problem can be directly translated into a sequence of assembly language instructions. Instead, the programmer must spend a significant amount of time in structuring the problem in a way that makes it easier to translate it into a sequence of primitive assembly language instructions.

Most problems are solved using high-level languages such as Pascal, Fortran, and C, because they provide powerful constructs to express complex computations. Examples of such constructs are not only repetition-statements such as for-, while-, and repeat-loops, but also conditionals such as if-then-else statements. Another important feature is the procedure, or subroutine concept. Procedures serve an important role in the structuring process in the program design: the programmer can break down the problem into smaller subproblems, which in turn can be broken down into smaller problems until a point when it is straightforward to translate them into assembly language instructions.

Not only must the programmer translate the problem description into a sequence of instructions. Another important aspect is to write the program in such a way that makes it possible for others to read and understand the code. There are several important means of enhancing the program readability. First, symbolic names can be used to express the intention behind their use. Second, data structures that are used by a certain subroutine, for example, should be declared close to that subroutine.

Finally, when the program has been designed, the programmer must make sure that it works correctly. Certain methods and tools are available to make testing easier.

The purpose of this chapter is severalfold. First, since the reader is familiar with high-level language programming, we will use an example language, in essence

Pascal, to present how commonly used high-level language constructs are translated into sequences of assembly language instructions for the M68000 in Section 5.1. In Section 5.2, we discuss a method of how to solve larger problems by using top-down design. We will also discuss the issue of structuring an assembly language program. Third, in order to illustrate how the design methodology can be applied to a realistic problem, we will present the design process for a larger program in Section 5.3. Finally, in Section 5.4, we will present methods and tools that can be used to test (debug) an assembly language program.

## 5.1 Translating high-level language constructs

One of the advantages of a high-level language is that it provides powerful constructs to design complex programs. As we have seen in the previous chapters, this is not the case for assembly language instructions. However, it is possible to build sequences of instructions that implement high-level language constructs. If such sequences are available, we can use the same methodology that is useful for high-level language programming design in the process of designing assembly language programs. We start with a high-level language description of the problem. We then translate the high-level language constructs into sequences of assembly language instructions almost in a mechanical fashion. This method is advantageous because of its robustness; common constructs have always the same assembly language instruction structure. This fact promotes correctness of the resulting program. A disadvantage with this method, however, is that it can result in an assembly language program that does not lead necessarily to the most efficient solution. Once we have a program working correctly, however, the programmer can then concentrate on efficiency issues and improve the solution separately.

In this section, we shall look at the implementation of commonly used high-level language constructs, in essence constructs in Pascal, using M68000 assembly language instructions.

### 5.1.1 Data structures

We noted in the previous chapter that symbolic names can be used to refer to memory locations. Symbolic names enhance program readability considerably. Other means of enhancing readability include the use of symbolic names for constants. Most assemblers support definition of symbolic names by special instructions to the assembler called *assembler directives*. In Table 5.1, we provide a list of commonly used directives for the M68000 assembler and their meaning. Note, however, that the names of the directives may differ from assembler to assembler.

Let us look at some examples to demonstrate the use of the directives in Table 5.1. Consider a string of characters `'HELLO WORLD!'` that is to be printed out on

**Table 5.1** Examples of assembler directives for the M68000.

| Assembler directive | | | Description |
|---|---|---|---|
| *sym* | EQU | *exp* | Assigns the constant called *sym* the value of expression *exp*. |
| | ORG | *n* | The instructions and declarations following this directive are to be placed in memory beginning at address *n*. |
| *sym* | DC.S | *n* | Initializes the operand of size S at the symbolic address *sym* to the value *n*. The size attribute S can denote a byte (B), word (W), or long word (L). |
| *sym* | DS.S | *n* | Reserves memory space for *n* operands, each of size S, where the first operand is stored at the symbolic address *sym*. |
| | EVEN | | Causes the next instruction or memory location to be stored at an even address. |
| | END | | This is a mandatory directive that must appear as the last statement in the assembly code. |

the terminal screen continuously. We want to write a subroutine PRSTR that prints out the string pointed to by A0. In order to do this, we make use of a predefined subroutine at address $F0432_{16}$ that prints out the character corresponding to the ASCII-code stored in the seven least significant bits of register D0.

```
PRINT   EQU      $F0432
NUL     EQU      0
        ORG      $9000
STRING  DC.B 'HELLO WORLD!',NUL
        EVEN
START   MOVEA.L  #STRING,A0   ; Let A0 point at STRING
        BSR      PRSTR        ; Print out the string
        BRA      START
PRSTR   CMPI.B   #NUL,(A0)    ; If character = NUL
        BEQ      PRRET        ; return from the subroutine
        MOVE.B   (A0)+,D0     ; Otherwise, print it out
        JSR      PRINT
        BRA      PRSTR
PRRET   RTS
        END
```

We make the following important observations on the use of assembler directives in this example. First, in order to use a more comprehensive subroutine name than its address, we have defined `PRINT` to denote the address of the subroutine that prints out a single character. We also use `EQU` to define a symbolic name for the `NUL` ASCII-character. The third line is the `ORG`-directive. It is used to tell the assembler that the subsequent line (a memory cell or an instruction) is to be located at address $9000_{16}$. After the `ORG`-directive follows the string '`HELLO WORLD!`' which is stored using the Define Constant directive with the first character `H` (ASCII $48_{16}$) at memory address `STRING` (in effect address $9000_{16}$). All twelve letters comprising the string are stored as bytes (`DC.B`) at address $9000_{16}$ to $900B_{16}$. `NUL` is stored at address $900C_{16}$. Immediately after `STRING` follows the `EVEN` directive. This is needed in order to make the instruction `MOVEA.L #STRING,A0` following `STRING` to be located at an even address. Since the last character (`NUL`) is located at address $900C_{16}$, the instruction would have been located at the odd address $900D_{16}$. It would introduce an address error since M68000 can only fetch instructions and operands of size word and long word on even addresses. The last statement is the `END` directive which should not be confused with the `STOP` instruction. The `END` directive tells the assembler that there are no more instructions to be translated.

Most high-level languages provide a means to declare a set of variables that is logically considered as a unit. One example is a record with information about a person such as name, address, and birth date. The next example demonstrates how to implement such a data structure, in essence the record concept in Pascal.

Consider a data structure for a buffer in which integers are retrieved in the front and inserted in the end. The size of the buffer is $100_{10}$ integers starting at address `LIST`. The current number of elements in the buffer is kept in a variable called `COUNT`.

```
const SIZE = 100;

type BUFFER  = record
  LIST        :array[0..SIZE-1] of integer;
  FIRST, LAST :integer;
  COUNT       :integer;
end;

var INBUF :BUFFER;
```

Variables `FIRST` and `LAST` are used to retrieve and insert elements in `LIST`. We assume that integers are implemented by long words (32 bits). The following assembler directives can implement this data structure:

```
SIZE   EQU   100
INBUF  DS.L  SIZE   ; Vector LIST
       DC.L  0      ; FIRST (initialized to 0)
       DC.L  0      ; LAST (initialized to 0)
       DC.L  0      ; COUNT (initialized to 0)
```

We will now give an example of an assembly language program that uses the record INBUF defined above. Consider the following Pascal program:

```
procedure INSERT(ITEM : integer);
begin
  INBUF.LIST[INBUF.LAST]:=ITEM;
  INBUF.LAST:=INBUF.LAST+1;
  INBUF.COUNT:=INBUF.COUNT+1;
end;
```

The above procedure inserts an integer named ITEM in the buffer. In order to implement procedure INSERT above, we will make considerable use of the addressing modes presented in Section 4.5. In order to do this, we note that the base address of the record INBUF is INBUF and that the address of the first element of LIST is INBUF+0. Secondly, the addresses of FIRST, LAST, and COUNT are INBUF+4*SIZE, INBUF+4*SIZE+4, and INBUF+4*SIZE+8, respectively. This makes it possible to refer to the record variables by displacements using the EQU-directive below.

```
LIST    EQU      0
FIRST   EQU      4*SIZE
LAST    EQU      4*SIZE+4
COUNT   EQU      4*SIZE+8

INSERT  MOVEA.L  #INBUF,A0
        MOVE.L   LAST(A0),D0
        MOVE.L   ITEM,LIST(A0,D0)   ; INBUF.LIST[INBUF.LAST]:=ITEM
        ADDI.L   #4,LAST(A0)        ; INBUF.LAST:=INBUF.LAST+1
        ADDI.L   #1,COUNT(A0)       ; INBUF.COUNT:=INBUF.COUNT+1
        RTS
```

Note that we add 4 to LAST. This is not exactly what the Pascal program does and the reason is simply that we use 32-bit integers so the next element in vector LIST appears four bytes higher up in the address space.

One could ask why we access the record variables using displacements instead of absolute addresses. But suppose that we defined more than one variable of type `BUFFER`. If we would have used absolute addressing to access all individual variables contained in the records, it would be a cumbersome task to deal with all absolute addresses for the individual record variables. By defining displacements, we can use the same displacements for all instances of the record `BUFFER`. For example, to access another instance `OUTBUF` of type `BUFFER`, we need only change the first instruction in the procedure `INSERT` to `MOVEA.L #OUTBUF,A0`.

In summary, assembler directives help the assembly programmer to write comprehensive assembly language programs. This is facilitated by means of defining symbols, constant values in memory, symbolic names for memory areas etc. Note that the assembler directives do not produce executable code and, therefore, should not be confused with the assembly instructions which, on the other hand, result in executable machine instructions.

### 5.1.2 Conditional statements

Most high-level languages contain various constructs for conditionals. Conditionals must be implemented with conditional branch-instructions in an assembly language program. For example, the statement

```
if A=B then A:=0;
```

can be translated into

```
        MOVE.L  A,D0
        CMP.L   B,D0  ; if A<>B then
        BNE     NEXT  ; goto NEXT
        CLR.L   A
NEXT
```

assuming that `A` and `B` are 32-bit integers.

We shall now turn our attention to the translation of a more general `if-then-else` construct of the following form:

`if A` *rel-op* `B then` *then-statement* `else` *else-statement*;

The relational operator *rel-op* is one of those found in Table 5.2. The construct is equivalent to

`if (A - B)` *rel-op* `0 then` *then-statement* `else` *else-statement*;

**Table 5.2** Relational operators (*rel-op*), their mnemonics (*cc*), and their inverses relevant for (a) Signed integers and (b) Unsigned integers.

**(a)**

| *rel-op* | *cc* | *rel-op'* | *cc'* |
|---|---|---|---|
| $<$ | LT | $\geq$ | GE |
| $\leq$ | LE | $>$ | GT |
| $=$ | EQ | $\neq$ | NE |
| $\neq$ | NE | $=$ | EQ |
| $\geq$ | GE | $<$ | LT |
| $>$ | GT | $\leq$ | LE |

**(b)**

| *rel-op* | *cc* | *rel-op'* | *cc'* |
|---|---|---|---|
| $<$ | CS | $\geq$ | CC |
| $\leq$ | LS | $>$ | HI |
| $=$ | EQ | $\neq$ | NE |
| $\neq$ | NE | $=$ | EQ |
| $\geq$ | CC | $<$ | CS |
| $>$ | HI | $\leq$ | LS |

By doing this, we can find the following solution to the problem

```
        MOVE.L  A,D0
        CMP.L   B,D0    ; if A-B rel-op 0 then
        Bcc     THEN    ; goto THEN
        else-statement
        BRA     NEXT
THEN
        then-statement
NEXT
```

where mnemonic *cc* is obtained from Table 5.2. Note that the mnemonics reflect their meaning; for instance, `LT` stands for 'Less Than (zero)' and `GT` stands for 'Greater Than (zero)'. This makes it easier to remember all conditional branch instructions.

Note that if `B` is a constant, we can replace the first two instructions `MOVE.L A,D0` and `CMP.L B,D0` by `CMPI.L #B,A`. Let us give an example. Implement the following `if-then-else` construct as a sequence of M68000 assembly language instructions, assuming that `A` and `X` are Signed 32-bit integers.

```
if A>=5 then X:=A else X:=0;
```

The following assembly language program implements this:

```
        CMPI.L  #5,A    ; if A-5 >= 0 then
        BGE     THEN    ; goto THEN
        CLR.L   X       ; X:=0 (else-statement)
        BRA     NEXT
THEN    MOVE.L  A,X     ; X:=A (then-statement)
NEXT
```

A word of warning concerning the tests we have devised is appropriate. As mentioned in Section 4.3, we noted that the conditional branch instructions that correspond to the relational operators in Table 5.2(a) are relevant for Signed integers only. When we are dealing with Unsigned integers, we must use the conditional branch instructions listed in Table 5.2(b) instead.

The translation of a conditional statement with nested else-clauses

```
if A=0 then
  then-statement1
else if A=1 then
  then-statement2
else
  else-statement;
```

is a natural extension of the scheme we have shown:

```
        CMPI.L  #0,A
        BEQ     THEN1
        CMPI.L  #1,A
        BEQ     THEN2
         else-statement
        BRA     NEXT
THEN1
        then-statement1
        BRA     NEXT
THEN2
        then-statement2
NEXT
```

### 5.1.3 Repetition statements

We will now show how to translate three commonly used high-level language constructs for repetition: for-loops; repeat-loops; and while-loops.

A for-loop in Pascal has the form:

```
for I:=START to STOP do BODY;
```

If `START` is greater than `STOP`, the loop-iteration `BODY` is not executed. Otherwise, `BODY` is executed (`STOP`−`START`+1) times. The for-loop is equivalent to the following statements

```
                I:=START;
                goto TEST;
FOR             BODY;
                I:=I+1;
TEST            if I <= STOP then goto FOR
NEXT
```

which we can translate into the following sequence of instructions

```
        MOVE.L  #START,D0   ; I:=START
        BRA     TEST        ; goto TEST
FOR     BSR     BODY        ; BODY
        ADDI.L  #1,D0       ; I:=I+1
TEST    CMPI.L  #STOP,D0    ; if I <= STOP then
        BLS     FOR         ; goto FOR
NEXT    ...
```

assuming that `I` is an Unsigned 32-bit integer and `START` and `STOP` are declared constants.

A repeat-loop is another example of a repetition statement. The difference between a for-loop and a repeat-loop is that one can have a general test condition for loop termination in a repeat-loop as shown in the example below.

`repeat BODY until A` *rel-op* `B;`

The repeat-loop terminates when A *rel-op* B. This is the same to say that it continues as long as A-B *rel-op'* 0, where *rel-op'* is the inverse relational operator of *rel-op*. The inverse relational operators are found in Table 5.2. For example, the inverse relational operator of '$<$' is '$\geq$' and the inverse relational operator of '$=$' is '$\neq$'. We also show the inverse mnemonics *cc'* for each relational operator in Table 5.2. Given these operators, the following Pascal-statements are equivalent to the repeat construct:

```
REPEAT      BODY;
            if A-B rel-op' 0 then goto REPEAT
NEXT
```

which becomes

```
REPEAT  BSR     BODY
        MOVE.L  A,D0
        CMP.L   B,D0
        Bcc'    REPEAT
NEXT
```

where mnemonic *cc'* can be obtained from Table 5.2.

The example program

```
repeat I:=I+1 until I > 5;
```

can be translated into

```
REPEAT  ADDI.L  #1,I    ; I:=I+1
        CMPI.L  #5,I    ; if I-5 <= 0 then
        BLE     REPEAT  ; goto REPEAT
NEXT
```

where we have used the inverse relational operator of $>$, that is $\leq$, to construct the test. Note that at least one iteration is executed in a repeat-loop. This is because the loop termination test is performed after each iteration.

In a while-loop, the test is performed before the iteration:

```
while A rel-op B do BODY;
```

which can be rewritten as

```
          goto TEST;
WHILE     BODY;
TEST      if A-B rel-op 0 then goto WHILE
NEXT
```

that is, the following sequence of instructions can implement the while-loop:

```
        BRA     TEST    ; goto TEST
WHILE   BSR     BODY    ; BODY
TEST    MOVE.L  A,D0
        CMP.L   B,D0    ; if A-B rel-op 0 then
        Bcc     WHILE   ; goto WHILE
NEXT    ...
```

For example,

```
while A <> B do A:=A+10;
```

can be translated into

```
        BRA     TEST    ; goto TEST
WHILE   ADDI.L  #10,A   ; A:=A+10
TEST    MOVE.L  A,D0
        CMP.L   B,D0    ; if A-B <> 0 then
        BNE     WHILE   ; goto WHILE
NEXT    ...
```

### 5.1.4 Parameter passing to subroutines

An important problem when using procedures or functions (collectively called subroutines) is how to pass parameters from the calling program to the subroutine. There are a number of solutions to this problem which we will discuss here. There are two basic approaches to pass parameters to subroutines: we can either pass the *value* or a *reference* (address) of the variable that comprises the parameter. These two methods are usually called *call-by-value* and *call-by-reference* in high-level languages.

Values can be passed either by using registers or, if the number of registers is not sufficient, by dedicated memory locations. For instance, in the Pascal-function

```
function ADDF(X,Y :integer):integer;
begin
   ADDF:=X+Y;
end;
```

we can pass X and Y using registers D0 and D1 and pass the function value in D1:

```
ADDF  ADD.L  D0,D1
      RTS
```

Now suppose that a subroutine uses a larger number of parameters than the number of available registers. Then it is not possible to use registers as a means to pass values. A solution to this problem is to associate a memory area with the subroutine in which the calling program puts all parameters. For instance, if the ADDF subroutine is to be used to add five numbers

```
function ADDF(X1,X2,X3,X4,X5 :integer):integer;
begin
   ADDF:=X1+X2+X3+X4+X5;
end;
```

we can use five memory locations associated with ADDF in the following way

```
X     DS.L     5          ; Space for X1 through X5

ADDF  MOVEA.L  #X,A0
      MOVE.L   #0,D1
      MOVE.B   #5,D2
LOOP  ADD.L    (A0)+,D1
      SUBI.B   #1,D2
      BNE      LOOP
      RTS
```

where the function value is returned in D1. Now suppose that we want to use the function to add two arbitrary memory locations, then the calling program would have to be responsible for moving the values into registers or memory locations before calling the ADDF function. It would be more efficient, in this case, to pass the addresses (call-by-reference) of the operands as in the following implementation:

```
ADDF  MOVE.L  (A0),D1
      ADD.L   (A1),D1
      RTS
```

In this implementation, the addresses of the operands are passed through address registers A0 and A1. The result is returned in register D1 also in this case.

## Exercises

**5.1** Implement the following procedure that retrieves an element from vector LIST, using the declarations on page 61.

```
procedure RETRIEVE; begin
   ITEM:=INBUF.LIST[INBUF.FIRST];
   INBUF.FIRST:=INBUF.FIRST+1;
   INBUF.COUNT:=INBUF.COUNT-1;
end;
```

**5.2** What sequence of instructions implements the following `if-then-else` construct

```
if A<B then A:=0 else A:=1;
```

assuming that A and B are 32-bit Signed integers?

**5.3** What sequence of instructions implements the following `if-then-else` construct

```
if A<B then A:=0 else A:=1;
```

assuming that A and B are 32-bit Unsigned integers?

**5.4** What sequence of instructions implements the following `if-then-else` construct

```
if (A=>5) and (A<=10) then A:=0 else A:=1;
```

assuming that A is a 32-bit Signed integer?
*Hint:* The above if-then-else construct can be rewritten as nested else-clauses as follows:

```
if A < 5 then
   A:=1
else if A > 10 then
   A:=1
else
   A:=0;
```

**5.5** What sequence of instructions implements the following `for`-loop

```
for I:=1 to 10 do
   J:=J+1;
```

assuming that I and J are 32-bit Unsigned integers?

**5.6** What sequence of instructions implements the following Pascal-statements

```
I:=0;
while I < 10 do
   I:=I+1;
```

assuming that I is a 32-bit Unsigned integer?

**5.7** What sequence of instructions implements the following Pascal-statements

```
I:=0;
repeat
   I:=I+1;
until I>20;
```

assuming that I is a 32-bit Signed integer?

---

## 5.2 Program design and structure

In the previous section, we showed how commonly used high-level language constructs such as conditionals and repetition-statements can be translated into sequences of assembly language instructions. In this section, we will present a method of how to design large assembly language programs. The approach is based on a commonly used design methodology referred to as *top-down design*. The general idea is to describe the solution to a problem using high-level primitives such as procedures and functions. Each high-level primitive is in turn described in terms of primitives on a more detailed level. This stepwise refinement continues until a point when it is straightforward to translate it into a sequence of assembly language instructions.

As a base for describing the solution of a problem, one can use an arbitrary high-level language. But it is also possible to mix this language with one's own inventions. A description using a high-level language mixed with one's own inventions is usually referred to as *pseudo-code*. We will illustrate this technique by using Pascal.

The specification and implementation of a program can be summarized by the

following five steps:

1. Specifying the task in pseudo-code
2. Refining the specification by breaking down high-level constructs into more manageable units.
3. Coding and documenting the program
4. Testing the program
5. Isolating and removing program errors, *debugging*

The first three steps constitute the design process. First, the solution is described using pseudo-code. Preferably, one uses a high-level language as long as possible. The advantage of this is that it provides well-defined constructs. Furthermore, by using standard translations, as we have shown in the previous section, one can reach a correct solution faster. Second, each primitive (i.e. a high-level procedure or function) is specified in terms of primitives at a more detailed level. This process continues until a point when it is obvious how to translate each primitive into a sequence of assembly language instructions. Third, the assembly language program is coded. At this point it is appropriate to add informative comments so as to enhance readability.

There are many ways to document a program. First, the comments ought to be informative. For example: in '`MOVE.B D0,D1 ; Move D0 to D1`' the comment is NOT informative. Comments should be problem-oriented rather than language-oriented – it should be assumed that any person that reads the program listings is familiar with the language (in this case the M68000 assembly language). One way of adding informative comments is to use the pseudo-code. Each high-level statement is used as an in-line comment to explain the corresponding sequence of instructions. Another good idea is to use headers of comments for each piece of code such as a subroutine. Below, we show what information should go into that header:

```
;===============================================================================
; NAME:          ADD64
; DESCRIPTION:   Adds two 64-bit numbers C := A + B
; INPUT:         D0 and D1 pass A, D2 and D3 pass B
; OUTPUT:        D2 and D3 return C
; REGISTERS:     D2 and D3
;===============================================================================

ADD64           ADD.L   D1,D3
                ADDX.L  D0,D2
                RTS
```

The header contains the name of the subroutine, a short description of what it does, how the input and output parameters are passed, and which registers are affected by the subroutine. This information is useful for other persons that use the subroutine. For example, if a subroutine uses register D2, it is important for

the program that performs a subroutine call to save register D2 provided that it uses register D2.

In the next section, we shall illustrate how all ideas developed in this chapter can be used to design a larger assembly language program. The last two issues regarding how to test and debug a program will be addressed in the last section in this chapter.

## 5.3 A large program design example

In this section, we shall make use of what we have learned about programming methodology and style by designing an assembly language program that performs a nontrivial task. The task we will consider is a program that retrieves information from a database consisting of a number of records of the following type:

```
type PERSON = record
                FNAME :array[0..20] of char;
                LNAME :array[0..20] of char;
                MALE :boolean;
                BYEAR :integer;
                NEXT :^PERSON;
              end;
```

Each record keeps information of a person's first and last name (`FNAME` and `LNAME`), a boolean variable (`MALE`) that is true if the person is a male, and finally, the birth year of the person (`BYEAR`). For simplicity, we use integers to express the boolean variable `MALE` (1 for true, and 0 for false). The last variable in the record (`NEXT`) is a pointer to the next record. In Figure 5.1, we show how the database keeps track of three records. The first record appears at address $9000_{16}$, while the last record starts at address $9068_{16}$.

The program will be able to insert a new record of type `PERSON`, list all persons that were born in a specific year, and list all persons of a specific sex. This task could be described using the following Pascal specification.

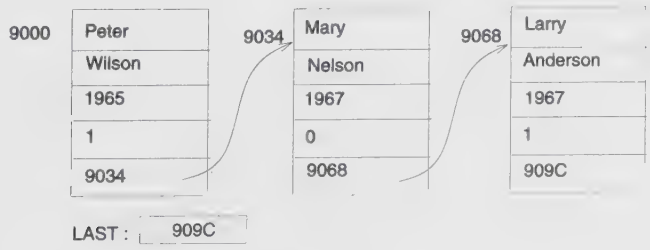



**Figure 5.1** An example of how the database keeps track of three records.

```
begin
  repeat
    CHOICE:=PRMEN;              (*** PRMEN prints the menu
                                     and returns the CHOICE ***)
    if CHOICE = 1 then
         INSREC                 (*** INSREC inserts a new person
                                     into the database ***)
       else if CHOICE = 2 then
              PRBORN            (*** PRBORN prints  all persons
                                     born a specific year ***)
            else if CHOICE = 3 then
                   PRSEX;       (*** PRSEX prints all persons
                                     of a specific sex ***)
   until CHOICE = -1;
end.
```

Although we haven't written the procedures that implement the desired function, we can structure the program at this very early stage of the design process. The next step is to refine the specification of each procedure and function.

```
function PRMEN;
begin
   PRSTR('1.  Insert new record');
   PRSTR('2.  Find all persons born in year...');
   PRSTR('3.  Find all persons with sex...');
   CHOICE:=READINT;
   if (CHOICE < 1) or (CHOICE > 3) then CHOICE:=-1;
   PRMEN:=CHOICE;
end;
```

PRMEN first prints out the menu and then reads an integer by calling a function named READINT. The menu choice is an integer between 1 and 3. If another integer is read, CHOICE is assigned $-1$.

```
procedure INSREC;
begin
  PRSTR('Input the first name'); LAST.FNAME:=READSTR;
  PRSTR('Input the last name'); LAST.LNAME:=READSTR;
  PRSTR('Input sex'); LAST.MALE:=READINT;
  PRSTR('Input year'); LAST.BYEAR:=READINT;
  LAST:=LAST.NEXT;
end;
```

INSREC reads all variables in the record from the terminal. This is done by using a function called READSTR. Note that LAST is a pointer to the last record to be

inserted in the database. In Figure 5.1, LAST contains the address to the record where the new person shall be inserted. It must be initialized in the main program to point to the first record. When a new record is inserted, LAST is updated to point to a new record.

```
procedure PRBORN;
begin
  PRSTR('Input year');
  YEAR:=READINT;
  REC:= 'First record';
  while REC <> LAST do
  begin
    if YEAR = REC.BYEAR then
    begin
      PRSTR(REC.FNAME);
      PRSTR(REC.LNAME);
    end;
    REC:=REC.NEXT;
  end;
end;
```

PRBORN reads an integer from the keyboard using the function READINT. It then traverses the records in the database according to Figure 5.1 to check for the occurrence of a person whose birth year matches the variable YEAR. If there is a match, the person's name is printed out on the terminal screen using the procedure PRSTR. The next procedure we consider, PRSEX, is similar in its structure:

```
procedure PRSEX;
begin
  PRSTR('Input sex');
  SEX:=READINT;
  REC:= 'First record';
  while REC <> LAST do
  begin
    if SEX = REC.MALE then
    begin
      PRSTR(REC.FNAME);
      PRSTR(REC.LNAME);
    end;
    REC:=REC.NEXT;
  end;
end;
```

In the specification of the procedures PRBORN and PRSEX, we have used a mix of English such as REC := 'First record' and fairly well-defined functions such as READINT which reads an integer from the keyboard. This has made it possible to specify the intention of each procedure in terms of other procedures. The net effect of this is that the only procedures that are left to be specified are PRSTR, READSTR, and READINT. These might be offered by the computer system in terms of device drivers in the so called operating system, or, can be coded directly in M68000 assembly code. We can now start to code the entire program, but before we do this, we need to agree upon a program structure.

Besides the advice given in the previous sections, we shall give some additional advice on how to structure a program such as the one we are dealing with here. In essence, it consists of a main program, a number of subroutines, and data structures. We will apply to the following structure:

- Main program. A header of comments describes its function.
- Subroutines. Each subroutine (procedure or function) should be preceded by a header specifying its name, a short description of its function, input as well as output parameters, and registers that are effected. In addition, the declarations and local variables should precede the subroutine code.
- Global data structures.

On the next few pages, we show the assembly code for the entire program starting with the main program.

```
;=============================================================================
; PROGRAM:      MAIN
; DESCRIPTION:  Inserts new records of persons and answers simple queries.
;=============================================================================

MAIN            BSR     PRMEN       ; repeat CHOICE:=PRMEN
                CMPI.L  #1,CHOICE   ; if CHOICE=1 then
                BEQ     THEN1
                CMPI.L  #2,CHOICE   ; else if CHOICE=2 then
                BEQ     THEN2
                CMPI.L  #3,CHOICE   ; else if CHOICE=3 then
                BEQ     THEN3
                BRA     TEST
THEN1           BSR     INSREC      ; INSREC
                BRA     TEST
THEN2           BSR     PRBORN      ; PRBORN
                BRA     TEST
THEN3           BSR     PRSEX       ; PRSEX
TEST            CMPI.L  #-1,CHOICE  ; until CHOICE=-1
                BNE     MAIN
                STOP    #$2700
```

On the previous page, we show the code for the main program. We note how the original specification in Pascal is used as comments in the main program. This way, it is easy to understand how the program works. Also note how we have made use of standard translations for the if-then-else statements and the repeat-statement that appear in the Pascal specification.

The main program will terminate if a menu choice outside the allowed range (1,2, and 3) is typed in. The STOP instruction that appears as the last instruction is then executed.

```
;===========================================================================
; NAME:          PRMEN
; DESCRIPTION:   Prints a menu and reads menu choice.
; INPUT:         None
; OUTPUT:        Menu choice 1,2,3 or -1 in memory location CHOICE.
; REGISTERS:     A0
;===========================================================================

STR1            DC.B     '1.  Insert new record',$0D,$0A,0
STR2            DC.B     '2.  Find all persons born in year...',$0D,$0A,0
STR3            DC.B     '3.  Find all persons with sex...',$0D,$0A,0
                EVEN
CHOICE          DS.L     1

PRMEN           MOVEA.L  #STR1,A0
                BSR      PRSTR          ; PRSTR('1.  Insert new
                                        ; record')
                MOVEA.L  #STR2,A0
                BSR      PRSTR          ; PRSTR('2.  Find all persons
                                        ; born in year...')
                MOVEA.L  #STR3,A0
                BSR      PRSTR          ; PRSTR('3.  Find all persons
                                        ; with sex...')
                MOVEA.L  #CHOICE,A0
                BSR      READINT        ; CHOICE:=READINT
                CMPI.L   #1,CHOICE      ; if CHOICE < 1 then
                BLT      MINUS1         ; BRA MINUS1, or
                CMPI.L   #3,CHOICE      ; if CHOICE > 3 then
                BGT      MINUS1         ; BRA MINUS1
                RTS
MINUS1          MOVE.L   #-1,CHOICE     ; CHOICE:=-1
                RTS
```

The function PRMEN above uses a variable named CHOICE to return the menu choice. The first thing PRMEN does is that it prints out the menu. Note how all strings are declared using the DC directive. Also note that each character string is terminated by the ASCII codes for carriage return (0D), line feed (0A), and NUL (0). While the first two ASCII characters cause the cursor on the screen to proceed to the beginning of the next line, the NUL character does not result in any output. It

is used by the PRSTR subroutine as an end-of-string mark. The PRSTR subroutine, which is used to print out the string, assumes that the address of the first character is stored in A0. This is why A0 is initialized before the subroutine is called.

```
;===============================================================================
; NAME:          INSREC
; DESCRIPTION:   Inserts a new record into the database.
; INPUT:         None
; OUTPUT:        None
; REGISTERS:     A0,A1
;===============================================================================

FNAME           EQU     0               ; Displacement for FNAME
LNAME           EQU     20              ; Displacement for LNAME
MALE            EQU     40              ; Displacement for MALE
BYEAR           EQU     44              ; Displacement for BYEAR
NEXT            EQU     48              ; Displacement for NEXT
NREC            EQU     52              ; Displacement to next record

LAST            DC.L    DATABASE

ISTR1           DC.B    'Input the first name',$0D,$0A,0
ISTR2           DC.B    'Input the last name',$0D,$0A,0
ISTR3           DC.B    'Input sex',$0D,$0A,0
ISTR4           DC.B    'Input year',$0D,$0A,0

                EVEN
INSREC          MOVEA.L LAST,A1
                MOVEA.L #ISTR1,A0
                BSR     PRSTR           ; PRSTR('Input the first name')
                LEA     FNAME(A1),A0
                BSR     READSTR         ; LAST.FNAME:=READSTR
                MOVEA.L #ISTR2,A0
                BSR     PRSTR           ; PRSTR('Input the last name')
                LEA     LNAME(A1),A0
                BSR     READSTR         ; LAST.LNAME:=READSTR
                MOVEA.L #ISTR3,A0
                BSR     PRSTR           ; PRSTR('Input sex')
                LEA     MALE(A1),A0
                BSR     READINT         ; LAST.MALE:=READINT
                MOVEA.L #ISTR4,A0
                BSR     PRSTR           ; PRSTR('Input year')
                LEA     BYEAR(A1),A0
                BSR     READINT         ; LAST.BYEAR:=READINT
                MOVEA.L A1,A0
                ADDA.L  #NREC,A1        ; Address of the next entry
                MOVE.L  A1,NEXT(A0)
                MOVE.L  A1,LAST         ; LAST:=LAST.NEXT
                RTS
```

In INSREC on the previous page, character strings and integers are read using the subroutines READSTR and READINT. Both these subroutines use A0 to point at the location in memory where to put the result. We have used a special instruction called LEA $a$,A$i$ that computes the absolute address of expression $a$ and stores it in address register A$i$. For example, LEA LNAME(A1),A0 performs the operation LNAME+(A1) → A0. For more details about this instruction please refer to Appendix B.

Also note that the address of the new record to be inserted is always present in the variable LAST. This is why A1 is initialized to contain the address stored in variable LAST. LAST must also be updated before the return instruction is executed. This is done by first adding the displacement NREC to A1 and then updating LAST with the content of A1.

```
;===============================================================================
; NAME:          PRBORN
; DESCRIPTION:   Prints all persons born in a specific year
; INPUT:         None
; OUTPUT:        None
; REGISTERS:     A0,A1,D0
;===============================================================================
PRSTR1          DC.B      'Input year',$0D,$0A,0
                EVEN
YEAR            DS.L      1

PRBORN          MOVEA.L   #PRSTR1,A0
                BSR       PRSTR          ; PRSTR('Input year')
                MOVEA.L   #YEAR,A0
                BSR       READINT        ; YEAR:=READINT;
                MOVEA.L   #DATABASE,A1   ; REC:= 'First record'
                BRA       WTEST1
WLOOP1          MOVE.L    YEAR,D0
                CMP.L     BYEAR(A1),D0   ; if YEAR = REC.BYEAR then
                BNE       CONT1
                LEA       FNAME(A1),A0
                BSR       PRSTR          ; PRSTR(REC.FNAME)
                LEA       LNAME(A1),A0
                BSR       PRSTR          ; PRSTR(REC.LNAME)
CONT1           MOVEA.L   NEXT(A1),A1    ; REC:=REC.NEXT
WTEST1          CMPA.L    LAST,A1        ; if REC - LAST <> 0 then
                BNE       WLOOP1         ; goto WLOOP1
                RTS                      ; end;
```

The assembly code for PRBORN appears above. The first thing it does is to read the year to be matched against all entries in the database from the keyboard (BSR READINT). The first record appears at address DATABASE. A1 is used to point to the current record in the database. This is why it is initialized to contain the absolute address corresponding to DATABASE. Since PRSEX is very similar in structure, we leave it as an exercise for the reader to design PRSEX.

Finally, we need to specify the subroutines PRSTR, READSTR, and READINT. They appear on the next two pages. Parameters to/from these subroutines are passed by reference using A0 to contain the address. PRSTR prints all characters until ASCII NUL (0) is encountered. READSTR reads characters until a carriage return ($0D_{16}$) is encountered. It then inserts ASCII NUL and exits. All these subroutines use two subroutines CHRIN and CHROUT that print and read an ASCII character to/from the terminal, respectively. In the next chapter, we will see how these subroutines can be implemented by extending the model of the computer system with input/output devices.

```
;===============================================================================
; NAME:          PRSTR
; DESCRIPTION:   Prints a string terminated by NUL on the terminal screen
; INPUT:         A string pointed to by A0
; OUTPUT:        None
; REGISTERS:     A0,D0
;===============================================================================

NUL             EQU     0
CHROUT          EQU     $F0432
PRSTR           CMPI.B  #NUL,(A0)
                BEQ     PREND
                MOVE.B  (A0)+,D0
                JSR     CHROUT      ; Print the character in D0
                BRA     PRSTR
PREND           RTS


;===============================================================================
; NAME:          READSTR
; DESCRIPTION:   Reads a string from the keyboard
; INPUT:         A string pointed to by A0
; OUTPUT:        None
; REGISTERS:     A0, D0
;===============================================================================

CR              EQU     $0D         ; Carriage return
CHRIN           EQU     $F0420

READSTR         JSR     CHRIN       ; Read a character into D0
                JSR     CHROUT      ; Print it on the screen
                CMPI.B  #CR,D0      ; Exit if Carriage Return
                BEQ     RSTEND
                MOVE.B  D0,(A0)+    ; else, insert it into buffer
                BRA     READSTR
RSTEND          MOVE.B  #NUL,(A0)
                RTS
```

READINT, that appears below, reads a decimal unsigned number from the keyboard and converts it into a 32-bit unsigned integer in the following way: If a character is not a decimal symbol, the subroutine simply ignores it. Otherwise, it converts the string of decimal symbols into a 32-bit unsigned integer. To do this, we make use of the instruction MULU which multiplies the source operand by the destination operand (register D1) to yield a 32-bit unsigned product. There is a corresponding instruction for signed multiplication called MULS. For additional information on how to use them, see Appendix B.

```
;=============================================================================
; NAME:          READINT
; DESCRIPTION:   Reads and converts an unsigned decimal number.
; INPUT:         An integer pointed to by A0
; OUTPUT:        None
; REGISTERS:     D0,D1
;=============================================================================
READINT         MOVE.L  #0,D1
READL           MOVE.L  #0,D0
                JSR     CHRIN     ; Read character
                JSR     CHROUT    ; Echo it
                CMPI.L  #CR,D0    ; Exit if Carriage Return
                BEQ     RINT
                SUBI.L  #$30,D0
                BLT     READL     ; Less than 0?, ignore it
                CMPI.L  #9,D0
                BGT     READL     ; Greater than 9?, ignore it
                MULU    #10,D1    ; D1:=10*(D1)
                ADD.L   D0,D1
                BRA     READL
RINT            MOVE.L  D1,(A0)
                RTS
```

Finally, we show the assembly code for the declaration of the database on the next page. It is implemented using a memory area that consists of 10*52 bytes, sufficient to store ten records. A critical issue in this application is how to prevent the user from inserting too many records. The solution to this problem is to test whether there is a sufficient amount of memory to insert a new record in INSREC. We have deliberately overlooked this test and will only mention that the program developed in this section is not guaranteed to work correctly if more than 10 records are inserted.

```
;===============================================================================
; Declaration of the DATABASE
;===============================================================================

DATABASE  DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          DS.B  52
          END
```

## Exercises

**5.8** Write a subroutine `CONVERT` that converts the ASCII-character in `D0` as follows: If it is upper-case (A,B,C,...) then it does nothing, if it is lower-case (a,b,c,...) it converts it into upper-case (Use the ASCII-table from Chapter 1).

**5.9** Use the subroutine from the previous exercise to write another subroutine `CSTR` that converts a string of characters pointed to by `A0` and ended by ASCII `NUL`.

**5.10** Define a table `TAB` with N integers, each occupying one word. Then write a subroutine that adds all these integers and returns the value in register `D0`.

**5.11** Write a program that calculates and prints all Fibonacci numbers $\leq 65\,535$. `PUTINT` can be used to print a 16-bit number stored in register `D0`. The Fibonacci numbers are defined recursively as:

$$a_0 = a_1 = 1$$
$$a_i = a_{i-1} + a_{i-2}, \qquad i > 1$$

**5.12** Implement the `PRSEX` procedure according to the Pascal specification on page 75.

## 5.4 Testing and debugging

Testing is the phase of the design process in which the programmer identifies program errors, bugs. The debugging phase aims at locating the bugs and removing them. These two phases are repeated until the program is virtually free from bugs. Virtually, because in practice it is impossible to prove correctness of large programs.

Many people tend to believe that the major part of the time to develop a program is spent at the specification and coding phase. This is not the true picture at all. It is not unusual that testing and debugging count for half the time of the design process.

One way of shortening this time is to follow the advice given in the previous sections. By using a structured approach in the design phase, the number of bugs will be reduced. Furthermore, bugs that are introduced will be easier to locate. It is not very likely that a program will work correctly the first time it is run. Most programmers will not experience this during their life-time, apart from very small problems. Therefore, an important part of the programming methodology is to devise some rules for testing and debugging, which will be done in this section.

The first kind of error test is carried out by the assembler. The assembler checks the program for syntactical errors. It can detect nonexistent instructions and symbolic names that are not defined. However, it cannot detect logical errors, which is a very important fact to be aware of. The kind of errors that the assembler can detect are called *assembly-time errors*. We will not discuss these in further detail. Instead, we will focus on program errors that pass the assembler and are detected at run-time. These are called *run-time errors*, or bugs.

The task of a program is to generate a certain output (result) for a certain input. This task is the core of the testing phase, namely, the programmer should devise some tests consisting of a number of input/output pairs.

Once a faulty output is detected, the debugging phase aims at locating the bug that caused this output. This is usually facilitated by a tool called a *debugger*. A debugger is a program that is run in order to aid the programmer in locating the bugs. We say that the processor is in *debug mode*, when the debugger is executed and in *user mode* when the program under test is run. At the assembly language level, a debugger usually provides the following facilities:

- *Single stepping.* The program can be executed one instruction at a time in user mode. The processor enters debug mode after each instruction.
- *Execution.* The program can be started at an arbitrary address and run at full speed.
- *Memory and register examination and modification.* The contents of memory locations and registers can be inspected and altered when the processor is in debug mode.
- *Running with breakpoints.* Addresses, so called *breakpoints*, can be specified so as to enter debug mode when the breakpoints are reached.

Let us look at how these facilities can be used in a typical debugging session. A common situation is that the program never terminates when it is run for the first time. In this case a first step is to make an educated guess as to where the bug is located. The second step is to set some breakpoints in order to isolate the location where the bug is and execute the program. If the breakpoints are reached, the task is to make sure that the register and variable contents are exactly as expected. If not, the bug is tracked. Otherwise, continue until the bug is found. One can also make use of the 'Single step' facility to chase the bug when you are close to it.

We end this chapter by presenting a few bugs that are common to novice assembly language programmers:

**Structure:**

```
        MOVE.L  #VAR,D0
LOOP    SUBI.L  #1,D0
VAR     DS.L    1
        BNE     LOOP
```

Although this program is syntactically correct (the assembler won't complain), it will not work as expected. The problem is that location `VAR` is defined in the middle of the code. The processor will interpret the content of `VAR` as an instruction, instead of fetching the branch instruction. Note that the assembler doesn't rearrange instructions – it simply translates them one after another.

**Operand size:**

```
MAIN    ORG       $8000
TAB     DC.B      1,2,3,4
        EVEN
START   MOVEA.W   #TAB,A0
        MOVE.B    #0,D0
LOOP    ADD.B     (A0)+,D0
        SUBI.L    #1,D0
        BNE       LOOP
```

Some programmers believe that the start address of an assembly language program is the one that corresponds to the first line, that is $8000_{16}$ in the example above. This is of course not true. Therefore, always make sure at what address the first instruction of a program starts.

Another important problem is the proper use of operand size attributes. In the example above, we have used operand size Word to load `A0`, which is in general wrong! The reason is that addresses are considered as 32-bit unsigned integers. In

general, the choice of operand size should be carefully considered.

The programmer should also avoid accessing a word or double word on odd addresses. This results in address error (which will be explained in Chapter 8) and is in particular difficult to locate. A related problem, which has been discussed earlier, is to avoid instructions to be loaded at odd addresses at memory. This could be avoided by inserting the EVEN assembler directive prior to the code.

**Addressing modes:**

```
MAIN    ORG       $8000
VAR     DC.L      START
START   MOVEA.L   #VAR,A0
        MOVE.B    (A0),D0
        MOVEA.L   VAR,A0
        MOVE.B    (A0),D0
```

Another common problem is to use the addressing modes incorrectly. The first instruction loads the **value** of VAR (= $8000_{16}$) into A0, while the third instruction loads the **content** of VAR (=START=$8004_{16}$) into A0. **Make sure that you have understood the difference between all the addressing modes we introduced in the previous chapter (see page 49)!**

**Branches:**

The reason for executing a program loop too many or too few times can be that the loop variable is initialized incorrectly:

```
        MOVE.L   #0,D0
LOOP    ADDI.L   #1,D0
        CMPI.B   #5,D0
        BGT      LOOP
```

Note that the branch is taken 4 times (and not 5) times. Another problem could be the incorrect use of conditional branch instructions.

**Improper base for constants:**

Forgetting a '$' or '%' when dealing with hexadecimal and binary numbers may introduce severe bugs.

## 5.5 Summary and concluding remarks

In this chapter, we presented a methodology for designing and testing assembly language programs. The primary objective is to design for reliability.

The assembler provides support for this by the assembler directives. These are used to define symbolic names for certain entities such as memory locations, constants etc.

By using a high-level language notation to specify the task of the program, we can use a top-down methodology in order to refine the specification for a final implementation by assembly language instructions. By using standard translations for certain common high-level language constructs, we can translate these in a mechanical fashion and thus support reliability. The pseudo-code can serve as in-line comments in the course of documentation.

We illustrated the methodology developed in the first two sections by the development of a fairly large example program. We imposed a structure in which we made considerable use of headers as a means of documenting the subroutines and global data structures.

Finally, we gave some advice on how to locate and remove program errors. An important tool in the course of debugging is the debugger which enables the programmer to execute the program in a controlled fashion by letting him examine the content of relevant registers and memory locations.

Chapter 6

# Input and Output Control

In this chapter, we will refine our model of the computer system introduced in Chapters 3 and 4 by adding the important concept of I/O. Other topics to be discussed in this chapter are how the return addresses from subroutines are handled, and some other features of the processor that enhance performance and reliability.

## 6.1 Input and output model

In the simplified models we introduced in the previous chapters, information can only be transferred between the processor and the memory. A computer that is not capable of exchanging information with the outside world is rather useless. In order to explain how this is performed, we need to extend our model to include the important concept of input/output (I/O).

In order for a computer to exchange information with the outside world, there are dedicated registers denoted I/O-ports. The I/O-ports are connected to input and output devices as shown in Figure 6.1. They are shared between the processor and the I/O-devices; an input device can write to a port while the processor can read from the same port. The implication of this is that it can be meaningful for a program to perform successive reads from a port, because an input device may have changed the content of the input register in between two successive read operations. Conversely, it can be meaningful for the processor to perform successive writes to a port, with no intervening read operations, because an output device may have read the last value written by the processor.

Some ports are dedicated to transferring information from the environment to the processor. These are called *input ports*. Conversely, some ports are dedicated for transferring information in the opposite direction. These are called *output ports*. From the discussion so far, it should be clear that it is not meaningful to read from output ports. By the same reason, it is not meaningful to write to an input port. In the example computer system in Figure 6.1, there is one input port, one output

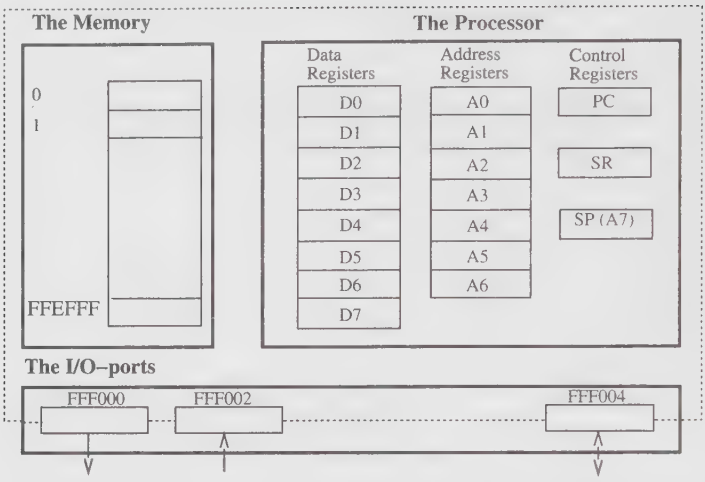



**Figure 6.1** A computer model including I/O-ports.

port, and a combined input/output port.

From the memory model's point of view, M68000 identifies parts of the address space as I/O-port addresses which are connected to dedicated I/O-ports. While the instruction set model is the same for all computers based on the M68000, the available amount of memory and I/O-ports may differ. The *memory map* of a computer system specifies exactly how the address space of the processor is distributed between available memory locations and I/O-ports. The manufacturer of a computer system usually offers this information to the system programmer. The example computer system in Figure 6.1 connects addresses 0 $\text{FFEFFF}_{16}$ to memory locations and addresses $\text{FFF000}_{16}$, $\text{FFF002}_{16}$, and $\text{FFF004}_{16}$ to I/O-ports.

The implications of connecting I/O-ports to memory addresses are twofold (i) I/O-ports can be read and written to using the addressing modes provided for memory locations, and (ii) there are no dedicated instructions for information transfer between the processor and the environment; information is simply transferred using the instruction set relevant for accessing memory locations. This type of I/O scheme is usually referred to as *memory-mapped* I/O.

Let us look at an example. The following instruction reads a byte from the input port at address $\text{FFF002}_{16}$ and copies it to memory location `IO`:

```
INPORT  EQU     $FFF002
        MOVE.B  INPORT,IO
```

Now suppose that an input device, such as a keyboard, is connected to an input port. Furthermore, suppose that we want to design a program that reads characters from the keyboard. We then run into the problem of how to detect when there is a

new character available in the input port. Conversely, suppose that an output port is connected to a printer. Now if a program transfers characters from the memory to the printer faster than the printer can receive them, characters that have not been printed out might be overwritten in the output port. Both these situations are examples of the general issue of how to synchronize the information transfer between the computer and an input/output device.

One commonly used technique to solve this problem is to associate a flag with each port that can be read by the processor at a dedicated I/O-address. For an input port, the flag is set when a new value is available and reset when the processor reads from the input port. For an output port, the flag is set when the value has been read by the output device, and reset when a new value has been written by the processor. Given such flags, we can design a program that synchronizes the information between the processor and any device.

For example, suppose that a flag is available in the least significant bit of port `FLAG`. This flag is set when data is available in input port `INPORT`. We can then write a program that reads from the input port when data is available as follows:

```
LOOP  ANDI.B  #%00000001,FLAG
      BEQ     LOOP
      MOVE.B  INPORT,(A0)+
      BRA     LOOP
```

Note how the program makes use of the logical AND instruction to determine whether the least significant bit of input port `FLAG` is set. If it is cleared, the result of this operation is zero and the subsequent branch is taken. If the flag is set, the execution continues at the next instruction which copies the content of the input port at address `INPORT` to a memory location pointed to by A0. The use of the logical AND instruction demonstrates a means of testing a single bit. We have used a constant $00000001_2$ to check bit 0. This constant is often called a *mask* because we mask all bits except for the least significant one. So, regardless of the contents of bits 1 through 7, the result is zero iff the content of bit 0 is zero.

There is a special instruction that tests a particular bit in a location, known as `BTST #C,`*a*. It sets the Z-flag if bit C in location *a* is cleared. We could use this instruction instead of the logical AND instruction:

```
LOOP  BTST    #0,FLAG
      BEQ     LOOP
      MOVE.B  INPORT,(A0)+
      BRA     LOOP
```

The technique we have demonstrated is known as *polling* or *busy-waiting* because the processor repeatedly asks whether the input or output device is ready to send

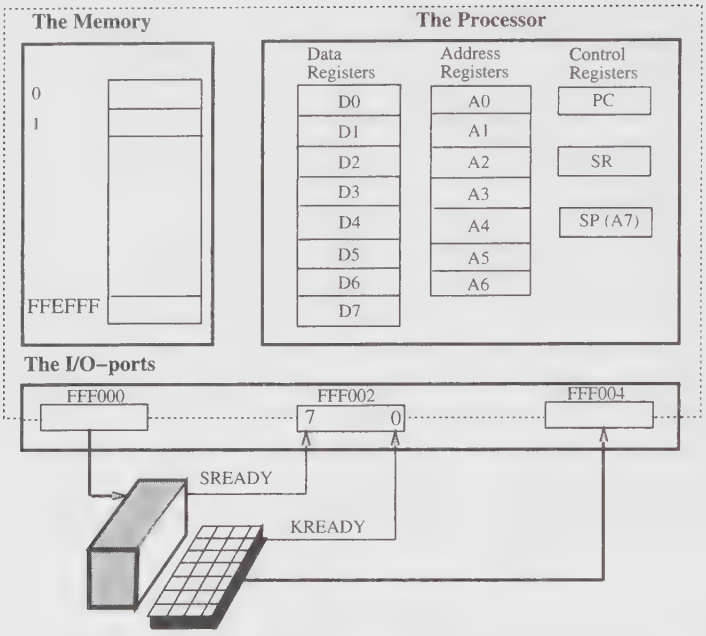



**Figure 6.2** An example of how a terminal is connected to a computer system through I/O ports.

or receive a new value. It is a simple and reliable scheme but can waste time if the transfer rate is small compared to the execution speed of the processor: in this case the processor will be occupied with testing the flag most of the time. A more attractive solution, in this case, would be to let the input device notify the processor when there is data that can be transferred. Using such a scheme would allow the processor to perform useful work instead of busy-waiting on a flag to be set. This scheme, which is known as *interrupt*, will be presented later in this chapter.

Our I/O model is useful in the implementation of various schemes that transfer information in between the computer system and the outside world. Consider the computer system in Figure 6.2. It consists of two input ports at addresses $FFF002_{16}$ and $FFF004_{16}$, respectively, and an output port at address $FFF000_{16}$. A keyboard is connected to $FFF004_{16}$. The output port can transfer ASCII characters to the terminal screen. Two flags are provided to indicate that a new ASCII character is available (`KREADY`) and that the terminal screen is ready to take care of a new ASCII character (`SREADY`). These facilities can be used to implement the two fundamental subroutines (`CHRIN` and `CHROUT`) we used in Chapter 5 to read and write characters to/from a terminal. The `CHRIN` subroutine can be implemented as

```
KREADY  EQU     0
STATUS  EQU     $FFF002
INPORT  EQU     $FFF004
CHRIN   BTST    #KREADY,STATUS
        BEQ     CHRIN
        MOVE.B  INPORT,D0
        RTS
```

and the `CHROUT` subroutine can be implemented as

```
SREADY   EQU     7
STATUS   EQU     $FFF002
OUTPORT  EQU     $FFF000
CHROUT   BTST    #SREADY,STATUS
         BEQ     CHROUT
         MOVE.B  D0,OUTPORT
         RTS
```

Note that the parameter (either the ASCII character to be read or to be written) is passed by register `D0`. The subroutines `CHRIN` and `CHROUT` that we have shown are often provided by the operating system, that is, the basic software that offers commonly used service routines to the programmer. A program that interacts with input and output devices is often called a *device driver*. In Chapter 7, we will go deeper into the details of designing device drivers.

When you input commands or any text to a computer system, all characters you type are usually printed out on the terminal screen. This process is called *echoing* because the processor echoes all characters that are read from the keyboard to the terminal screen. Below, we combine the programs that read and write characters so that all characters that are typed on the keyboard are echoed on the terminal screen.

```
CHRIN   BTST    #KREADY,STATUS
        BEQ     CHRIN
        MOVE.B  INPORT,D0       ; Read the character
CHROUT  BTST    #SREADY,STATUS
        BEQ     CHROUT
        MOVE.B  D0,OUTPORT      ; Echo the character
        BRA     CHRIN
```

### Exercises

**6.1** Write a sequence of instructions that reads the content of input port at address $FFF100_{16}$, multiplies it by 4 and writes the result to output port $FFF102_{16}$ without using polling.

**6.2** Two input ports are available at addresses $FFF100_{16}$ and $FFF102_{16}$. An output port is available at address $FFF104_{16}$. Write a program that repeatedly performs bitwise logical AND between the input ports and writes the result to the output port without using polling.

**6.3** Given eight input devices. Input device $i$, where $i = 0, 1, \ldots, 7$ is connected to input port $i$ at address $(FFF000_{16} + i)$. When bit $i$ in input port `STATUS` at address $FFF008_{16}$ is set, data from device $i$ is available at its input port. Write a program that repeatedly checks input port `STATUS` through polling. When data is available at input port $i$, its content is copied to memory location $(9000_{16} + i)$.

## 6.2 Stacks and subroutines

A branch-to-subroutine instruction is like a branch in that the PC (program counter) is loaded with a new value, but a call must also save the old value (that is, the address of the instruction following the branch-to-subroutine instruction) somewhere so that it can return correctly. We refer to this address as the *return address*.

To support subroutine calls and returns, there are a number of approaches that turn out to have serious weaknesses. One solution would be to have a special location for the return address as illustrated in the program according to Figure 6.3. In Figure 6.3, a subroutine call is implemented by storing the return address at location `RETADDR` (`MOVE.L #RET1,RETADDR`) and an unconditional branch to the subroutine address (`JMP SUBR1`). The return-from-subroutine instruction (`RTS`) is implemented by a branch-instruction to the address stored at location `RETADDR` (`JMP (A0)` performs a branch to address ((A0))). This solution has a major drawback. It prevents a subroutine from calling another subroutine because, in this case, the content of location `RETADDR` will be overwritten, and the old return address is forgotten. For example, when `SUBR2` is called, the return address `RET1` is overwritten.

This problem could be solved by associating a location to store return addresses

```
RETADDR  DS.L     1

START    MOVE.L   #RET1,RETADDR  ; Save return address
         JMP      SUBR1          ; Branch to SUBR2
RET1     ...
         ...
SUBR1    ...
         MOVE.L   #RET2,RETADDR  ; Save return address
         JMP      SUBR2          ; Branch to SUBR2
RET2     MOVEA.L  RETADDR,A0
         JMP      (A0)           ; 'RTS' Branch to the address
                                 ; that is stored at RETADDR
SUBR2    ...
         MOVEA.L  RETADDR,A0
         JMP      (A0)           ; 'RTS' Branch to the address
                                 ; that is stored at RETADDR
```

**Figure 6.3** Using a special location to store the return address.

with each subroutine. We will still have a problem, namely, it will prevent a subroutine from calling itself, a so-called *recursive subroutine call.* The fact that subroutines should be able to call other subroutines (and as a special case calling themselves) has led to a technique to handle return addresses which is simplest to understand by looking at the following example:

```
START  BSR  SUBR1
RET1   ...
       ...
SUBR1  BSR  SUBR2
RET2   ...
       RTS
SUBR2  BSR  SUBR3
RET3   ...
       RTS
SUBR3  ...
       RTS
```

Consider the sequence of subroutine calls generated by the execution of the above program. This sequence is SUBR1, SUBR2, and SUBR3. The sequence of subroutine calls gives rise to the following sequence of return addresses: RET1, RET2, and RET3. Now when SUBR3 has been executed, the execution shall continue at address RET3.

When SUBR2 has been executed, the execution continues at address RET2, and finally, when SUBR1 has been executed, the execution continues at address RET1 in the main program. We make the following important observations. First, return addresses are generated in the order the subroutines are called (i.e. RET1, RET2, and RET3), and the return addresses are used in the reverse order (i.e. RET3, RET2, and RET1). Given a data structure that could keep track of return addresses in the same way as plates are stored in a dish well in a cafeteria (the plate that is retrieved from the dish well is the last one that was put into the dish well), we have solved the problem.

A data structure that handles objects (e.g. return addresses) in this way is called a *stack*. A stack is a list in which items (e.g. return addresses) can be stored and retrieved in reverse order. There are two primitive operations, PUSH and POP, associated with a stack. A PUSH operation stores a new item on the top whereas a POP operation retrieves the top item and a new item comes to the top. When a branch-to-subroutine instruction (BSR *address* or JSR *address*) is executed, the return address is simply PUSHed by the processor onto the stack and when a return-from-subroutine instruction is executed, the return address on top of the stack is retrieved and loaded into the PC.

The stack is implemented using a segment of the memory space in conjunction with a dedicated location called Stack Pointer (SP for short), which keeps track of the address of the top of the stack. The processor designer has to make a decision about whether to let the stack grow towards higher or lower addresses, as well as whether to let the SP point to the top item, or the first empty item, of the stack. In the M68000, SP points to the top item and the stack grows towards lower addresses. In Figure 6.4 we show how the stack is managed in the M68000. Initially $(SP) = i + 1$. A PUSH operation decrements the SP before a new item is put on top of the stack ($(SP) = i$). A POP operation removes an item from the top of the stack and increments the SP.

Before we look more closely into how return addresses are handled by the processor, we want to show how the machine language for the M68000 supports user-defined stacks. PUSH and POP operations can be implemented using the address registers introduced in Section 4.5 in conjunction with the indirect with post- and predecrement addressing modes. We illustrate the semantics of PUSH and POP by using address register A0:

```
MOVEA.L  #BOTTOM,A0  ; Initialize the stack
MOVE.L   D0,-(A0)    ; PUSH D0
MOVE.L   (A0)+,D0    ; POP D0
```

The first instruction initializes the Stack Pointer (in our example address register A0). The second instruction PUSHes the content of D0 onto the top of the stack, or formally: (A0) − 4 → A0 and (D0) → (A0). The third instruction POPs

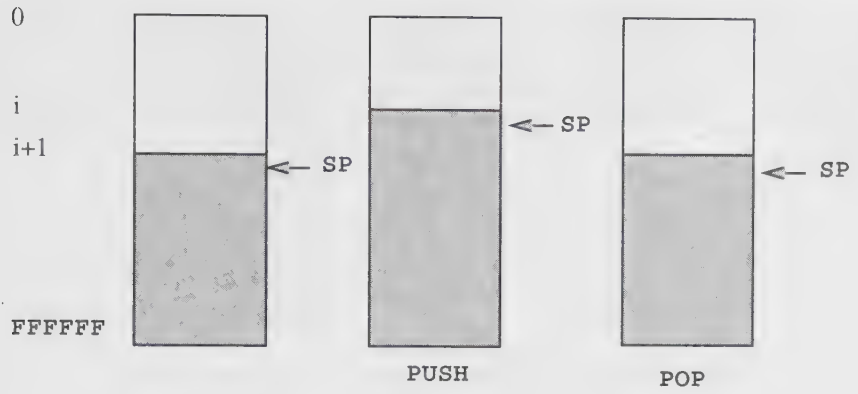



**Figure 6.4** The Stack Pointer (SP) and the function of PUSH and POP in the M68000.

and places the top item of the stack in D0, or formally: ((A0)) → D0 and (A0) + 4 → A0. Note that the stack pointer is decremented **before** the content of D0 is copied onto the top of the stack and incremented **after** the top element of the stack is removed.

In Figure 6.1, we have introduced the Stack Pointer (SP) in the model of the processor. SP (or A7) is from the point of view of the instruction set simply another address register in the sense that all instructions relevant for address registers (for example those listed in Table 4.6 in Section 4.5) are relevant for SP. However, an important feature of SP distinguishes it from the other address registers when a branch-to-subroutine instruction (BSR or JSR) is executed, the processor **automatically** performs a PUSH operation on PC using SP as a stack pointer. Likewise, when a return-from-subroutine instruction (RTS) is executed, the processor **automatically** performs a POP operation and places the content of the top of the stack in PC (see Table 6.1).

**Table 6.1** Semantics of the BSR and RTS instructions.

| Name | | Operation | |
|---|---|---|---|
| BSR | *address* | (SP)−4 | → SP; |
| | | (PC) | → (SP); |
| | | *address* | → PC |
| RTS | | ((SP)) | → PC; |
| | | (SP) + 4 | → SP |

Besides the automatic use of the stack by the subroutine instructions, the stack can be used to store temporary data by using the indirect addressing modes avail-

able for address registers. In the previous chapter, we had to state explicitly which registers are used by a subroutine in the commentary header associated with each subroutine. Sometimes it is important to leave all registers unaffected. This can be done by using the stack as a temporary memory space for the contents of the registers:

```
8000  MOVE.L  #$1234,D0
8006  MOVE.L  #$5678,D1
800A  BSR     SUBR
800C  ...

SUBR  MOVE.L  D0,-(SP)    ; PUSH D0
      MOVE.L  D1,-(SP)    ; PUSH D1
      ...
      MOVE.L  (SP)+,D1    ; POP D1
      MOVE.L  (SP)+,D0    ; POP D0
      RTS
```

In the above example program, the absolute addresses of some of the instructions appear to the left. In subroutine `SUBR`, we have used the stack to save the contents of registers D0 and D1. Note that we POP them from the stack in the reverse order (D1 before D0).

It is important to understand how the stack changes during the execution of the program above. In Figure 6.5, we show how the stack is affected by the program above. Assume that (SP)=$9000_{16}$ ((1) in Figure 6.5), initially. When the `BSR` instruction has been executed, the return address ($800C_{16}$) has been PUSHed onto the stack ((2) in Figure 6.5) and (SP)=$8FFC_{16}$. After the execution of the second instruction in the subroutine, D0 and D1 both have been PUSHed onto the stack as long words and (SP)=$8FF4_{16}$ (8 less than before, see (3) in Figure 6.5). Note how the contents of D0 and D1 are stored in the stack space in Figure 6.5. Finally, immediately before the `RTS` instruction has been executed, the content of SP is $8FFC_{16}$.

We have assumed that the stack is infinitely large, which in practice means that it is sufficiently large. A critical point is that the stack pointer must be initialized so that there is sufficient memory space to prevent the stack from overflowing. This could happen as a result of too many nested subroutine calls; if the program to be supported by the stack can perform $n$ nested subroutine calls, the stack space must exceed $4n$ bytes in order to have space for $n$ return addresses.

A word of warning is justified. It turns out be a common mistake that the stack is incorrectly handled. Consider the following erroneous subroutine:

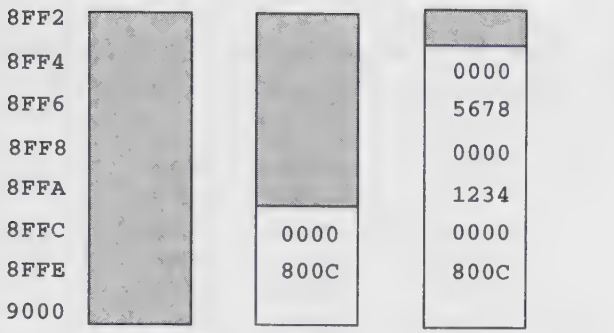



**Figure 6.5** The content of the SP and the stack at different places in the example program.

```
SUB  MOVE.L  D0,-(SP)
     RTS
```

The first instruction PUSHes the content of D0 onto the stack, while RTS POPs the top element of the stack (that is the old value of D0) and copies it into PC. This means that PC will not be loaded with the correct return address. Problems of this kind result in bugs that are especially hard to locate.

## Exercises

---

**6.4** In this exercise we shall implement a recursive algorithm, that computes the sum of all integers between 1 and N:

```
function NSUM( N );
begin
  if N=1 then NSUM:=1
         else NSUM:= N + NSUM( N-1 );
end;
```

If NSUM > 1 then NSUM must be called again. Note that the sum is computed first when NSUM has been called N times. This implies that one copy of N must exist for every instance of NSUM in the calling sequence, which means that we cannot use a fixed location for N. Write NSUM as a subroutine that implements the recursion above. *Hint:* Use the stack to save N.

**6.5** In the example program below, we show the addresses to the left. Analyze the program and answer the questions below:

```
8000          BSR      NSUM
8004          STOP     #$2700
8008  NSUM    MOVE.L   D0,-(SP)
800A          MOVE.L   #0,D1
8010          SUBI.L   #1,D0
8016          BEQ      OUT
801A          BSR      NSUM
801C          ADD.L    D0,D1
801E  OUT     MOVE.L   (SP)+,D0
8020          RTS
```

(a) Assume that (SP)=$9000_{16}$ and that (D0)= 3, initially. Show the content of the stack and the stack pointer each time `SUBI.L` is executed.(b) What does D1 contain when subroutine `NSUM` has been executed if (D0) = 3, initially?

**6.6** In the program below, `PREGS` pushes all data registers denoted by the word immediately following the subroutine call instruction in the following way: if bit $i = 1$ then D$i$ is pushed. `PREGS` also ensures that the return address is the address of the instruction immediately following the word (that is, `MOVE.L D1,D2`). Write a subroutine that implements `PREGS`.

```
JSR      PREGS
DC.W     %000000000100101   ; D5,D2,D0 are pushed
MOVE.L   D1,D2              ; Next instruction...
```

**6.7** We shall perform calculations using a stack which is maintained by `A0` as a stack pointer. Implement the following subroutines:

`ENTER`: Pushes the content of D0 onto the stack.
`ADDSTACK`: Pops the top of stack item and adds it to D0.
`SUBSTACK`: Pops the top of stack item and subtracts it from D0.
`POPSTACK`: Pops the top of stack item into D0.

**6.8** Consider the following program which uses the subroutines in the previous exercise

```
MOVEA.L   #$8000,A0
MOVE.L    #$10000005,D0
BSR       ENTER
MOVE.L    #$10000006,D0
BSR       ENTER
BSR       ADDSTACK
BSR       SUBSTACK
```

Show the content of the stack (and A0) after the execution of each subroutine.

---

## 6.3 Instruction execution rate

We have now introduced almost all important features relevant to the machine language programmer in order fully to take advantage of the functionality of a computer system. Our model so far has been functional and we have not addressed maybe the most important objective of using computers – the processing speed of a computer.

Efficiency is one of the primary objectives of all computing. The main explanation of the 'revolution of computers' is that computers can perform operations at a high rate. Typically, standard computers of today can perform in the order of 1,000,000 instructions per second, although this number is rapidly changing. Some instructions take a longer time to execute than others. Therefore, computer manufacturers use an average measure of instruction execution rate called the MIPS-*rate* (Million Instructions Per Second). Consequently, a 1-MIPS computer executes $10^6$ instructions/s on average.

In order to get an idea of how the instruction execution time differs for different instructions, we shall pick a few instructions from the instruction set of the M68000. In Table 6.2, we show the execution time of some instructions for specific addressing modes in terms of cycles. The cycle time may differ from computer to computer. To get a rough estimate of the execution time, we shall assume that the cycle time is 100 ns ($10^{-7}$ s).

From Table 6.2 we note that the execution time ranges from 4 cycles to 18 cycles, that is, close to five times. One would like to know the reason for this discrepancy.

The number of memory accesses carried out by an instruction has a first order effect on the execution time. For instance, if we compare the execution time of

**Table 6.2** Execution time in terms of cycles for some M68000 instructions.

| Instruction | | Cycles |
|---|---|---|
| MOVE.W | D0,D1 | 4 |
| ADD.W | D0,D1 | 4 |
| ADD.W | 10,D1 | 16 |
| BRA | LOOP | 10 |
| B*cc* | LOOP | 10/8 |
| BSR | SUBR | 18 |
| RTS | | 16 |

the two ADD instructions, we see that if the source operand is pointed out using absolute addressing, we get an increase by 12 cycles compared to register-direct addressing. The reason is twofold: (i) the instruction occupies three words (the operation word and two words for the absolute address) instead of one, and (ii) the operand to be fetched requires one extra word to be fetched from memory. This results in three more words to be fetched from memory.

The reason why a branch-to-subroutine instruction takes almost twice as long to execute than an unconditional branch-instruction has also to do with extra memory accesses. The difference between these two instructions is that the return address has to be PUSHed onto the stack before the branch is taken, thus requiring two words to be transferred to memory.

We also show the execution time for conditional branch-instructions. The execution time differs depending on whether the branch is taken or not. If the branch is taken, the execution time is 10 cycles. Otherwise it is 8 cycles. The reason for this discrepancy is that if the branch is taken, the PC must be loaded with a new address. This is not needed if the branch is not taken because the PC already points to the next instruction to be fetched, namely, the instruction that immediately follows the branch-instruction.

From this discussion we can conclude that when performance is crucial to a program, one should carefully consider the choice of instructions. Since most of the time is spent in executing loops, one should especially try to optimize these. Let us give a rough estimate of the execution time of the program below:

```
START   MOVE.L  #10000,D0
LOOP    SUBI.L  #1,D0
        BNE     LOOP
        STOP    #$2700
```

The loop is executed 10,000 times so the number of instructions executed is $1+2\times 10,000+1 = 20,002$. Assuming that each instruction takes 10 cycles to execute, the execution time is about $2{\cdot}10^5 10^{-7}$ s $= 2{\cdot}10^{-2}$ s, assuming that the cycle time is 100 ns.

## 6.4 Interrupts

In Section 6.1, we studied a scheme to synchronize an external event, such as a keyboard input, with the program execution. We called it polling, because the processor is polling an external device to find out whether it needs service. Now recall the example program from Section 6.1:

```
LOOP  BTST    #0,FLAG
      BEQ     LOOP
      MOVE.B  INPORT,(A0)+
      BRA     LOOP
```

Let us assume that the above program reads characters from a keyboard, and that a secretary is writing at a speed of ten characters each second. (This is tough even for an extremely experienced secretary!) What is the fraction of time the processor spends on useful computation? Considering what is performed in the program above, we must admit that the only useful computation is when the character is stored at the location pointed to by A0. This instruction takes about $10^{-6}$ s to execute, so the magnitude of the fraction of useful work is only $10^{-6}/10^{-1} = 10^{-5}$. This is of course not acceptable given the fact that the processor could perform 999990 useful operations each second instead of repeatedly asking the question 'Is there anything to me?' over and over again. A better approach would be to let the keyboard, or more general, the external device notify the processor when an action needs to be taken.

The mechanism that implements this concept is the *interrupt*. In the data input example above it works as follows: The synchronization flag is connected to an interrupt input on the processor. When a new data value is loaded into the port the processor automatically senses this. Instead of executing the instruction pointed to by PC, the processor **automatically** performs a subroutine call to a special subroutine called an *interrupt service routine*, which takes care of the input value. The interrupt service routine is ended with a special return instruction which resumes execution of the interrupted program.

The interrupt service routine is similar in structure with an ordinary subroutine. However, there is a fundamental difference between the two. While a subroutine is called from specific points in the program, determined by the programmer, an interrupt service routine can be called at an arbitrary point in the program because of the unpredictability of external events. This means that the programmer cannot

predict when the interrupt service routine is called so special actions need to be taken. Especially, an interrupt service routine must always return leaving **all** registers unaffected. We call the contents of all registers the *processor context*. This means that **if the interrupt service routine uses any registers, their contents must first be saved and later restored**. The interrupt service routine has the following basic structure:

```
INT   MOVEM.L   reg-list,-(SP)   ; Save registers used
                                 ; by the service routine
      ...                        ; Instructions that
      ...                        ; implement the service
      MOVEM.L   (SP)+,reg-list   ; Restore registers used
                                 ; by the service routine
      RTE                        ; Return from exception
```

There are a few remarks that need to be made regarding this example. First, besides pushing the return address onto the stack, the processor also automatically saves the status register. The reason for this is that almost all instructions affect the condition code register (CCR) which is part of the status register (recall Section 4.3). Second, all registers used by the interrupt service routine need to be saved. A safe way to do this is to use the stack. Note that we use a new instruction called `MOVEM`. It takes a list of register names as operands and stores them onto the top of the stack (we will talk about this instruction more in detail later in this chapter). Third, the dots represent the action that is to be taken to service the external event. Fourth, we need to restore the registers by making the reverse POP operation to copy the register contents from the stack. Finally, we use a special return-from-subroutine instruction (`RTE`). The reason for this is that, unlike an ordinary subroutine call, the status register is PUSHed onto the stack and needs to be restored. This is exactly what `RTE` does, besides POPing PC as the ordinary return-from-subroutine instruction `RTS` does.

Let us give an example on the use of interrupts. Consider a system in which a certain character string which is stored at location `STRING` shall be written to the screen as soon as an interrupt occurs. We use the `PRSTR` subroutine from Chapter 5 to solve this problem. This subroutine uses registers `A0` and `D0`, which means we have to save them:

```
INT   MOVEM.L   D0/A0,-(SP)
      MOVEA.L   #STRING,A0
      BSR       PRSTR
      MOVEM.L   (SP)+,D0/A0
      RTE
```

We have not discussed the issue of how the processor knows at what address the interrupt service routine is located. To make the problem even worse, the processor may have a number of interrupt inputs, each with an associated call address of its own. With several interrupt inputs, a priority order is usually defined so that the behavior is determined when more than one interrupt occurs simultaneously.

M68000 supports interrupts according to various schemes. In the following, we will present a scheme that is often used in computer systems with a small number of interrupts. It is referred to as the *autovector* interrupt scheme. In Chapter 7, we will look at an extended model of the interrupt system called *vectored interrupts*. The base for the autovector interrupt system is seven interrupts which we denote $I_1$ to $I_7$. In order for the M68000 to keep track of the addresses of the interrupt service routines that service these interrupts, there is a table called *exception vector table* which consists of one entry for each interrupt input. The machine language programmer is responsible for initializing the entries in this table. The address of the interrupt service routine that is connected to interrupt input $I_n$ is stored at address $4(18_{16} + n)$. In the table below, we show the addresses of the entries for $I_1$ through $I_7$.

| Interrupt input | Address |
|---|---|
| $I_1$ | $64_{16}$ |
| $I_2$ | $68_{16}$ |
| $I_3$ | $6C_{16}$ |
| $I_4$ | $70_{16}$ |
| $I_5$ | $74_{16}$ |
| $I_6$ | $78_{16}$ |
| $I_7$ | $7C_{16}$ |

Since there are more than one interrupt inputs, there must be a rule for how to deal with several interrupts that are activated at the same time. The general rule is that if two interrupts are activated at the same time, the one with the highest number will get service first.

Another question is how to prevent an interrupt input from getting service. This is solved by associating a priority level with the processor. We refer to this as the *current priority level* (CPL). The general rule is that a certain interrupt input $I_n$ can interrupt the processor provided that $n >$CPL. There is one exception to this rule, namely, if $n = 7$. Interrupt input $I_7$, known as *non-maskable interrupt*, can always interrupt the processor.

The machine language programmer can set the CPL which is done by accessing the status register. In Figure 6.6, we show the computer system model again, highlighting all the bits contained in the status register (SR). The CPL is controlled by bits 8 -10 called C1, C2, and C3. They can be modified by the following move instruction that sets the current priority level to 5:

```
MOVE  #$2500,SR
```

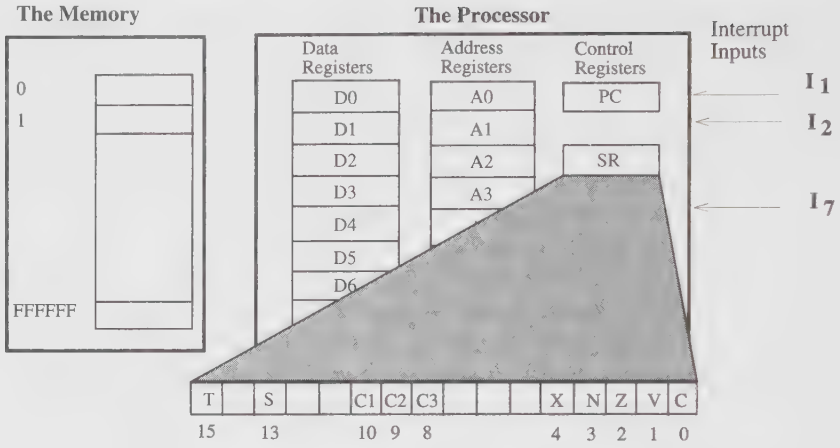



**Figure 6.6** The control bits in the status register.

Note that the above instruction sets (C1 C2 C3) = $101_2$, that is, the CPL is binary coded by bits 8–10 in SR. There are two other bits named T and S (bits 15 and 13) whose function we will describe in Chapter 8. We shall only mention that the S-bit must be set in order to change the CPL. This is why bit 13 in the word $2500_{16}$ ($= 00101010000_2$) is set. We say that an interrupt is *enabled* when the processor can be interrupted by this interrupt. For M68000, we note that interrupt $n$ is enabled when CPL $< n$, $n < 7$. Interrupt $I_7$ is always enabled.

We are now able to write a program that takes care of more than one interrupt. In the following example, we want to service interrupt inputs $I_2$ and $I_5$ with two interrupt service routines `INT2` and `INT5`. We show the necessary initializations for these interrupts:

```
START   MOVE.L   #INT2,$68   ; Initialize INT2
        MOVE.L   #INT5,$74   ; Initialize INT5
        MOVE     #$2100,SR   ; Enable interrupts with CPL>1
                             ; Interrupts are enabled here...
LOOP    BRA      LOOP

INT2    ...
        RTE
INT5    ...
        RTE
```

In the program above, we first initialize the exception vector table entries that correspond to `INT2` and `INT5`. We then enable these interrupts by setting CPL=1 (why 1?). Note that it is important to enable the interrupts after the exception

vector table has been initialized. Otherwise, the interrupt service routines cannot be called and the system can act unpredictably. In general, therefore, it is important to consider carefully when to enable the interrupts because an interrupt can theoretically happen immediately after this point in the program.

To summarize, in order to design a program in which a certain service is to be achieved when an interrupt is encountered, the machine language programmer is responsible for the following:

- Design an interrupt service routine which saves and restores all registers affected by the routine.
- Initialize the exception vector table and enable the interrupts.

We are now able to write interrupt service routines but have not discussed what happens when the processor encounters an interrupt which is to be done now. While interrupts can occur at any point in time, the processor checks whether an interrupt has occurred after each instruction only. Recall the instruction cycle from Chapter 3:

**Step 1**: Fetch the instruction at the memory address specified by PC.
**Step 2**: Update PC.
**Step 3**: Execute the instruction.
**Step 4**: Check if there are any pending interrupts.

We have augmented the instruction cycle with the interrupt check (Step 4). When an interrupt from interrupt input $I_n$ occurs, the processor performs the following tasks automatically, without involving any machine language program:

**Step 1**: If $n \leq$ CPL, nothing is done. Otherwise,
**Step 2**: it makes an internal copy of the content of SR (status register).
**Step 3**: CPL:=$n$ in SR, in order to prevent further interrupts at the same or lower priority level.
**Step 4**: It sets the S-bit and clears the T-bit in SR.
**Step 5**: It pushes PC onto the stack. Then it pushes its internal copy of SR (the old value) onto the stack.
**Step 6**: It loads PC with the entry $4(18_{16} + n)$ and performs a branch to this address (the interrupt service routine).

When the interrupt service routine has been executed, that is, when the `RTE` instruction is executed, the processor resumes execution at the point in the program where it was interrupted and with the same SR content as before the interrupt occurred. The processor performs this by popping SR and PC from the stack (recall that these values were pushed in the reverse order). Note that since SR is restored, the priority level that was set before the interrupt occurred is restored.

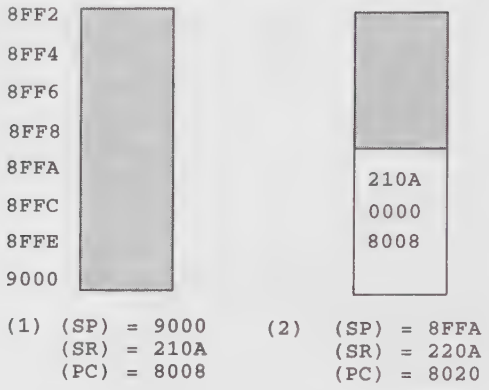



**Figure 6.7** The content of the SP at various places during the interrupt.

Consider the following program:

```
        ...
8000    MOVE.B  #$7F,D0
8004    ADDI.B  #1,D0     ; An interrupt occurs here
8008    MOVE.B  #1,D1     ; This instruction is executed
                          ; when the interrupt has been
                          ; handled
8020    ...
        RTE
```

Assume that an interrupt occurs at interrupt input $I_2$ when the `ADDI` instruction is being executed. The corresponding interrupt service routine is located at address 8020. When the interrupt service routine has been executed, the execution continues at address 8008. We further assume that the initial contents of the status register and the stack pointer are (SR)=2100 and (SP)=9000, respectively. The content of the condition code register will be changed by the `ADDI` instruction so that the content of the status register is (SR)=210A when the interrupt occurs (confirm this as an exercise). We will now look at how the contents of the stack, the program counter, and the status register change when the interrupt service routine is invoked. The contents of SP, SR, and PC when `ADDI` is being executed are shown to the left in Figure 6.7. When the interrupt is being processed by the processor, PC and SR will be pushed onto the top of the stack ((2) in Figure 6.7). Note also that the current priority level is changed to the same priority as the interrupt ((SR)=220A, i.e. CPL=2) when the first instruction of the interrupt service routine is to be executed. When finally the interrupt has been processed, the content of SP, SR, and PC are exactly the same as before the interrupt occurred ((1) in Figure 6.7).

We will now look at an application in which interrupts from a timer form the base. A timer is connected to interrupt input $I_5$ in such a way that an interrupt is generated each millisecond. We will implement a program that keeps track of the actual time using four memory locations: TICK, SEC, MIN, and HOUR. These locations contain the actual time in milliseconds (TICK), seconds (SEC), minutes (MIN), and hours (HOUR).

We first show the algorithm using Pascal-like code to explain what the interrupt service routine is supposed to do:

```
procedure TIME;
begin
   TICK:=TICK+1;
   if TICK=1000 then
   begin
      TICK:=0; SEC:=SEC+1;
      if SEC=60 then
      begin
        SEC:=0; MIN:=MIN+1;
        if MIN=60 then
        begin
          MIN:=0; HOUR:=HOUR+1;
          if HOUR=24 then
             HOUR:=0;
        end;
      end;
  end;
end; (*** RTE ***)
```

The interrupt service routine to be invoked every millisecond appears in Figure 6.8. Note that we do not need to save and restore any registers, because the interrupt service routine does not affect any of them. The initializations that are needed to get it to work are as follows:

```
MAIN  MOVE.L  #0,TICK     ; TICK:=0
      MOVE.L  #0,SEC      ; SEC:=0
      MOVE.L  #0,MIN      ; MIN:=0
      MOVE.L  #0,HOUR     ; HOUR:=0
      MOVE.L  #TIME,$74   ; Initialize exception table
      MOVE    #$2400,SR   ; Enable I5
LOOP  BRA     LOOP
```

```
TIME  ADDI.L  #1,TICK      ; TICK:=TICK+1;
      CMPI.L  #1000,TICK   ; if TICK=1000 then
      BNE     BACK
      MOVE.L  #0,TICK      ; TICK:=0;
      ADDI.L  #1,SEC       ; SEC:=SEC+1;
      CMPI.L  #60,SEC      ; if SEC=60 then
      BNE     BACK
      MOVE.L  #0,SEC       ; SEC:=0;
      ADDI.L  #1,MIN       ; MIN:=MIN+1;
      CMPI.L  #60,MIN      ; if MIN=60 then
      BNE     BACK
      MOVE.L  #0,MIN       ; MIN:=0;
      ADDI.L  #1,HOUR      ; HOUR:=HOUR+1;
      CMPI.L  #24,HOUR     ; if HOUR=24 then
      BNE     BACK
      MOVE.L  #0,HOUR      ; HOUR:=0;
BACK  RTE
```

**Figure 6.8** Assembly code for the interrupt service routine that implements the clock.

Note that we must initialize the locations `TICK`, `SEC`, `MIN`, `HOUR` before we enable the interrupt.

## 6.5 Additional useful instructions

We will end this chapter by presenting in this section the instructions we have introduced in this chapter and make some remarks on the use of them. They appear in Table 6.3.

The `BTST` instruction tests the bit in *a* denoted by a data register $D_i$ or a constant (using immediate addressing). It only affects the Z-flag.

The `MOVEM` *reg-list,b* instruction copies the contents of the registers denoted by *reg-list* to the consecutive addresses where the first address is denoted by operand *b*. *b* may designate absolute addressing, indirect addressing, and indexed (with displacement) addressing. The syntax of the register list is as follows: A range of registers such as `D0, D1, D2` is denoted `D0-D3`, while a list of registers such as `D1,A3,A5` is denoted `D1/A3/A5`. For example, this program

```
MOVEA.L  #$9000,A0
MOVEM.L  D0-D2/D5/A0-A3,(A0)
```

**Table 6.3** Program control instructions introduced in this Chapter.*S* denotes W or L. *b* is a restricted set of addressing modes (see text)

| Name | | Operation |
|---|---|---|
| BTST | $D_i$,*a* | if bit ($D_i$) of (*a*) is set then 0 → Z else 1 → Z |
| BTST | #C,*a* | if bit C of (*a*) is set then 0 → Z else 1 → Z |
| MOVEM.S | *reg-list,b* | (*reg-list*) → *b* |
| MOVEM.S | *b,reg-list* | (*b*) → *reg-list* |
| MOVE | *a*,SR | (*a*) → SR |
| MOVE | SR,*a* | (SR) → *a* |
| RTE | | ((SP))→ SR; (SP)+2→ SP; ((SP)) → PC; (SP) + 4→ SP |

copies the contents of D0,D1,D2,D5,A0,A1,A2,A3 to addresses $9000_{16} - 901F_{16}$ because each register occupies 4 bytes which adds up to 32 bytes.

The instruction MOVEM *b,reg-list,* performs the opposite operation. We need to make a remark when the source operand uses indirect addressing with predecrement (which was used in the generic interrupt service routine on page 102). In this case it is important to pop the registers in the reverse order, the MOVEM *b,reg-list* instruction is smart enough to do so:

```
MOVEM.L  D0-D2/D5/A0-A3,-(A0)
MOVEM.L  (A0)+,D0-D2/D5/A0-A3
```

In this example, the registers are pushed in the order specified by the sequence in the register list. However, they will be popped in the reverse order.

The MOVE SR,*a* and MOVE *a*,SR facilitates copying of SR to any location as specified by the addressing mode denoted by *a*. Since SR is a 16-bit register, the default size is word. For additional information on the use of these instructions, please refer to Appendix B.

## Exercises

**6.9** Consider the interrupt service routine TIME on page 108. A switch is connected to interrupt input $I_2$. Write an interrupt service routine connected to this interrupt that sets the variables TICK, MIN etc. to zero. Also, modify the main program to initialize this interrupt.

**6.10** A timer generates an interrupt each 100 ms at $I_5$. Fifteen ASCII-character displays are connected to output ports at addresses $5700_{16} - 570E_{16}$. The ASCII-codes of the string '68000 FOR EVER' are stored at address $6000_{16} - 600E_{16}$ (ended by NUL). In the subsequent exercises, we shall design a complete program that results in the following successive sample printouts on the displays each second:

```
68000 FOR EVER
8000 FOR EVER
000 FOR EVER 6
00 FOR EVER 68
```

Write a subroutine MCHAR according to the Pascal-specification below:

```
procedure MCHAR;
begin
  POINTER:=POSITION;
  for I:=0 to 14 do begin
     if STRING[POINTER] = NUL then
        POINTER:=0;
     DISPLAY[I]:=STRING[POINTER];
     POINTER:=POINTER+1
  end;
  POSITION:=POSITION+1;
  if POSITION > 14 then
     POSITION:=0;
end;
```

**6.11** Write an interrupt service routine that each second calls subroutine MCHAR in the previous exercise. All registers must be left unaffected when the routine is exited.

**6.12** Write a main program that initializes interrupt $I_5$ and the variable POSITION in the previous two exercises.

---

## 6.6 Summary and concluding remarks

In this chapter, we have looked into special program control features needed to support such things as I/O and subroutines. All communication with the outside

world is achieved through I/O-ports, which in the M68000 is done through special memory locations. They are memory locations in the sense that information transfer is supported by the same subset of the instruction set that is used to transfer information to/from memory locations. However, they are special in the sense that the information written to an output port can generally not be read afterwards. Similarly, information read from an input port may change from one time to another without having written to it in between.

An important scheme to synchronize an external event to the actions taken by the machine language program was to check repeatedly a dedicated I/O-location that is affected by the external device that needs service. This scheme is known as polling or busy-waiting.

In order to support nested subroutines, most computers manage the return addresses by a data structure called a stack. Upon a subroutine call, the PC (containing the return address) is automatically pushed onto the stack. Upon a return from the subroutine, the return address is popped from the stack. The stack can also be used to store register contents, in order to leave all registers unaffected.

Another synchronization scheme known as interrupt was also investigated. The motivation behind this scheme is when the external events are rare. In this case, the polling scheme will waste most of the capacity of the processor to check for external events. Therefore, most computers have dedicated interrupt inputs that, when activated, makes the processor perform a subroutine call to an interrupt service routine. The machine language programmer must design the interrupt service routine and connect it to a dedicated interrupt through something we called exception vector table. Furthermore, the machine language programmer must enable the interrupt at an appropriate point in time, usually when all initializations have been performed.

In order to handle several external events, most computers have not only one interrupt input, but a number of them. This raises the issue of priority. Each interrupt input is associated with an interrupt priority. This is used to resolve several interrupts that occur at the same time.

Chapter 7

# Programmable Input/Output Interfaces

In the previous chapter, we studied how to transfer information between the computer system and the environment through I/O-ports. The simplified model we presented viewed the I/O-ports as special memory cells. We did not discuss the issue of how a particular input/output device such as e.g. a printer or a terminal from one manufacturer can be connected to a computer system from another manufacturer so that they can communicate with each other. In general when we have two units, such as a computer system and a printer, a rule is established for how they are supposed to communicate. Such a rule is called a *communications protocol*. The communications protocol establishes a set of requirements to make it possible to transfer information between two units.

Let us consider an example. Suppose that we have purchased a printer from one manufacturer and a computer system from another. Our task is to connect these two units and implement a printer device driver so that we can send characters to this unit. There are two separate issues involved in this task: (i) how do we connect the units, and once they are connected, (ii) what is the synchronization scheme supposed to look like? Are we going to use polling or interrupt-driven communication?

Information between two units is usually transferred using a cable with one or several lines, where each line can transfer a bit. Since each ASCII-coded character makes use of seven bits, a common way to connect a computer system to a printer is to use seven lines, one for each bit. This enables us to transfer one character at a time, in parallel. In order for a computer system to support parallel transfers to a printer, for example, there is often a *parallel interface*, that is, a connection to a port at which an input or output device can be connected. The parallel interface can be used if the physical distance between the computer system and the printer is small, typically less than a meter. However, suppose that the printer is located in a room far away from the computer system. One imagines that it would not be convenient to transfer characters in a parallel fashion because of the prohibitive cost of the cable that connects the printer to the computer system. There are also

electrical restrictions that make this alternative less attractive but this discussion is outside the scope of this text. A viable alternative in this case would be to use a single line to transfer one bit one after another. In essence, using the printer example, an ASCII-coded character could be transferred by seven successive bit transfers. There are special interfaces on most computer systems that transfer information in such a bit-serial fashion called *serial interfaces.*

In this chapter, we shall look at how communications protocols are established by special devices that support parallel and serial communication. Because of the variety of devices that can be purchased, these interfaces are often programmable in the sense that the programmer can set up the communications protocol so as to meet the requirement of the device that is to be connected to the computer system. Therefore, they are called *programmable interfaces.* In Subsection 7.1, we discuss parallel communication and an example programmable interface from Motorola that supports parallel communications protocols. In Subsection 7.2, we discuss bit-serial communication and how the communication protocol is set up with another programmable interface from Motorola. In the previous chapter, we noted that Motorola has seven interrupt inputs. A natural question is how we support more than seven interrupts. In fact, M68000 can support 192 interrupts by requiring that the unit that generates the interrupt identifies itself by an 8-bit number called an *interrupt vector.* In Subsection 7.3, we present the vectored interrupt mode of the M68000.

## 7.1 Parallel input and output

### 7.1.1 Bit I/O and handshake protocols

We will start the discussion in this section by considering a simple interface problem. Suppose that we want to connect two lamps and six switches to a computer. The computer is supposed to read the status of the switches and switch on the light of any of the lamps. This is an example of a common situation where single bits are to be read or written to a device. It is called *bit* I/O. We can use a combined input/output port with eight bits to solve this interface problem as shown in Figure 7.1. In Figure 7.1, we use two bits as output ports for the lamps, whereas six bits are used as input ports for the switches. In another situation, we may want to use six lamps and two switches, that is; six output ports and two input ports are needed. One realizes that it would be convenient if a single-bit port could be programmed to act as an input in one situation and as an output in another. In fact, most manufacturers offer such programmable interfaces. The system programmer can program these interfaces to act in a way that suits the application, using a sequence of machine language instructions. We will present a parallel programmable interface from Motorola where each individual bit in its ports can be programmed either as an input or output. There are other more complicated situations where

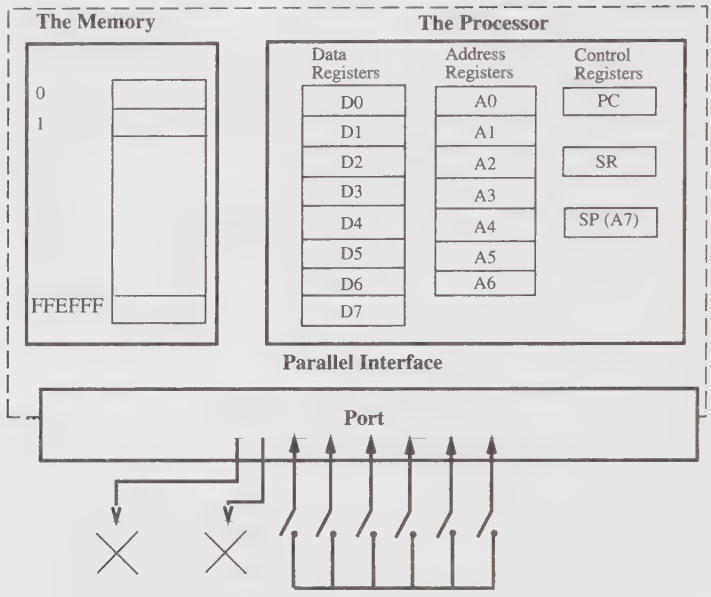



**Figure 7.1** A computer that is interfaced to two lamps and six switches.

we want to change the way information is transferred between the computer system and the environment. In the next example, we consider a commonly used communications protocol to synchronize the transfer between two devices.

The next example in this section considers a computer system that is supposed to transfer characters to a printer. As shown in Figure 7.2, both units have a parallel interface that makes it possible to interconnect them.

The data that are to be transferred to the printer, in essence ASCII-coded characters, make use of seven lines. In the previous chapter, we noted that the program that transfers data simply writes each character to the output port. At almost the same time, the character is available in the input port of the printer. However, we run into the problem of how the printer knows when a new character is available and how the computer system knows when the printer has taken care of the previously transferred character. We are now about to present a commonly used technique, called *handshaking* that solves this problem.

In Figure 7.2, two lines are associated with the parallel connection, denoted Ready and Send, to help synchronize the transfer in the following way. The sending side (the M68000-based computer system) notifies the receiving side (the printer) that data is available by asserting the Send line to a one in the example as shown in Figure 7.3. When the receiving side notices that Send is set, it reads the data and acknowledges the reception of the data by asserting the Ready line to a logical one. The sending side now knows that data is received and can send a new data item by again asserting Send. Send and Ready are called *handshake lines*, because

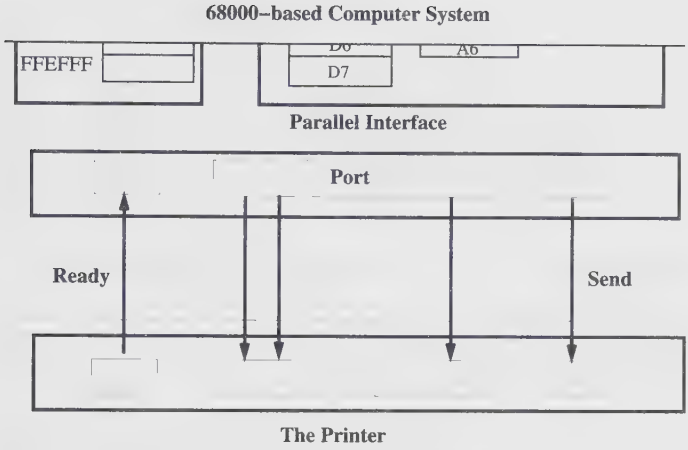



**Figure 7.2** Parallel I/O with handshaking between a computer system and a printer.

of the function they have; the communicating devices 'shake their hands' so as to agree that data has been transferred.

Note that the above situation is just one example of a handshake protocol. While some devices may signal that they have accepted data by resetting Ready, other devices may signal that data is available by resetting the Send signal. One realizes that a programmable interface should be able to support various handshake protocols. This is exactly what the programmable interface we will present next is able to do.

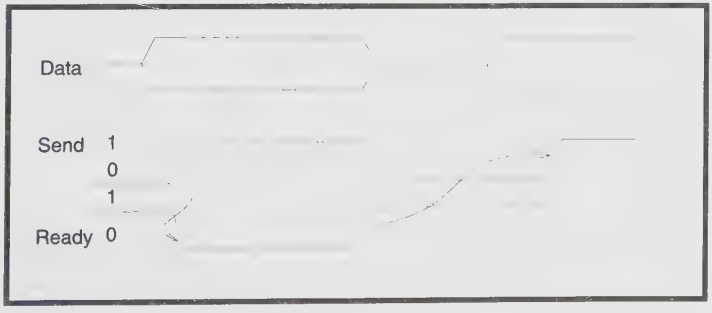



**Figure 7.3** Timing diagram for the handshake signals Send and Ready.

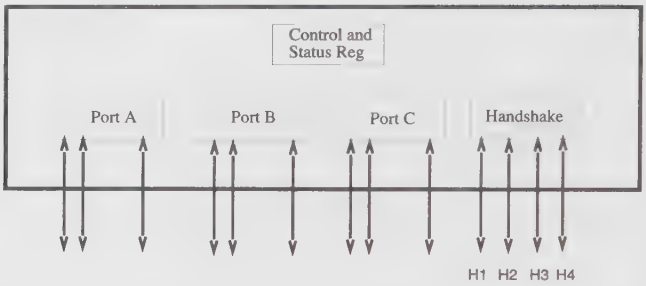



**Figure 7.4** Schematic diagram of the M68230 parallel interface.

**Table 7.1** Operation modes of the Motorola PI/T.

| | **Mode 0** (8-bit, unidir.) | | **Mode 1** (16-bit, unidir.) | | **Mode 2** (8-bit, bidir.) | **Mode 3** (16-bit, bidir.) |
|---|---|---|---|---|---|---|
| **Sub-mode** | Input | (00) | Input | (X0) | | |
| | Output | (01) | Output | (X1) | | |
| | Bit I/O | (1X) | | | | |

### 7.1.2 The PI/T – an example parallel interface

We are now ready to present an example of a programmable parallel interface from Motorola, called M68230 PI/T. M68230 can be thought of as containing two separate units: the parallel interface (PI) and the timer (T). We will only present the parallel interface. The interested reader should consult the complete data sheet from Motorola for more details. We will not give an exhaustive treatment of all features of the PI/T. In fact, it is almost 'infinitely programmable' and it would cover almost as many pages as this book contains to clarify fully all the possibilities that this programmable device provides. We will rather discuss those features that are relevant for the discussion in this section.

The parallel interface of the M68230 contains three 8-bit ports, called Port A, B, and C, according to Figure 7.4. These ports can be programmed in a variety of ways. In addition, there are four handshake lines, denoted H1 through H4, that can be programmed according to various handshake protocols.

In order to program the device, there are a large number of control registers. The interface is programmed by writing a sequence of codes to its control registers. There is also a status register that can be used if we desire to use polling to synchronize transitions on H1 through H4.

In Table 7.1, we list the various operation modes that can be selected by programming the PI/T. There are four main operation modes constructed by the key terms 8-bit/16-bit and unidirectional/bidirectional. A transfer is *unidirectional* if

the direction of the data transfer between ports in two communicating devices is the same all the time. Sometimes it is convenient to let two communicating devices X and Y use the same lines to transfer data from X to Y as the opposite transfer from Y to X. Doing this, we do not have to dedicate a pair of ports to data transfers in one direction. Connections between two communicating devices that can change data direction are referred to as *bidirectional*. The main operation mode also gives provision for combining the A and B port to form a 16-bit port. For the 8-bit modes, each of the ports are controlled individually, whereas for the 16-bit modes, A and B are considered as a 16-bit port and cannot be programmed individually.

From Table 7.1, we see that a number of alternatives are available for some of the main operation modes. These alternatives are referred to as submodes. For example, three submodes are associated with Mode 0, whereas no submodes are associated with modes 2 and 3. In the following, we will restrict the discussion to the submodes of Mode 0.

The input/output submodes are used when a handshake protocol is desired. According to Figure 7.4, four handshake lines are available. H1 and H2 are associated with port A, while H3 and H4 are associated with port B. The third submode is used in bit-I/O operations; each bit in ports A, B, and C can be individually programmed as either an input or output.

Besides the actual ports in the PI/T, there are nine 8-bit control and status registers that control its actions. In Table 7.2, we summarize almost all registers that support the parallel interface of the PI/T. From left to right, we show the register address, the individual control and data bits of all registers, and finally to the right, the name of each register. The manufacturer of a computer system has specified a particular address, called *base address*, for a parallel interface such as the PI/T. The address of a certain register within the PI/T can be obtained by adding the Register Select Offset in Table 7.2 to the base address. For example, if the base address is $FFF000_{16}$, the address of the PACR is $FFF006_{16}$. Note that in some systems the register select offset is used differently.

We will now show how various communications protocols can be supported by using Port A as an example. In Table 7.3, we show a subset of all registers in the PI/T that are relevant for the operation of Port A.

In the first example, we want to use Port A to be able to read the logical levels of the switch settings and, in addition, to control the lamps in Figure 7.1. To do this, we need to program Port A to be used in bit-I/O operation; bits 6 and 7 should be outputs while bits 0-5 should be inputs. This is done by selecting the bit-I/O submode of Mode 0 (see Table 7.1).

The main operation mode (Mode 0) is selected by the two most significant bits of the Port General Control Register (PGCR) according to Table 7.3 by simply writing the mode number to these bits. The submode of Port A is selected by the two most significant bits of the Port A Control Register (PACR) by simply using the codes that are found in Table 7.1 (1X denotes Bit I/O, where X means that the setting of bit 6 is irrelevant). The remaining bits of the PGCR and of the PACR

**Table 7.2** Ports, control, and status registers for the parallel interface in the PI/T.

| Register Select Offset (hex.) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | Port Mode Control | | H34 Enable | H12 Enable | H4 Sense | H3 Sense | H2 Sense | H1 Sense | Port General Control Register (PGCR) |
| 1 | * | SVCRQ Select | | IPF Select | | Port Interrupt Priority Control | | | Port Service Request Register (PSRR) |
| 2 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port A Data Direction Register (PADDR) |
| 3 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port B Data Direction Register (PBDDR) |
| 4 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port C Data Direction Register (PCDDR) |
| 5 | Interrupt Vector Number | | | | | | * | * | Port Interrupt Vector Register (PIVR) |
| 6 | Port A Submode | | H2 Control | | | H2 Int Enable | H1 SVCRQ Enable | H1 Stat Ctrl | Port A Control Register (PACR) |
| 7 | Port B Submode | | H4 Control | | | H4 Int Enable | H3 SVCRQ Enable | H3 Stat Ctrl | Port B Control Register (PBCR) |
| 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port A Data Register (PADR) |
| 9 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port B Data Register (PBDR) |
| C | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port C Data Register (PCDR) |
| D | H4 Level | H3 Level | H2 Level | H1 Level | H4S | H3S | H2S | H1S | Port Status Register (PSR) |

**Table 7.3** A subset of the PI/T registers that are relevant for Port A.

| Register Select Offset (hex.) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | Port Mode Control | | H34 Enable | H12 Enable | H4 Sense | H3 Sense | H2 Sense | H1 Sense | Port General Control Register (PGCR) |
| 2 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port A Data Direction Register (PADDR) |
| 6 | Port A Submode | | H2 Control | | | H2 Int Enable | H1 SVCRQ Enable | H1 Stat Ctrl | Port A Control Register (PACR) |
| 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Port A Data Register (PADR) |

affect the operation of the handshake lines and will be discussed later. We therefore ignore their settings for a while. The following sequence of instructions programs the PI/T according to the specification:

```
PGCR    EQU     $FFF000             ; Address to the PGCR
PACR    EQU     $FFF006             ; Address to the PACR
PADDR   EQU     $FFF002             ; Address to the PADDR

INIT    MOVE.B  #%00000000,PGCR     ; Mode 0
        MOVE.B  #%10000000,PACR     ; Bit-I/O submode
        MOVE.B  #%11000000,PADDR    ; Bits 6-7 outputs and
                                    ; bits 0-5 inputs
```

To program the PI/T, we have assumed that its base address is $FFF000_{16}$. The addresses of the individual registers are obtained by adding the offset of each register from Table 7.3 to the base address of the PI/T. Note that we have arbitrarily assigned zero to bit 0-5 in the PGCR. In addition, bits 6 and 7 have been assigned $10_2$ to set up the bit-I/O submode and the rest of the bits in the PACR are arbitrarily assigned zero. The third move-instruction sets up the data direction of the eight single-bit ports in Port A. A zero indicates that the corresponding bit is an input while a one indicates an output. Since we want the two most significant bits in Port A to become outputs (the lamps are connected to these bits, see Figure 7.1), these bits are set in the PADDR.

After the initialization, it is possible to read the switch settings and control the lamps. For example, in the example below

```
PADR  EQU      $FFF008            ; Address to the PADR

      MOVE.B   PADR,D0            ; Read data from Port A
      MOVE.B   #%11000000,PADR    ; Switch on both lamps
```

the switch settings are read and the lamps are controlled using the same port-address. This might seem strange. However, the designer of the PI/T has ensured that a value written to bit 0–5 is ignored because these bits are inputs.

Now suppose that we want to program Port A so that bits 6 and 7 are inputs and bits 5-0 are outputs. The following sequence of instructions will do

```
INIT  MOVE.B   #%00000000,PGCR    ; Mode 0
      MOVE.B   #%10000000,PACR    ; Bit-I/O submode
      MOVE.B   #%00111111,PADDR   ; Bits 6-7 inputs and
                                  ; bits 0-5 outputs
```

The only thing that differs from the previous example is how the PADDR is set up. Note that Port B can be programmed similarly by using the Port B Control Register (PBCR) to determine the submode; Port B Data Direction Register to determine the data direction of all individual bits in Port B (See Table 7.2), and finally data is accessed from the Port B Data Register (PBDR).

We will now look at more complex initializations of the PI/T. Suppose that we want to use Port A to output data to a printer according to the handshake protocol of Figures 7.2 and 7.3. In Figure 7.5, we show how the handshake lines H1 and H2 can be used to facilitate the signalling of Ready (the printer has taken care of the character) and Send (the processor has written a new character to Port A) in a handshake protocol. The handshake lines can be programmed in a variety of ways as inputs and outputs. In the following, we will show how the protocol according to Figure 7.2 is set up. After that point, we will show how other handshake protocols can be supported.

**Table 7.4** Layout of the Port Status Register (PSR) in the PI/T.

| Register Select Offset (hex.) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| D | H4 Level | H3 Level | H2 Level | H1 Level | H4S | H3S | H2S | H1S | Port Status Register (PSR) |

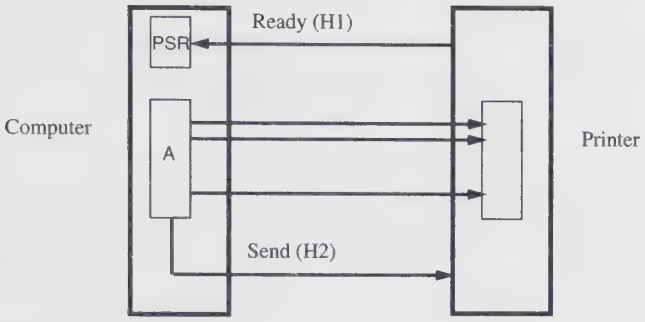



**Figure 7.5** Output handshake protocol using Port A and the H1 and H2 handshake lines.

The Ready line (H1) can be read by reading the content of the Port Status Register (PSR) (see Table 7.4). As can be seen from Table 7.4, there are two bits associated with H1 (H1 Level and H1S(ense)). Simply speaking, the level-bit reflects the direct value of H1 while the sense-bit reflects transitions on H1. For instance, if the PI/T is set up to sense when the H1-line is set, the H1S-bit will be set when there is a transition on the H1-line from zero to one. However, if the H1-line is reset shortly thereafter, H1S will remain set while H1-Level will reflect the direct change. The idea behind this is to let the PI/T do most of the work in a handshake protocol - when the printer signals Ready, the H1S-bit is set. The processor can sense this by a polling scheme that repeatedly tests the H1S-bit in the PSR. When the processor writes a new value to the PI/T, H1S is automatically reset, and the H2-line (Send) is automatically asserted. We now show how the PI/T can be set up to conform to this protocol. We show the entire sequence of control words needed to set up the PI/T below:

```
PGCR    EQU     $FFF000             ; Address of the PGCR
PACR    EQU     $FFF006             ; Address of the PACR
PADDR   EQU     $FFF002             ; Address of the PADDR

INIT    MOVE.B  #%00010011,PGCR     ; Set mode 0.  H1 and H2 are
                                    ; asserted when set
        MOVE.B  #%01110000,PACR     ; Set the output submode
                                    ; and H2 handshake
        MOVE.B  #%11111111,PADDR    ; All bits are outputs
```

To explain the codes that are used to set up the PI/T, we take a closer look at the control registers starting with the PGCR. In Table 7.5, we show the detailed layout of the PGCR and the PACR control registers. The four least significant bits of the PGCR specify at what logical level a certain handshake line is asserted. If we want

**Table 7.5** Layout of the Port General Control Register and the Port A Control Register.

| Register Select Offset (hex.) | 7 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | Port Mode Control | H34 Enable | H12 Enable | H4 Sense | H3 Sense | H2 Sense | H1 Sense | Port General Control Register (PGCR) |
| 6 | Port A Submode | H2 Control | | | H2 Int Enable | H1 SVCRQ Enable | H1 Stat Ctrl | Port A Control Register (PACR) |

H1 to assert to a logical one, as in our example, we must set the corresponding bit. If the bit is cleared assertion will be a logical zero. Since we want H1 and H2 to be asserted when set, the corresponding bits are set in the PGCR (see initialization sequence above). In order for the handshake-line pairs H1 and H2 to be enabled according to the handshake protocol, we also need to set bit 4.

Continuing on the set up of the PACR, we note that bits 7 and 6 determine the submode. Since we want a handshake protocol for output, we have set these bits to $01_2$ (see Table 7.1). Bits 3 5 control the handshake protocol that H2 is supposed to follow. Below, we provide a list of all the options that are available for the output submode:

$$\text{bit } 5\ 4\ 3 = \begin{cases} 0\ \ X\ \ X & \text{Input} - \text{status only} \\ 1\ \ 0\ \ 0 & \text{Output} - \text{always negated} \\ 1\ \ 0\ \ 1 & \text{Output} - \text{always asserted} \\ 1\ \ 1\ \ 0 & \text{Output} - \text{interlocked handshake} \\ 1\ \ 1\ \ 1 & \text{Output} - \text{pulsed handshake} \end{cases}$$

In the above list, X denotes that the setting of the corresponding bit is irrelevant. To choose H2 to be an arbitrary input that can be read using the Port Status Register, we simply code bits 3-5 as e.g. 000. 100 and 101 make H2 to act as an output that is constantly either negated (not asserted) or asserted to a logical level according to the H2 sense bit in the PGCR. 110 denotes that H2 is to follow an interlocked handshake protocol. Interlocked means that it is asserted when the processor writes to the data port (Port A in this example) and negated when the output device (the printer in this example) has taken care of the data and signaled asserted H1 (Ready). This is exactly what we want to achieve, which is why bits 3–5 in the PACR are assigned 110. Unlike the interlocked handshake protocol, H2 will remain only asserted for a short while in the pulsed handshake protocol (111). Bits 1 and 2 in PACR determine whether an interrupt is to be generated when H1

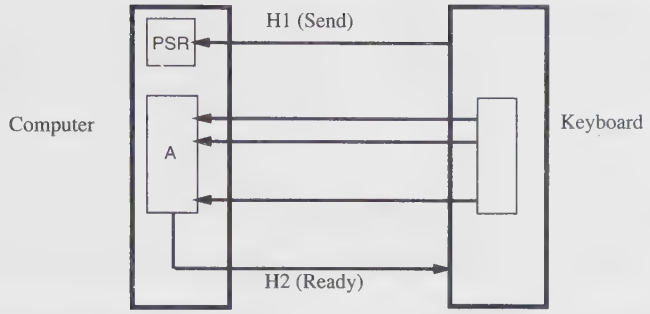



**Figure 7.6** Input handshake protocol using Port A and the H1 and H2 handshake lines.

and H2 are asserted. We will discuss the support for interrupt-driven I/O that the PI/T provides later in Section 7.3.

Note that the PI/T takes care of all the handshake signalling. In the following example program, we use the handshake scheme to transfer a buffer of characters to a printer under the handshake protocol we have described.

```
PADR   EQU      $FFF008      ; Port A Data Register
PSR    EQU      $FFF00D      ; Port Status Register
NUL    EQU      0            ; ASCII-code for NUL

INIT   MOVEA.L  #BUFFER,A0
LOOP   CMP.B    #NUL,(A0)
       BEQ      NEXT

POLL   BTST     #0,PSR       ; Test H1S in the PSR
       BEQ      POLL         ; If not asserted, try again
       MOVE.B   (A0)+,PADR   ; Write to Port A
       BRA      POLL
NEXT   RTS
```

We make the following important observations. First, after initialization, the processor will busy-wait on the least significant bit of the PSR (H1S) (see Table 7.4) until the printer asserts Ready (H1). Second, at this point, the processor writes the next character to the Port A data register (PADR). When the PI/T senses this, it automatically asserts H2 to notify the printer that a new character is available.

We have now seen how the PI/T can be set up to meet various protocols. Before we close this section, we will provide another example in which Port A is used as an input port and the data transfer is synchronized by a similar handshake protocol according to Figures 7.6 and 7.7.

The differences between this handshake protocol and the previous one are the


**Figure 7.7** Timing diagram for the input handshake protocol.

following. The input device (the keyboard) signals when data is available by asserting the Send handshake line (H1). The 68000-based computer system signals that data has been read by asserting the Ready line (H2) according to Figure 7.6. The logical levels for assertion also differ. According to Figure 7.7, both Send and Ready are asserted to logical level zero - when the key is pressed, the Send signal goes from one to zero. Similarly, when data is read from Port A, Ready goes from one to zero. We shall now look at the initialization sequence to program the PI/T to conform to this protocol.

```
PGCR    EQU     $FFF000             ; Address of the PGCR
PACR    EQU     $FFF006             ; Address of the PACR
PADDR   EQU     $FFF002             ; Address of the PADDR

INIT    MOVE.B  #%00010000,PGCR     ; Set mode 0.  H1 and H2 are
                                    ; asserted when reset
        MOVE.B  #%00110000,PACR     ; Set the input submode
                                    ; and H2 handshake
        MOVE.B  #%00000000,PADDR    ; All bits are inputs
```

First, the assertion level of H1 and H2 is changed; bits 0 and 1 of the PGCR are now reset. Second, the submode is now 00 for input. Third, all Port A bits are directed to serve as input ports in the PADDR.

## Exercises

**7.1** What sequence of instructions is needed to program the PI/T so that bits 0–3 of Port A are inputs and bits 4–7 are outputs?

**7.2** What sequence of instructions is needed to program the PI/T so that bits 0–3 of Port B are inputs and bits 4–7 are outputs?

**7.3** What sequence of instructions is needed to program the PI/T so that H1 is asserted when zero in the output handshake protocol according to Figure 7.5?

**7.4** What sequence of instructions is needed to program the PI/T so that Port B conforms to the output handshake protocol according to Figure 7.5? Note that H3 and H4 are replaces by H1 and H2 in the handshake protocol.

**7.5** Write a device driver to the keyboard protocol in Figure 7.6 that returns the character in D0 when data is available in Port A.

---

## 7.2 Serial input and output

In the previous section, we have assumed that a computer system communicates with a device by sending several bits in parallel at a time. To connect a printer to a computer system would require at least seven bits in order to transfer an ASCII-coded character in parallel. One could imagine that the cabling cost of connecting a computer system in one room in a large building with a printer in another room far away from the computer system would be prohibitive. In this section, we shall look at an interesting alternative, namely, the use of a single line to transfer the bits in a word in a serial fashion.

### 7.2.1 Asynchronous bit-serial communication

The fact that we want to use a single line to transfer information serially poses the same synchronization problem we have seen in this book several times by now – how is the receiving side supposed to know when a new bit is available? We could of course use the handshake protocol that we have described in the previous section, but that would be terribly inefficient; the program would have to check the status register for each transferred bit. Also, it would take away most of the advantages of having a single line to bring down the costs of cabling. Instead, a commonly used protocol, called *asynchronous serial communication*, has been developed to solve this problem.

To be able to use a single line necessitates that the information carried along this line must bring synchronization support. The basic assumption for the asynchronous protocol to work correctly is that the sending and the receiving side conforms to the same transfer rate, usually denoted *baud rate* and measured in baud (1 baud = 1 bit/second). In Figure 7.8, we show how two characters 'A' ($1000001_2$) and 'B' ($1000010_2$) are transferred serially without any support for


**Figure 7.8** Two successive serial character transfers without support for synchronization.

synchronization. The duration of each transferred bit is the same (i.e. the time between two consecutive ticks in Figure 7.8 is the same for all bits). When the sender has nothing to send, the line is constantly set. Even if both the sending side and the receiving side agree on the same baud rate, the receiving side will have the problem of knowing when the transfer of a character begins. In the asynchronous protocol we will present this problem is solved by assuming that a zero is sent to indicate the beginning of a new character, according to Figure 7.9. This bit is called a *start bit*. A start bit is detected by the receiver as a transition from the idle one to zero. When the receiver detects this transition, it can determine where the actual data transmission starts.

A problem related to long-distance transfers is that the transfer is not always undistorted. In fact, the nice square-shaped pulses as shown in Figure 7.8 can be so heavily distorted so that a one is interpreted as a zero and vice versa. A simple scheme to detect single-bit transmission errors is to append a so called *parity bit*. The parity bit can either be used to detect *odd parity* or *even parity*. Under an even parity scheme, the parity bit is set so that the total number of bits in the data item including the parity bit is even. For instance, if character 'A' is transmitted ($1000001_2$), the parity bit is zero (see Figure 7.9), while if character 'B' is transmitted, the parity bit is one. For odd parity calculation, the total number of ones including the parity bit is odd. Now suppose that a zero is distorted so that the receiver will interpret it as a one. The receiver can then detect that a transmission error has occurred by counting the number of ones and checking the parity bit. However, note that two errors within the same character cannot be detected.

The last issue to be introduced is that of allowing the receiver to start synchronizing for a new character to arrive. One often requires that the smallest distance between two consecutive character transfers is one or two bits. These bits are denoted *stop bits* because they indicate that the last bit has been sent. The stop bits are indistinguishable from the idle state, that is, they are detected as ones. In Figure 7.9, we show the transmission of character 'A' with the support for synchronization by means of a start bit preceding each character, and support for error detection by means of an even parity bit, and a stop bit to indicate that the last bit has been received.

The asynchronous serial protocol shown in Figure 7.9 is widely used to connect e.g. VDTs (Video Display Terminals) and printers to computer systems. Therefore, there are special interfaces that take care of the conversion of a character to a

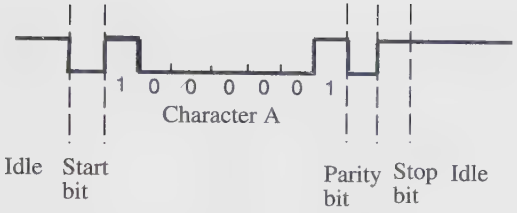



**Figure 7.9** Transmission of the character 'A' with an asynchronous protocol with even parity and one stop bit.

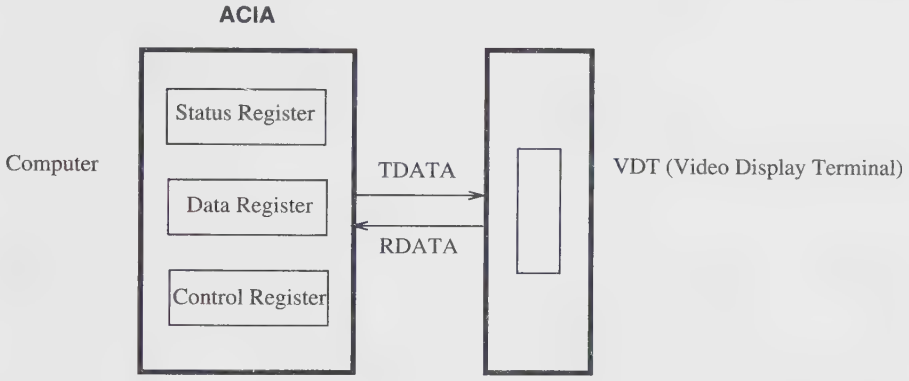



**Figure 7.10** The organization of the ACIA and how it is connected to a Video Display Terminal with bit-serial lines.

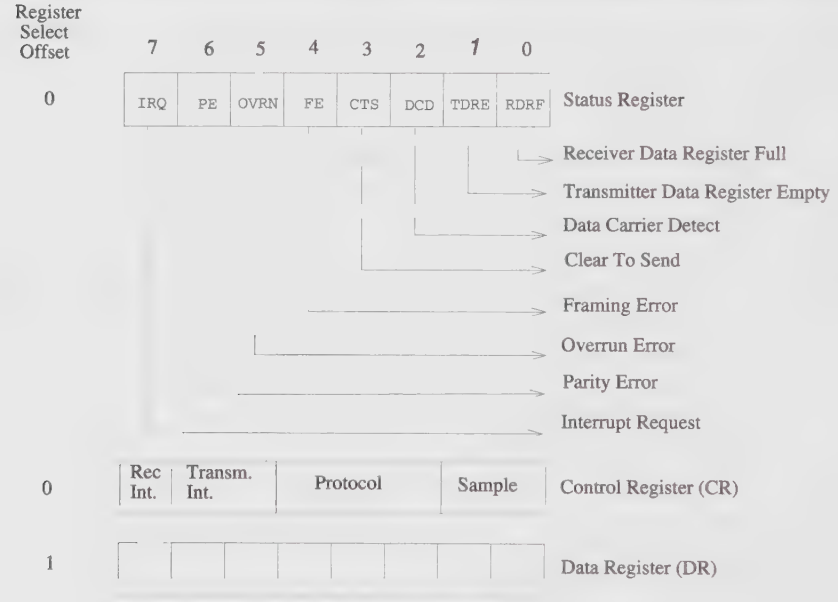
sequence of bits including the start, parity, and stop bits and which detects when a transmission error occurs. Such interfaces are called UARTs (Universal Asynchronous Receiver Transmitter). Although the basic protocol is the same, some of the operational parameters may vary. First, the transfer rate can be varied from 110 baud (about 10 characters per second) to 19,200 baud (about 2000 characters per second). Second, the parity check can be odd or even, and, finally; one or two stop bits can be used. Next we present a programmable UART from Motorola which has the marketing name ACIA (Asynchronous Communications Interface Adapter), or MC6850.

### 7.2.2 The ACIA – an example serial interface

In Figure 7.10, we show the basic organization of the ACIA and how it is connected to a VDT. Before we look at how the protocol parameters can be set up, we note that the function of the ACIA is to convert a word (eight or seven bits depending on how it is programmed) into a serial bit-stream and to add a start, parity, and

one or two stop bits according to Figure 7.9. The word to be transmitted (along the TDATA-line in Figure 7.10) is simply written to the data register in Figure 7.10. The ACIA can also receive data (along the RDATA-line in Figure 7.10) according to the asynchronous bit-serial protocol and detect whether a single-bit transmission error has occurred by checking the parity bit.

**Table 7.6** Port, control, and status register for the serial interface ACIA.




The protocol parameters of the ACIA are set up by writing an 8-bit control word to the control register (CR). In Table 7.6, we show the layout of the registers contained in the ACIA. Note that the control and status registers have the same addresses. The designer has decided this by noting that the control register is write-only while the status register is read-only.

Let us start to see how we set up the protocol parameters for the ACIA. This is done by a 3-bit code in bits 2–4 of the control register as follows:

$$\text{Protocol} = \begin{cases} 0\ 0\ 0 & \text{7 data bits, even parity, 2 stop bits} \\ 0\ 0\ 1 & \text{7 data bits, odd parity, 2 stop bits} \\ 0\ 1\ 0 & \text{7 data bits, even parity, 1 stop bit} \\ 0\ 1\ 1 & \text{7 data bits, odd parity, 1 stop bit} \\ 1\ 0\ 0 & \text{8 data bits, no parity, 2 stop bits} \\ 1\ 0\ 1 & \text{8 data bits, no parity, 1 stop bits} \\ 1\ 1\ 0 & \text{8 data bits, even parity, 1 stop bit} \\ 1\ 1\ 1 & \text{8 data bits, odd parity, 1 stop bit} \end{cases}$$

We can choose between 7 or 8 bits of data, odd or even parity, and finally, we can have one or two stop bits. The baud rate is usually defined by a timer that is connected to the ACIA. How the baud rate is changed may differ between computer systems.

In order for the ACIA to synchronize on the start bit to determine when to read the first bit of the data word, it must sample the serial line. This is done with a frequency that is typically a magnitude higher than the baud rate. The higher this frequency is, the better the ACIA will be able to cope with deviations in the baud rate between the sending and receiving devices. The ACIA permits the user to specify the number of samples per bit. It is possible to choose between 1, 16, 64 samples per bit through bits 0 and 1 in the control register according to

$$\text{Sample} = \begin{cases} 0\ 0 & \text{1 sample/bit} \\ 0\ 1 & \text{16 samples/bit} \\ 1\ 0 & \text{64 samples/bit} \\ 1\ 1 & \text{master reset} \end{cases}$$

Usually, 16 bits per sample is used. In order to program the ACIA, we need to perform a master reset. This is done by writing $11_2$ to bits 0 and 1 in the control register. Bits 5–7 specify whether the ACIA should generate an interrupt when the output buffer of the data register is empty, that is, when the ACIA is ready to take care of a new character to be transmitted, and whether an interrupt is to be generated when a character is ready in the input buffer. These bits will not concern us.

We are now ready to provide some examples of how to program the ACIA. Suppose that we want to use 7 bits, odd parity, 2 stop bits, and a sample rate of 64 samples per bit. The proper initialization of the ACIA is as follows, assuming that the base address is $\text{FFF000}_{16}$

```
CR    EQU     $FFF000          ; Address to control register

INIT  MOVE.B  #%00000011,CR    ; Master reset of the ACIA
      MOVE.B  #%00000110,CR    ; 7 bits, odd parity,
                               ; 2 stop bits, and 64 samples
```

In the second example, we assume 8 bits, even parity, and 1 stop bit. We get

```
CR    EQU     $FFF000          ; Address to control register

INIT  MOVE.B  #%00000011,CR    ; Master reset of the ACIA
      MOVE.B  #%00011010,CR    ; 8 bits, even parity,
                               ; 1 stop bit, and 64 samples
```

Note that once we have initialized the ACIA, it is ready for use. We now turn our attention to how the processor is supposed to know when the ACIA has received a new character, and when it is ready to send a new character. Also, we will see how certain transmission errors, such as single-bit errors, are detected. This information is provided by the status register (see Figure 7.6).

Starting from the most significant bit of the status register, bit 7 (IRQ) is set when an interrupt is generated (provided that the interrupt control is enabled). Bit 6 (PE) is set when a parity error has occurred, that is, the receiver has detected a single-bit transmission error. Bit 5 (OVRN) detects an *overrun error*. An overrun error results if a character that has not been read by the processor is overwritten by a new character that is received by the ACIA. Bit 4 (FE) indicates that the wrong number of stop bits have been detected or that the start bit is not correctly received. For instance, if the ACIA has been programmed for two stop bits and it detects a start bit immediately following the first stop bit, a *framing error* occurs. Bits 0 and 1 indicate when a new character is available in the ACIA (bit 0) and when the ACIA is ready to send a new character (bit 1).

Suppose that we want to transfer a buffer of characters to a VDT. The following polling scheme will do

```
SR     EQU      $FFF000      ; Status Register
DATA   EQU      $FFF001      ; Data Register
NUL    EQU      0            ; ASCII-code for NUL

INIT   MOVEA.L  #BUFFER,A0
LOOP   CMP.B    #NUL,(A0)
       BEQ      NEXT

POLL   BTST     #1,SR        ; Test TDRE in the SR
       BEQ      POLL         ; If not asserted, try again
       MOVE.B   (A0)+,DATA   ; Write to the ACIA
       BRA      POLL
NEXT   RTS
```

Note that we test bit 1 in the status register. This bit is one when the ACIA is ready to transmit the next character. At this point, we can write the new character to the data register in the ACIA.

Conversely, if we want to read a character string from the ACIA, which is terminated by NUL, and if we want to detect if any error has occurred, we can do as follows

```
SR      EQU     $FFF000     ; Status Register
DATA    EQU     $FFF001     ; Data Register
NUL     EQU     0           ; ASCII-code for NUL

INIT    MOVEA.L #BUFFER,A0

POLL    BTST    #0,SR       ; Test RDRF in the SR
        BEQ     POLL        ; If not asserted, try again
        BTST    #6,SR       ; Parity error?
        BNE     PERROR      ; Yes, branch to PERROR
        BTST    #5,SR       ; Overrun error?
        BNE     OERROR      ; Yes, branch to OERROR
        BTST    #4,SR       ; Framing error?
        BNE     FERROR      ; Yes, branch to FERROR
        MOVE    DATA,(A0)+  ; Read from the ACIA
LOOP    CMP.B   #NUL,(A0)
        BEQ     NEXT
        BRA     POLL
PERROR  ...                 ; Parity error ...
OERROR  ...                 ; Overrun error...
FERROR  ...                 ; Framing error...
NEXT    RTS
```

Note how the error flags in the status registers are tested to find out about any transmission error that might have occurred. A possible action, upon detection of a parity error, could be to request the sender to retransmit the character.

## Exercises

**7.6** What sequence of instructions is needed to set up the ACIA with 7 bits, even parity, 1 stop bit, and 16 samples per bit?

**7.7** What sequence of instructions is needed to set up the ACIA with 7 bits, even parity, 2 stop bits, and 64 samples per bit?

**7.8** What sequence of instructions is needed to set up the ACIA with 8 bits, no parity, 2 stop bits, and 16 samples per bit?

**7.9** Write a subroutine that polls the RDRF bit in the status register of the ACIA. When a character has been received, it is returned in D0. In addition, an error code should be returned in D1 as follows: 0=no error, 1=parity error, 2=overrun error, and 3=framing error.

---

## 7.3 Vectored interrupts

The interrupt mechanism provides a means to let the processor perform useful work instead of actively checking whether an external event has happened which is the case for polling schemes. However, it is important to note that polling is useful if interrupts from a single input device occur frequently and can sometimes be more efficient than using interrupts. The reason is as follows. When an interrupt occurs, the processor needs to store the program counter and the status register on top of the stack. In addition, it has to invoke the interrupt service routine. All these actions take a substantial number of cycles to perform. Consequently, the response time from the point when the interrupt occurred until the service is performed is in general longer in interrupt-driven exception handling than in a polling scheme. This is true if the processor only needs to service a few external events. Now assume that we want to design a polling scheme for a large number of external events. In the below example, we show such a scheme for five events:

```
LOOP    BTST  #1,EVENT   ; Is event-flag 1 active?
        BNE   EVENT1     ; Yes, handle it
        BTST  #2,EVENT   ; Is event-flag 2 active?
        BNE   EVENT2     ; Yes, handle it
        ...
        BTST  #5,EVENT   ; Is event-flag 5 active?
        BNE   EVENT5     ; Yes, handle it
        BRA   LOOP
EVENT1  ...              ; Handle event 1
        BRA   LOOP
EVENT2  ...              ; Handle event 2
        BRA   LOOP
        ...
EVENT5  ...              ; Handle event 5
        BRA   LOOP
```

In the above scheme, five event-flags are available in an inport at address `EVENT`. Each event-flag is tested in turn; when event-flag 5 has been tested, event-flag 1 is tested again etc. Now suppose that event 5 occurs immediately after it has been polled. It will now take five tests for the processor to again test whether event 5 has occurred. One realizes that if the number of events is large, the response time becomes extremely long. In such situations it is more efficient to use interrupts.

In Chapter 6, we noted that there are only seven interrupt inputs. What do we do if we want to support more than seven interrupts. In fact, the M68000 as well as most computers supports a large number of interrupts called *vectored interrupts*. In this section, we will take a look at how one can extend the number of interrupt inputs beyond seven by means of user-defined interrupt vectors.

M68000 supports seven interrupt priority levels which are denoted $I_1$ to $I_7$. We say that $I_n$ has *interrupt priority level* $n$. In Chapter 6, we noted that an interrupt with an interrupt priority level $n$ can be taken care of by the processor provided that $n >$ CPL (the current priority level). In 68000-based computer systems that only need to take care of at the most seven interrupts, one can associate each interrupt with a distinct interrupt priority level. The scheme we presented in Chapter 6 assumed that an entry in the exception vector table is associated with each interrupt priority level. This scheme is called *autovector* mode, because M68000 calculates the address of the entry in the exception vector table based on the interrupt priority level. We noted that the address in the exception vector table is given by $4(18_{16} + n)$, assuming that the interrupt priority level of the interrupt is $n$. The autovector for interrupt $n$ is $v = 18_{16} + n$.

M68000 can support more than seven interrupts by letting the interface (or device) that generates the interrupt provide the processor with a vector number. In this section, we will show how the parallel interface PI/T can be programmed to supply M68000 with a vector number when an interrupt occurs. There are 192 vector numbers ranging from $v = 40_{16}$ to $v = \mathrm{FF}_{16}$. In general, if an interface generates an interrupt with a vector $v$, M68000 will fetch the address of the corresponding interrupt service routine at address $4v$. For example, if an interface provides the vector $v = 40_{16}$, the address of the interrupt service routine is available at address $4v = 100_{16}$. In Table 7.7, we show the exception vector table with the addresses and vector numbers for all vectored and autovectored interrupts.

It is possible to program the PI/T, and many other programmable interfaces, to provide a certain vector when an interrupt from the interface occurs. However, it is necessary that the designer of the computer system has connected the PI/T to the processor in such a way that it can generate a vector number. To explain this is outside the scope of this text. The only thing we shall bother about is how the vector number is programmed and how we initialize the exception vector table to connect the interrupt to a corresponding interrupt service routine. In Table 7.8, we show the layout of the Port Interrupt Vector Register (PIVR) and the PACR of the PI/T. Recalling Section 7.1, we noted that four handshake lines are available which are denoted H1 through H4. Each of these handshake lines can cause an interrupt. The programmer can associate a vector with each of these handshake

**Table 7.7** Vector numbers and addresses in the exception vector table for autovectors and user-defined interrupt vectors.

| Interrupt | Vector number | Address |
|---|---|---|
| Level 1 Interrupt Autovector | $19_{16}$ | $64_{16}$ |
| Level 2 Interrupt Autovector | $1A_{16}$ | $68_{16}$ |
| Level 3 Interrupt Autovector | $1B_{16}$ | $6C_{16}$ |
| Level 4 Interrupt Autovector | $1C_{16}$ | $70_{16}$ |
| Level 5 Interrupt Autovector | $1D_{16}$ | $74_{16}$ |
| Level 6 Interrupt Autovector | $1E_{16}$ | $78_{16}$ |
| Level 7 Interrupt Autovector | $1F_{16}$ | $7C_{16}$ |
| ..... | | |
| ..... | | |
| ..... | | |
| User Interrupt Vector 1 | $40_{16}$ | $100_{16}$ |
| User Interrupt Vector 2 | $41_{16}$ | $104_{16}$ |
| User Interrupt Vector 3 | $42_{16}$ | $108_{16}$ |
| ..... | | |
| User Interrupt Vector 192 | $FF_{16}$ | $3FC_{16}$ |

lines by a 6-bit vector number. The additional two least significant bits in the PIVR are assigned by the PI/T itself according to the table below:

| Source | Low order bits of the PIVR |
|---|---|
| H1 | 00 |
| H2 | 01 |
| H3 | 10 |
| H4 | 11 |

So given that the six bits are 010000, the vector numbers for H1 through H4

**Table 7.8** Layout of the Port Interrupt Vector Register and the PACR of the PI/T.

| Register Select Offset (hex.) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | Interrupt Vector Number | | | | | | * | * | Port Interrupt Vector Register (PIVR) |
| 6 | Port A Submode | | H2 Control | | | H2 Int Enable | H1 SVCRQ Enable | H1 Stat Ctrl | Port A Control Register (PACR) |

are 01000000,...,01000011. A consequence of the predetermined low order bits in the PIVR is that H1 through H4 will get four consecutive vector numbers. The programmer must specify the high order six bits. Note that the resulting 8-bit vector number must be in the range [$40_{16}$,$FF_{16}$]. For example, suppose that interrupts caused by H1 through H4 shall generate the vector numbers [$40_{16}$, $43_{16}$]. then the following instruction initializes the PI/T to supply these vector numbers:

```
PIVR  EQU     $FFF005
      MOVE.B  #%01000000,PIVR
```

assuming that the base address of the PI/T is $FFF000_{16}$. In order to connect four interrupt service routines to H1 through H4, the programmer must initialize the exception vector table and enable the interrupts. Assuming that the designer of the computer system has decided that the PI/T generates interrupts at interrupt priority level 3, the necessary initializations of the interrupt system is as follows:

```
INIT   MOVE.B  #%01000000,PIVR
       MOVE.L  #H1INT,$100       ; Exception address for H1
       MOVE.L  #H2INT,$104       ; Exception address for H2
       MOVE.L  #H3INT,$108       ; Exception address for H3
       MOVE.L  #H4INT,$10C       ; Exception address for H4
       MOVE    #$2200,SR         ; Set interrupt priority 2
       ...                       ; Interrupts are enabled
H1INT  ...
       RTE
H2INT  ...
       RTE
H3INT  ...
       RTE
H4INT  ...
       RTE
```

It is important to note that the interrupt priority level and the vector number for an interrupt is not the same. Since there are only seven interrupt priority levels, the system designer must let many devices generate interrupts at the same priority. An important issue now arises. How can we disable some devices to generate an interrupt at a certain priority level and still let other devices at the same priority level be able to generate interrupts. Most parallel interfaces have provision for enabling and disabling a certain interrupt. In the PI/T, it is possible to enable/disable interrupts caused by the handshake lines. In order to enable interrupts caused by the H1 handshake line, bit 1 in the PACR must be set (see Table 7.8).

## 7.4 Summary and concluding remarks

In this chapter, we have seen how a variety of communications protocols can be supported by programmable interfaces. The simplest kind of interface problem is to control output devices, such as lamps, and read status of input devices such as switches. This type of I/O is called bit I/O. Programmable interfaces usually contain ports where each individual bit can be programmed as either an input or an output.

When information is exchanged between two devices, such as between computers and printers or terminals, the information flow needs to be controlled in one way or another. We have seen how handshaking can be used to synchronize data transmission; two handshake lines are sufficient to design such protocols. When the sender has a data item to transfer, it notifies the receiver by asserting its handshake line. The receiver, on the other hand, uses another handshake line to notify the sender when data has been read.

When the distance between two devices is small, one can use bit-parallel data transfers. Several bytes can then be transferred at the same time. We have looked at a parallel programmable interface which supports various handshake protocols. Its ability to take care of all handshake control, frees the processor from this task. The processor can either use polling or interrupts to synchronize when data is available or ready to send.

When the distance between two devices is large, the cost of connecting two devices by means of parallel interfaces soon becomes prohibitively expensive. In such situations, we can use a bit-serial communications protocol. The asynchronous bit-serial protocol that we have seen in this chapter is commonly used to connect e.g. terminals with computers. It is asynchronous in the sense that the synchronization information is provided in the bit-stream itself. The start bit is used to notify the receiver that a new data item is on its way. In order for the receiver to perform single-bit error detection, a simple method called parity calculation is often used. The parity bit indicates whether the number of ones contained in the data word is even or odd. If the receiver counts the number of ones, it can decide whether a single-bit error has occurred by simply checking the parity bit. All these functions are provided by UART-interfaces (Universal Asynchronous Receiver/Transmitter). We looked at one such example, namely the ACIA.

When a computer system needs to take care of a large number of interrupts, it can use vectored interrupts by letting each device identify itself by a vector number. The processor uses the vector number to find out about the address of the interrupt service routine.

Chapter 8

# Real-Time Applications

We have now introduced the most important concepts of a computer system from the machine language programmer's point of view. This chapter aims at taking a broader look at what we have learned by applying it to the important domain of real-time applications.

The flexibility of a computer is that it can be programmed to perform various tasks. Many microprocessor systems are parts of equipment that aim at controlling various processes. Such systems are known as *embedded* in the sense that the microprocessor system has a specific task in the entire system. The task to be performed can, for instance, be to regulate the temperature in a chemical process or to regulate the flaps in an aircraft etc. Typical for these applications is that time plays an important role. Therefore, we call them time-critical or real-time applications.

This chapter gives us some understanding about the basic problems that need to be addressed in the course of real-time applications. We will focus on how these concepts can be implemented rather than giving an exhaustive treatment of the issue, that can be found in almost every text on operating system design. Before we look at the implementation of certain real-time mechanisms, we shall extend the model of the M68000 by looking at the important concept of supervisor and user mode in Section 8.1. We will then look at exception management in general, that is, the management of all exceptional events such as interrupts in Section 8.2. In Sections 8.3 and 8.4, we present the implementation of a simple time-sharing and real-time operating system.

## 8.1 Supervisor and user mode

In Chapter 6, we introduced the concept of interrupt. The interrupt system provides a mechanism to make efficient use of the processing power of the processor. The reason for this is that the processor can perform useful work until it is notified

that an external device needs service. Since an event can cause the processor to interrupt at any time, it is important that the interrupted program is unaffected by the interrupt service routine. To be more specific, the *processor context*, that is, the contents of all registers must be unaffected by the interrupt service routine.

We solved this problem by saving the contents of all registers on the stack at the beginning of the interrupt service routine. By doing this, it appears in the interrupted program as if nothing had happened other than the fact that the interrupt service routine 'steals' cycles from the processor. Another way of looking at it is that the processor (including all its registers) is shared between the main program and the interrupt service routine. From the viewpoint of these two programs (the interrupted program and the interrupt service routine), it appears as if they have the processor on their own. The idea of assigning a 'virtual' processor to a program is fruitful because since the processor is very fast we could indeed share it between several user (or application) programs.

Let us consider $N$ programs that are to be executed on the processor. Each program produces a result every $T$ seconds and the execution time is $KT$ seconds. Furthermore, a user is associated with each program and waits for it to produce a new result. The question is how we should share the processor among these programs.

One approach would be to run each program until completion and then start the next one. This would result in having the first user to obtain the first result after $T$ seconds while the last user would have to wait $KT(N-1)+T$ seconds. Although the first user is happy the last user would certainly consider this approach to be unfair.

In order to make all users wait about the same amount of time, we could choose to execute the first program for a short while (say 1 millisecond) and then the second one etc. Theoretically, each user would have to wait $NT$ seconds for the first result, $2NT$ seconds for the second result and $KTN$ seconds for the last result. We call this approach *time-sharing* because each program, often referred to as a *process*, is assigned a *time-slice* and the processor is **time-shared** between the processes. Figure 8.1 shows how the time-sharing scheme affects the execution of each process; Proc0, Proc1,...,ProcN-1 are executed a time-slice in turn. After ProcN-1 has been executed for a while, Proc0 is executed again and so on.

One way of implementing the time-sharing scheme is to use a timer that generates an interrupt periodically. The interrupt service routine saves all registers of the executing process in a dedicated memory area and restores the register contents of the next process before it is restarted. In general, if we have $N$ processes and process $i$ is executing, the task of the interrupt service routine is to save the registers of process $i$ and restart a new process $i+1 (\mathrm{mod}\ N)$. What we have achieved with this scheme is that the processor is assigned to each process in turn, something often referred to as a *round-robin* policy. The processor assigns a time-slice as dictated by the timer to each process. The action of swapping off a process and restarting a new one is called a *context-switch* because the context of the running process is saved and the context of the next process in turn is restored. In order


**Figure 8.1** Time-sharing among $N$ processes according to round-robin.

to implement the above scheme, we need to make precise what we mean by the context of a process. The context is basically all information needed to restart a program and making it execute at the same point it was interrupted when the timer went off. Below we list the information that specifies the context:

- *Registers.* All registers can potentially be used by a program, that is, D0-D7, A0-A6, SP, and SR.
- *Program counter* (PC). We must save the PC to be able to restart a process at the point it was interrupted.
- *Stack pointer.* This is a special case of the registers.
- *Stack space.* Each process should have its own stack in order to prevent other processes from destroying it.

In Chapter 6, all registers and the program counter were saved on the stack when an interrupt service routine was executed. This is a simple method of saving the context because we could restore the information by popping the stack. When we have more than one process, this scheme is not possible to use — each process must have its own stack. In order to cope with this, many computers such as the M68000 provide two stack pointers called supervisor stack pointer (SSP) and user stack pointer (USP). The processor can be operated in two modes called *supervisor mode* and *user mode.*

When the processor encounters an interrupt it enters supervisor mode. There are also other exceptions that cause the processor to enter supervisor mode such as traps which will be treated in the next section. When the supervisor mode is entered, the processor uses the supervisor stack pointer to store data. For instance, the return address of the interrupted program is saved on to the supervisor stack.

The supervisor mode also makes it possible to execute certain *privileged instructions.* An example of a privileged instruction is `MOVE` $a$`,SR`, which is used to change the content of the SR (status register). This instruction may only be executed in supervisor mode. Bit 13 in SR (see Figure 6.6 at page 104) controls the operating mode; if S=1, then the processor is in supervisor mode, otherwise, it is in user mode. If the processor executes in user mode, register `SP` (or A7) refers to `USP` and otherwise, it refers to `SSP`.

We are now ready to present a scheme that saves and restores the context of a process. Below, we show parts of the code for two processes PROC0 and PROC1.

```
PROC0   ...         ; Here starts PROC0
        ...
        ...         ; The last instruction

; Process control block for PROC0

CNTX0   DS.L  15    ; D0 - D7, A0 - A6
        DS.L  1     ; SP (USP)
        DS.W  1     ; SR
        DS.L  1     ; PC
        DS.L  10    ; Stack space for process 0
STACK0  DS.L  1

PROC1   ...         ; Here starts PROC1
        ...
        ...         ; The last instruction

; Process control block for PROC1

CNTX1   DS.L  15    ; D0 - D7, A0 - A6
        DS.L  1     ; SP (USP)
        DS.W  1     ; SR
        DS.L  1     ; PC
        DS.L  10    ; Stack space for process 1
STACK1  DS.L  1
```

We have associated a memory area, called a *process control block* (pcb), with each process that has space for all registers (D0 – D7, A0 – A6, USP, SR, and PC). In addition, we allocate 10 long words of stack space for each process. Note that the stack grows towards lower addresses. This is why we associate the symbolic addresses STACK0 and STACK1 with the last long word in the stacks.

The following sequence of instructions initializes the processor to begin executing at address PROC0:

```
MOVEA.L  #STACK0,A0   ; Stack for PROC0
MOVE     A0,USP       ; Initialize the USP
MOVE     #0,SR        ; Enter user mode
BRA      PROC0
```

The two first instructions initialize the user stack pointer to contain the address of the first element of stack STACK0. We have used a special form of the MOVE

instruction. (See Appendix B.)

We now turn our attention to the part of the interrupt service routine that implements the context switch. This is usually a part of the operating system called *scheduler*. What we wish to do is to save the context of the currently running process in its process control block. Suppose that PROC0 is running and that an interrupt is encountered. From Chapter 6 we learned that the processor will set the S-bit (recall the actions taken when an interrupt is generated from page 105). It will then push PC and the old value of SR onto the supervisor stack before it branches to the entry point of the interrupt service routine. Having this in mind, the following sequence of instructions saves the context in the process control block of PROC0:

```
A0OFF   EQU     32                  ; Displacement to A0
USPOFF  EQU     60                  ; Displacement to USP
SROFF   EQU     64                  ; Displacement to SR
PCOFF   EQU     66                  ; Displacement to PC

SCHED   MOVE.L  A0,-(SP)            ; Push A0 using SSP
        MOVEA.L #CNTX0,A0           ; Address to pcb
        MOVEM.L D0-D7/A0-A6,(A0)    ; Save D0-D7 and A0-A6
        MOVE.L  (SP)+,A0OFF(A0)     ; Pop A0 using SSP
                                    ; and save it
        MOVE    USP,A1
        MOVE.L  A1,USPOFF(A0)       ; Save USP
        MOVE.W  (SP)+,SROFF(A0)     ; Save SR
        MOVE.L  (SP)+,PCOFF(A0)     ; Save PC
```

SCHED is the entry point of the scheduler. The first thing to be done is to release one of the address registers (A0) to be used to point at the process control block. We have declared displacements that can be used to access certain entities in the process control block. For instance, when all data and address registers have been saved by MOVEM.L D0-D7/A0-A6,(A0), we need to save the old value of A0 that we temporarily have stored in the supervisor stack. This is done by the instruction MOVE.L (SP)+,A0OFF(A0). Note the order in which we access the return address and the content of SR from the system stack. This is the reverse order from which the processor pushed them during the interrupt cycle.

The next sequence of instructions performs the opposite operation; it copies the context from process PROC1 to the processor registers:

```
MOVEA.L   #CNTX1,A0          ; Address to pcb
MOVEA.L   USPOFF(A0),A1
MOVE      A1,USP             ; Restore user stack pointer
MOVE.L    PCOFF(A0),-(SP)    ; Restore program counter
MOVE.W    SROFF(A0),-(SP)    ; Restore status register
MOVEM.L   (A0),D0-D7/A0-A6   ; Restore D0-D7 and A0-A6
RTE                          ; Activate PROC1
```

An interesting observation from the above sequence of instructions is how the PC and the SR are restored. We push their values from the process control block onto the supervisor stack. When `RTE` is executed, PC and SR are restored from the supervisor stack and `PROC1` is restarted correctly.

In summary, we have presented a scheme that can be used to save and restore the context of a process. By providing two operation modes; the user and supervisor mode, we can have separate stacks for all user programs and the scheduler. Later in this chapter, we will present the code of a simple time-sharing operating system that enables us to execute an arbitrary number of processes on the same processor reliably.

## 8.2 Exceptions

We have only met one kind of exception, namely interrupts. Interrupts are examples of external events that cause the processor to perform a desired action. There are also exceptions caused by internal events. These are often referred to as *traps*. Examples of traps are:

- *Address error.* An attempt to execute an instruction or access a word or long word at an odd address.
- *Illegal instruction.* An attempt to execute an operation word that does not correspond to a valid instruction.
- *Trap on Overflow.* Explicit trap when the V-flag is set.
- *Zero divide.* An attempt to divide by zero.
- *Privilege violation.* An attempt to execute a privileged instruction in user mode.
- *Trace enabled.* The trace-bit is set (bit 15 in SR).
- *Explicit traps.* An explicit `TRAP` instruction has been executed.

When any of the above mentioned traps are generated, the processor performs the same actions as for interrupts (see page 105) except that it will fetch the address of the trap service routine, called a *trap handler*, at another place in the exception vector table. Another difference is that the current priority level of the processor

**Table 8.1** Entries in the exception vector table for various traps.

| Exception | Address in exception vector table |
|---|---|
| Address error | $C_{16}$ |
| Illegal instruction | $10_{16}$ |
| Zero divide | $14_{16}$ |
| Overflow trap (TRAPV) | $1C_{16}$ |
| Privilege violation | $20_{16}$ |
| Trace enabled | $24_{16}$ |
| Explicit traps | $80_{16} + 4n$ |

is not changed. Table 8.1 shows the entries for the different traps in the exception vector table.

M68000 provides two division instructions `DIVS` *a*`,D`*i* and `DIVU` *a*`,D`*i* for signed (`DIVS`) and unsigned (`DIVU`) division of a 32-bit number by a 16-bit number, where the destination operand (a data register) is divided by the source operand. The result of the execution is that the quotient is available in the 16 least significant bits of D*i* and the remainder is available in the 16 most significant bits. For additional information, please refer to Appendix B. Now if the divisor (the source operand) is zero, a Divide-by-Zero trap is generated.

An overflow trap can explicitly be used by inserting the `TRAPV` instruction in the code. If the V-flag is set, the processor will automatically invoke the trap handler whose address is stored at address $1C_{16}$ (see Table 8.1). If a privileged instruction is executed in user mode, a privilege-violation trap is generated.

Now recall the T-bit (Trace bit) in the SR from Chapter 6. An important feature of a debugger is to interrupt the execution of a program at a specific address (breakpoint) or after each instruction (single-step). The Trace-bit can be used to cause a trap after the execution of each instruction. When the T-bit is set, a Trace trap is generated after each instruction. Note that it is only possible to trace a program in user mode this way because the T-bit is reset when an interrupt or a Trace trap is handled.

There are 16 explicit trap instructions available. They are denoted `TRAP #`*n*, where $n$= [0,15]. `TRAP #i` results in a trap to address $80_{16} + 4n$. For instance, the following sequence of instructions initializes the `TRAP #0` instruction to perform a trap to address `TRAP0`:

```
        MOVE.L  #TRAP0,$80  ; Initialize the exception
                            ; vector table
        TRAP    #0
        ...
TRAP0   ...
        RTE
```

When the `TRAP #0` instruction is executed, the processor performs the actions involved when an interrupt is generated and continues to execute at address `TRAP0`.

Note that the last instruction to be executed in a trap handler is `RTE`, that is, the same as we used to exit from an interrupt service routine. Explicit `TRAP` instructions can be used to enter supervisor mode and in a controlled fashion execute a certain piece of code.

To summarize, traps and interrupts both cause the processor to perform a subroutine call to a trap handler or interrupt service routine. Traps and interrupts are collectively called *exceptions* and the routine that services an exception is called an *exception handler*.

## 8.3 Time-sharing operating systems

An *operating system* of a computer system is responsible for all its resources including the processor and I/O-devices. In this section, we will look at the piece of code used to manage the processor resource. Recall the round-robin policy from Section 8.1 which in turn assigned a time-slice to each of a number of processes. We shall here present the complete code of this scheduler.

A timer is connected to interrupt input $I_5$. It generates an interrupt periodically (typically each millisecond). When an interrupt is generated, the following actions are taken by the scheduler:

- Save the context of the currently running process.
- Choose the next process to be restarted.
- Restore the context of the next process and restart it.

Figure 8.2 shows how N processes and the timer interrupts interact with the scheduler. The scheduler is simply an interrupt service routine that performs a context-switch on each timer interrupt.

On the next few pages, we show the code of a scheduler and two processes. The first part of the program initializes the interrupt system and starts process `PROC0`. The scheduler (`SCHED`) is designed to be able to handle an arbitrary number of processes in the sense that we can add more processes without having to rewrite

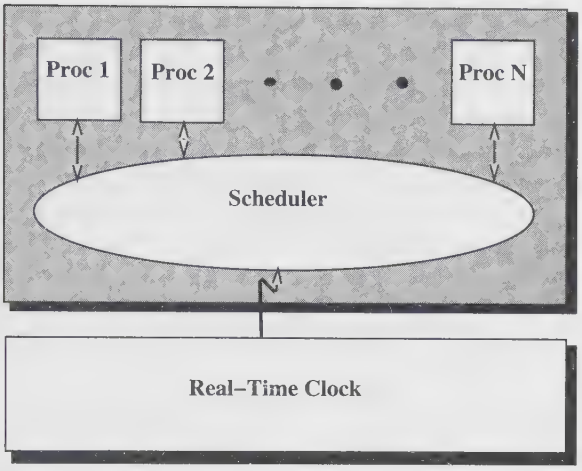



**Figure 8.2** The structure of the time-sharing operating system.

the scheduler code. All we have to do is to add the code and a process control block for the new process and insert a constant in the data structure that appears at the end of the assembly code:

```
A0OFF    EQU      32                ; Displacement to A0
USPOFF   EQU      60                ; Displacement to USP
SROFF    EQU      64                ; Displacement to SR
PCOFF    EQU      66                ; Displacement to PC
NUMPRO   EQU      2                 ; Number of processes

;================== Initializations ================================

START    MOVEA.L  #CNTX1,A0         ; Initialize the pcb of PROC1
         MOVE.W   #$0400,SROFF(A0)  ; Initialize SR
         MOVE.L   #PROC1,PCOFF(A0)  ; Initialize PC
         MOVE.L   #STACK1,SPOFF(A0) ; Initialize SP
         MOVEA.L  #STACK0,A0
         MOVE     A0,USP            ; Initialize USP of PROC0
         MOVE.L   #SCHED,$74        ; Initialize exception vector
         MOVE     #$0400,SR         ; Enable timer interrupts and
                                    ; enter user mode
         BRA      PROC0             ; Start PROC0
```

The initialization part in the beginning of the program must initialize parts of the process control block of PROC1 in order for the scheduler to start it at the entry point with correct status register and user stack pointer contents. In addition, we initialize the user stack pointer for PROC0 to point to the stack of process PROC0. Then we initialize the interrupt system with the exception vector entry equal to the address of the interrupt service routine SCHED. Note that we assume that the timer is using an autovector with the interrupt priority level 5. We therefore set the current processor priority (CPL) to 4 in order to enable timer interrupts. We enter user mode by resetting the S-bit in the status register (bit 13) at the same time. The two processes PROC0 and PROC1 are defined according to the code at page 140.

```
SCHED
; SAVE_CNTXT
        MOVE.L   A0,-(SP)          ; Push A0 onto the stack
        MOVE.L   D0,-(SP)          ; Push D0 onto the stack
        MOVEA.L  #CNTXTS,A0        ; Calculate the address
        MOVE.L   ACTIVE,D0         ; to the pcb of the
        ASL.L    #2,D0             ; currently running process
        MOVEA.L  0(A0,D0),A0       ; A0 contains the address
                                   ; to the pcb
        MOVE.L   (SP)+,D0          ; Restore D0
        MOVEM.L  D0-D7/A0-A6,(A0)  ; Save D0-D7 and A0-A6
        MOVE.L   (SP)+,A0OFF(A0)   ; Pop A0 from the stack
                                   ; and save it in the pcb
        MOVE     USP,A1
        MOVE.L   A1,USPOFF(A0)     ; Save USP
        MOVE.W   (SP)+,SROFF(A0)   ; Save SR
        MOVE.L   (SP)+,PCOFF(A0)   ; Save PC

; SELECT_NEXT
        ADDI.L   #1,ACTIVE
        CMPI.L   #NUMPRO,ACTIVE
        BNE      RSTORE
        MOVE.L   #0,ACTIVE

; RESTORE_CNTXT
RSTORE  MOVEA.L  #CNTXTS,A0        ; Calculate the address
        MOVE.L   ACTIVE,D0         ; of the pcb of the
        ASL.L    #2,D0             ; next running process
        MOVEA.L  0(A0,D0),A0       ; A0 contains the address
                                   ; of the pcb of next process
        MOVEA.L  USPOFF(A0),A1
        MOVE     A1,USP            ; Restore USP
        MOVE.L   PCOFF(A0),-(SP)   ; Restore PC
        MOVE.W   SROFF(A0),-(SP)   ; Restore SR
        MOVEM.L  (A0),D0-D7/A0-A6  ; Restore D0-D7 and A0-A6
        RTE                        ; Activate next process
```

We next show the data structure used by the scheduler.

```
;================== Data structure for the scheduler ================

CNTXTS   DC.L   CNTX0,CNTX1   ; Addresses to each pcb

ACTIVE   DC.L   0             ; Current process ID
         END
```

The scheduler SCHED has the following general structure

```
procedure SCHED;
begin
   SAVE_CNTXT;
   SELECT_NEXT;
   RESTORE_CNTXT;
end;
```

The scheduler keeps track of the currently running process by a variable called ACTIVE which is initialized to 0. The first part of the scheduler (see assembly code on the previous page) saves the context of the currently running process. The addresses of the process control blocks are stored in a vector at address CNTXTS. In SAVE_CNTXT, the context of the currently running process is stored in its process control block. The first part of SAVE CNTXT aims at calculating the address of the process control block of the current process (see the assembly code). This can be expressed in terms of the content of ACTIVE as: CNTXTS + 4(ACTIVE). The multiplication by four is implemented by shifting the contents of ACTIVE twice (ASL.L #2,D0).

The second part of the scheduler (SELECT_NEXT) aims at selecting the next process to be restarted. This is done by incrementing ACTIVE modulo NUMPRO which is a constant that specifies the number of processes. The third part of the scheduler (RESTORE_CNTXT) restores the process control block of the selected process and restarts it.

We have structured the scheduler this way to be able to add new processes without having to change the code for the scheduler. A new process is added by adding a new framework consisting of the code and a process control block. The constant NUMPRO must be incremented by one, and finally, the address of the process control block for the new process must be inserted in the table at address CNTXTS.

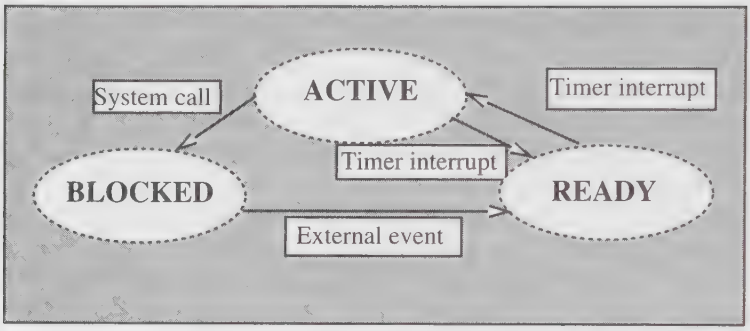



**Figure 8.3** State diagram of a process.

## 8.4 Real-time control

The scheduler in the previous section is only meaningful if all processes can perform useful computation all the time. If that is the case, the processor is efficiently shared between all processes.

Now assume that a process is waiting for a human user to input data. Then the scheduler would make more efficient use of the processor if it did not assign any time at all for a process that is waiting. What the scheduler needs, is to be able to handle two kinds of processes; those ready for execution and those blocked because they are waiting for an external event such as manual input from a keyboard.

In this section, we shall extend the functionality of the scheduler to take care of external events. Each process can be in exactly one of the following three states: ACTIVE, READY, or BLOCKED.

In Figure 8.3, we show these states and what actions that cause a process to transit from one state to another. Note that at most one process can be active at a time. The number of processes in state READY ranges from zero to the total number of processes minus one (the one that is in state ACTIVE), whereas the number of processes in state BLOCKED ranges from zero to the total number of processes. However, the sum of processes in all states is of course the total number of processes.

A state transition is triggered by an event. An event can be internal (a system call) or external (a timer interrupt or another type of interrupt). The purpose of the scheduler is to make a choice as to which process to run next based on the type of event and to move processes in between the states. The facts that there can be zero processes that are ready for execution and more than one process in state BLOCKED or READY have two important implications. First, in case all processes are BLOCKED, the scheduler cannot activate a new process. Therefore, an idle process called the NULL process must be available. Second, since more than one process can be BLOCKED or READY, a queue must be associated with these states. The NULL process is designed in the same manner as an ordinary

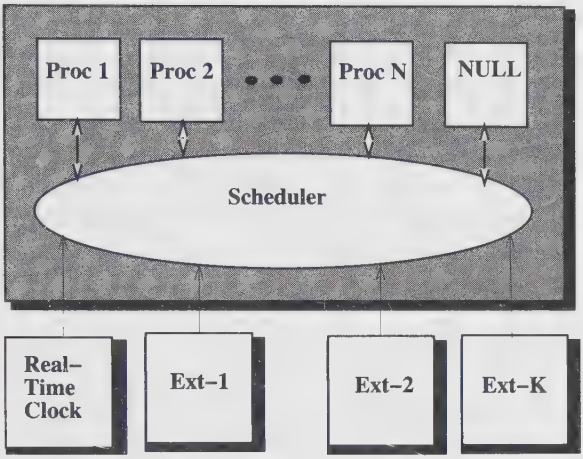



**Figure 8.4** The structure of a simple real-time operating system.

process. Thus, it has a process control block in order to make it possible to treat it in the same way as any other processes. However, the NULL process does not perform any useful task; it simply executes an infinite loop.

Figure 8.4 shows the general structure of a simple real-time operating system we will present on the next few pages. It supports $N$ processes and can handle $K$ events in the following way. As long as no process needs service, all processes will be in the ready state except for one process which is in the active state, and thus is executing. For each timer-interrupt, the scheduler selects a new process in the ready queue according to a round-robin. When a process needs service (e.g. is waiting for a keyboard input) it performs a system call. The system call invokes the scheduler which removes the process from the active state and puts it into the queue for blocked processes (see Figure 8.3). The process will be blocked until the external event occurs that it is awaiting. At this point, the scheduler is invoked again to move the process from the BLOCKED state to the READY state.

Note that the basic structure of the scheduler is the same as before: `SAVE_CNTXT`, `SELECT_NEXT`, and `RESTORE_CNTXT`. The only part that differs from the time-sharing operating system is the `SELECT_NEXT` procedure. This procedure is presented below in a Pascal-like notation. We start with the additional data structures needed to control the action of the scheduler, in essence, the events and the queues associated with the BLOCKED and READY states.

```
type  EVENT_TYPE = (SYS_CALL,TIMER,EXT_EVENT);
      EXT_TYPE   = (NON_BLOCK,EXT_1,EXT_2,...,EXT_K);
      READY_TYPE = (NOT_READY,READY);

const NUMPROC = 10;
      NULL = NUMPROC;
```

```
var
      EVENT : EVENT_TYPE;
      SYS   : EXT_TYPE;

      BLOCK_Q : array[0..NUMPROC-1] of EXT_TYPE;
      READY_Q : array[0..NUMPROC-1] of READY_TYPE;

      ACTIVE : integer;
```

We have defined three types of events that can result in the scheduler to be invoked SYS_CALL, TIMER, and EXT_EVENT which are caused by a system call, a timer interrupt, and an external event, respectively. When the scheduler is invoked, the event type is reflected by the variable EVENT. Each system call corresponds to a service. For example, when input from a keyboard is needed, the process simply performs a system call. The corresponding process is then blocked until an interrupt from the keyboard occurs. To support such external services, we associate an external event with each system call. There are $K$ such external events named EXT_1, EXT_2,..., EXT_K. When either a system call or an external event occurs, the type of the external event is available in variable SYS. The BLOCKED and READY queues are implemented by two vectors (BLOCK_Q and READY_Q) that have the same number of elements as the number of processes (NUMPROC). Assuming that process $i$ is blocked due to an external event EXT_k, then BLOCK_Q[$i$] = EXT_k and READY_Q[$i$] = NOT_READY. Let us now look at the SELECT_NEXT procedure.

```
procedure SELECT_NEXT;
begin
  if EVENT = SYS_CALL then
    BLOCK_Q[ACTIVE]:=SYS
  else if EVENT = TIMER then
         READY_Q[ACTIVE]:=READY
       else if EVENT = EXT_EVENT then
              MAKE_READY;
  ACTIVE:=NEXT_PROC;
end;
```

The purpose of the SELECT_NEXT procedure is to move processes between the different states according to the state-transition graph in Figure 8.3. Note that the identity of the currently running process is stored in the variable ACTIVE. First, if a process performs a system call, the scheduler puts the currently running process into the BLOCKED state simply by marking the vector element that corresponds to the currently active process with the type of the external event (BLOCK_Q[ACTIVE]:=SYS).

Second, if the scheduler is invoked as a result of a timer interrupt (EVENT = TIMER), the currently running process will be put into the READY queue. Finally,

if the scheduler is invoked as a result of an external event (EVENT = EXT_EVENT), the process that was blocked due to this event is removed from the BLOCKED queue and inserted into the READY queue (MAKE_READY).

Independent of the type of event that invoked the scheduler, a new process must be selected as the next running process. This task is accomplished by the function NEXT_PROC.

We now look at the implementation of NEXT_PROC and MAKE_READY. We list the specification of these functions and subroutines in a Pascal-like notation below.

```
function NEXT_PROC:integer;
begin
  i:=0;   NEXT:=ACTIVE;
  repeat
    i:=i+1;
    NEXT:=NEXT+1;
    if NEXT = NUMPROC then
       NEXT:=0;
    until (i=NUMPROC) or READY_Q[NEXT] = READY;

    if i<>NUMPROC then
    begin
       NEXT_PROC:=NEXT;
       READY_Q[NEXT]:=NOT_READY;
    end;
    else
       NEXT_PROC:=NULL;
end;
```

The NEXT_PROC function selects the next process to be activated by performing a round-robin policy among the processes that are marked READY in the READY queue. If all processes are marked NOT_READY, the NULL process will be the next process to run. We next look at the MAKE_READY procedure.

```
procedure MAKE_READY;
begin
  i:=0;
  while BLOCK_Q[i] <> SYS do
     i:=i+1;
  BLOCK_Q[i]:=NON_BLOCK;
  READY_Q[i]:=READY;
end;
```

The purpose of the MAKE_READY procedure is to find the identity of the process that is blocked due to the external event SYS. By examining the BLOCKED queue, the index of the element that matches SYS is the identity of the process that has been blocked due to this external event. This process is moved to the READY queue and removed from the BLOCKED queue.

Note that this real-time scheduler is simplified; it assumes that exactly one process can be blocked for each external event. Despite this limitation, we will look at the implementation of the scheduler next.

To simplify the presentation, we will assume that there are two system calls and external events. The external events are caused by two distinct interrupts at priority 1 and 2, respectively. The corresponding system calls are performed by executing TRAP #1 and TRAP #2. Below, we show the necessary initializations of the exception vector table for these traps and interrupts. We also assume that the timer generates an autovectored interrupt with priority 5.

```
INIT  MOVE.L  #TRAP1,$84   ; Entry for TRAP #1
      MOVE.L  #TRAP2,$88   ; Entry for TRAP #2
      MOVE.L  #INT1,$64    ; Entry for interrupt 1
      MOVE.L  #INT2,$68    ; Entry for interrupt 2
      MOVE.L  #INT5,$74    ; Entry for timer interrupt
      MOVE    #$2000,SR    ; Enable all interrupts
```

Given the entry points of the system calls and the interrupts above, we show the data structures needed to implement the real-time scheduler below:

```
; EVENT_TYPE

SYS_CALL    EQU  0
TIMER       EQU  1
EXT_EVENT   EQU  2

; EXT_TYPE

NON_BLOCK   EQU  0
EXT_1       EQU  1
EXT_2       EQU  2
```

```
; READY_TYPE

NOT_READY   EQU   0
READY       EQU   1
NUMPROC     EQU   10
NULL        EQU   NUMPROC

; Variables

EVENT       DS.L  1
SYS         DS.L  1
BLOCK_Q     DS.B  NUMPROC
READY_Q     DS.B  NUMPROC
            EVEN
ACTIVE      DS.L  1
```

Before we show the implementation of the SELECT_NEXT procedure, we show the entry points for the traps and the interrupts below:

```
TRAP1   MOVE.L  #SYS_CALL,EVENT    ; EVENT:=SYS_CALL
        MOVE.L  #EXT_1,SYS         ; SYS:=EXT_1
        BRA     SCHED

TRAP2   MOVE.L  #SYS_CALL,EVENT    ; EVENT:=SYS_CALL
        MOVE.L  #EXT_2,SYS         ; SYS:=EXT_2
        BRA     SCHED

INT1    MOVE.L  #EXT_EVENT,EVENT   ; EVENT:=EXT_EVENT
        MOVE.L  #EXT_1,SYS         ; SYS:=EXT_1
        BRA     SCHED

INT2    MOVE.L  #EXT_EVENT,EVENT   ; EVENT:=EXT_EVENT
        MOVE.L  #EXT_2,SYS         ; SYS:=EXT_2
        BRA     SCHED

INT5    MOVE.L  #TIMER,EVENT       ; EVENT:=TIMER
        BRA     SCHED
```

For each system call and interrupt, we properly set up the variables that keep track of the event type (EVENT) and the type of external event (SYS). After that

point, a branch is taken to the entry point of the scheduler (SCHED). For example, if a system call is performed by executing TRAP #1, the variables EVENT and SYS are assigned SYS_CALL and EXT_1, respectively. We are now ready to present the implementation of the SELECT_NEXT procedure (we call the entry point SEL below).

```
; SELECT_NEXT

SEL   CMPI.L   #SYS_CALL,EVENT   ; if EVENT = SYS_CALL
      BEQ      SYST              ; then goto SYST
      CMPI.L   #TIMER,EVENT      ; else if EVENT = TIMER
      BEQ      TIME              ; then goto TIME
      BSR      MAKE_READY        ; MAKE_READY
      BRA      ACT
SYST  MOVEA.L  #BLOCK_Q,A0
      MOVE.L   ACTIVE,D0
      MOVE.L   SYS,0(A0,D0)      ; BLOCK_Q[ACTIVE]:=SYS
      BRA      ACT
TIME  MOVEA.L  #READY_Q,A0
      MOVE.L   ACTIVE,D0
      MOVE.L   #READY,0(A0,D0)   ; READY_Q[ACTIVE]:=READY
ACT   BSR      NEXT_PROC         ; ACTIVE:=NEXT_PROC
      RTS
```

The implementations of NEXT_PROC and MAKE_READY are straightforward and left to the reader as an exercise. Before we close this chapter, we want to note the following. The simple real-time scheduler we have outlined does not address several important issues. For example, in real-life implementations, there are often some processes that should be given more time than others. Therefore, one often has a number of READY queues with different priorities. Processes with the same priority are competing with each other using a round-robin scheme. Another problem is that mutual exclusion is needed for shared resources. For instance, a keyboard should be owned by one process at a time – imagine how strange it could be if one process is reading from a keyboard and another steals some of the characters. Therefore, there are dedicated mechanisms in an operating system that guarantee mutual exclusion. To go into further detail of this is definitely outside the scope of this text. However, the interested reader should consult a text on operating system principles.

## 8.5 Summary and concluding remarks

In this chapter, we have introduced some new features needed to support the execution of multiple programs on the same processor. The crucial point is to save the context of each program called a process control block. The context includes the content of all registers and the stack of each process.

In order to handle multiple stacks, we can use the supervisor and user mode concepts. This makes it possible for the supervisor (in our example the scheduler) to have its own stack, while all user processes have their own private stacks which are controlled by the user stack pointer.

We also talked about exceptions in general. Exceptions can be external events such as interrupts, but also internally generated events such as certain arithmetic conditions and explicit system calls. These are referred to as traps.

We implemented a time-sharing operating system which makes it possible for multiple programs to share the same processor. The scheduler executes each process for a short while, a time-slice. It then picks a new process. This makes it appear as if all processes are executing at the same time.

Finally, we generalized the scheduler to take care of system calls which enabled us to make more efficient use of the processor. A program (or process) may be waiting for an external event. When it is waiting (blocked) it does not load the processor. Therefore, we introduced three states: ACTIVE, READY, and BLOCKED. The process that currently uses the processor is denoted ACTIVE, while all other processes that are ready are kept in state READY. Those processes that are waiting for an event are kept in the state BLOCKED. Since it is possible that all processes are kept in state BLOCKED, a system process called NULL is needed. This process has the same basic structure as the other processes but executes in an infinite loop.

Appendix A

# Solutions to Exercises

**1.1** $42_{10}$

**1.2** $2303_{10}$

**1.3** $0921_{16}$

**1.4** $011101_2 = 11101_2$

**1.5** $100110101000_2$

**1.6** $272\mathrm{E}_{16}$

**1.7** $[0, 255]$

**1.8** $[0, 63]$

**1.9** $[-128, 127]$

**1.10** $[-32, 31]$

**1.11** 00000111

**1.12** 11111001

**1.13** Interpretation as an unsigned integer: $9_{10}$
Interpretation as a signed (two's complement) integer: $-7_{10}$

**1.14** $46_{16} = 1000110_2$

**1.15** $2\mathrm{B}_{16} = 0101011_2$

**1.16** $61_{16} = 1100001_2$

**1.17** 48 45 4C 4C 4F. Hexadecimal representation

**1.18** 68000

---

**2.1** $A = 0111_2 = 7_{10}, B = 0001_2 = 1_{10}, A+B = 1000_2 = 8_{10}$. The addition did not result in overflow.

**2.2** Range $[0, 31]$ $A = 00100_2 = 4_{10}, B = 11110_2 = 30_{10}, A+B = 00010_2 = 2_{10}$. The addition resulted in overflow.

**2.3** Range $[0, 63]$ $A = 011000_2 = 24_{10}, B = 000001_2 = 1_{10}, A + B = 011001_2 = 25_{10}$. The addition did not result in overflow.

**2.4** Range $[0, 255]$ $A = 10000000_2 = 128_{10}, B = 10000000_2 = 128_{10}, A + B = 00000000_2 = 0_{10}$. The addition resulted in overflow.

**2.5** $A = 0111_2 = 7_{10}, B = 0001_2 = 1_{10}, A + B = 1000_2 = -8_{10}$. The addition resulted in overflow since both numbers have the same sign (positive) and the sum has opposite sign (negative).

**2.6** Range $[-16, 15]$ $A = 00100_2 = 4_{10}, B = 11110_2 = -2_{10}, A + B = 000010_2 = 2_{10}$. The addition did not result in overflow.

**2.7** Range $[-32, 31]$ $A = 011000_2 = 24_{10}, B = 000001_2 = 1_{10}, A + B = 011001_2 = 25_{10}$. The addition did not result in overflow.

**2.8** Range $[-128, 127]$ $A = 10000000_2 = -128_{10}, B = 10000000_2 = -128_{10}, A + B = 00000000_2 = 0_{10}$. The addition resulted in overflow.

**2.9** 0010

**2.10** 0011. It is the same because $A = 1111$. $1 \wedge X = X$

**2.11** 1010. It is the same because $A = 0000$. $0 \vee X = X$

**2.12** 0000.

**2.13** 1111. The strings in the previous exercise did not differ in any position. The strings in this exercise differ in all positions

---

**3.1** 22 bits

**3.2** $2^{20}$

**3.3** $10_{16}$

**3.4** $23_{16}$

**3.5** $8004_{16}$

**3.6** $(45_{16}) = \mathrm{FF}_{16}$

**3.7** $(45_{16}) = 00$

**3.8** $(45_{16}) = \mathrm{AA}_{16}$

---

**4.1** (a) (D0) = $12345687_{16}$ (b) (D0) = $12348765_{16}$ (c) (D0) = $87654321_{16}$

**4.2** (a) (D0) = $01010188_{16}$ (b) (D0) = $01018866_{16}$ (c) (D0) = $88664422_{16}$

**4.3** (a) (D0) = $\mathrm{AAAAAA00}_{16}$ (b) (D0) = $\mathrm{AAAA00AA}_{16}$ (c) (D0) = $\mathrm{00AA00AA}_{16}$

4.4

```
MOVE.B  $21F,$2FA
ADDI.B  #25,$2FA
```

4.5

```
MOVE.B  $1234,$25
SUBI.B  #25,$25
```

4.6

```
ORI.B  #4,$3
```

4.7

```
MOVE.B  $FFF,D0
ADD.B   D0,$ABC
ADD.B   D0,$DEF
```

4.8

```
MOVE.B  ROW,D0
ADD.B   COL,D0
ADDI.B  #1,D0
MOVE.B  D0,MAT
```

4.9

```
ADDI.B  #22,LOC
```

4.10

```
SUBI.B  #NUM,VAR
NEG.B   VAR
```

4.11

```
ADDI.B  #1,NUM1
ADDI.B  #2,NUM2
ADDI.B  #3,NUM3
```

**4.12**

```
        MOVE.B  NUM,D1
        ADD.B   D1,NUM
```

**4.13**

```
        MOVE.B  #7,D0
        MOVE.B  NUM,D1
LOOP    ADD.B   D1,NUM
        SUBI.B  #1,D0
        BNE     LOOP
```

**4.14**

```
        MOVE.B  #0,P
        MOVE.B  M1,D0
        MOVE.B  M2,D1
        CMPI.B  #0,D0
        BEQ     DONE
LOOP    ADD.B   D1,P
        SUBI.B  #1,D0
        BNE     LOOP
DONE
```

**4.15**

```
        CMPI.B  #1,A
        BEQ     THEN
        CMPI.B  #2,A
        BNE     ELSE
THEN    MOVE.B  A,B
        BRA     NEXT
ELSE    MOVE.B  B,A
NEXT    ...
```

**4.16**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BHI     THEN
        MOVE.L  #0,B
        BRA     DONE
THEN    MOVE.L  #0,A
DONE
```

**4.17**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BLS     THEN
        MOVE.L  #0,B
        BRA     DONE
THEN    MOVE.L  #0,A
DONE
```

**4.18**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BGT     THEN
        MOVE.L  #0,B
        BRA     DONE
THEN    MOVE.L  #0,A
DONE
```

**4.19**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BLE     THEN
        MOVE.L  #0,B
        BRA     DONE
THEN    MOVE.L  #0,A
DONE
```

**4.20**

```
        ADD.L   D3,D7
        ADDX.L  D2,D6
        ADDX.L  D1,D5
        ADDX.L  D0,D4
```

**4.21**

```
        SUB.L   D3,D7
        SUBX.L  D2,D6
        SUBX.L  D1,D5
        SUBX.L  D0,D4
```

**4.22**

```
        ROR.L   #5,D0
```

**4.23**

```
        ROL.B   #2,D0
```

**4.24**

```
        ROR   NUM
```

**4.25**

```
        ASR   NUM
```

**4.26**

```
        ASL   NUM
```

**4.27**

```
        MOVEA.L  #$100,A0
        MOVE.B   #2,D0
        MOVE.W   (A0)+,D2
LOOP    ADD.W    (A0)+,D2
        SUBI.B   #1,D0
        BNE      LOOP
        MOVE.W   D2,(A0)
```

**4.28**

```
        MOVEA.L  #$100,A0
        MOVE.B   #N,D0
        MOVE.W   (A0)+,D2
LOOP    ADD.W    (A0)+,D2
        SUBI.B   #1,D0
        BNE      LOOP
        MOVE.W   D2,(A0)
```

**4.29**

```
        MOVEA.L  #$0,A0
        MOVE.B   #N,D0
        MOVE.W   $100(A0),D2
        ADDA.L   #2,A0
LOOP    ADD.W    $100(A0),D2
        ADDA.L   #2,A0
        SUBI.B   #1,D0
        BNE      LOOP
        MOVE.W   D2,$100(A0)
```

**4.30**

```
        MOVEA.L  #$100,A0
        MOVE.B   #N,D0
        MOVE.L   #0,D1
        MOVE.W   0(A0,D1),D2
        ADDI.L   #2,D1
LOOP    ADD.W    0(A0,D1),D2
        ADDI.L   #2,D1
        SUBI.B   #1,D0
        BNE      LOOP
        MOVE.W   D2,0(A0,D1)
```

**4.31**

```
MAX     CMP.L    D0,D1
        BLS      DONE
        MOVE.L   D1,D0
DONE    RTS
```

**4.32**

```
        MOVEA.L  #VEC,A0
        MOVE.L   #N+1,D2
        MOVE.L   #0,D0
LOOP    MOVE.L   (A0)+,D1
        BSR      MAX
        SUBI.L   #1,D2
        BNE      LOOP
```

**4.33**

```
DIV2    ASR.L  D1,D0
        RTS
```

**4.34**

```
        MOVEA.L  #VEC,A0
        MOVE.L   #0,D1
        MOVE.L   #0,D2
LOOP    MOVE.L   (A0)+,D0
        BSR      DIV2
        ADD.L    D0,D2
        ADDI.L   #1,D1
        CMPI.L   #N+1,D1
        BNE      LOOP
```

**4.35** a) DC85 b) D63C 00B6 c) D479 0005 3254

**4.36** a) DF92 b) D41B c) D44A

**4.37** a) 6002 b) 60FC c) 60EE

---

**5.1**

```
ITEM      DS.L     1

RETRIEVE  MOVEA.L  #INBUF,A0
          MOVE.L   FIRST(A0),D0
          MOVE.L   LIST(A0,D0),ITEM   ; ITEM:=LIST[FIRST];
          ADDI.L   #4,FIRST(A0)       ; FIRST:=FIRST+1;
          SUBI.L   #1,COUNT(A0)       ; COUNT:=COUNT-1;
          RTS
```

**5.2**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BLT     THEN
        MOVE.L  #1,A
        BRA     NEXT
THEN    CLR.L   A
NEXT    ...
```

**5.3**

```
        MOVE.L  A,D0
        CMP.L   B,D0
        BCS     THEN
        MOVE.L  #1,A
        BRA     NEXT
THEN    CLR.L   A
NEXT    ...
```

**5.4**

```
        CMPI.L  #5,A
        BLT     THEN
        CMPI.L  #10,A
        BGT     THEN
        CLR.L   A
        BRA     NEXT
THEN    MOVE.L  #1,A
NEXT    ...
```

**5.5**

```
        MOVE.L  #1,D0    ; I:=1
        BRA     TEST     ; goto TEST
FOR     ADDI.L  #1,J     ; J:=J+1
        ADDI.L  #1,D0    ; I:=I+1
TEST    CMPI.L  #10,D0   ; if I <= 10 then
        BLS     FOR      ; goto FOR
NEXT    ...
```

**5.6**

```
        MOVE.L  #0,I     ; I:=0;
        BRA     TEST     ; goto TEST
WHILE   ADDI.L  #1,I     ; I:=I+1
TEST    CMPI.L  #10,I    ; if I-10 < 0 then
        BCS     WHILE    ; goto WHILE
NEXT    ...
```

**5.7**

```
        MOVE.L  #0,I     ; I:=0;
REPEAT  ADDI.L  #1,I     ; I:=I+1;
        CMPI.L  #20,I    ; if I <= 20
        BLE     REPEAT   ; then goto REPEAT
NEXT
```

**5.8**

```
CONVERT  CMPI.B  #$61,D0
         BCS     OUT
         CMPI.B  #$7A,D0
         BHI     OUT
         SUBI.B  #$20,D0
OUT      RTS
```

**5.9**

```
NUL    EQU     0

CSTR   CMPI.B  #NUL,(A0)
       BEQ     FINE
       MOVE.B  (A0),D0
       BSR     CONVERT
       MOVE.B  D0,(A0)+
       BRA     CSTR
FINE   RTS
```

**5.10**

```
N      EQU      5
TAB    DC.W     0,1,2,3,4

ADDF   MOVEA.L  #TAB,A0
       MOVE.W   #0,D0
       MOVE.L   #N,D1
ADDL   ADD.W    (A0)+,D0
       SUBI.L   #1,D1
       BNE      ADDL
       RTS
```

**5.11** Example solution:

```
AOLD := 1;
A:=1;
repeat
  TEMP:=A;
  A:=A+AOLD;
  PUTINT(A);
  AOLD:=TEMP;
until A > 65535;
```

```
AOLD    DS.L    1
A       DS.L    1
TEMP    DS.L    1

FIB     MOVE.W  #1,AOLD     ; AOLD:=1
        MOVE.W  #1,A        ; A:=1
REP     MOVE.W  A,TEMP      ; TEMP:=A
        MOVE.W  AOLD,D0
        ADD.W   A,D0        ; D0:=A+AOLD
        BSR     PUTINT      ; PUTINT(A)
        MOVE.W  D0,A
        MOVE.W  TEMP,AOLD   ; AOLD:=TEMP
        CMPI.W  #65535,A
        BLS     REP
```

### 5.12

```
;===============================================================================
; NAME:          PRSEX
; DESCRIPTION:   Prints all persons with a specific sex
; INPUT:         None
; OUTPUT:        None
; REGISTERS:     A0,A1,D0
;===============================================================================

PRSTR2          DC.B    'Input sex',$0D,$0A,0
                EVEN
PSEX            DS.L    1

PRSEX           MOVEA.L #PRSTR2,A0
                BSR     PRSTR           ; PRSTR('Input sex');
                MOVEA.L #PSEX,A0
                BSR     READINT         ; SEX:=READINT;
                MOVEA.L #DATABASE,A1    ; REC:= "First record";
                BRA     WTEST2

WLOOP2          MOVE.L  PSEX,D0
                CMP.L   MALE(A1),D0     ; if PSEX = REC.MALE then
                BNE     CONT2
                LEA     FNAME(A1),A0
                BSR     PRSTR           ; PRSTR(REC.FNAME);
                LEA     LNAME(A1),A0
                BSR     PRSTR           ; PRSTR(REC.LNAME);
CONT2           MOVEA.L NEXT(A1),A1     ; REC:=REC.NEXT;
WTEST2          CMPA.L  LAST,A1         ; if REC - LAST <> 0 then
                BNE     WLOOP2          ; goto WLOOP2
                RTS                     ; end;
```

---

### 6.1

```
MOVE.B  $FFF100,D0
ASL.B   #2,D0
MOVE.B  D0,$FFF102
```

**6.2**

```
IN1     EQU     $FFF100
IN2     EQU     $FFF102
OUT     EQU     $FFF104

LOOP    MOVE.B  IN1,D0
        AND.B   IN2,D0
        MOVE.B  D0,OUT
        BRA     LOOP
```

**6.3**

```
INDEV   EQU     $FFF000
STATUS  EQU     $FFF008
MEM     EQU     $9000

START   MOVEA.L #INDEV,A0
        MOVEA.L #MEM,A1
        MOVE.L  #0,D0
LOOP    BTST    D0,STATUS
        BNE     COPY
        ADDA.L  #1,A0
        ADDA.L  #1,A1
TEST    ADDI.L  #1,D0
        CMPI.L  #8,D0
        BNE     LOOP
        BRA     START

COPY    MOVE.B  (A0)+,(A1)+
        BRA     TEST
```

**6.4** We use D0 to pass N to the subroutine and D1 to return the function value NSUM.

```
NSUM    MOVE.L  D0,D1       ; NSUM:=N;
        CMPI.L  #1,D0       ; if N=1 then
        BEQ     NEND        ; return
        MOVE.L  D0,-(SP)    ; Push D0
        SUBI.L  #1,D0
        BSR     NSUM        ; D1:=NSUM(N-1);
        MOVE.L  (SP)+,D0    ; Pop D0
        ADD.L   D0,D1       ; NSUM:=N+NSUM(N-1);
NEND    RTS
```

**6.5** (a) SP1 shows the content of the stack and the stack pointer the first time SUBI.L is executed, SP2 the second time etc.

```
SP        Content     Address
==========================
SP3-->    0000        8FE8
          0001        8FEA
          0000        8FEC
          801C        8FEE
SP2-->    0000        8FF0
          0002        8FF2
          0000        8FF4
          801C        8FF6
SP1-->    0000        8FF8
          0003        8FFA
          0000        8FFC
          8004        8FFE
```

(b) (D1) = 3.

**6.6**

```
PREGS   MOVEA.L  (SP)+,A0  ; Get address to the word
        ROR      (A0)      ; Shift one step to the right
        BCC      REG1      ; Check least significant bit
        MOVE.L   D0,-(SP)  ; If set, push D0
REG1    ROR      (A0)
        BCC      REG2
        MOVE.L   D1,-(SP)
REG2    ROR      (A0)
        BCC      REG3
        MOVE.L   D2,-(SP)
REG3    ROR      (A0)
        BCC      REG4
        MOVE.L   D3,-(SP)
REG4    ...
        ...
        ...
REG7    ROR      (A0)
        BCC      FINE
        MOVE.L   D7,-(SP)
FINE    ADDA.L   #2,A0     ; Modify return address
        MOVE.L   A0,-(SP)
        RTS
```

**6.7**

```
ENTER     MOVE.L  D0,-(A0)
          RTS
ADDSTACK  ADD.L   (A0)+,D0
          RTS
SUBSTACK  SUB.L   (A0)+,D0
          RTS
POPSTACK  MOVE.L  (A0)+,D0
          RTS
```

**6.8**

```
          ENTER          ENTER        ADDSTACK       SUBSTACK

7FF8                --> 1000           1000           1000
7FFA                    0006           0006           0006
7FFC  --> 1000          1000       --> 1000           1000
7FFE      0005          0005           0005           0005
8000                                                -->
          (A0)=7FFC     (A0)=7FF8      (A0)=7FFC      (A0)=8000
```

**6.9** The interrupt service routine:

```
SWITCH  MOVE.L  #0,TICK
        MOVE.L  #0,SEC
        MOVE.L  #0,MIN
        MOVE.L  #0,HOUR
        RTE
```

We need to modify the main program as follows:

```
MAIN    ...
        MOVE.L  #SWITCH,$68   ; Add this line...
        MOVE.L  #TIME,$74
        MOVE    #$2100,SR     ; ...and modify this one.
```

**6.10**

```
NUL       EQU      0

STRING    EQU      $6000
DISPLAY   EQU      $5700

POSITION  DS.L     1

MCHAR     MOVE.B   #0,D0           ; I:=0
          MOVE.L   POSITION,D1
          MOVEA.L  #DISPLAY,A0
          MOVEA.L  #STRING,A1
MLOOP     CMPI.B   #14,D0          ; for I:= 0 to 14 do
          BHI      NEXT1
          CMPI.B   #NUL,0(A1,D1)   ; if STRING[POINTER] = NUL
          BNE      NEXT
          MOVE.L   #0,D1           ; then POINTER:=0
NEXT      MOVE.B   0(A1,D1),(A0)+  ; DISPLAY[I]:=
                                   ; STRING[POINTER]
          ADDI.L   #1,D1           ; POINTER:=POINTER+1
          ADDI.L   #1,D0
          BRA      MLOOP
NEXT1     ADDI.L   #1,POSITION
          CMPI.L   #14,POSITION
          BLT      GOBACK
          MOVE.L   #0,POSITION
GOBACK    RTS
```

**6.11**

```
COUNT     DS.B     1

          EVEN

PRINT     MOVEM.L  D0-D1/A0-A1,-(SP)
          ADDI.B   #1,COUNT
          CMPI.B   #10,COUNT
          BNE      POPREG
          MOVE.B   #0,COUNT
          BSR      MCHAR
POPREG    MOVEM.L  (SP)+,D0-D1/A0-A1
          RTE
```

**6.12**

```
MAIN    MOVE.L  #0,POSITION   ; POSITION:=0
        MOVE.B  #0,COUNT      ; COUNT:=0
        MOVE.L  #PRINT,$74    ; Exception vector
        MOVE    #$2400,SR
        ....
```

---

**7.1**

```
PGCR    EQU     $FFF000
PACR    EQU     $FFF006
PADDR   EQU     $FFF002

INIT    MOVE.B  #%00000000,PGCR   ; Mode 0
        MOVE.B  #%10000000,PACR   ; Bit-I/O submode
        MOVE.B  #%11110000,PADDR  ; Bits 0-3 inputs and
                                  ; bits 4-7 outputs
```

**7.2**

```
PGCR    EQU     $FFF000
PBCR    EQU     $FFF007
PBDDR   EQU     $FFF003

INIT    MOVE.B  #%00000000,PGCR   ; Mode 0
        MOVE.B  #%10000000,PBCR   ; Bit-I/O submode
        MOVE.B  #%11110000,PBDDR  ; Bits 0-3 inputs and
                                  ; bits 4-7 outputs
```

**7.3**

```
PGCR    EQU     $FFF000
PACR    EQU     $FFF006
PADDR   EQU     $FFF002

INIT    MOVE.B  #%00010010,PGCR   ; Mode 0, H1 asserts to 0
                                  ; but H2 asserts to 1
        MOVE.B  #%01110000,PACR   ; Output submode
        MOVE.B  #%11111111,PADDR  ; Bits 0-7 outputs
```

7.4

```
PGCR   EQU     $FFF000
PBCR   EQU     $FFF007
PBDDR  EQU     $FFF003

INIT   MOVE.B  #%00101100,PGCR   ; Mode 0, H3 and H4
                                 ; assert to 1
       MOVE.B  #%01110000,PBCR   ; Output submode
       MOVE.B  #%11111111,PBDDR  ; Bits 0-7 outputs
```

7.5

```
PSR    EQU     $FFF00D
PADR   EQU     $FFF008

CHRIN  BTST    #0,PSR    ; if H1 is not asserted
       BNE     CHRIN     ; goto CHRIN
       MOVE.B  PADR,D0   ; Read from Port A
       RTS
```

7.6

```
CR     EQU     $FFF000

INIT   MOVE.B  #%00000011,CR  ; Master reset
       MOVE.B  #%00001001,CR  ; 7 bits, even parity,
                              ; 1 stop bit, 16 samples
```

7.7

```
CR     EQU     $FFF000

INIT   MOVE.B  #%00000011,CR  ; Master reset
       MOVE.B  #%00000010,CR  ; 7 bits, even parity,
                              ; 2 stop bits, 64 samples
```

7.8

```
CR     EQU     $FFF000

INIT   MOVE.B  #%00000011,CR  ; Master reset
       MOVE.B  #%00010001,CR  ; 8 bits, no parity,
                              ; 2 stop bits, 16 samples
```

**7.9**

```
SR      EQU     $FFF000  ; Status Register
DATA    EQU     $FFF001  ; Data Register

POLL    BTST    #0,SR    ; Test RDRF in the SR
        BEQ     POLL     ; If not asserted,
                         ; try again
        BTST    #6,SR    ; Parity error?
        BNE     PERROR   ; Yes, branch to PERROR
        BTST    #5,SR    ; Overrun error?
        BNE     OERROR   ; Yes, branch to OERROR
        BTST    #4,SR    ; Framing error?
        BNE     FERROR   ; Yes, branch to FERROR
        MOVE.B  DATA,D0  ; Read from the ACIA
        MOVE.B  #0,D1    ; No error
        RTS
PERROR  MOVE.B  #1,D1    ; Parity error
        RTS
OERROR  MOVE.B  #2,D1    ; Overrun error
        RTS
FERROR  MOVE.B  #3,D1    ; Framing error
        RTS
```

Appendix B

# 68000 Instruction Set

This appendix provides detailed information on the use of most instructions available for the M68000. For additional information, please refer to *M68000 User's Manual* issued by Motorola.

For each instruction, information is provided about the syntax, including the available addressing modes and the operand size; a description of the operation that is carried out; and how the condition codes are affected. The example below shows the information that is available for each instruction.

| **ADD** | Add |
|---|---|
| **Syntax:** | `ADD.S` $a_1, D_i$ or `ADD.S` $D_i, a_3$ |
| **Operand size:** | S = (B,W,L) |
| **Operation:** | $(a_1) + (D_i) \rightarrow D_i$ or $(D_i) + (a_3) \rightarrow a_3$ |
| **Condition codes:** | X N Z V C<br>* * * * * |

The syntax specifies the name of the instruction and the addressing modes available by means of either a register name (e.g. $D_i$ or $A_i$) or a set of addressing modes (e.g. $a_1$) that are applicable to the source or destination operand of the instruction. The operation is described either using the notation in Chapter 3, or verbally. Moreover, it is shown how the instruction affects the condition codes using special symbols. In the following, we will present the shorthand notations used in the instruction summaries.

In Table B.1, we show all addressing modes treated in the textbook and an abbreviation for each of them. Since only a subset of all available addressing modes are applicable to a specific operand in an instruction, we define nine addressing-mode subsets $(a_1, a_2, \ldots, a_9)$ in Table B.2 using the abbreviations from Table B.1. An 'X' in a specific position means that the corresponding addressing mode is available. For example, $a_1$ means that all addressing modes can be used to specify

**Table B.1** All addressing modes introduced in the text.

| Abbreviation | Addressing mode | Example |
|---|---|---|
| DRD | Data register direct | `MOVE.B D1,D0` |
| ARD | Address register direct | `MOVE.L A0,D0` |
| ABS | Absolute | `MOVE.B 1,D0` |
| IMM | Immediate | `MOVE.B #1,D0` |
| IND | Indirect | `MOVE.B (A0),D0` |
| IDI | Indirect with postincrement | `MOVE.B (A0)+,D0` |
| IDD | Indirect with predecrement | `MOVE.B -(A0),D0` |
| AID | Address register indirect with displacement | `MOVE.B 10(A0),D0` |
| AII | Address register indirect with index | `MOVE.B 10(A0,D1),D0` |
| PID | Program counter indirect with displacement | `MOVE.B 10(PC),D0` |
| PII | Program counter indirect with index | `MOVE.B 10(PC,A0),D0` |

**Table B.2** Addressing-mode subsets as used by various instructions.

| | DRD | ARD | ABS | IMM | IND | IDI | IDD | AID | AII | PID | PII |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_1$ | X | X | X | X | X | X | X | X | X | X | X |
| $a_2$ | X | | X | | X | X | X | X | X | | |
| $a_3$ | | | X | | X | X | X | X | X | | |
| $a_4$ | X | | X | X | X | X | X | X | X | X | X |
| $a_5$ | | | X | | X | | X | X | X | | |
| $a_6$ | | | X | | X | X | | X | X | X | X |
| $a_7$ | | | X | | X | | | X | X | X | X |
| $a_8$ | X | X | X | | X | X | X | X | X | | |
| $a_9$ | X | | X | | X | X | X | X | X | X | X |

**Table B.3** Notation for how the condition codes are affected by each instruction.

| Description | Notation |
|---|---|
| Common case | * |
| Not affected | – |
| The flag is set | 1 |
| The flag is cleared | 0 |
| The flag is undefined | U |
| Special meaning | ! |

**Table B.4** The common case for how the condition codes are affected.

| Flag | Condition |
|---|---|
| X | Set if the C-flag is set. Cleared otherwise. |
| N | Set when the result is negative. Cleared otherwise. |
| Z | Set when the result is zero. Cleared otherwise. |
| V | Set when two's complement operation results in overflow. Cleared otherwise. |
| C | Set when carry/borrow is generated. Cleared otherwise. |

an operand whereas $a_4$ means that all addressing modes are available except for ARD (address register direct).

To specify how the condition codes are affected by each instruction, we use the symbols found in Table B.3. The common case (denoted by an asterisk '*') means that the flags are set in a way that conforms to their meaning. We specify precisely what the common case means in Table B.4. For some instructions, a flag can be set according to special rules. We denote this case by an exclamation mark and will describe the special setting of the flag in the instruction summary.

| **ADD** | **Add** |
|---|---|
| **Syntax:** | `ADD.S` $a_1, D_i$ or `ADD.S` $D_i, a_3$ |
| **Operand size:** | S = (B,W,L) |
| **Operation:** | $(a_1) + (D_i) \rightarrow D_i$ or $(D_i) + (a_3) \rightarrow a_3$ |
| **Condition codes:** | X N Z V C<br>* * * * * |

## ADDA — Add Address

**Syntax:** `ADDA.S` $a_1$`,`$A_i$

**Operand size:** S = (W,L)

**Operation:** $(a_1) + (A_i) \rightarrow A_i$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## ADDI — Add Immediate

**Syntax:** `ADDI.S #`$n$`,`$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $n + (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## ADDQ — Add Quick

**Syntax:** `ADDQ.S #`$n$`,`$a_8$

**Operand size:** S = (B,W,L)

**Operation:** $n + (a_8) \rightarrow a_8$, where $1 \leq n \leq 8$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** This instruction occupies one word only.

## ADDX — Add Extended

**Syntax:** `ADDX.S` $D_i$`,`$D_j$ or

`ADDX.S` $-(A_i), -(A_j)$

**Operand size:** S = (B,W,L)

**Operation:** $(D_i) + (D_j) + (X) \rightarrow D_j$ or
$(A_i) - k \rightarrow A_i$; $(A_j) - k \rightarrow A_j$;
$((A_i)) + ((A_j)) + (X) \rightarrow (A_j)$
where $k$ depends on S.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## AND — And

**Syntax:** `AND.S` $a_4$`,`$D_i$ or `AND.S` $D_i$`,`$a_3$

**Operand size:** S = (B,W,L)

**Operation:** $(a_4) \wedge (D_i) \rightarrow D_i$ or $(D_i) \wedge (a_3) \rightarrow a_3$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## ANDI — And Immediate

**Syntax:** `ANDI.S #`$n$`,`$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $n \wedge (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## ANDI to CCR — And Immediate to Condition Codes

**Syntax:** `ANDI #`$n$`,CCR`

**Operand size:** Byte

**Operation:** $n \wedge$ (CCR) $\rightarrow$ CCR

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## ANDI to SR — And Immediate to Status Register

**Syntax:** `ANDI #`$n$`,SR`

**Operand size:** Word

**Operation:** $n \wedge$ (SR) $\rightarrow$ SR

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** Privileged instruction

## ASL — Arithmetic Shift Left

**Syntax:**

`ASL.S` $D_i$,$D_j$ or

`ASL.S` `#`$n$,$D_j$ or

`ASL` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Shifts the bits in the destination operand to the left the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Zeros are shifted into the least significant bits.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | ! | ! |

V Set iff the most significant bit is changed at any time during the shift operation.

C Set according to the last bit shifted out.

## ASR — Arithmetic Shift Right

**Syntax:**

`ASR.S` $D_i$,$D_j$ or

`ASR.S` `#`$n$,$D_j$ or

`ASR` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Shifts the bits in the destination operand to the right the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Zeros are shifted into the most significant bits.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | 0 | ! |

C Set according to the last bit shifted out.

## B*cc* — Branch Conditionally

**Syntax:** B*cc* *label*

**Operation:** If Condition *cc* then *label* → PC. Condition *cc* specifies one of the following conditions:

```
CC  (C)'
CS  (C)
NE  (Z)'
EQ  (Z)
VC  (V)'
VS  (V)
PL  (N)'
MI  (N)
LS  (C) ∨ (Z)
HI  (C)' ∧ (Z)'
LT  ((N) ∧ (V)') ∨ ((N)' ∧ (V))
LE  (Z) ∧ (((N) ∧ (V)') ∨ ((N)' ∧ (V)))
GT  (Z)' ∧ (((N) ∧ (V)) ∨ ((N)' ∧ (V)'))
GE  ((N) ∧ (V)) ∨ ((N)' ∧ (V)')
```

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## BCHG — Test a Bit and Change

**Syntax:** BCHG $D_i$,$a_2$ or

BCHG #$n$,$a_2$

**Operand size:** When the destination is a data register, the operand size is Long word; otherwise it is Byte.

**Operation:** Tests the bit in the destination operand denoted by the source operand and sets the Z-flag accordingly. The tested bit is then inverted.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | ! | - | - |

Z See operation above.

## BCLR — Test a Bit and Clear

**Syntax:** `BCLR` $D_i, a_2$ or

`BCLR` `#`$n, a_2$

**Operand size:** When the destination is a data register, the operand size is Long word. Otherwise it is Byte.

**Operation:** Tests the bit in the destination operand denoted by the source operand and sets the Z-flag accordingly. The tested bit is then cleared.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | ! | - | - |

Z See operation above.

## BRA — Branch Unconditionally

**Syntax:** `BRA` *label*

**Operation:** *label* $\rightarrow$ PC

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## BSET — Test a Bit and Set

**Syntax:** `BSET` $D_i, a_2$ or

`BSET` `#`$n, a_2$

**Operand size:** When the destination is a data register, the operand size is Long word. Otherwise it is Byte.

**Operation:** Tests the bit in the destination operand denoted by the source operand and sets the Z-flag accordingly. The tested bit is then set.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | ! | - | - |

Z See operation above.

## BSR — Branch to Subroutine

**Syntax:** `BSR` *label*

**Operation:** (SP) − 4 → SP;
(PC) → (SP);
*label* → PC

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## BTST — Test a Bit

**Syntax:** `BTST` $D_i$`,`$a_2$ or

`BTST #`$n$`,`$a_9$

**Operand size:** When the destination is a data register, the operand size is Long word. Otherwise it is Byte.

**Operation:** Tests the bit in the destination operand denoted by the source operand and sets the Z-flag accordingly.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | ! | - | - |

Z See operation above.

## CHK — Check Register Against Bounds

**Syntax:** `CHK` $a_4$`,`$D_i$

**Operand size:** Word

**Operation:** if ($D_i$)< 0 or ($D_i$) > ($a_4$), a trap (vector number 6) occurs.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | ! | U | U | U |

N Set if ($D_i$)< 0; cleared if ($D_i$) > ($a_4$). Undefined otherwise.

## CLR — Clear an Operand

**Syntax:** `CLR.S` $a_2$

**Operand size:** S = (B,W,L)

**Operation:** 0 → $a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | 0 | 1 | 0 | 0 |

## CMP — Compare

**Syntax:** `CMP.S` $a_1$`,`$D_i$

**Operand size:** S = (B,W,L)

**Operation:** $(D_i)-(a_1)$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | * | * |

## CMPA — Compare Address

**Syntax:** `CMPA.S` $a_1$`,`$A_i$

**Operand size:** S = (W,L)

**Operation:** $(A_i)-(a_1)$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | * | * |

## CMPI — Compare Immediate

**Syntax:** `CMPI.S #`$n$`,`$a_9$

**Operand size:** S = (B,W,L)

**Operation:** $(a_9)-n$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | * | * |

## CMPM — Compare Memory

**Syntax:** `CMPM.S (`$A_i$`)+,(`$A_j$`)+`

**Operand size:** S = (B,W,L)

**Operation:** $(A_j)-(A_j)$; $(A_i)+k \rightarrow A_i$; $(A_j)+k \rightarrow A_j$
where $k$ depends on the operand size

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | * | * |

## DB*cc* — Test Condition, Decrement, and Branch

**Syntax:** `DBcc` $D_i$,*label*

**Operand Size:** Word

**Operation:** If Condition *cc* nothing is done. Otherwise, $(D_i)-1 \rightarrow D_i$; if $(D_i) \neq -1$ then *label* $\rightarrow$ PC.
Condition *cc* is one of those listed under the `Bcc` instruction and in addition the following:

`F` always FALSE
`T` always TRUE

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## DIVS — Signed Divide

**Syntax:** `DIVS.W` $a_4$,$D_i$

**Operand Size:** Word

**Operation:** Divides the signed destination operand (32 bits) by the signed source operand (16 bits). The result is a signed quotient in the least significant 16 bits and the remainder in the most significant 16 bits. The sign of the remainder is the same as the sign of the dividend. The instruction results in a trap if the divisor is zero.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | ! | ! | ! | 0 |

N Common case but undefined when overflow occurs.
Z Common case but undefined when overflow occurs.
V Common case but undefined if divide by zero occurs.

## DIVU — Unsigned Divide

**Syntax:** `DIVU.W` $a_4$,$D_i$

**Operand Size:** Word

**Operation:** Divides the unsigned destination operand (32 bits) by the unsigned source operand (16 bits). The result is an unsigned quotient in the least significant 16 bits and the remainder in the most significant 16 bits. The instruction results in a trap if the divisor is zero.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | ! | ! | ! | 0 |

N Common case but undefined when overflow occurs.
Z Common case but undefined when overflow occurs.
V Common case but undefined if divide by zero occurs.

## EOR — Exclusive OR

**Syntax:** `EOR.S` $D_i$,$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $(D_i) \oplus (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## EORI — Exclusive OR Immediate

**Syntax:** `EORI.S` #$n$,$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $n \oplus (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## EORI to CCR

**Exclusive OR Immediate to Condition Codes**

**Syntax:** `EORI #`*n*`,CCR`

**Operand size:** Byte

**Operation:** $n \oplus (\text{CCR}) \rightarrow \text{CCR}$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## EORI to SR

**Exclusive OR Immediate to Status Register**

**Syntax:** `EORI #`*n*`,SR`

**Operand size:** Word

**Operation:** $n \oplus (\text{SR}) \rightarrow \text{SR}$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** Privileged instruction

## EXG

**Exchange Registers**

**Syntax:**
`EXG` $D_i, D_j$ or
`EXG` $A_i, A_j$ or
`EXG` $D_i, A_j$ or
`EXG` $A_i, D_j$

**Operand size:** Long word

**Operation:** Exchanges the contents of the source and the destination operands.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## EXT

**Sign Extend**

**Syntax:** `EXT.S` $D_i$

**Operand size:** S=(W,L)

**Operation:** Extends the sign bit of the operand. If the operand size is Word, the least significant 8 bits are sign extended to a Word, and if operand size is Long word, the least significant 16 bits are sign extended to a Long word.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## JMP — Jump

**Syntax:** JMP $a_7$

**Operation:** $a_7 \rightarrow$ PC

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## JSR — Jump to Subroutine

**Syntax:** JSR $a_7$

**Operation:** (SP) $-$ 4 $\rightarrow$ SP;
(PC) $\rightarrow$ (SP);
$a_7 \rightarrow$ PC

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## LEA — Load Effective Address

**Syntax:** LEA $a_7, A_i$

**Operand size:** Long word

**Operation:** $a_7 \rightarrow A_i$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## LSL — Logical Shift Left

**Syntax:**

`LSL.S` $D_i$,$D_j$ or

`LSL.S` `#`$n$,$D_j$ or

`LSL` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Shifts the bits in the destination operand to the left the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Zeros are shifted into the least significant bits.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | 0 | ! |

C Set according to the last bit shifted out.

## LSR — Logical Shift Right

**Syntax:**

`LSR.S` $D_i$,$D_j$ or

`LSR.S` `#`$n$,$D_j$ or

`LSR` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Shifts the bits in the destination operand to the right the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Zeros are shifted into the most significant bits.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | 0 | ! |

C Set according to the last bit shifted out.

## MOVE — Move Data from Source to Destination

**Syntax:** `MOVE.S` $a_1$,$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $(a_1) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## MOVEA — Move Address

**Syntax:** `MOVEA.S` $a_1$,$A_i$

**Operand size:** S = (W,L)

**Operation:** $(a_1) \rightarrow A_i$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## MOVE — Move to Condition Code Register

**Syntax:** `MOVE` $a_4$,`CCR`

**Operand size:** Word

**Operation:** $(a_4) \rightarrow$ CCR

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## MOVE — Move to Status Register

**Syntax:** `MOVE` $a_4$,`SR`

**Operand size:** Word

**Operation:** $(a_4) \rightarrow$ SR

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** Privileged instruction

## MOVE — Move User Stack Pointer

**Syntax:** `MOVE` $A_i$`,USP` or `MOVE USP,`$A_i$

**Operand size:** Long word

**Operation:** $(A_i) \rightarrow$ USP or (USP) $\rightarrow A_i$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

**Remark:** Privileged instruction

## MOVEM — Move Multiple Registers

**Syntax:** `MOVEM.S` *register list*,$a_5$ or
`MOVEM.S` $a_6$,*register list*

**Operand size:** S=(W,L)

**Operation:** Moves the contents of the selected registers to (first form) or from (second form) consecutive memory locations.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## MOVEQ — Move Quick

**Syntax:** `MOVEQ #`$n$`,`$D_i$

**Operand size:** Long word

**Operation:** $n \rightarrow D_i$, where $n$ is an 8-bit two's complement number which is sign extended in the destination.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

**Remark:** This instruction occupies one word only.

## MULS — Signed Multiply

**Syntax:** `MULS.W` $a_4$,$D_i$

**Operand Size:** Word

**Operation:** Multplies two signed 16-bit operands yielding a signed 32-bit result.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## MULU — Unsigned Multiply

**Syntax:** `MULU.W` $a_4$,$D_i$

**Operand Size:** Word

**Operation:** Multplies two unsigned 16-bit operands yielding an unsigned 32-bit result.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## NEG — Negate

**Syntax:** `NEG.S` $a_2$

**Operand size:** S = (B,W,L)

**Operation:** $0 - (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## NEGX — Negate with Extend

**Syntax:** `NEGX.S` $a_2$

**Operand size:** S = (B,W,L)

**Operation:** $0 - (a_2) - (X) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | ! | * | * |

Z Cleared if the result is non-zero.
Unaffected otherwise.

## NOP — No Operation

**Syntax:** `NOP`

**Operation:** Performs no operation.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## NOT — Logical Inverse

**Syntax:** `NOT.S` $a_2$

**Operand size:** S = (B,W,L)

**Operation:** Inverts all bits in the destination.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## OR — Inclusive OR

**Syntax:** `OR.S` $a_4, D_i$ or `OR.S` $D_i, a_5$

**Operand size:** S = (B,W,L)

**Operation:** $(a_4) \vee (D_i) \rightarrow D_i$ or $(D_i) \vee (a_5) \rightarrow a_5$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## ORI — Inclusive OR Immediate

**Syntax:** `ORI.S` #$n, a_2$

**Operand size:** S = (B,W,L)

**Operation:** $n \vee (a_2) \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## ORI to CCR — Inclusive OR Immediate to Condition Codes

**Syntax:** `ORI` #$n$,`CCR`

**Operand size:** Byte

**Operation:** $n \vee (\text{CCR}) \rightarrow \text{CCR}$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## ORI to SR — OR Immediate to Status Register

**Syntax:** `ORI` #$n$,`SR`

**Operand size:** Word

**Operation:** $n \vee (\text{SR}) \rightarrow \text{SR}$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** Privileged instruction

## PEA — Push Effective Address

**Syntax:** `PEA` $a_7$

**Operand size:** Long word.

**Operation:** (SP)−4 → SP; $a_7$ → (SP)

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## ROL — Rotate Left

**Syntax:** `ROL.S` $D_i, D_j$ or

`ROL.S` `#`$n, D_j$ or

`ROL` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Rotates the bits in the destination operand to the left the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Bits shifted out from the most significant bit are shifted into the least significant bit.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | ! |

C Set according to the last bit shifted out.

## ROR — Rotate Right

**Syntax:**

`ROR.S` $D_i, D_j$ or

`ROR.S` `#`$n, D_j$ or

`ROR` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Rotates the bits in the destination operand to the right the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Bits shifted out from the least significant bit are shifted into the most significant bit.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | ! |

C Set according to the last bit shifted out.

## ROXL — Rotate Left with Extend

**Syntax:**

`ROXL.S` $D_i, D_j$ or

`ROXL.S` `#`$n, D_j$ or

`ROXL` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Rotates the bits in the destination operand to the left the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Bits shifted out from the most significant bit are shifted into the X-flag and the X-flag is shifted into the least significant bit.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | 0 | ! |

C Set according to the last bit shifted out.

## ROXR — Rotate Right with Extend

**Syntax:**

`ROXR.S` $D_i, D_j$ or

`ROXR.S` `#`$n, D_j$ or

`ROXR` $a_3$

**Operand size:** S=(B,W,L). The last form assumes Word.

**Operation:** Rotates the bits in the destination operand to the right the number of steps denoted by the source operand. If the source operand is a data register, the shift count is $(D_i)$ mod 64. If the source operand is a constant, the shift count is $n = [1, 8]$, and if the destination is a memory word (last form), the shift count is one. Bits shifted out from the least significant bit are shifted into the X-flag and the X-flag is shifted into the most significant bit.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | 0 | ! |

C Set according to the last bit shifted out.

## RTE — Return from Exception

**Syntax:** `RTE`

**Operation:** ((SP)) → SR; (SP) +2 → SP;
((SP)) → PC; (SP) +4 → SP

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** Privileged instruction

## RTR — Return and Restore Condition Codes

**Syntax:** `RTR`

**Operation:** ((SP)) → CCR; (SP) +2 → SP;
((SP)) → PC; (SP) +4 → SP

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## RTS — Return from Subroutine

**Syntax:** `RTE`

**Operation:** ((SP)) → PC; (SP) +4 → SP

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## S*cc* — Set According to Condition

**Syntax:** `S`*cc* $a_2$

**Operand Size:** Byte

**Operation:** If Condition *cc* then $11111111_2 \rightarrow a_2$
else $0 \rightarrow a_2$
Condition *cc* is one of those listed under the `DB`*cc* instruction.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## STOP — Load Status Register and Stop

**Syntax:** `STOP #`*n*

**Operation:** $n \rightarrow$ SR; execution stops. An exception resumes execution.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## SUB — Subtract

**Syntax:** `SUB.S` $a_1$`,`$D_i$ or `SUB.S` $D_i$`,`$a_3$

**Operand size:** S = (B,W,L)

**Operation:** $(D_i) - (a_1) \rightarrow D_i$ or $(a_3) - (D_i) \rightarrow a_3$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## SUBA — Subtract Address

**Syntax:** `SUBA.S` $a_1$`,`$A_i$

**Operand size:** S = (W,L)

**Operation:** $(A_i) - (a_1) \rightarrow A_i$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## SUBI — Sub Immediate

**Syntax:** `SUBI.S #`$n$`,`$a_2$

**Operand size:** S = (B,W,L)

**Operation:** $(a_2) - n \rightarrow a_2$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## SUBQ — Sub Quick

**Syntax:** `SUBQ.S #`$n$`,`$a_8$

**Operand size:** S = (B,W,L)

**Operation:** $(a_8) - n \rightarrow a_8$, where $1 \leq n \leq 8$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

**Remark:** This instruction occupies one word only.

## SUBX — Subtract Extended

**Syntax:** `SUBX.S` $D_i, D_j$ or

`SUBX.S` $-(A_i), -(A_j)$

**Operand size:** S = (B,W,L)

**Operation:** $(D_j) - (D_i) - (X) \rightarrow D_j$ or

$(A_i) - k \rightarrow A_i$; $(A_j) - k \rightarrow A_j$; $((A_j)) - ((A_i)) - (X) \rightarrow (A_j)$

where $k$ depends on the operand size

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| * | * | * | * | * |

## SWAP — Swap Register Halves

**Syntax:** `SWAP` $D_i$

**Operand size:** Word

**Operation:** Exchanges the contents of the 16 most significant bits and the 16 least significant bits.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

## TRAP — Trap

**Syntax:** `TRAP #`$n$

**Operation:** Causes a trap to exception with vector number $n$, where $n = [0, 15]$.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## TRAPV — Trap on overflow

**Syntax:** `TRAPV`

**Operation:** If (V)=1 then cause a trap to exception with vector number 7.

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | - | - | - | - |

## TST — Test an Operand

**Syntax:** `TST.S` $a_9$

**Operand size:** S = (B,W,L)

**Operation:** $(a_9) - 0$

**Condition codes:**

| X | N | Z | V | C |
|---|---|---|---|---|
| - | * | * | 0 | 0 |

# ASCII Table

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **0** | NUL | DLE | SP | 0 | @ | P | ‘ | p |
| **1** | SOH | DC1 | ! | 1 | A | Q | a | q |
| **2** | STX | DC2 | " | 2 | B | R | b | r |
| **3** | ETX | DC3 | # | 3 | C | S | c | s |
| **4** | EOT | DC4 | $ | 4 | D | T | d | t |
| **5** | ENQ | NAK | % | 5 | E | U | e | u |
| **6** | ACK | SYN | & | 6 | F | V | f | v |
| **7** | BEL | ETB | ’ | 7 | G | W | g | w |
| **8** | BS | CAN | ( | 8 | H | X | h | x |
| **9** | HT | EM | ) | 9 | I | Y | i | y |
| **A** | LF | SUB | * | : | J | Z | j | z |
| **B** | VT | ESC | + | ; | K | [ | k | { |
| **C** | FF | FS | , | < | L | \ | l | \| |
| **D** | CR | GS | - | = | M | ] | m | } |
| **E** | SO | RS | . | > | N | ^ | n | ~ |
| **F** | SI | US | / | ? | O | _ | o | DEL |

# Index

The index consists of two parts. The first part lists all important concepts that are used in the book. The second part lists all 68000-instructions that are discussed and/or used in the book. For index terms that have multiple entries, the first entry usually corresponds to where the term is introduced or used in an important way. Exceptions to this rule are marked by typing the second or any subsequent entries in **boldface** type.

## Concepts

## 68000-instructions

THIS BOOK IS AN INTRODUCTION to microcomputer system organization and assembly language programming, in particular for the Motorola 68000.

Experience in high-level language programming, such as an introductory course in Pascal, is all that is needed to learn how a computer works that is stripped of all the layers of software it is usually clothed in. From this starting point, the book systematically introduces the programming model and organization of microcomputers.

The instruction set model of a state-of-the-art microprocessor is often difficult to understand mainly because of its complexity. This book aims to dispel this difficulty by starting with a simple model of a computer, in essence a 68000-based system, and then successively refining this model to include more functionality. When the complete instruction set model has been introduced, the book shows how high-level language constructs, essentially Pascal-constructs, can be translated into sequences of assembly language instructions.

Control systems play an important role as embedded systems in microcomputers. This book emphasizes this application area by examining the concepts of I/O (polling, interrupts and programmable interfaces) and the design of I/O drivers. A case study provides an introduction to designing schedules for time-sharing and real-time operating systems.

This textbook contains several worked examples to highlight the basic ideas, and in addition there are a large number of exercises. The appendices contain solutions to all these exercises, a summary of most instructions for the Motorola 68000, and an ASCII table.

FEATURES

- *Basic mechanisms needed to support time-sharing and real-time operating systems.*
- *Symbol representation and elementary computer arithmetic*
- *Assembly-language programming methodology based on high-level language (HLL) programming techniques*
- *Low-level communication schemes between computer systems and external devices using programmable interfaces*

ABOUT THE AUTHOR

PER STENSTRÖM is an assistant professor in computer engineering at Lund University, Sweden. He has taught computer organization and architecture since 1981 and has published more than ten articles on advanced topics in computer architecture.

ISBN 0-13-584855-5

PRENTICE HALL

COVER DESIGN BY DESIGNERS AND PARTNERS, OXFORD.