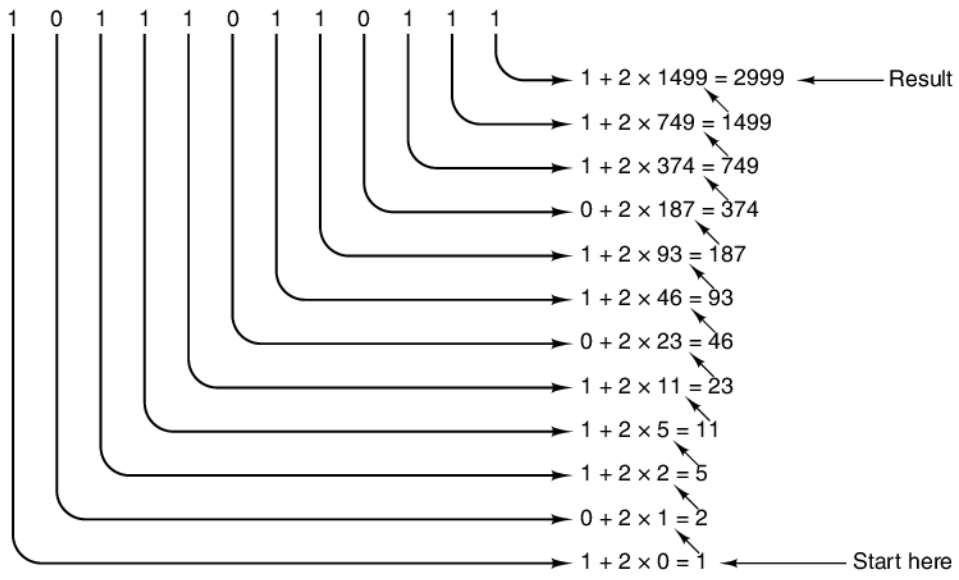

Figure A-6. Conversion of the binary number 101110110111 to decimal by successive doubling, starting at the bottom. Each line is formed by doubling the one below it and adding the corresponding bit. For example, 749 is twice 374 plus the 1 bit on the same line as 749.

to the result. Binary addition is the same as decimal addition except that a carry is generated if the sum is greater than 1 rather than greater than 9. For example, converting 6 to two's complement is done in two steps:

00000110 (+6)
11111001 (–6 in one's complement)
11111010 (–6 in two's complement)

If a carry occurs from the leftmost bit, it is thrown away.

The fourth system, which for m-bit numbers is called **excess 2^{m-1}**, represents a number by storing it as the sum of itself and 2^{m-1}. For example, for 8-bit numbers, $m = 8$, the system is called excess 128 and a number is stored as its true value plus 128. Therefore, –3 becomes –3 + 128 = 125, and –3 is represented by the 8-bit binary number for 125 (01111101). The numbers from –128 to +127 map onto 0 to 255, all of which are expressible as an 8-bit positive integer. Interestingly enough, this system is identical to two's complement with the sign bit reversed. Figure A-7 gives examples of negative numbers in all four systems.

Both signed magnitude and one's complement have two representations for zero: a plus zero, and a minus zero. This situation is undesirable. The two's complement system does not have this problem because the two's complement of plus zero is also plus zero. The two's complement system does, however, have a dif-

N decimal	N binary	–N signed mag.	–N 1's compl.	–N 2's compl.	–N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

Figure A-7. Negative 8-bit numbers in four systems.

ferent singularity. The bit pattern consisting of a 1 followed by all 0s is its own complement. The result is to make the range of positive and negative numbers unsymmetric; there is one negative number with no positive counterpart.

The reason for these problems is not hard to find: we want an encoding system with two properties:

1. Only one representation for zero.
2. Exactly as many positive numbers as negative numbers.

The problem is that any set of numbers with as many positive as negative numbers and only one zero has an odd number of members, whereas m bits allow an even number of bit patterns. There will always be either one bit pattern too many or one bit pattern too few, no matter what representation is chosen. This extra bit pattern can be used for –0 or for a large negative number, or for something else, but no matter what it is used for it will always be a nuisance.

A.5 BINARY ARITHMETIC

The addition table for binary numbers is given in Fig. A-8.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

Figure A-8. The addition table in binary.

Two binary numbers can be added, starting at the rightmost bit and adding the corresponding bits in the addend and the augend. If a carry is generated, it is carried one position to the left, just as in decimal arithmetic. In one's complement arithmetic, a carry generated by the addition of the leftmost bits is added to the rightmost bit. This process is called an end-around carry. In two's complement arithmetic, a carry generated by the addition of the leftmost bits is merely thrown away. Examples of binary arithmetic are shown in Fig. A-9.

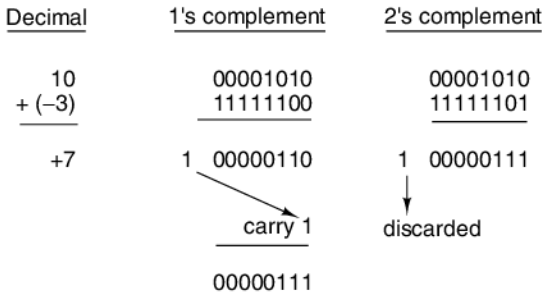

Figure A-9. Addition in one's complement and two's complement.

If the addend and the augend are of opposite signs, overflow error cannot occur. If they are of the same sign and the result is of the opposite sign, overflow error has occurred and the answer is wrong. In both one's and two's complement arithmetic, overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit. Most computers preserve the carry out of the sign bit, but the carry into the sign bit is not visible from the answer. For this reason, a special overflow bit is usually provided.

PROBLEMS

1. Convert the following numbers to binary: 1984, 4000, 8192.

2. What is 1001101001 (binary) in decimal? In octal? In hexadecimal?

3. Which of the following are valid hexadecimal numbers? BED, CAB, DEAD, DECADE, ACCEDED, BAG, DAD.

4. Express the decimal number 100 in all radices from 2 to 9.

5. How many different positive integers can be expressed in k digits using radix r numbers?

6. Most people can only count to 10 on their fingers; however, computer scientists can do better. If you regard each finger as one binary bit, with finger extended as 1 and finger touching palm as 0, how high can you count using both hands? With both hands and both feet? Now use both hands and both feet, with the big toe on your left foot as a sign bit for two's complement numbers. What is the range of expressible numbers?

7. Perform the following calculations on 8-bit two's complement numbers.

00101101	11111111	00000000	11110111
+ 01101111	+ 11111111	− 11111111	− 11110111

8. Repeat the calculation of the preceding problem but now in one's complement.

9. Consider the following addition problems for 3-bit binary numbers in two's complement. For each sum, state
a. Whether the sign bit of the result is 1.
b. Whether the low-order 3 bits are 0.
c. Whether an overflow occurred.

000	000	111	100	100
+ 001	+ 111	+ 110	+ 111	+ 100

10. Signed decimal numbers consisting of n digits can be represented in $n + 1$ digits without a sign. Positive numbers have 0 as the leftmost digit. Negative numbers are formed by subtracting each digit from 9. Thus the negative of 014725 is 985274. Such numbers are called nine's complement numbers and are analogous to one's complement binary numbers. Express the following as three-digit nine's complement numbers: 6, −2, 100, −14, −1, 0.

11. Determine the rule for addition of nine's complement numbers and then perform the following additions.

0001	0001	9997	9241
+ 9999	+ 9998	+ 9996	+ 0802

12. Ten's complement is analogous to two's complement. A ten's complement negative number is formed by adding 1 to the corresponding nine's complement number, ignoring the carry. What is the rule for ten's complement addition?

13. Construct the multiplication tables for radix 3 numbers.

14. Multiply 0111 and 0011 in binary.

15. Write a program that takes in a signed decimal number as an ASCII string and prints out its representation in two's complement in binary, octal, and hexadecimal.

16. Write a program that takes in two 32-character ASCII strings containing 0s and 1s, each representing a two's complement 32-bit binary number. The program should print their sum as a 32-character ASCII string of 0s and 1s.

B

FLOATING-POINT NUMBERS

In many calculations the range of numbers used is very large. For example, a calculation in astronomy might involve the mass of the electron, 9×10^{-28} grams, and the mass of the sun, 2×10^{33} grams, a range exceeding 10^{60}. These numbers could be represented by

```
000000000000000000000000000000000.000000000000000000000000000009
200000000000000000000000000000000.000000000000000000000000000000
```

and all calculations could be carried out keeping 34 digits to the left of the decimal point and 28 places to the right of it. Doing so would allow 62 significant digits in the results. On a binary computer, multiple-precision arithmetic could be used to provide enough significance. However, the mass of the sun is not even known accurately to five significant digits, let alone 62. In fact few measurements of any kind can (or need) be made accurately to 62 significant digits. Although it would be possible to keep all intermediate results to 62 significant digits and then throw away 50 or 60 of them before printing the final results, doing this is wasteful of both CPU time and memory.

What is needed is a system for representing numbers in which the range of expressible numbers is independent of the number of significant digits. In this appendix, such a system will be discussed. It is based on the scientific notation commonly used in physics, chemistry, and engineering.

B.1 PRINCIPLES OF FLOATING POINT

One way of separating the range from the precision is to express numbers in the familiar scientific notation

$$n = f \times 10^e$$

where f is called the **fraction**, or **mantissa**, and e is a positive or negative integer called the **exponent**. The computer version of this notation is called **floating point**. Some examples of numbers expressed in this form are

$$\begin{array}{llll} 3.14 & = 0.314 & \times 10^1 & = 3.14 \times 10^0 \\ 0.000001 & = 0.1 & \times 10^{-5} & = 1.0 \times 10^{-6} \\ 1941 & = 0.1941 & \times 10^4 & = 1.941 \times 10^3 \end{array}$$

The range is effectively determined by the number of digits in the exponent and the precision is determined by the number of digits in the fraction. Because there is more than one way to represent a given number, one form is usually chosen as the standard. In order to investigate the properties of this method of representing numbers, consider a representation, R, with a signed three-digit fraction in the range $0.1 \le |f| < 1$ or zero and a signed two-digit exponent. These numbers range in magnitude from $+0.100 \times 10^{-99}$ to $+0.999 \times 10^{+99}$, a span of nearly 199 orders of magnitude, yet only five digits and two signs are needed to store a number.

Floating-point numbers can be used to model the real-number system of mathematics, although there are some differences. Figure B-1 gives an exaggerated schematic of the real number line. The real line is divided up into seven regions:

1. Large negative numbers less than -0.999×10^{99}.
2. Negative numbers between -0.999×10^{99} and -0.100×10^{-99}.
3. Small negative numbers with magnitudes less than 0.100×10^{-99}.
4. Zero.
5. Small positive numbers with magnitudes less than 0.100×10^{-99}.
6. Positive numbers between 0.100×10^{-99} and 0.999×10^{99}.
7. Large positive numbers greater than 0.999×10^{99}.

One major difference between the set of numbers representable with three fraction and two exponent digits and the real numbers is that the former cannot be used to express any numbers in regions 1, 3, 5, or 7. If the result of an arithmetic operation yields a number in regions 1 or 7—for example, $10^{60} \times 10^{60} = 10^{120}$—**overflow error** will occur and the answer will be incorrect. The reason for this is due to the finite nature of the representation for numbers and is thus unavoidable. Similarly, a result in regions 3 or 5 cannot be expressed either. This situation is called

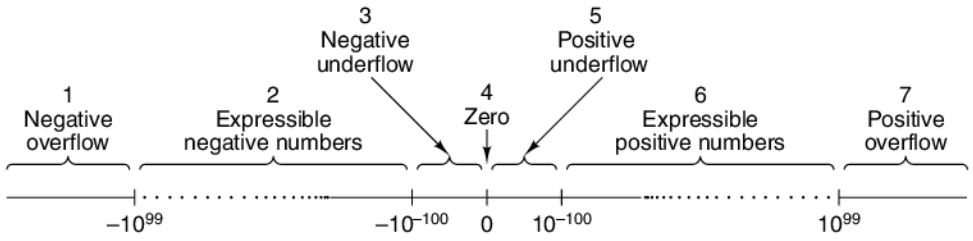

Figure B-1. The real number line can be divided into seven regions.

underflow error. Underflow error is less serious than overflow error, because 0 is often a satisfactory approximation to numbers in regions 3 and 5. A bank balance of 10^{-102} dollars is hardly better than a bank balance of 0.

Another important difference between floating-point numbers and real numbers is their density. Between any two real numbers, x and y, is another real number, no matter how close x is to y. This property comes from the fact that for any distinct real numbers, x and y, $z = (x + y)/2$ is a real number between them. The real numbers form a continuum.

Floating-point numbers, in contrast, do not form a continuum. Exactly 179,100 positive numbers can be expressed in the five-digit, two-sign system used above, 179,100 negative numbers, and 0 (which can be expressed in many ways), for a total of 358,201 numbers. Of the infinite number of real numbers between -10^{+100} and $+0.999 \times 10^{99}$, only 358,201 of them can be specified by this notation. They are symbolized by the dots in Fig. B-1. It is quite possible for the result of a calculation to be one of the other numbers, even though it is in region 2 or 6. For example, $+0.100 \times 10^{3}$ divided by 3 cannot be expressed *exactly* in our system of representation. If the result of a calculation cannot be expressed in the number representation being used, the obvious thing to do is to use the nearest number that can be expressed. This process is called **rounding**.

The spacing between adjacent expressible numbers is not constant throughout region 2 or 6. The separation between $+0.998 \times 10^{99}$ and $+0.999 \times 10^{99}$ is vastly more than the separation between $+0.998 \times 10^{0}$ and $+0.999 \times 10^{0}$. However, when the separation between a number and its successor is expressed as a percentage of that number, there is no systematic variation throughout region 2 or 6. In other words, the **relative error** introduced by rounding is approximately the same for small numbers as large numbers.

Although the preceding discussion was in terms of a representation system with a three-digit fraction and a two-digit exponent, the conclusions drawn are valid for other representation systems as well. Changing the number of digits in the fraction or exponent merely shifts the boundaries of regions 2 and 6 and changes the number of expressible points in them. Increasing the number of digits in the fraction increases the density of points and therefore improves the accuracy

of approximations. Increasing the number of digits in the exponent increases the size of regions 2 and 6 by shrinking regions 1, 3, 5, and 7. Figure B-2 shows the approximate boundaries of region 6 for floating-point decimal numbers for various sizes of fraction and exponent.

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Figure B-2. The approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers.

A variation of this representation is used in computers. For efficiency, exponentiation is to base 2, 4, 8, or 16 rather than 10, in which case the fraction consists of a string of binary, base-4, octal, or hexadecimal digits. If the leftmost of these digits is zero, all the digits can be shifted one place to the left and the exponent decreased by 1, without changing the value of the number (barring underflow). A fraction with a nonzero leftmost digit is said to be **normalized**.

Normalized numbers are generally preferable to unnormalized numbers, because there is only one normalized form, whereas there are many unnormalized forms. Examples of normalized floating-point numbers are given in Fig. B-3 for two bases of exponentiation. In these examples a 16-bit fraction (including sign bit) and a 7-bit exponent using excess 64 notation are shown. The radix point is to the left of the leftmost fraction bit—that is, to the right of the exponent.

B.2 IEEE FLOATING-POINT STANDARD 754

Until about 1980, each computer manufacturer had its own floating-point format. Needless to say, all were different. Worse yet, some of them actually did arithmetic incorrectly because floating-point arithmetic has some subtleties not obvious to the average hardware designer.

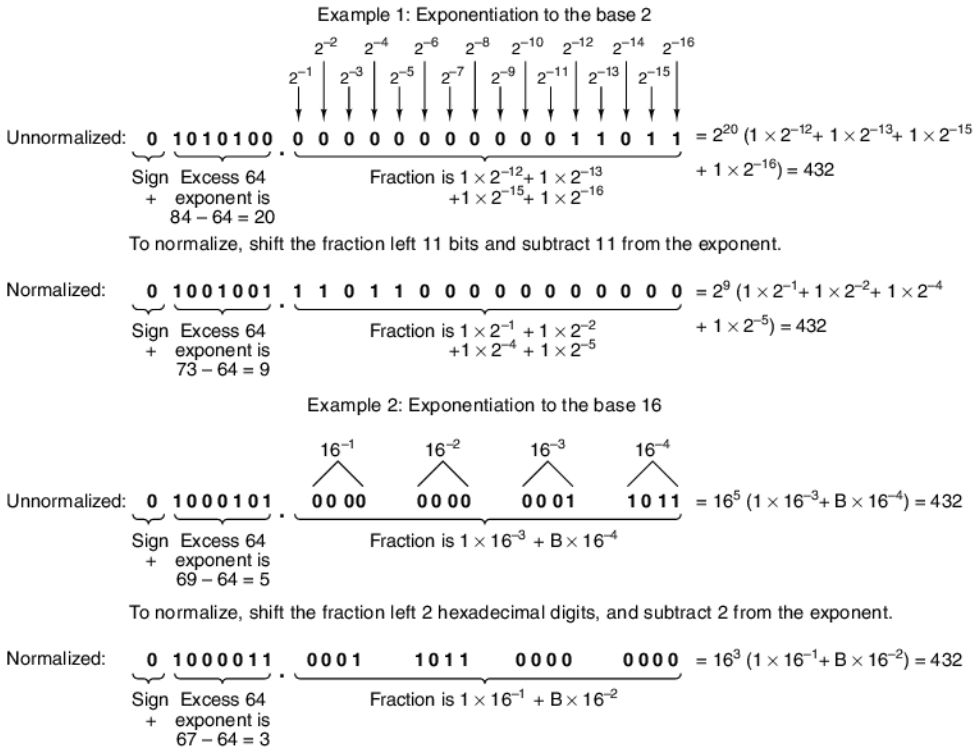

Figure B-3. Examples of normalized floating-point numbers.

To rectify this situation, in the late 1970s IEEE set up a committee to standardize floating-point arithmetic. The goal was not only to permit floating-point data to be exchanged among different computers but also to provide hardware designers with a model known to be correct. The resulting work led to IEEE Standard 754 (IEEE, 1985). Most CPUs these days (including the Intel and JVM ones studied in this book) have floating-point instructions that conform to the IEEE floating-point standard. Unlike many standards, which tend to be wishy-washy compromises that please no one, this one is not bad, in large part because it was primarily the work of one person, Berkeley math professor William Kahan. The standard will be described in the remainder of this section.

The standard defines three formats: single precision (32 bits), double precision (64 bits), and extended precision (80 bits). The extended-precision format is intended to reduce roundoff errors. It is used primarily inside floating-point arithmetic units, so we will not discuss it further. Both the single- and double-precision formats use radix 2 for fractions and excess notation for exponents. The formats are shown in Fig. B-4.

Both formats start with a sign bit for the number as a whole, 0 being positive and 1 being negative. Next comes the exponent, using excess 127 for single

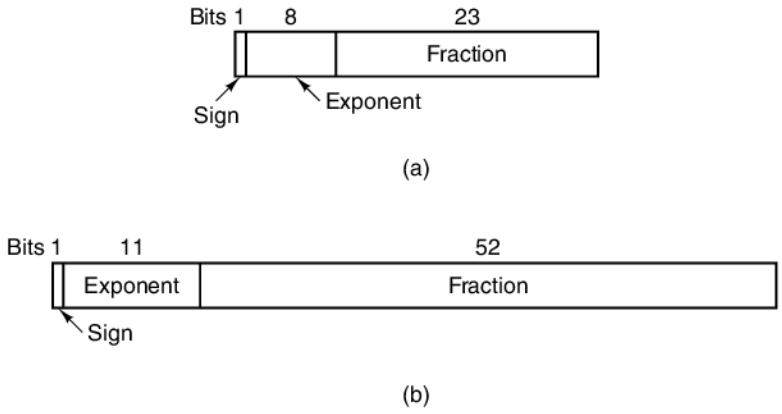

Figure B-4. IEEE floating-point formats. (a) Single precision. (b) Double precision.

precision and excess 1023 for double precision. The minimum (0) and maximum (255 and 2047) exponents are not used for normalized numbers; they have special uses described below. Finally, we have the fractions, 23 and 52 bits, respectively.

A normalized fraction begins with a binary point, followed by a 1 bit, and then the rest of the fraction. Following a practice started on the PDP-11, the authors of the standard realized that the leading 1 bit in the fraction does not have to be stored, since it can just be assumed to be present. Consequently, the standard defines the fraction in a slightly different way than usual. It consists of an implied 1 bit, an implied binary point, and then either 23 or 52 arbitrary bits. If all 23 or 52 fraction bits are 0s, the fraction has the numerical value 1.0; if all of them are 1s, the fraction is numerically slightly less than 2.0. To avoid confusion with a conventional fraction, the combination of the implied 1, the implied binary point, and the 23 or 52 explicit bits is called a **significand** instead of a fraction or mantissa. All normalized numbers have a significand, s, in the range $1 \leq s < 2$.

The numerical characteristics of the IEEE floating-point numbers are given in Fig. B-5. As examples, consider the numbers 0.5, 1, and 1.5 in normalized single-precision format. These are represented in hexadecimal as 3F000000, 3F800000, and 3FC00000, respectively.

One of the traditional problems with floating-point numbers is how to deal with underflow, overflow, and uninitialized numbers. The IEEE standard deals with these problems explicitly, borrowing its approach in part from the CDC 6600. In addition to normalized numbers, the standard has four other numerical types, described below and shown in Fig. B-6.

A problem arises when the result of a calculation has a magnitude smaller than the smallest normalized floating-point number that can be represented in this system. Previously, most hardware took one of two approaches: just set the result to zero and continue, or cause a floating-point underflow trap. Neither of these is

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Figure B-5. Characteristics of IEEE floating-point numbers.

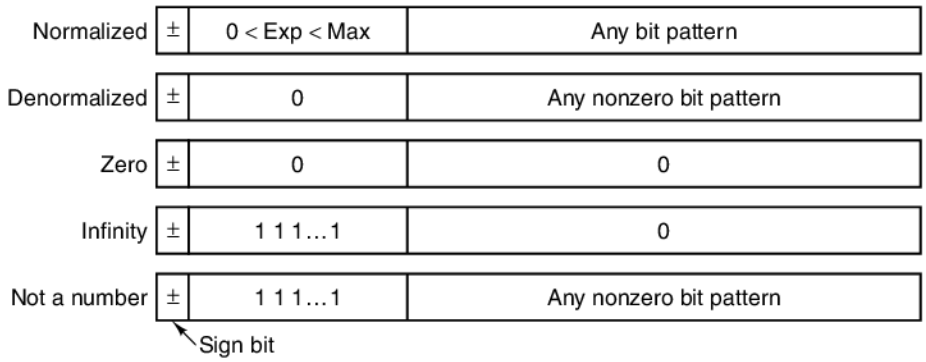

Figure B-6. IEEE numerical types.

really satisfactory, so IEEE invented **denormalized numbers**. These numbers have an exponent of 0 and a fraction given by the following 23 or 52 bits. The implicit 1 bit to the left of the binary point now becomes a 0. Denormalized numbers can be distinguished from normalized ones because the latter are not permitted to have an exponent of 0.

The smallest normalized single precision number has a 1 as exponent and 0 as fraction, and represents 1.0×2^{-126}. The largest denormalized number has a 0 as exponent and all 1s in the fraction, and represents about $0.9999999 \times 2^{-126}$, which is almost the same thing. One thing to note however, is that this number has only 23 bits of significance, versus 24 for all normalized numbers.

As calculations further decrease this result, the exponent stays put at 0, but the first few bits of the fraction become zeros, reducing both the value and the number of significant bits in the fraction. The smallest nonzero denormalized number consists of a 1 in the rightmost bit, with the rest being 0. The exponent represents

2^{-126} and the fraction represents 2^{-23} so the value is 2^{-149}. This scheme provides for a graceful underflow by giving up significance instead of jumping to 0 when the result cannot be expressed as a normalized number.

Two zeros are present in this scheme, positive and negative, determined by the sign bit. Both have an exponent of 0 and a fraction of 0. Here too, the bit to the left of the binary point is implicitly 0 rather than 1.

Overflow cannot be handled gracefully. There are no bit combinations left. Instead, a special representation is provided for infinity, consisting of an exponent with all 1s (not allowed for normalized numbers), and a fraction of 0. This number can be used as an operand and behaves according to the usual mathematical rules for infinity. For example infinity plus anything is infinity, and any finite number divided by infinity is zero. Similarly, any finite number divided by zero yields infinity.

What about infinity divided by infinity? The result is undefined. To handle this case, another special format is provided, called **NaN** (**Not a Number**). It too, can be used as an operand with predictable results.

PROBLEMS

1. Convert the following numbers to IEEE single-precision format. Give the results as eight hexadecimal digits.

a. 9
b. 5/32
c. −5/32
d. 6.125

2. Convert the following IEEE single-precision floating-point numbers from hex to decimal:

a. 42E48000H
b. 3F880000H
c. 00800000H
d. C7F00000H

3. The format of single-precision floating-point numbers on the 370 has a 7-bit exponent in the excess 64 system, and a fraction containing 24 bits plus a sign bit, with the binary point at the left end of the fraction. The radix for exponentiation is 16. The order of the fields is sign bit, exponent, fraction. Express the number 7/64 as a normalized number in this system in hex.

4. The following binary floating-point numbers consist of a sign bit, an excess 64, radix 2 exponent, and a 16-bit fraction. Normalize them.

a. 0 1000000 0001010100000001
b. 0 0111111 0000001111111111
c. 0 1000011 1000000000000000

5. To add two floating-point numbers, you must adjust the exponents (by shifting the fraction) to make them the same. Then you can add the fractions and normalize the result, if need be. Add the single-precision IEEE numbers 3EE00000H and 3D800000H and express the normalized result in hexadecimal.

6. The Tightwad Computer Company has decided to come out with a machine having 16-bit floating-point numbers. The Model 0.001 has a floating-point format with a sign bit, 7-bit, excess 64 exponent, and 8-bit fraction. The Model 0.002 has a sign bit, 5-bit, excess 16 exponent, and 10-bit fraction. Both use radix 2 exponentiation. What are the smallest and largest positive normalized numbers on each model? About how many decimal digits of precision does each have? Would you buy either one?

7. There is one situation in which an operation on two floating-point numbers can cause a drastic reduction in the number of significant bits in the result. What is it?

8. Some floating-point chips have a square root instruction built in. A possible algorithm is an iterative one (e.g., Newton-Raphson). Iterative algorithms need an initial approximation and then steadily improve it. How can one obtain a fast approximate square root of a floating-point number?

9. Write a procedure to add two IEEE single-precision floating-point numbers. Each number is represented by a 32-element Boolean array.

10. Write a procedure to add two single-precision floating-point numbers that use radix 16 for the exponent and radix 2 for the fraction but do not have an implied 1 bit to the left of the binary point. A normalized number has 0001, 0010, ..., 1111 as the leftmost 4 bits of the fraction, but not 0000. A number is normalized by shifting the fraction left 4 bits and subtracting 1 from the exponent.

This page intentionally left blank

C

ASSEMBLY LANGUAGE PROGRAMMING

Evert Wattel
Vrije Universiteit
Amsterdam, The Netherlands

Every computer has an **ISA** (**Instruction Set Architecture**), which is a set of registers, instructions, and other features visible to its low-level programmers. This ISA is commonly referred to as **machine language**, although the term is not entirely accurate. A program at this level of abstraction is a long list of binary numbers, one per instruction, telling which instructions to execute and what their operands are. Programming with binary numbers is very difficult to do, so all machines have an **assembly language**, a symbolic representation of the instruction set architecture, with symbolic names like ADD, SUB, and MUL, instead of binary numbers. This appendix is a tutorial on assembly language programming for one specific machine, the Intel 8088, which was used in the original IBM PC and was the base from which the modern Core i7 grew. The appendix also covers the use of some tools that can be downloaded to help learn about assembly language programming.

The purpose of this appendix is not to turn out polished assembly language programmers, but to help the reader learn about computer architecture through hands-on experience. For this reason, a simple machine—the Intel 8088— has

been chosen as the running example. While 8088s are rarely encountered any more, every Core i7 is capable of executing 8088 programs, so the lessons learned here are still applicable to modern machines. Furthermore, most of the Core i7's basic instructions are the same as the 8088's, only using 32-bit registers instead of 16-bit registers. Thus, this appendix can also be seen as a gentle introduction to Core i7 assembly language programming.

In order to program any machine in assembly language, the programmer must have a detailed knowledge of the machine's instruction set architecture. Accordingly, Sections C.1 through C.4 of this appendix are devoted to the architecture of the 8088, its memory organization, addressing modes, and instructions. Section C.5 discusses the assembler, which is used in this appendix and which is available for free, as described later. The notation used in this appendix is the one used by this assembler. Other assemblers use different notations, so readers already familiar with 8088 assembly programming should be alert for differences. Section C.6 discusses an interpreter/tracer/debugger tool, which can be downloaded to help the beginner programmer get programs debugged. Section C.7 describes the installation of the tools, and how to get started. Section C.8 contain programs, examples, exercises and solutions.

C.1 OVERVIEW

We will start our tour of assembly language programming with a few words on assembly language and then give a small example to illustrate it.

C.1.1 Assembly Language

Every assembler uses **mnemonics**, that is, short words such as ADD, SUB, and MUL for machine instructions such as add, subtract, and multiply, to make them easy to remember. In addition, assemblers allow the use of **symbolic names** for constants and **labels** to indicate instruction and memory addresses. Also, most assemblers support some number of **pseudoinstructions**, which do not translate into ISA instructions, but which are commands to the assembler to guide the assembly process.

When a program in assembly language is fed to a program called an **assembler**, the assembler converts the program into a **binary program** suitable for actual execution. This program can then be run on the actual hardware. However, when beginners start to program in assembly language, they often make errors and the binary program just stops, without any clue as to what went wrong. To make life easier for beginners, it is sometimes possible to run the binary program not on the actual hardware, but on a simulator, which executes one instruction at a time and gives a detailed display of what it is doing. In this way, debugging is much

easier. Programs running on a simulator run slowly, of course, but when the goal is to learn assembly language programming, rather than run a production job, this loss of speed is not important. This appendix is based on a toolkit that includes such a simulator, called the **interpreter** or **tracer**, as it interprets and traces the execution of the binary program step by step as it runs. The terms "simulator," "interpreter," and "tracer" will be used interchangeably throughout this appendix. Usually, when we are talking about just executing a program, we will speak of the "interpreter" and when we are talking about using it as a debugging tool, we will call it the "tracer," but it is the same program.

C.1.2 A Small Assembly Language Program

To make some of these abstract ideas a bit more concrete, consider the program and tracer image of Fig. C-1. An image of the tracer screen is given in Fig. C-1. Fig. C-1(a) shows a simple assembly language program for the 8088. The numbers following the exclamation marks are the source line numbers, to make it easier to refer to parts of the program. A copy of this program can be found in the accompanying material, in the directory *examples* in the source file *HlloWrld.s*. This assembly program, like all assembly programs discussed in this appendix, has the suffix *.s*, which indicates that it is an assembly language source program. The tracer screen, shown in Fig. C-1(b), contains seven windows, each containing different information about the state of the binary program being executed.

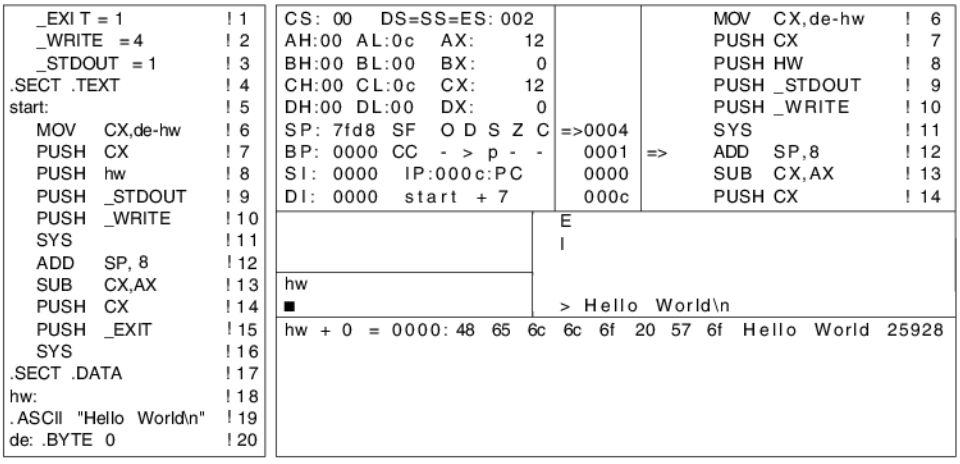

Figure C-1. (a) An assembly language program. (b) The corresponding tracer display.

Let us now briefly examine the seven windows of Fig. C-1(b). On the top are three windows, two larger ones and a smaller one in the middle. The top left

window shows the contents of the processor, consisting of the current values of the segment registers, CS, DS, SS, and ES, the arithmetic registers, AH, AL, AX, and others.

The middle window in the top row contains the stack, an area of memory used for temporary values.

The right-hand window in the top row contains a fragment of the assembly language program, with the arrow showing which instruction is currently being executed. As the program runs, the current instruction changes and the arrow moves to point to it. The strength of the tracer is that by hitting the return key (labeled Enter on PC keyboards), one instruction is executed and all the windows are updated, making it possible to run the program in slow motion.

Below the left window is a window that contains the subroutine call stack, here empty. Below it are commands to the tracer itself. To the right of these two windows is a window for input, output, and error messages.

Below these windows is a window that shows a portion of memory. These windows will be discussed in more detail later, but the basic idea should be clear: the tracer shows the source program, the machine registers, and quite a bit of information about the state of the program being executed. As each instruction is executed the information is updated, allowing the user to see in great detail what the program is doing.

C.2 THE 8088 PROCESSOR

Every processor, including the 8088, has an internal state, where it keeps certain crucial information. For this purpose, the processor has a set of **registers** where this information can be stored and processed. Probably the most important of these is the PC (**program counter**), which contains the memory location, that is, the **address**, of the next instruction to be executed. This register is also called IP (**Instruction Pointer**). This instruction is located in a part of the main memory, called the **code segment**. The main memory on the 8088 may be up to slightly more the 1 MB in size, but the current code segment is only 64 KB. The CS register in Fig. C-1 tells where the 64-KB code segment begins within the 1-MB memory. A new code segment can be activated by simply changing the value of the CS register. Similarly, there is also a 64-KB data segment, which tells where the data begins. In Fig. C-1 its origin is given by the DS register, which can also be changed as needed to access data outside the current data segment. The CS and DS registers are needed because the 8088 has 16-bit registers, so they cannot directly hold the 20-bit addresses needed to reference the entire 1-MB memory. This is why the code and data segment registers were introduced.

The other registers contain data or pointers to data in the main memory. In assembly language programs, these registers can be directly accessed. Apart from

these registers, the processor also contains all the necessary equipment to perform the instructions, but these parts are available to the programmer only through the instructions.

C.2.1 The Processor Cycle

The operation of the 8088 (and all other computers) consists of executing instructions, one after another. The execution of a single instruction can be broken down into the following steps:

1. Fetch the instruction from memory from the code segment using PC.
2. Increment the program counter.
3. Decode the fetched instruction.
4. Fetch the necessary data from memory and/or processor registers.
5. Perform the instruction.
6. Store the results of the instruction in memory and/or registers.
7. Go back to step 1 to start the next instruction.

The execution of an instruction is somewhat like running a very small program. In fact, some machines really do have a little program, called a **microprogram**, to execute their instructions. Microprograms are described in detail in Chap. 4.

From the point of view of an assembly programmer, the 8088 has a set of 14 registers. These registers are in some sense the scratch pad where the instructions operate and are in constant use, although the results stored in them are very volatile. Figure C-2 gives an overview of these 14 registers. It is clear that this figure and the register window of the tracer of Fig. C-1 are very similar because they represent the same information.

The 8088 registers are 16 bits wide. No two registers are completely functionally equivalent, but some of them share certain features, so they are subdivided into groups in Fig. C-2. We will now discuss the different groups.

C.2.2 The General Registers

The registers in the first group, AX, BX, CX, and DX are the **general registers**. The first register of this group, AX, is called the **accumulator register**. It is used to collect results of computations and is the target of many of the instructions. Although each of the registers can perform a host of tasks, in some instructions this AX is the implied destination, for example, in multiplication.

The second register of this group is BX, the **base register**. For many purposes BX can be used in the same way as AX, but it has one power AX does not have. It is

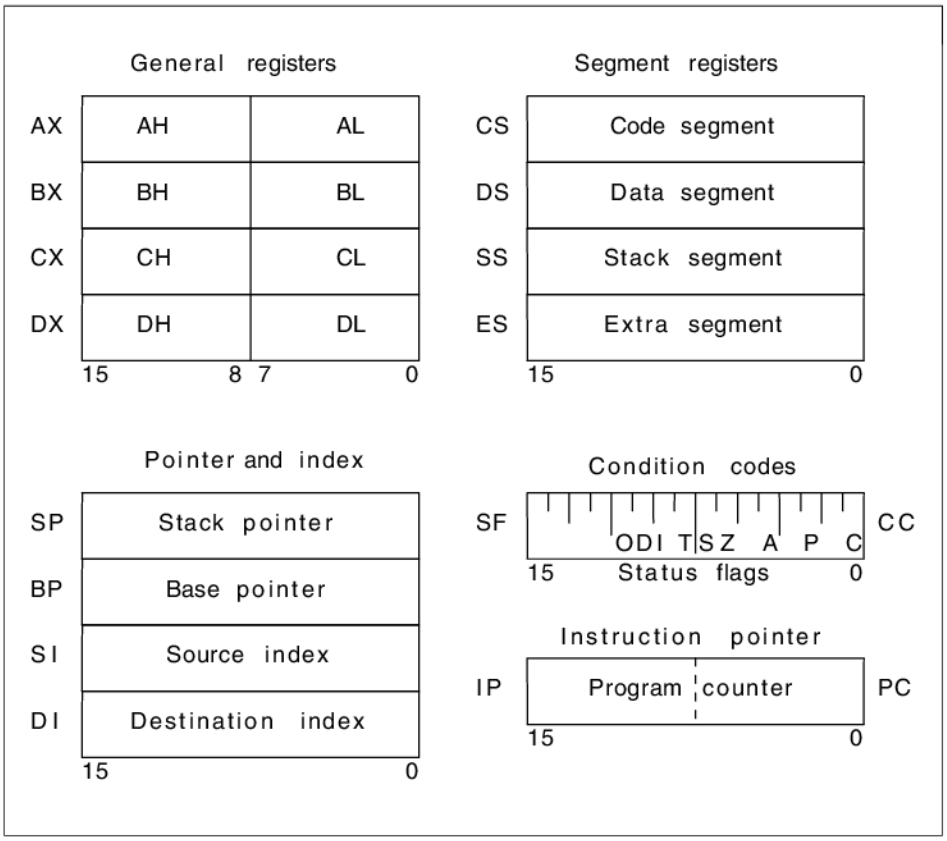

Figure C-2. The 8088 registers.

possible to put a memory address in BX and then execute an instruction whose operand comes from the memory address contained in BX. In other words, BX can hold a pointer to memory, AX cannot. To show this, we compare two instructions. First we have

```
MOV AX,BX
```

which copies to AX the contents of BX. Second we have

```
MOV AX,(BX)
```

which copies to AX the contents of the memory word whose address is contained in BX. In the first example, BX contains the source operand; in the second one it points to the source operand. In both of these examples, note that the MOV

instruction has a source and a destination operand, and that the destination is written before the source.

The next general register is CX, the **counter register**. Besides fulfilling many other tasks, this register is specifically used to contain counters for loops. It is automatically decremented in the LOOP instruction, and loops are usually terminated when CX reaches zero.

The fourth register of the general group is DX, the **data register**. It is used together with AX in double word length (i.e., 32-bit) instructions. In this case, DX contains the high-order 16 bits and AX contains the low-order 16 bits. Usually, 32-bit integers are indicated by the term **long**. The term **double** is usually reserved for 64-bit floating point values, although some people use "double" for 32-bit integers. In this tutorial, there will be no confusion because we will not discuss floating-point numbers at all.

All of these general registers can be regarded either as a 16-bit register or as a pair of 8-bit registers. In this way, the 8088 has precisely eight different 8-bit registers, which can be used in byte and character instructions. None of the other registers can be split into 8-bit halves. Some instructions use an entire register, such as AX, but other instructions use only half of a register, such as AL or AH. In general, instructions doing arithmetic use the full 16-bit registers, but instructions dealing with characters usually use the 8-bit registers. It is important, however, to realize that AL and AH are just names for both halves of AX. When AX is loaded with a new value, both AL and AH are changed to the lower and upper halves of the 16-bit number put in AX, respectively. To see how AX, AH, and AL interact, consider the instruction

```
MOV AX,258
```

which loads the AX register with the decimal value 258. After this instruction, the byte register AH contains the value 1, and the byte register AL contains the number 2. If this instruction is followed by the byte add instruction

```
ADDB AH,AL
```

then the byte register AH is incremented by the value in AL (2) so that it now contains 3. The effect on the register AX of this action is that its value is now 770, which is equivalent to 00000011 00000010 in binary notation or 0x03 0x02 in hexadecimal notation. The eight byte-wide registers are almost interchangeable, with the exception that AL always contains one of the operands in the MULB instruction, and is the implied destination of this operation, together with AH. DIVB also uses the AH : AL pair for the dividend. The lower byte of the counter register CL can be used to hold the number of cycles in shift and rotate instruction.

Section C.8, example 2, shows some of the properties of the general registers by means of a discussion of the program *GenReg.s*.

C.2.3 Pointer Registers

The second group of registers consists of the **pointer and index registers**. The most important register of this group is the **stack pointer**, which is denoted by SP. Stacks are important in most programming languages. The stack is a segment of memory that holds certain context information about the running program. Usually, when a procedure is called, part of the stack is reserved for holding the procedure's local variables, the address to return to when the procedure has finished, and other control information. The portion of the stack relating to a procedure is called its **stack frame**. When a called procedure calls another procedure, an additional stack frame is allocated, usually just below the current one. Additional calls allocate additional stack frames below the current ones. While not mandatory, stacks almost always grow downward, from high addresses to low addresses. Nevertheless, the lowest numerical address occupied on the stack is always called the top of the stack.

In addition to their use for holding local variables, stacks can also hold temporary results. The 8088 has an instruction, PUSH, which puts a 16-bit word on top of the stack. This instruction first decrements SP by 2, then stores its operand at the address SP is now pointing to. Similarly, POP removes a 16-bit word from the top of the stack by fetching the value on top of the stack and then incrementing SP by 2. The SP register points to the top of the stack and is modified by PUSH, POP, and CALL instructions, being decremented by PUSH, incremented by POP, and decremented by CALL.

The next register in this group is BP, the **base pointer**. It usually contains an address in the stack. Whereas SP always points to the top of the stack, BP can point to any location within the stack. In practice, a common use for BP is to point to the beginning of the current procedure's stack frame, in order to make it easy to find the procedure's local variables. Thus, BP often points to the bottom of the current stack frame (the stack frame word with the highest numerical value) and SP points to the top (the stack frame word with the lowest numerical value). The current stack frame is thus delimited by BP and SP.

In this register group, there are two index registers: SI, the **source index,** and DI, the **destination index**. These registers are often used in combination with BP to address data in the stack, or with BX to compute the addresses of data memory locations. More extensive treatment of these registers will be deferred to the section on addressing modes.

One of the most important registers, which is a group by itself, is the **instruction pointer**, which is Intel's name for the program counter (PC). This register is not addressed directly by the instructions, but contains an address in the program code segment of the memory. The processor's instruction cycle starts by fetching the instruction pointed to by PC. This register is then incremented before the rest of the instruction is executed. In this way this program counter points to the first instruction beyond the current one.

The **flag register** or **condition code register** is actually a set of single-bit registers. Some of the bits are set by arithmetic instructions and relate to the result, as follows:

Z - result is zero

S - result is negative (sign bit)

V - result generated an overflow

C - result generated a carry

A - Auxilary carry (out of bit 3)

P - parity of the result

Other bits in this register control operation of certain aspects of the processor. The *I* bit enables interrupts. The *T* bit enables tracing mode, which is used for debugging. Finally, the *D* bit controls the direction of the string operations. Not all 16 bits of this flag register are used; the unused ones are hardwired to zero.

There are four registers in the **segment register group**. Recall that the stack, the data and the instruction codes all reside in main memory, but usually in different parts of it. The segment registers govern these different parts of the memory, which are called **segments**. These registers are called CS for the code segment register, DS for the data segment register, SS for the stack segment register, and ES for the extra segment register. Most of the time, their values are not changed. In practice, the data segment and stack segment use the same piece of memory, with the data being at the bottom of the segment and the stack being at the top. More about these registers will be explained in Sec. C.3.1.

C.3 MEMORY AND ADDRESSING

The 8088 has a somewhat ungainly memory organization due to its combination of a 1-MB memory and 16-bit registers. With a 1-MB memory, it takes 20 bits to represent a memory address. Consequently, it is impossible to store a pointer to memory in any of the 16-bit registers. To get around this problem, memory is organized as segments, each of them 64 KB, so an address within a segment can be represented in 16 bits. We will now go into the 8088 memory architecture in more detail.

C.3.1 Memory Organization and Segments

The memory of the 8088, which consists simply of an array of addressable 8-bit bytes, is used for the storage of instructions as well as for the storage of data and for the stack. In order to separate the parts of the memory which are used for

these different purposes, the 8088 uses **segments** which are chunks of the memory set apart for a certain uses. In the 8088, such a segment consists of 65,536 consecutive bytes. There are four segments:

1. The code segment.
2. The data segment.
3. The stack segment.
4. The extra segment.

The code segment contains the program instructions. The contents of the PC register are always interpreted as a memory address in the code segment. A PC value of 0 refers to the lowest address in the code segment, not absolute memory address zero. The data segment contains the initialized and uninitialized data for the program. When BX contains a pointer, it points to this data segment. The stack segment contains local variables and intermediate results pushed on the stack. Addresses in SP and BP are always in this stack segment. The extra segment is a spare segment register that can be placed anywhere in memory that it is needed.

For each of the segments, there exists a corresponding segment register: the 16-bit registers CS, DS, SS, and ES. The starting address of a segment is the 20-bit unsigned integer which is constructed by shifting the segment register by 4 bits to the left, and putting zero's in the four right-most positions. This means that segment registers always indicate multiples of 16, in a 20-bit address space. The segment register points to the base of the segment. Addresses within the segment can be constructed by converting the 16-bit segment register value to its true 20-bit address by appending four zero bits to the end and adding the offset to that. In effect, an absolute memory address is computed by multiplying the segment register by 16 and then adding the offset to it. For example, if DS is equal to 7, and BX is 12, then the address indicated by BX is $7 \times 16 + 12 = 124$. In other words, the 20-bit binary address implied by DS = 7 is 00000000000001110000. Adding the 16-bit offset 0000000000001100 (decimal 12) to the segment's origin gives the 20-bit address 00000000000001111100 (decimal 124).

For *every* memory reference, one of the segment registers is used to construct the actual memory address. If some instruction contains a direct address without reference to a register, then this address is automatically in the data segment, and DS is used to determine the base of the segment. The physical address is found by adding this bottom to the address in the instruction. The physical address in memory of the next instruction code is obtained by shifting the contents of CS by four binary places and adding the value of the program counter. In other words, the true 20-bit address implied by the 16-bit CS register is first computed, then the 16-bit PC is added to it to form a 20-bit absolute memory address.

The stack segment is made up of 2-byte words and so the stack pointer, SP, should always contain an even number. The stack is filled up from high addresses to low addresses. Thus, the PUSH instruction decreases the stack pointer by 2 and then stores the operand in the memory address computed from SS and SP. The POP command retrieves the value, and increments SP by 2. Addresses in the stack segment which are lower than those indicated by SP are considered free. Stack cleanup is thus achieved by merely increasing SP. In practice, DS and SS are always the same, so a 16-bit pointer can be used to refer to a variable in the shared data/stack segment. If DS and SS were different, a 17th bit would be needed on each pointer to distinguish pointers into the data segment from pointers into the stack segment. In retrospect, having a separate stack segment at all was probably a mistake.

If addresses in the four segment registers are chosen to be far apart, then the four segments will be disjointed, but if the available memory is restricted, it is not necessary to make them disjoint. After compilation, the size of the program code is known. It is then efficient to start the data and stack segments at the first multiple of 16 after the last instruction. This assumes that the code and data segment will never use the same physical addresses.

C.3.2 Addressing

Almost every instruction needs data, either from memory or from the registers. To name this data, the 8088 has a reasonably versatile collection of addressing modes. Many instructions contain two operands, usually called **destination** and **source**. Think, for instance, about the copy instruction, or the add instruction:

```
MOV AX,BX
```

or

```
ADD CX,20
```

In these instructions, the first operand is destination and the second is the source. (The choice of which goes first is arbitrary; the reverse choice could also have been made.) It goes without saying that, in such a case, the destination must be a **left value** that is, it must be a place where something can be stored. This means that constants can be sources, but not destinations.

In its original design, the 8088 required that at least one operand in a two-operand instruction be a register. This was done so that the difference between **word instructions,** and **byte instructions** could be seen by checking whether the addressed register was a **word register** or a **byte register**. In the first release of the processor, this idea was so strictly enforced that it was impossible to push a constant, because neither the source nor the destination was a register in that instruction. Later versions were not as strict, but the idea influenced the design

anyway. In some cases, one of the operands is not mentioned. For example, in the MULB instruction, only the AX register is powerful enough to act as a destination.

There are also a number of one-operand instructions, such as increments, shifts, negates, etc. In these cases, there is no register requirement, and the difference between the word and byte operations has to be inferred from the opcodes (i.e., instruction types) only.

The 8088 supports four basic data types: 1-byte **byte,** the 2-byte **word,** the 4-byte **long**, and **binary coded decimal**, in which two decimal digits are packed into a word. The latter type is not supported by the interpreter.

A memory address always refers to a byte, but in case of a word or a long, the memory locations directly above the indicated byte are implicitly referred to as well. The word at 20 is in the memory locations 20 and 21. The long at address 24 occupies the addresses 24, 25, 26 and 27. The 8088 is **little endian,** meaning that the low-order part of the word is stored at the lower address. In the stack segment, words should be placed at even addresses. The combination AX DX, in which AX holds the low-order word, is the only provision made for longs in the processor registers.

The table of Fig. C-3 gives an overview of the 8088 addressing modes. Let us now briefly discuss them. The topmost horizontal block of the table lists the registers. They can be used as operands in nearly all instructions, both as sources and as destinations. There are eight word registers and eight byte registers.

The second horizontal block, data segment addressing, contains addressing modes for the data segment. Addresses of this type always contain a pair of parentheses, to indicate that the contents of the address instead of the value is meant. The easiest addressing mode of this type is **direct addressing**, in which the data address of the operand is in the instruction itself. Example

```
ADD CX,(20)
```

in which the contents of the memory word at address 20 and 21 is added to CX. Memory locations are usually represented by labels instead of by numerical values in the assembly language, and the conversion is made at assembly time. Even in CALL and JMP instructions, the destination can be stored in a memory location addressed by a label. The parentheses around the labels are essential (for the assembler we are using) because

```
ADD CX,20
```

is also a valid instruction, only it means add the constant 20 to CX, not the contents of memory word 20. In Fig. C-3, the # symbol is used to indicate a numerical constant, label, or constant expression involving a label.

In **register indirect addressing,** the address of the operand is stored in one of the registers BX, SI, or DI. In all three cases the operand is found in the data

Mode	Operand	Examples
Register addressing		
Byte register	Byte register	AH,AL,BH,BL,CH,CL,DH,DL
Word register	Word register	AX,BX,CX,DX,SP,BP,SI,DI
Data segment addressing		
Direct address	Address follows opcode	(#)
Register indirect	Address in register	(SI), (DI), (BX)
Register displacement	Address is register+displ.	#(SI), #(DI), #(BX)
Register with index	Address is BX + SI/DI	(BX)(SI), (BX)(DI)
Register index displacement	BX + SI DI + displacement	#(BX)(SI), #(BX)(DI)
Stack segment address		
Base Pointer indirect	Address in register	(BP)
Base pointer displacement	Address is BP + displ.	#(BP)
Base Pointer with index	Address is BP + SI/DI	(BP)(SI), (BP)(DI)
Base pointer index displ.	BP+SI/DI + displacement	#(BP)(SI), #(BP)(DI)
Immediate data		
Immediate byte/word	Data part of instruction	#
Implied address		
Push/pop instruction	Address indirect (SP)	PUSH, POP, PUSHF, POPF
Load/store flags	status flag register	LAHF, STC, CLC, CMC
Translate XLAT	AL, BX	XLAT
Repeated string instructions	(SI), (DI), (CX)	MOVS, CMPS, SCAS
In / out instructions	AX, AL	IN #, OUT #
Convert byte, word	AL, AX, DX	CBW,CWD

Figure C-3. Operand addressing modes. The symbol # indicates a numerical value or label.

segment. It is also possible to put a constant in front of the register, in which case the address is found by adding the register to the constant. This type of addressing, called **register displacement**, is convenient for arrays. If, for example, SI contains 5, then the fifth character of the string at the label *FORMAT* can be loaded in AL by

```
MOVB AL,FORMAT(SI).
```

The entire string can be scanned by incrementing or decrementing the register in each step. When word operands are used, the register should be changed by two each time.

It is also possible to put the base (i.e., lowest numerical address) of the array in the BX register, and keep the SI or DI register for counting. This is called **register with index** addressing. For example:

```
PUSH (BX)(DI)
```

fetches the contents of the data segment location whose address is given by the

sum of the BX and DI registers. This value is then pushed onto the stack. The last two types of addresses can be combined to get **register with index and displacement** addressing, as in

```
NOT 20(BX)(DI)
```

which complements the memory word at BX + DI + 20 and BX + DI + 21.

All the indirect addressing modes in the data segment also exist for the stack segment, in which case the base pointer BP is used instead of the base register BX. In this way (BP) is the only register indirect stack addressing mode, but more involved modes also exist, up to base pointer indirect with index and displacement –1(BP)(SI). These modes are valuable for addressing local variables and function parameters, which are stored in stack addresses in subroutines. This arrangement is described further in Sec. C.4.5.

All the addresses which comply with the addressing modes discussed up to now can be used as sources and as destinations for operations. Together they are defined to be **effective addresses**. The addressing mode in the remaining two blocks cannot be used as destinations and are not referred to as effective addresses. They can only be used as sources.

The addressing mode in which the operand is a constant byte or word value in the instruction itself is called **immediate addressing**. Thus, for example,

```
CMP AX,50
```

compares AX to the constant 50 and sets bits in the flag register, depending on the results.

Finally, some of the instructions use **implied addressing**. For these instructions, the operand or operands are implicit in the instruction itself. For example, the instruction

```
PUSH AX
```

pushes the contents of AX onto the stack by decrementing SP and then copying AX to the location now pointed to by SP. SP is not named in the instruction itself, however; the mere fact that it is a PUSH instruction implies that SP is used. Similarly, the flag manipulation instructions implicitly use the status flags register without naming it. Several other instructions also have implicit operands.

The 8088 has special instructions for moving (MOVS), comparing (CMPS), and scanning (SCAS) strings. With these string instructions, the index registers SI and DI are automatically changed after the operation. This behavior is called **auto increment** or **auto decrement** mode. Whether SI and DI are incremented or decremented depends on the **direction flag** in the status flags register. A direction flag value of 0 increments, whereas a value of 1 decrements. The change is 1 for byte instructions and 2 for word instructions. In a way, the stack pointer is also auto increment and auto decrement: it is decremented by 2 at the start of a PUSH and incremented by 2 at the end of a POP.

C.4 THE 8088 INSTRUCTION SET

The heart of every computer is the set of instructions it can carry out. To really understand a computer, it is necessary to have a good understanding of its instruction set. In the following sections, we will discuss the most important of the 8088's instructions. Some of them are shown in Fig. C-4, where they are divided into 10 groups.

C.4.1 Move, Copy and Arithmetic

The first group of instructions is the copy and move instructions. By far, the most common is the instruction MOV, which has an explicit source and an explicit destination. If the source is a register, the destination can be an effective address. In this table a register operand is indicated by an *r* and an effective address by an *e*, so this operand combination is denoted by *e*←*r*. This is the first entry in the *Operands* column for MOV. Since, in the instruction syntax, the destination is the first operand and the source is the second operand, the arrow ← is used to indicate the operands. Thus, *e*←*r* means that a register is copied to an effective address.

For the MOV instruction, the source can also be an effective address and the destination a register, which will be denoted by *r*←*e*, the second entry in the *Operands* column of the instruction. The third possibility is immediate data as source, and effective address as destination, which yields *e*←#. Immediate data in the table is indicated by the sharp sign (#). Since both the word move MOV and the byte move MOVB exist, the instruction mnemonic ends with a *B* between parentheses. Thus, the line really represents six different instructions.

None of the flags in the condition code register are affected by a move instruction, so the last four columns have the entry "-". Note that the move instructions do not move data. They make copies, meaning that the source is not modified as would happen with a true move.

The second instruction in the table is XCHG, which exchanges the contents of a register with the contents of an effective address. For the exchange the table uses the symbol ↔. In this case, there exists a byte version as well as a word version. Thus, the instruction is denoted by XCHG and the *Operand* field contains *r*↔*e*. The next instruction is LEA, which stands for Load Effective Address. It computes the numerical value of the effective address and stores it in a register.

Next is PUSH, which pushes its operand onto the stack. The explicit operand can either be a constant (# in the *Operands* column) or an effective address (*e* in the *Operands* column). There is also an implicit operand, SP, which is not mentioned in the instruction syntax. What the instruction does is decrement SP by 2, then store the operand at the location now pointed to by SP.

Then comes POP, which removes an operand from the stack to an effective address. The next two instructions, PUSHF and POPF, also have implied operands, and push and pop the flags register, respectively. This is also the case for XLAT

which loads the byte register AL from the address computed from AL + BX . This instruction allows for rapid lookup in tables of size 256 bytes.

Officially defined in the 8088, but not implemented in the interpreter (and thus not listed in Fig. C-4), are the IN and OUT instructions. These are, in fact, move instructions to and from an I/O device. The implied address is always the AX register, and the second operand in the instruction is the port number of the desired device register.

In the second block of Fig. C-4 are the addition and subtraction instructions. Each of these has the same three operand combinations as MOV: effective address to register, register to effective address, and constant to effective address. Thus, the *Operands* column of the table contains *r*←*e*, *e*←*r*, and *e*←#. In all four of these instructions, the overflow flag, *O*, the sign flag, *S*, the zero flag, *Z*, and the carry flag, *C* are all set, based on the result of the instruction. This means, for example, that *O* is set if the result cannot be correctly expressed in the allowed number of bits, and cleared if it can be. When the largest 16-bit number, 0x7fff (32,767 in decimal), is added to itself, the result cannot be expressed as a 16-bit signed number, so *O* is set to indicate the error. Similar things happen to the other status flags in these operations. If an instruction has an effect on a status flag, an asterisk (*) is shown in the corresponding column. In the instructions ADC and SBB, the carry flag at the start of the operation is used as an extra 1 (or 0), which is seen as a carry or borrow from the previous operation. This facility is especially useful for representing 32-bit or longer integers in several words. For all additions and subtractions, byte versions also exist.

The next block contains the multiplication and division instructions. Signed integer operands require the IMUL and IDIV instructions; unsigned ones use MUL and DIV. The AH : AL register combination is the implied destination in the byte version of these instructions. In the word version, the implied destination is the *AX*: DX register combination. Even if the result of the multiplication is only a word or a byte, the DX or AH register is rewritten during the operation. The multiplication is always possible because the destination contains enough bits. The overflow and carry bits are set when the product cannot be represented in one word, or one byte. The zero and the negative flags are undefined after a multiply.

Division also uses the register combinations DX : AX or AH : AL as the destination. The quotient goes into AX or AL and the remainder into DX or AH. All four flags, carry, overflow, zero and negative, are undefined after a divide operation. If the divisor is 0, or if the quotient does not fit into the register, the operation executes a **trap**, which stops the program unless a trap handler routine is present. Moreover, it is sensible to handle minus signs in software before and after the divide, because in the 8088 definition the sign of the remainder equals the sign of the dividend, whereas in mathematics, a remainder is always nonnegative.

The instructions for binary coded decimals, among which Ascii Adjust for Addition (AAA), and Decimal Adjust for Addition (DAA), are not implemented by the interpreter and not shown in Fig. C-4.

Mnemonic	Description	Operands	Status flags			
			O	S	Z	C
MOV(B)	Move word, byte	r ← e, e ← r, e ← #	-	-	-	-
XCHG(B)	Exchange word	r ↔ e	-	-	-	-
LEA	Load effective address	r ← #e	-	-	-	-
PUSH	Push onto stack	e, #	-	-	-	-
POP	Pop from stack	e	-	-	-	-
PUSHF	Push flags	-	-	-	-	-
POPF	Pop flags	-	-	-	-	-
XLAT	Translate AL	-	-	-	-	-
ADD(B)	Add word	r ← e, e ← r, e ← #	*	*	*	*
ADC(B)	Add word with carry	r ← e, e ← r, e ← #	*	*	*	*
SUB(B)	Subtract word	r ← e, e ← r, e ← #	*	*	*	*
SBB(B)	Subtract word with borrow	r ← e, e ← r, e ← #	*	*	*	*
IMUL(B)	Multiply signed	e	*	U	U	*
MUL(B)	Multiply unsigned	e	*	U	U	*
IDIV(B)	Divide signed	e	U	U	U	U
DIV(B)	Divide unsigned	e	U	U	U	U
CBW	Sign extend byte-word	-	-	-	-	-
CWD	Sign extend word-double	-	-	-	-	
NEG(B)	Negate binary	e	*	*	*	*
NOT(B)	Logical complement	e	-	-	-	-
INC(B)	Increment destination	e	*	*	*	-
DEC(B)	Decrement destination	e	*	*	*	-
AND(B)	Logical and	e ← r, r ← e, e ← #	0	*	*	0
OR(B)	Logical or	e ← r, r ← e, e ← #	0	*	*	0
XOR(B)	Logical exclusive or	e ← r, r ← e, e ← #	0	*	*	0
SHR(B)	Logical shift right	e ← 1, e ← CL	*	*	*	*
SAR(B)	Arithmetic shift right	e ← 1, e ← CL	*	*	*	*
SAL(B) (=SHL(B))	shift left	e ← 1, e ← CL	*	*	*	*
ROL(B)	Rotate left	e ← 1, e ← CL	*	-	-	*
ROR(B)	Rotate right	e ← 1, e ← CL	*	-	-	*
RCL(B)	Rotate left with carry	e ← 1, e ← CL	*	-	-	*
RCR(B)	Rotate right with carry	e ← 1, e ← CL	*	-	-	*
TEST(B)	Test operands	e ↔ r, e ↔ #	0	*	*	0
CMP(B)	Compare operands	e ↔ r, e ↔ #	*	*	*	*
STD	Set direction flag (↓)	-	-	-	-	-
CLD	Clear direction flag (↑)	-	-	-	-	-
STC	Set carry flag	-	-	-	-	1
CLC	Clear carry flag	-	-	-	-	0
CMC	Complement carry	-	-	-	-	*
LOOP	Jump back if decremented CX ≥ 0	label	-	-	-	-
LOOPZ LOOPE	Back if Z=1 and DEC(CX)≥0	label	-	-	-	-
LOOPNZ LOOPNE	Back if Z=0 and DEC(CX)≥0	label	-	-	-	-
REP REPZ REPNZ	Repeat string instruction	string instruction	-	-	-	-
MOVS(B)	Move word string	-	-	-	-	-
LODS(B)	Load word string	-	-	-	-	-
STOS(B)	Store word string	-	-	-	-	-
SCAS(B)	Scan word string	-	*	*	*	*
CMPS(B)	Compare word string	-	*	*	*	*
JCC	Jump according conditions	label	-	-	-	-
JMP	Jump to label	e, label	-	-	-	-
CALL	Jump to subroutine	e, label	-	-	-	-
RET	Return from subroutine	-, #	-	-	-	-
SYS	System call trap	-	-	-	-	-

Figure C-4. Some of the most important 8088 instructions.

C.4.2 Logical, Bit and Shift Operations

The next block contains instructions for sign extension, negation, logical complement, increment and decrement. The sign extend operations have no explicit operands, but act on the DX : AX or the AH : AL register combinations. The single operand for the other operations of this group can be found at any effective address. The flags are affected in the expected way in case of the NEG, INC and DEC, except that the carry is not affected in the increment and decrement, which is quite unexpected and which some people regard as a design error.

The next block of instructions is the two-operand logical group, all of whose instructions behave as expected. In the shift and rotate group, all operations have an effective address as their destination, but the source is either the byte register CL or the number 1. In the shifts, all four flags are affected; in the rotates, only the carry and the overflow are affected. The carry always gets the bit that is shifted or rotated out of the high-order or low-order bit, depending on the direction of the shift or rotate. In the rotates with carry, RCR, RCL, RCRB, and RCLB, the carry together with the operand at the effective address, constitutes a 17-bit or a 9-bit circular shift register combination, which facilitates multiple word shifts and rotates.

The next block of instructions is used to manipulate the flag bits. The main reason for doing this is to prepare for conditional jumps. The double arrow ($\leftrightarrow$) is used to indicate the two operands in compare and test operations, which do not change during the operation. In the TEST operation, the logical AND of the operands is computed to set or clear the zero flag and the sign flag. The computed value itself is not stored anywhere and the operand is unmodified. In the CMP, the difference of the operands is computed and all four flags are set or cleared as a result of the comparison. The direction flag, which determines whether the SI and DI registers should be incremented or decremented in the string instructions, can be set or cleared by STD and CLD, respectively.

The 8088 also has a **parity flag** and an **auxiliary carry flag**. The parity flag gives the parity of the result (odd or even). The auxiliary flag checks whether overflow was generated in the low (4-bit) nibble of the destination. There are also instructions LAHF and SAHF, which copy the low-order byte of the flag register in AH, and vice versa. The overflow flag is in the high-order byte of the condition code register and is not copied in these instructions. These instructions and flags are mainly used for backward compatibility with the 8080 and 8085 processors.

C.4.3 Loop and Repetitive String Operations

The following block contains the instructions for looping. The LOOP instruction decrements the CX register and jumps back to the label indicated if the result is positive. The instructions LOOPZ, LOOPE, LOOPNZ and LOOPNE also test the zero flag to see whether the loop should be aborted before CX is 0.

The destination for all LOOP instructions must be within 128 bytes of the current position of the program counter because the instruction contains an 8-bit signed offset. The number of *instructions* (as opposed to bytes) that can jumped over cannot be calculated exactly because different instructions have different lengths. Usually, the first byte defines the type of an instruction, and so some instructions take only one byte in the code segment. Often, the second byte is used to define the registers and register modes of the instruction, and if the instructions contain displacements or immediate data, the instruction length can increase to four or six bytes. The average instruction length is typically about 2.5 bytes per instruction, so the LOOP cannot jump further back than approximatel 50 instructions.

There also exist some special string instruction looping mechanisms. These are REP, REPZ, and REPNZ. Similarly, the five string instructions in the next block of Fig. C-4 all have implied addresses and all use auto increment or auto decrement mode on the index registers. In all of these instructions, the SI register points into the **data segment**, but the DI register refers to the **extra segment**, which is based on ES. Together with the REP instruction, the MOVSB can be used to move complete strings in one instruction. The length of the string is contained in the CX register. Since the MOVSB instruction does not affect the flags, it is not possible to check for an ASCII zero byte during the copy operation by means of the REPNZ, but this can be fixed by using first a REPNZ SCASB to get a sensible value in CX and later a REP MOVSB. This point will be illustrated by the string copy example in Sec. C.8. For all of these instructions, extra attention should be paid to the segment register ES, unless ES and DS have the same value. In the interpreter a small memory model is used, so that ES = DS = SS.

C.4.4 Jump and Call Instructions

The last block is about conditional and unconditional jumps, subroutine calls, and returns. The simplest operation here is the JMP instruction. It can have a label as destination or the contents of any effective address. A distinction is made between a **near jump** and a **far jump**. In a near jump, the destination is in the current code segment, which does not change during the operation. In a far jump, the CS register is changed during the jump. In the direct version with a label, the new value of the code segment register is supplied in the call after the label, in the effective address version, a long is fetched from memory, such that the low word corresponds to the destination label, and the high word to the new code segment register value.

It is, of course, not surprising, that such a distinction exists. To jump to an arbitrary address within a 20-bit address space, some provision has to be made for specifying more than 16 bits. The way it is done is by giving new values for CS and PC.

Conditional jumps

The 8088 has 15 conditional jumps, a few of which have two names (e.g., JUMP GREATER OR EQUAL is the same instruction as JUMP NOT LESS THAN). They are listed in Fig. C-5. All of these allow only jumps with a distance of up to 128 bytes from the instruction. If the destination is not within this range, a jump over jump construction has to be used. In such a construction, the jump with the opposite condition is used to jump over the next instruction. If the next instruction contains an unconditional jump to the intended destination, then the effect of these two instructions is just a longer-ranging jump of the intended type. for example

```
        JB FARLABEL
```

becomes

```
        JNA 1f
        JMP FARLABEL
1:
```

In other words, if it is not possible to do JUMP BELOW, then a JUMP NOT ABOVE to a nearby label *1* is placed, followed by an unconditional jump to *FARLABEL*. The effect is the same, at a slightly higher cost in time and space. The assembler generates these jump over jumps automatically when the destination is expected to be too distant. Doing the calculation correctly is a bit tricky. Suppose that the distance is close to the edge, but some of the intervening instructions are also conditional jumps. The outer one cannot be resolved until the sizes of the inner ones are known, and so on. To be safe, the assembler errs on the side of caution. Sometimes it generates a jump over jump when it is not strictly necessary. It only generates a direct condition jump when it is certain that the target is within range.

Most conditional jumps depend on the status flags, and are preceded by a compare or test instruction. The CMP instruction subtracts the source from the destination operand, sets the condition codes and discards the result. Neither of the operands is changed. If the result is zero or has the sign bit on (i.e., is negative), the corresponding flag bit is set. If the result cannot be expressed in the allowed number of bits, the overflow flag is set. If there is a carry out of the high-order bit, the carry flag is set. The conditional jumps can test all of these bits.

If the operands are considered to be signed, the instructions using GREATER THAN and LESS THAN should be used. If they are unsigned, the ones using ABOVE and BELOW should be used.

C.4.5 Subroutine Calls

The 8088 has an instruction used to call procedures, usually known in assembly language as **subroutines**. In the same way as in the jump instructions, there exist **near call** instructions and **far call** instructions. In the interpreter, only the

Instruction	Description	When to jump
JNA, JBE	Below or equal	CF=1 or ZF=1
JNB, JAE, JNC	Not below	CF=0
JE, JZ	Zero, equal	ZF=1
JNLE, JG	Greater than	SF=OF and ZF=0
JGE, JNL	Greater equal	SF=OF
JO	Overflow	OF=1
JS	Sign negative	SF=1
JCXZ	CX is zero	CX=0
JB, JNAE, JC	Below	CF=1
JNBE, JA	Above	CF=0&ZF=0
JNE, JNZ	Nonzero, nonequal	ZF=0
JL, JNGE	Less than	SF≠OF
JLE, JNG	Less or equal	SF≠OF or ZF=1
JNO	Nonoverflow	OF=0
JNS	Nonnegative	SF=0

Figure C-5. Conditional jumps.

near call is implemented. The destination is either a label or can be found at an effective address. Parameters needed in the subroutines have to be pushed onto the stack in reverse order first, as illustrated in Fig. C-6. In assembly language, parameters are usually called **arguments**, but the terms are interchangeable. Following these pushes the CALL instruction is executed. The instruction starts by pushing the current program counter onto the stack. In this way the return address is saved. The return address is the address at which the execution of the calling routine has to be resumed when the subroutine returns.

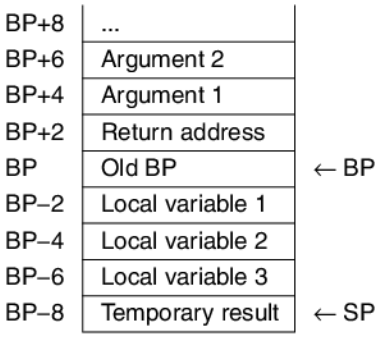

Figure C-6. An example stack.

Next the new program counter is loaded either from the label, or from the effective address. If the call is far, then the CS register is pushed before PC and

both the program counter and the code segment register are either loaded from immediate data or from the effective address. This finishes the CALL instruction.

The return instruction, RET, just pops the return address from the stack, stores it in the program counter and the program continues at the instruction immediately after the CALL instruction. Sometimes the RET instruction contains a positive number as immediate data. This number is assumed to be the number of bytes of arguments that were pushed onto the stack before the call; it is added to SP to clean up the stack. In the far variant, RETF, the code segment register is popped after the program counter, as would be expected.

Inside the subroutine, the arguments need to be accessible. Therefore the subroutine starts often by pushing the base pointer and copying the current value of SP into BP. This means that the base pointer points to its previous value. Now the return address is at BP + 2 and the first and second arguments can be found at the effective addresses BP + 4 and BP + 6, respectively. If the procedure needs local variables, then the required number of bytes can be subtracted from the stack pointer, and those variables can be addressed from the base pointer with negative offsets. In the example of Fig. C-6, there are three single-word local variables, located at BP – 2, BP – 4, and BP – 6, respectively. In this way, the entire set of current arguments and local variables is reachable through the BP register.

The stack is used in the ordinary way to save intermediate results, or for preparing arguments for the next call. Without computing the amount of stack used in the subroutine, the stack can be restored before the return by copying the base pointer into the stack pointer, popping the old BP and finally executing the RET instruction.

During a subroutine call, the values of the processor registers sometimes change. It is good practice to use some type of convention such that the calling routine need not be aware of the registers used by the called routine. The simplest way to do this is to use the same conventions for system calls and ordinary subroutines. It is assumed that the AX and DX can change in the called routine. If one of these registers contains valuable information then it is advisable for the calling routine to stack them before pushing the arguments. If the subroutine uses other registers as well, those can be pushed onto the stack immediately at the start of the subroutine, and popped before the RET instruction. In other words, a good convention is for the caller to save AX and DX if they contain anything important, and for the callee to save any other registers it overwrites.

C.4.6 System Calls and System Subroutines

In order to separate the tasks of opening, closing, reading, and writing files from assembly programming, programs are run on top of an operating system. To allow the interpreter to run on multiple platforms, a set of seven system calls and five functions are supported by the interpreter. They are listed in Fig. C-7.

Nr	Name	Arguments	Return value	Description
5	_OPEN	*name, 0/1/2	file descriptor	Open file
8	_CREAT	*name, *mode	file descriptor	Create file
3	_READ	fd, buf, nbytes	# bytes	Read nbytes in buffer buf
4	_WRITE	fd, buf, nbytes	# bytes	Write nbytes from buffer buf
6	_CLOSE	fd	0 on success	Close file with fd
19	_LSEEK	fd, offset(long), 0/1/2	position (long)	Move file pointer
1	_EXIT	status		Close files Stop process
117	_GETCHAR		read character	Read character from std input
122	_PUTCHAR	char	write byte	Write character to std output
127	_PRINTF	*format, arg		Print formatted on std output
121	_SPRINTF	buf, *format, arg		Print formatted in buffer buf
125	_SSCANF	buf, *format, arg		Read arguments from buffer buf

Figure C-7. Some UNIX system calls and subroutines in the interpreter.

These twelve routines can be activated by the standard calling sequence; first push the necessary arguments on the stack in reverse order, then push the call number, and finally execute the system trap instruction SYS without operands. The system routine finds all the necessary information on the stack, including the call number of the required system service. Return values are put either in the AX register, or in the DX : AX register combination (when the return value is a long).

It is guaranteed that all other registers will keep their values over the SYS instruction. Also, the arguments will still be on the stack after the call. Since they are not needed any more, the stack pointer should be adjusted after the call (by the caller), unless they are needed for a subsequent call.

For convenience, the names of the system calls can be defined as constants at the start of the assembler program, so that they can be called by name instead of by number. In the examples, several system calls will be discussed, so in this section only a minimum of necessary detail is supplied.

In these system calls, files are opened either by the OPEN or by the CREAT call. In both cases, the first argument is the address of the start of a string containing the file name. The second argument in the OPEN call is either 0 (if the file should be opened for reading), 1 (if it should be opened for writing), or 2 (for both). If the file should allow writes, and does not exist, it is created by the call. In the CREAT call an empty file is created, with permission set according to the second argument. Both the OPEN and the CREAT call return a small integer in the AX register, which is called the **file descriptor** and which can be used for reading, writing or closing the file. A negative return value means the call failed. At the start of the program, three files are already opened with file descriptors: 0 for standard input, 1 for standard output, and 2 for standard error output.

The READ and WRITE calls have three arguments: the file descriptor, a buffer to hold the data, and the number of bytes to transfer. Since the arguments are stacked

in reverse order, we first push the number of bytes, then the address of the start of the buffer, then the file descriptor and finally the call number (READ or WRITE). This order of stacking the arguments was chosen to be the same as the standard C language calling sequence in which

```
read(fd, buffer, bytes);
```

is implemented by pushing the parameters in the order *bytes*, *buffer*, and finally *fd*.

The CLOSE call requires just the file descriptor and returns 0 in AX if the file could be closed successfully. The EXIT call requires the exit status on the stack and does not return.

The LSEEK call changes the **read/write pointer** in an open file. The first argument is the file descriptor. Since the second argument is a long, first the high-order word, then the low word should be pushed onto the stack, even when the offset would fit into a word. The third argument indicates whether the new read/write pointer should be computed relative to the start of the file (case 0), relative to the current position (case 1), or relative to the end of the file (case 2). The return value is the new position of the pointer relative to the start of a file, and can be found as a long in the DX : AX register combination.

Now we come to the functions that are not system calls. The GETCHAR function reads one character from standard input, and puts it in AL. AH is set to zero. On failure, AX is set to −1. The call PUTCHAR writes a byte on standard output. The return value for a successful write is the byte written; on failure it is −1.

The call PRINTF outputs formatted information. The first argument to the call is the address of a format string, which tells how to format the output. The string "%d" indicates that the next argument is an integer on the stack, which is converted to decimal notation when printed. In the same way, "%x" converts to hexadecimal and "%o" converts to octal. Furthermore, "%s" indicates that the next argument is a null-terminated string, which is passed to the call through a memory address on the stack. The number of extra arguments on the stack should match the number of conversion indications in the format string.

For example, the call

```
printf("x = %d and y = %d\n", x, y);
```

prints the string with the numerical values of *x* and *y* substituted for the "%d" strings in the format string. Again, for compatibility with C, the order in which the arguments are pushed is "y", "x", and finally, the address of the format string. The reason for this convention is that *printf* has a variable number of parameters, and by pushing them in the reverse order the format string itself is always the last one and thus can be located. If the parameters were pushed from left to right, the format string would be deep in the stack and the *printf* procedure would not know where to find it.

In the call PRINTF, the first argument is the buffer, to receive the output string, instead of standard output. The other arguments are the same as in PRINTF. The

SSCANF call is the converse of the PRINTF in the sense that the first argument is a string, which can contain integers in decimal, octal, or hexadecimal notation, and the next argument is the format string, which contains the conversion indications. The other arguments are addresses of memory words to receive the converted information. These system subroutines are very versatile and an extensive treatment of the possibilities is far beyond the scope of this appendix. In Sec. C.8, several examples show how they can be used in different situations.

C.4.7 Final Remarks on the Instruction Set

In the official definition of the 8088, there exists a **segment override** prefix, which facilitates the possibility of using effective addresses from a different segment; that is, the first memory address following the override is computed using the indicated segment register. For example, the instruction

```
ESEG MOV DX,(BX)
```

first computes the address of BX using the extra segment, and then moves the contents to DX. However, the stack segment, in the case of addresses using SP, and the extra segment, in the case of string instructions with the DI register, cannot be overridden. The segment registers SS, DS and ES can be used in the MOV instruction, but it is impossible to move immediate data into a segment register, and those registers cannot be used in an XCHG operation. Programming with changing segment registers and overrides is quite tricky and should be avoided whenever possible. The interpreter uses fixed segment registers, so these problems do not arise here.

Floating-point instructions are available in most computers, sometimes directly in the processor, sometimes in a separate coprocessor, and sometimes only interpreted in the software through a special kind of floating point trap. Discussion of those features is outside the scope of this appendix.

C.5 THE ASSEMBLER

We have now finished our discussion of the 8088 architecture. The next topic is the software used to program the 8088 in assembly language, in particular the tools we provide for learning assembly language programming. We will first discuss the assembler, then the tracer, and then move on to some practical information for using them.

C.5.1 Introduction

Up until now, we have referred to instructions by their **mnemonics**, that is, by short easy-to-remember symbolic names like ADD and CMP. Registers were also called by symbolic names, such as AX and BP. A program written using symbolic

names for instructions and registers is called an **assembly language program**. To run such a program, it is first necessary to translate it into the binary numbers that the CPU actually understands. The program that converts an assembly language program into binary numbers is the **assembler**. The output of the assembler is called an **object file**. Many programs make calls to subroutines that have been previously assembled and stored in libraries. To run these programs, the newly-assembled object file and the library subroutines it uses (also object files) must be combined into a single **executable binary file** by another program called a **linker**. Only when the linker has built the executable binary file from one or more object files is the translation fully completed. The operating system can then read the executable binary file into memory and execute it.

The first task of the assembler is to build a **symbol table**, which is used to map the names of symbolic constants and labels directly to the binary numbers that they represent. Constants that are directly defined in the program can be put in the symbol table without any processing. This work is done on pass one.

Labels represent addresses whose values are not immediately obvious. To compute their values, the assembler scans the program line by line in what is called the **first pass**. During this pass, it keeps track of a **location counter** usually indicated by the symbol ".", pronounced **dot**. For every instruction and memory reservation that is found in this pass, the location counter is increased by the size of the memory necessary to contain the scanned item. Thus, if the first two instructions are of size 2 and 3 bytes, respectively, then a label on the third instruction will have numerical value 5. For example, if this code fragment is at the start of a program, the value of *L* will be 5.

```
MOV AX,6
MOV BX,500
L:
```

At the start of the **second pass**, the numerical value of every symbol is known. Since the numerical values of the instruction mnemonics are constants, **code generation** can now begin. One at a time, instructions are read again and their binary values are written into the object file. When the last instruction has been assembled, the object file is complete.

C.5.2 The ACK-Based Assembler, as88

This section describes the details of the assembler/linker *as88*, which is provided on the CD-ROM and website and which works with the tracer. This assembler is Amsterdam Compiler Kit (ACK) and is patterned after UNIX assemblers rather than MS-DOS or Windows assemblers. The comment symbol in this assembler is the exclamation mark (!). Anything following an exclamation mark until the end of the line is a comment and does not affect the object file produced. In the same way, empty lines are allowed, but ignored.

This assembler uses three different sections, in which the translated code and data will be stored. Those sections are related to the memory segments of the machine. The first is the **TEXT section**, for the processor instructions. Next is the **DATA section** for the initialization of the memory in the data segment, which is known at the start of the process. The last is the **BSS** (**Block Started by Symbol**), section, for the reservation of memory in the data segment that is not initialized (i.e., initialized to 0). Each of these sections has its own location counter. The purpose of having sections is to allow the assembler to generate some instructions, then some data, then some instructions, then more data, and so on, and then have the linker rearrange the pieces so that all the instructions are together in the text segment and all the data words are together in the data segment. Each line of assembly code produces output for only one section, but code lines and data lines can be interleaved. At run time, the TEXT section is stored in the text segment and the data and BSS sections are stored (consecutively) in the data segment.

An instruction or data word in the assembly language program can begin with a label. A label may also appear all by itself on a line, in which case it is as though it appeared on the next instruction or data word. For example, in

```
    CMP AX,ABC
    JE L
    MOV AX,XYZ
L:
```

L is a label that refers to the instruction of data word following it. Two kinds of labels are allowed. First are the **global labels**, which are alphanumeric identifiers followed by a colon (:). These must all be unique, and cannot match any keyword or instruction mnemonic. Second, in the TEXT section only, we can have **local labels**, each of which consists of a single digit followed by a colon (:). A local label may occur multiple times. When a program contains an instruction such as

```
JE 2f
```

this means JUMP EQUAL forward to the next local label *2*. Similarly,

```
JNE 4b
```

means JUMP NOT EQUAL backward to the closest label *4*.

The assembler allows constants to be given a symbolic name using the syntax

```
identifier = expression
```

in which the identifier is an alphanumeric string, as in

```
BLOCKSIZE = 1024
```

Like all identifiers in this assembly language, only the first eight characters are significant, so *BLOCKSIZE* and *BLOCKSIZZ* are the same symbol, namely, *BLOCKSIZ*. Expressions can be constructed from constants, numerical values, and

operators. Labels are considered to be constants because at the end of the first pass their numerical values are known.

Numerical values can be **octal** (starting with a 0), **decimal**, or **hexadecimal** (starting with 0X or 0x). Hexadecimal numbers use the letters a–f or A–F for the values 10–15. The integer operators are +, –, *, /, and %, for addition, subtraction, multiplication, division and remainder, respectively. The logical operators are &, ^, and ~, for bitwise AND, bitwise OR and logical complement (NOT) respectively. Expressions can use the square brackets, [and] for grouping. Parentheses are *NOT* used, to avoid confusion with the addressing modes.

Labels in expressions should be handled in a sensible way. Instruction labels cannot be subtracted from data labels. The difference between comparable labels is a numerical value, but neither labels nor their differences are allowed as constants in multiplicative or logical expressions. Expressions which are allowed in constant definitions can also be used as constants in processor instructions. Some assemblers have macro facility, by which multiple instructions can be grouped together and given a name, but *as88* does not have this feature.

In every assembly language, there are some directives that influence the assembly process itself but which are not translated into binary code. They are called **pseudoinstructions**. The *as88* pseudoinstructions are listed in Fig. C-8.

Instruction	Description
.SECT .TEXT	Assemble the following lines in the TEXT section
.SECT .DATA	Assemble the following lines in the DATA section
.SECT .BSS	Assemble the following lines in the BSS section
.BYTE	Assemble the arguments as a sequence of bytes
.WORD	Assemble the arguments as a sequence of words
.LONG	Assemble the arguments as a sequence of longs
.ASCII "str"	Store str as an ASCII string without a trailing zero byte
.ASCIZ "str"	Store str as an ASCII string with a trailing zero byte
.SPACE n	Advance the location counter n positions
.ALIGN n	Advance the location counter up to an n-byte boundary
.EXTERN	Identifier is an external name

Figure C-8. The *as88* pseudoinstructions.

The first block of pseudoinstructions determines the section in which the following lines should be processed by the assembler. Usually such a section requirement is made on a separate line and can be put anywhere in the code. For implementation reasons, the first section to be used must be the TEXT section, then the DATA section, then the BSS section. After these initial references, the sections can be used in any order. Furthermore, the first line of a section should have a global label. There are no other restrictions on the ordering of the sections.

The second block of pseudoinstructions contains the data type indications for the data segment. There are four types: .BYTE, .WORD, .LONG, and string. After an optional label and the pseudoinstruction keyword, the first three types expect a comma-separated list of constant expressions on the remainder of the line. For strings there are two keywords, ASCII, and ASCIZ, with the only difference being that the second keyword adds a zero byte to the end of the string. Both require a string between double quotes. Several escapes are allowed in string definitions. These include those of Fig. C-9. In addition to these, any specific character can be inserted by a backslash and an octal representation, for example, \377 (at most three digits, no 0 required here).

Escape symbol	Description
\n	New line (line feed)
\t	Tab
\\	Backslash
\b	Back space
\f	Form feed
\r	Carriage return
\"	Double quote

Figure C-9. Some of the escapes allowed by *as88*.

The SPACE pseudoinstruction simply requires the location pointer to be incremented by the number of bytes given in the arguments. This keyword is especially useful following a label in the BSS segment to reserve memory for a variable. ALIGN keyword is used to advance the location pointer to the first 2-, 4-, or 8-byte boundary in memory to facilitate the assembly of words, longs, etc. at a suitable memory location. Finally, the keyword EXTERN announces that the routine or memory location mentioned will be made available to the linker for external references. The definition need not be in the current file; it can also be somewhere else, as long as the linker can handle the reference.

Although the assembler itself is fairly general, when it is used with the tracer some small points are worth noting. The assembler accepts keywords in either uppercase or lowercase but the tracer always displays them in uppercase. Similarly, the assembler accepts both "\r" (carriage return) and "\n" (line feed) as the new line indication, but the tracer always uses the latter. Moreover, although the assembler can handle programs split over multiple files, for use with the tracer, the entire program must be in a single file with extension ".$". Inside it, include files can be requested by the command

```
#include filename
```

In this case, the required file is also written in the combined ".$" file at the position of the request. The assembler checks whether the include file was already

processed and loads only one copy. This is especially useful if several files use the same header file. In this case, only one copy is included in the combined source file. In order to include the file, the *#include* must be the first token of the line without leading white space, and the file path must be between double quotes.

If there is a single source file, say *pr.s*, then it is assumed that the project name is *pr*, and the combined file will be *pr.$*. If there is more than one source file, then the basename of the first file is taken to be the projectname, and used for the definition of the *.$* file, which is generated by the assembler by concatenating the source files. This behavior can be overridden if the command line contains a "–o projname" flag before the first source file, in which case the combined file will be *projname*. $.

Note that there are some drawbacks in using include files and more than one source. It is necessary that the names of labels, variables and constants are different for all sources. Moreover, the file which is eventually assembled to the load file is the *projname*. $ file, so the line numbers mentioned by the assembler in case of errors and warnings are determined with respect to this file. For very small projects, it is sometimes simplest to put the entire program in one file and avoid *#include*.

C.5.3 Some Differences with Other 8088 Assemblers

The assembler, *as88*, is patterned after the standard UNIX assembler, and, as such, differs in some ways from the Microsoft Macro Assembler MASM and the Borland 8088 assembler TASM. Those two assemblers were designed for the MS-DOS operating system, and in places the assembler issues and the operating system issues are closely interrelated. Both MASM and TASM support all 8088 memory models allowed by MS-DOS. There is, for example, the **tiny** memory model, in which all code and data must fit in 64 KB, the **small** model, in which the code segment and the data segment each can be 64 KB, and **large** models, which contain multiple code and data segments. The difference between those models depends on the use of the segment registers. The large model allows far calls and changes in the DS register. The processor itself puts some restrictions on the segment registers (e.g., the CS register is not allowed as destination in a MOV instruction). To make tracing simpler, the memory model used in *as88* resembles the small model, although the assembler without the tracer can handle the segment registers without additional restrictions.

These other assemblers do not have a .BSS section, and initialize memory only in the DATA sections. Usually the assembler file starts with some header information, then the DATA section, which is indicated by the keyword .data, followed by the program text after the keyword .code. The header has a keyword title to name the program, a keyword .model to indicate the memory model, and a keyword .stack to reserve memory for the stack segment. If the intended binary is a *.com*

file, then the tiny model is used, all segment registers are equal, and at the head of this combined segment 256 bytes are reserved for a "Program Segment Prefix."

Instead of the .WORD, .BYTE, and ASCIZ directives, these assemblers have keywords DW for define word and DB for define byte. After the DB directive, a string can be defined inside a pair of double quotes. Labels for data definitions are not followed by a colon. Large chunks of memory are initialized by the DUP keyword, which is preceded by a count and followed by an initialization. For example, the statement

```
LABEL  DB 1000 DUP (0)
```

initializes 1000 bytes of memory with ASCII zero bytes at the label *LABEL*.

Furthermore, labels for subroutines are not followed by a colon, but by the keyword PROC. At the end of the subroutine, the label is repeated and followed by the keyword ENDP, so the assembler can infer the exact scope of a subroutine. Local labels are not supported.

The keywords for the instructions are identical in MASM, TASM, and *as88*. Also, the source is put after the destination in two operand instructions. However, it is common practice to use registers for the passing of arguments to functions, instead of on the stack. If, however, assembly routines are used inside C or C++ programs, then it is advisable to use the stack in order to comply with the C subroutine calling mechanism. This is not a real difference, since it is also possible to use registers instead of the stack for arguments in *as88*.

The biggest difference between the MASM, TASM and *as88* is in making system calls. The system is called in MASM and TASM by means of a system interrupt INT. The most common one is INT 21H, which is intended for the MS-DOS function calls. The call number is put in AX, so again we have passing of arguments in registers. For different devices there are different interrupt vectors, and interrupt numbers, such as INT 16H for the BIOS keyboard functions and INT 10H for the display. In order to program these functions, the programmer has to be aware of a great deal of device-dependent information. In contrast, the UNIX system calls available in *as88* are much easier to use.

C.6 THE TRACER

The tracer-debugger is meant to run on a 24 × 80 ordinary (VT100) terminal, with the ANSI standard commands for terminals. On UNIX or Linux machines, the terminal emulator in the X-window system usually meets the requirements. On Windows machines, the *ansi.sys* driver usually has to be loaded in the system initialization files as described below. In the tracer examples, we have already seen the layout of the tracer window. As can be seen in Fig. C-10, the tracer screen is subdivided into seven windows.

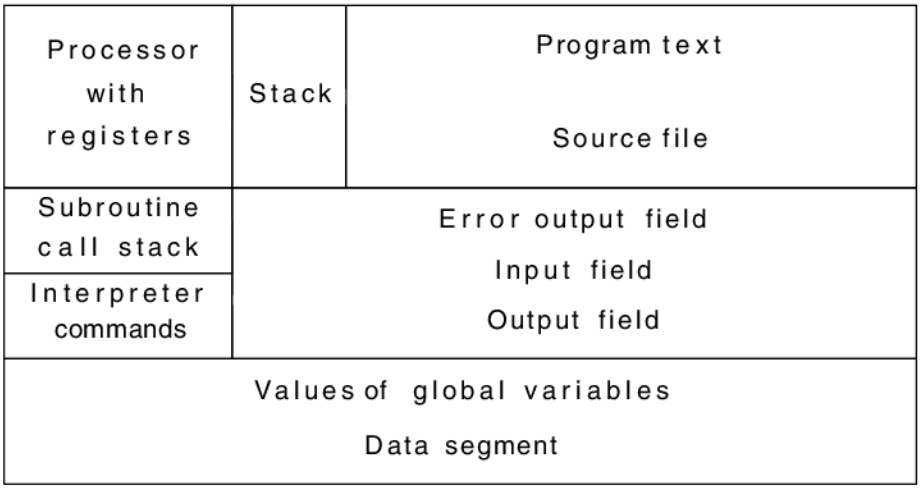

Figure C-10. The tracer's windows.

The upper left window is the processor window, which displays the general registers in decimal notation and the other registers in hexadecimal. Since the numerical value of the program counter is not very instructive, the position in the program source code with respect to the previous global label is supplied on the line below it. Above the program counter field, five condition codes are shown. Overflow is indicated by a "v", the direction flag by ">" for increasing and by "<" for decreasing. The sign flag is either "n", for negative or "p" for zero and positive. The zero flag is "z" if set, and the carry flag set is "c". A "–" indicates a cleared flag.

The upper middle window is used for the stack, displayed in hexadecimal. The stack pointer position is indicated with an arrow =>". Return addresses of subroutines are indicated by a digit in front of the hexadecimal value. The upper right window displays a part of the source file in the neighborhood of the next instruction to be executed. The position of the program counter is also indicated by an arrow "=>".

In the window under the processor, the most recent source code subroutine call positions are displayed. Directly under it is the tracer command window, which has the previously-issued command on top and the command cursor on the bottom. Note that every command needs to be followed by a carriage return (labeled Enter on PC keyboards).

The bottom window can contain six items of global data memory. Every item starts with a position relative to some label, followed by the absolute position in the data segment. Next comes a colon, then eight bytes in hexadecimal. The next 11 positions are reserved for characters, followed by four decimal word representations. The bytes, the characters, and the words each represent the same memory

contents, although for the character representation we have three extra bytes. This is convenient, because it is not clear from the start whether the data will be used as signed or unsigned integers, or as a string.

The middle right window is used for input and output. The first line is for error output of the tracer, the second line for input, and then there are some lines left for output. Error output is preceded by the letter "E", input by an "I", and standard output by a ">". In the input line there is an arrow "->" to indicate the pointer which is to be read next. If the program calls *read* or *getchar*, the next input in the tracer command line is going into the input field. Also, in this case, it is necessary to close the input line with a return. The part of the line which has not yet processed can be found after the "->" arrow.

Usually, the tracer reads both its commands and its input from standard input. However, it is also possible to prepare a file of tracer commands and a file of input lines to be read before the control is passed to the standard input. Tracer command files have extensions *.t* and input files *.i*. In the assembly language, both uppercase and lowercase characters can be used for keywords, system subroutines and pseudoinstructions. During the assembly process, a file with extension *.$* is made in which those lowercase keywords are translated into uppercase and carriage return characters are discarded. In this way, for each project, say, *pr* we can have up to six different files:

1. *pr.s* for the assembly source code.
2. *pr.$* for the composite source file.
3. *pr.88* for the load file.
4. *pr.i* for preset standard input.
5. *pr.t* for preset tracer commands.
6. *pr.#* for linking the assembly code to the load file.

The last file is used by the tracer to fill the upper right window and the program counter field in the display. Also, the tracer checks whether the load file has been created after the last modification of the program source; if not it issues a warning.

C.6.1 Tracer Commands

Figure C-11 lists the tracer commands. The most important ones are the single return command, which is at the first line of the table and which executes exactly one processor instruction, and the quit command *q*, at the bottom line of the table. If a number is given as a command, then that number of instructions is executed. The number *k* is equivalent to typing a return *k* times. The same effect is achieved if the number is followed by an exclamation mark, *!*, or an *X*.

The command *g* can be used to go to a certain line in the source file. There are three versions of this command. If it is preceded by a line number, then the tracer executes until that line is encountered. With a label */T*, with or without *+#*, the line number at which to stop is computed from the instruction label *T*. The *g* command, without any indication preceding it, causes the tracer to execute commands until the current line number is again encountered.

Address	Command	Example	Description
			Execute one instruction
#	, ! , X	24	Execute # instructions
/T+#	g , ! ,	/start+5g	Run until line # after label T
/T+#	b	/start+5b	Put breakpoint on line # after label T
/T+#	c	/start+5c	Remove breakpoint on line # after label T
#	g	108g	Execute program until line #
	g	g	Execute program until current line again
	b	b	Put breakpoint on current line
	c	c	Remove breakpoint on current line
	n	n	Execute program until next line
	r	r	Execute until breakpoint or end
	\&=	\&=	Run program until same subroutine level
	-	-	Run until subroutine level minus 1
	+	+	Run until subroutine level plus 1
/D+#		/buf+6	Display data segment on label+#
/D+#	d , !	/buf+6d	Display data segment on label+#
	R , CTRL L	R	Refresh windows
	q	q	Stop tracing, back to command shell

Figure C-11. The tracer commands. Each command must be followed by a carriage return (the Enter key). An empty box indicates that just a carriage return is needed. Commands with no *Address* field listed above have no address. The # symbol represents an integer offset.

The command */label* is different for an instruction label and a data label. For a data label, a line in the bottom window is filled or replaced with a set of data starting with that label. For an instruction label, it is equivalent to the *g* command. The label may be followed by a plus sign and a number (indicated by # in Fig. C-11), to obtain an offset from the label.

It is possible to set a **breakpoint** at an instruction. This is done with the command *b*, which can be optionally preceded by an instruction label, possibly with an offset. If a line with a breakpoint is encountered during execution, the tracer stops. To start again from a breakpoint, a return or run command is required. If the label and the number are omitted, then the breakpoint is set at the current line. The

breakpoint can be cleared by a breakpoint clear command, *c*, which can be preceded by labels and numbers, like the command *b*. There is a run command, *r*, in which the tracer executes until either a breakpoint, an exit call, or the end of the commands is encountered.

The tracer also keeps track of the subroutine level at which the program is running. This is shown in the window below the processor window and can also be seen through the indication numbers in the stack window. There are three commands that are based on these levels. The – command causes the tracer to run until the subroutine level is one less than the current level. What this command does is execute instructions until the current subroutine is finished. The converse is the + command, which runs the tracer until the next subroutine level is encountered. The = command runs until the same level is encountered, and can be used to execute a subroutine at the *CALL* command. If = is used, the details of the subroutine are not shown in the tracer window. There is a related command, *n*, which runs until the next line in the program is encountered. This command is especially useful when issued as a *LOOP* command; execution stops exactly when the bottom of the loop is executed.

C.7 GETTING STARTED

In this section, we will explain how to use the tools. First of all, it is necessary to locate the software for your platform. We have precompiled versions for Solaris, UNIX, for Linux and for Windows. The tools are located on the CD-ROM and on the Web at *www.prenhall.com/tanenbaum*. Once there, click on the *Companion Web Site* for this book and then click on the link in the left-hand menu. Unpack the selected zip file to a directory *assembler*. This directory and its subdirectories contain all the necessary material. On the CD-ROM, the main directories are *Bigendnx*, *LtlendNx*, and *MSWindos*, and in each there is a subdirectory *assembler* which contains the material. The three top-level directories are for Big-Endian UNIX (e.g. Sun workstations), Little-Endian UNIX (e.g., Linux on PCs), and Windows systems, respectively.

After unpacking or copying, the assembler directory should contain the following subdirectories and files: *READ_ME*, *bin*, *as_src*, *trce_src*, *examples*, and *exercise*. The precompiled sources can be found in the *bin* directory but, for convenience, there is also a copy of the binaries in the *examples* directory.

To get a quick preview of how the system works, go to the *examples* directory and type the command

```
t88 HlloWrld
```

This command corresponds to the first example in Sec. C.8.

The source code for the assembler is in the directory *as_src*. The source code files are in the language C, and the command *make* should recompile the sources.