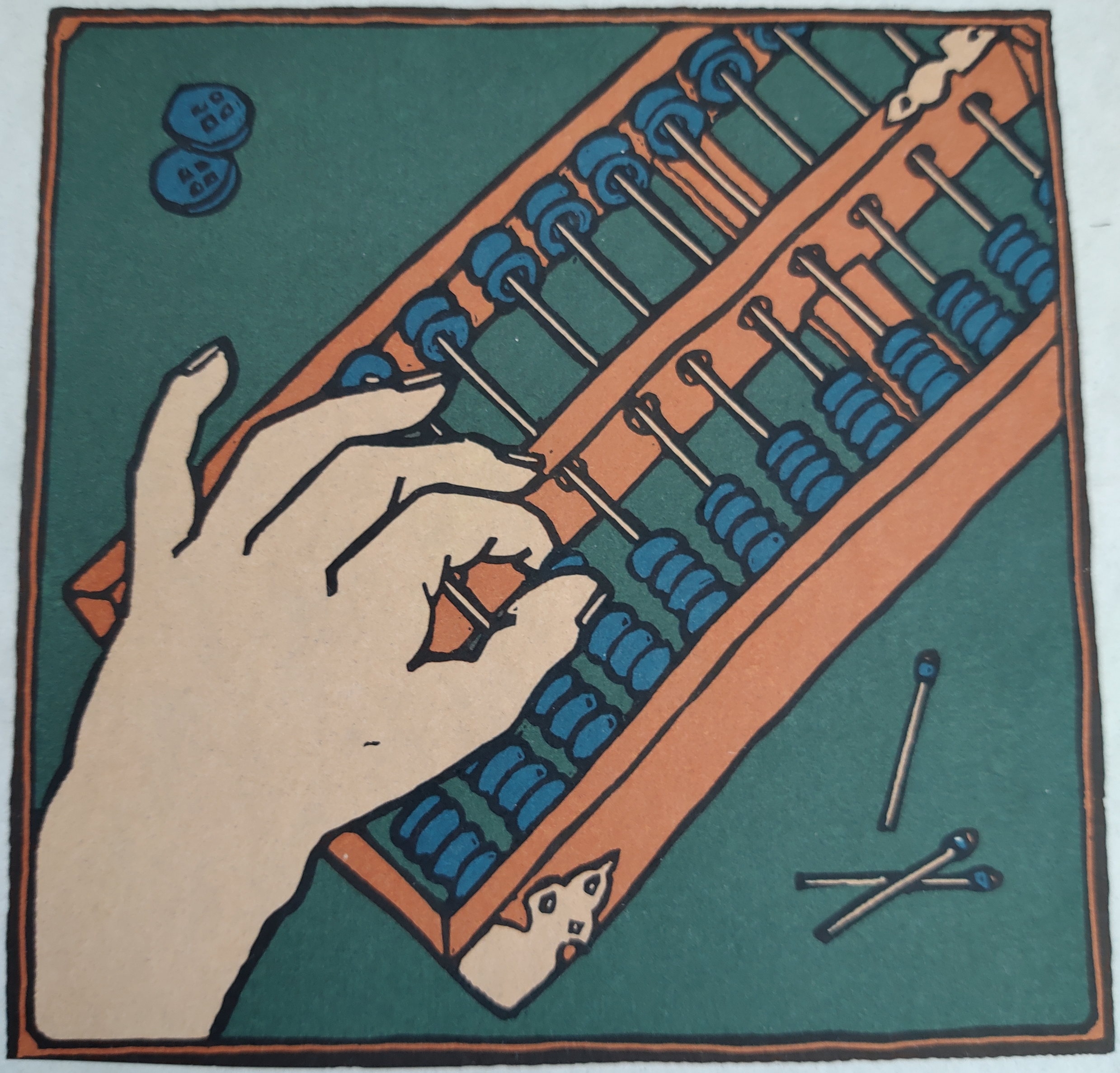
COMPUTER ORGANIZATION & DESIGN
THE HARDWARE / SOFTWARE INTERFACE



DAVID A. PATTERSON · JOHN L. HENNESSY

Trend Micro, Inc.
EX1008-1

Senior Editor: Bruce M. Spatz
Production Manager: Yonie Overton
Editorial Coordinator: Douglas Sery
Copyediting: Steve Hiatt and Gary Morris
Text Design: Ross Carron Design
Illustration: Alexander Teshin Associates
Composition/Color Separation/Postscript Programming: Edward W. Sznyter, Babel Press
Cover Design: David Lance Goines
Additional Cover Mechanical Art: Patty King
Chapter Opener Illustrations: Jo Jackson
Indexing: Steve Rath
Proofreading: Gary Morris
Electronic Prepress: The Courier Connection
Printer: Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104

© 1994 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

97 96 95 94 5 4 3 2

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

Advice, Praise, and Errors: Any correspondence related to this publication or intended for the authors should be addressed to the editorial offices of Morgan Kaufmann Publishers, Inc., Dept. P&H APE. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction at the time of their implementation in a reprint. Electronic mail can be sent to errors@cs.berkeley.edu.

Instructor Support: For information on the SPIM software simulator and other instructor materials available to adoptors, please contact the editorial offices of Morgan Kaufmann Publishers, Inc.

Cataloging-in-Publication Data

Patterson, David A.
Computer organization and design: the hardware/software interface
/ David A. Patterson, John L. Hennessy.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-281-X
1. Computer organization. 2. Computers--Design and construction.
3. Computer interfaces. I. Hennessy, John L. II. Title
QA76.9.C643P37 1994
004.2'2--dc20

94-17639
CIP

Trend Micro, Inc.
EX1008-2

take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast.

Chapters 5 and 6 show the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is a key to program performance.

Example

It is the compiler's job to associate program variables with registers. Take, for instance, the C statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` can be assigned to the registers `$16`, `$17`, `$18`, `$19`, and `$20`, respectively. What is the compiled MIPS assembly code?

Answer

The compiled program is

```
add $8,$17,$18      # register $8 contains g+h
add $9,$19,$20      # register $9 contains i+j
sub $16,$8,$9       # f gets $8-$9, or (g+h)-(i+j)
```

Registers `$8` and `$9` correspond to `t0` and `t1` in the earlier example.

Programming languages have simple variables that contain single data elements as in these examples, but they also have more complex data structures such as arrays. These complex data structures can contain many more data elements than there are registers in a machine. How can a computer represent and access such large structures?

Recall the five components of a computer introduced in Chapter 1 and depicted on the second page of this chapter (page 93). The processor can store only a small amount of data in registers, but computer memory contains millions of data elements. Hence data structures, like arrays, are kept in memory.

As explained above, arithmetic operations occur only on registers in MIPS; thus MIPS must include instructions that transfer data between memory and registers. Such instructions are called *data transfer* instructions. To access a word in memory, the instruction must supply its *address*. Memory is really just a large, single-dimensional array, with the address acting as the index to that array. Addresses start at 0. For example, in Figure 3.2, the address of the third data element is 2, and the value of Memory[2] is 1000.

Trend Micro, Inc.
EX1008-3

The Big Picture

Today's computers are built on two key principles:

1. Instructions are represented as numbers; and
2. Programs can be stored in memory to be read or written just like numbers.

This is the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 3.7 shows the power of the concept; specifically, memory can contain the C code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

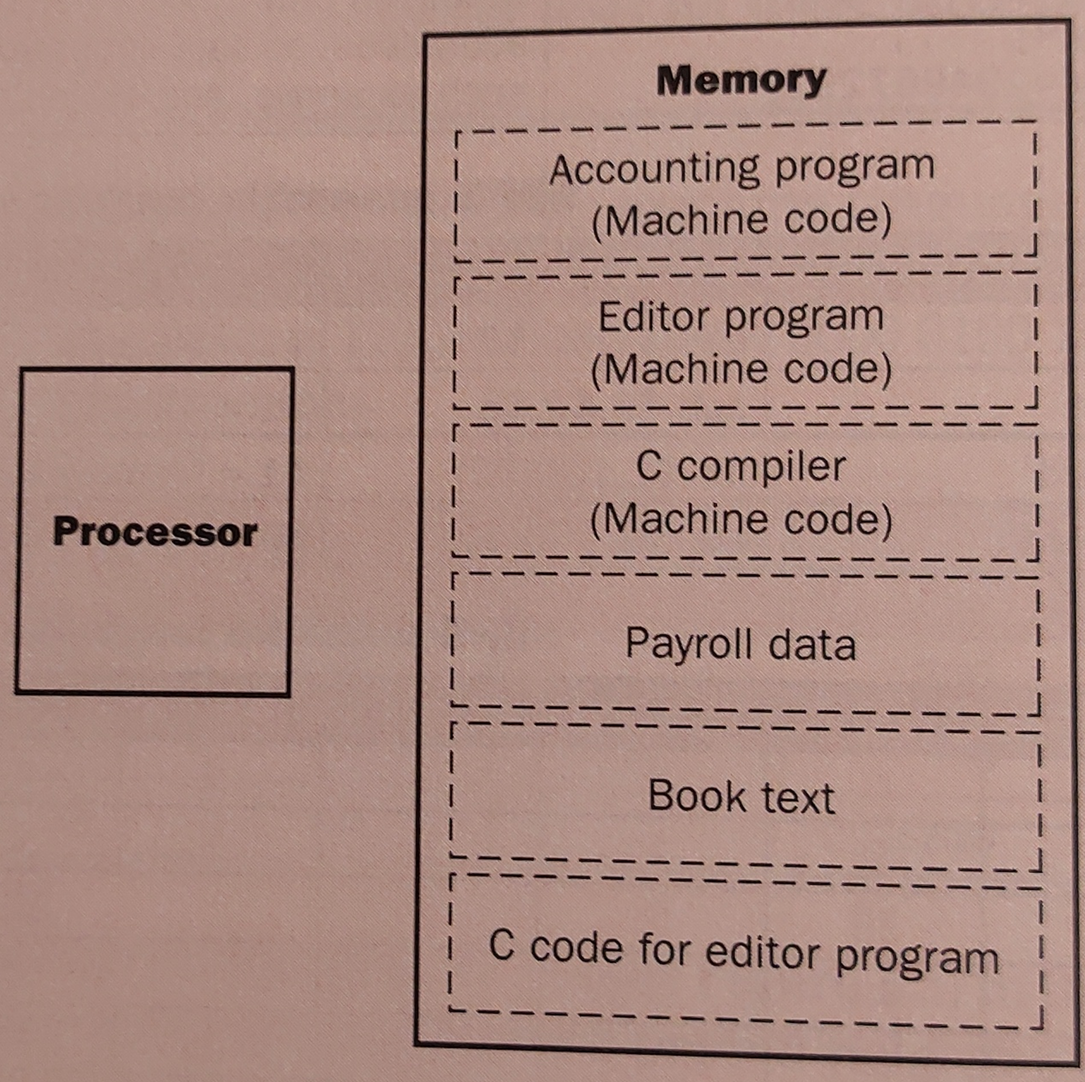


FIGURE 3.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the machine can understand.

Trend Micro, Inc.
EX1008-4

Hardware Software Interface

Recall from Chapter 1 that the symbolic representation of instructions is called the *assembly language* of a computer. To avoid confusion between the symbolic and numerical forms of programs, we traditionally call the numerical equivalent that the machine executes the *machine language*. The translation from assembly language to machine language is called *assembly*, and the program that translates is called an *assembler*. Figure 3.8 shows the translation hierarchy: A C program is first translated into an assembly language program by the compiler, then the assembler translates the assembly language program into machine language. The program that places the machine language program into memory for execution is called a *loader* (see section A.4 on page A-19).

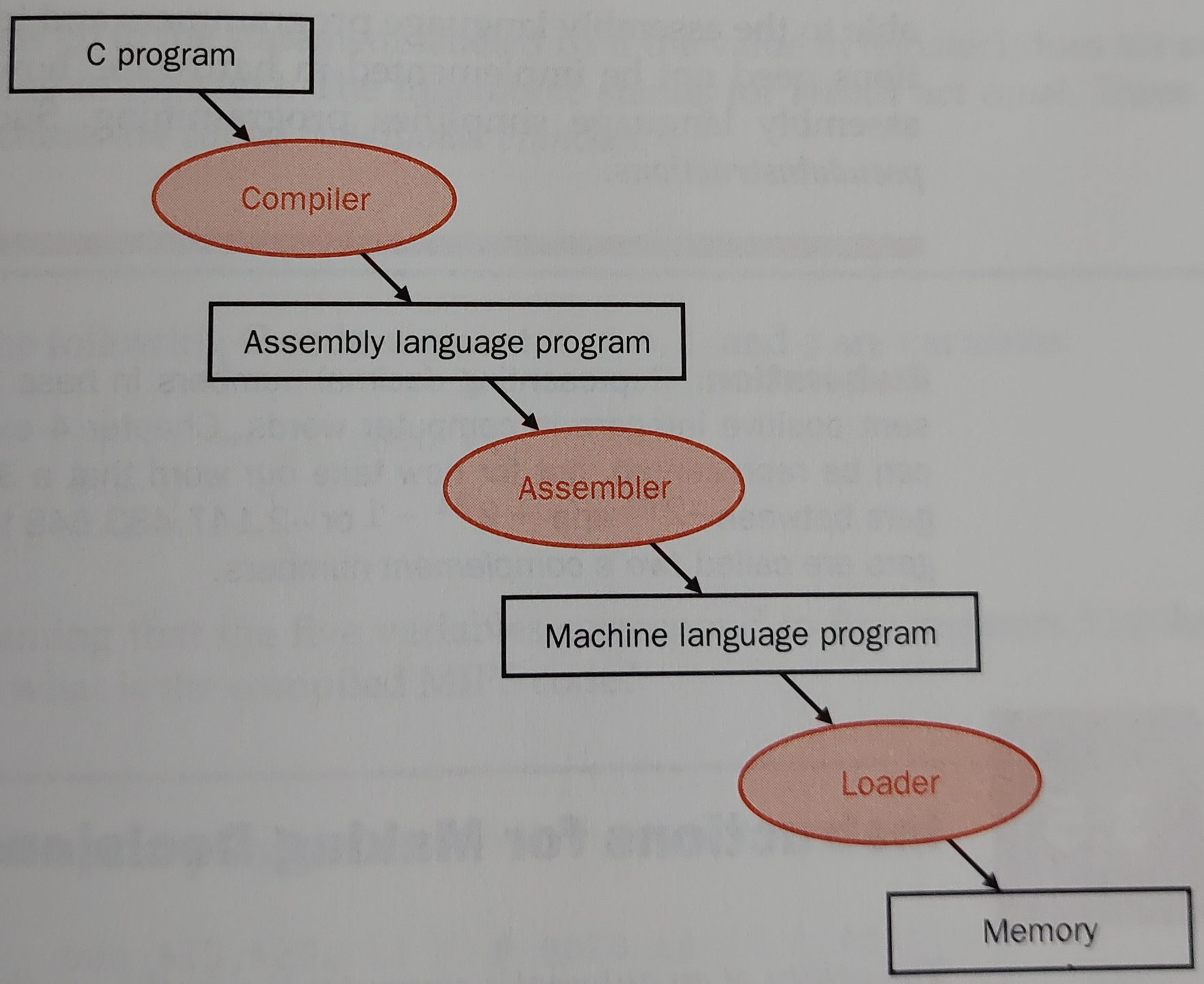


FIGURE 3.8 A translation hierarchy. A high-level language program is first compiled into an assembly language program and then assembled into a machine language program. The loader then places the machine code into the proper memory locations for execution by the processor.

Trend Micro, Inc.
EX1008-5

Assembly language is obviously a great improvement over writing numbers. In addition to replacing symbols with numbers, assemblers treat common variations of machine language instructions as if they were instructions in their own right.

For example, the MIPS hardware makes sure that register $0 always has the value 0. That is, whenever register $0 is used, it supplies a 0, and the programmer cannot change the value of register $0. Register $0 is used to create the assembly language instruction `move` that copies the contents of one register to another. Thus the MIPS assembler accepts this instruction even though it is not found in the MIPS architecture:

```
move $8,$18          # register $8 gets register $18
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
add  $8,$0,$18       # register $8 gets 0 + register $18
```

The MIPS assembler effectively increases the number of instructions available to the assembly language programmers and to compilers. These instructions need not be implemented in hardware; however, their appearance in assembly language simplifies programming. Such instructions are called *pseudoinstructions*.

Elaboration: Representing decimal numbers in base 2 gives an easy way to represent positive integers in computer words. Chapter 4 explains how negative numbers can be represented, but for now take our word that a 32-bit word can represent integers between -2^{31} and $+2^{31} - 1$ or –2,147,483,648 to +2,147,483,647. Such integers are called *two's complement* numbers.

3.5 Instructions for Making Decisions

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer

Trend Micro, Inc.
EX1008-6

[instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during the computation, different instructions are executed. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *goto* statements and labels. MIPS includes two decision making instructions, similar to an *if* statement with a *goto*:

```
beq register1, register2, L1
```

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic stands for *branch equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic stands for *branch not equal*. These two instructions are called *conditional branches*.

Example

In the following C code segment, `f`, `g`, `h`, `i`, and `j` are variables:

```
     if (i == j) goto L1;
     f = g + h;
L1:  f = f - i;
```

Assuming that the five variables correspond to five registers `$16` through `$20`, what is the compiled MIPS code?

Answer

The compiled program is

```
     beq $19,$20, L1    # goto L1 if i equals j
     add $16,$17,$18    # f = g + h (skipped if i equals j)
L1:  sub $16,$16,$19    # f = f - i (always executed).
```

Trend Micro, Inc.
EX1008-7

Instructions are stored in memory in stored-program computers; hence, instructions must have memory addresses just like other words in memory. The label L1 thus corresponds to the address of the subtract instruction. Notice that the assembler relieves the compiler or the assembly language programmer from the tedium of calculating branch addresses, just as it does for calculating data addresses for loads and stores (see section A.1 on page A-3 in Appendix A or the example on page 106).

Hardware Software Interface

Compilers frequently create branches and labels when they do not appear in the programming language. Using the same variables and registers from the previous example, the C code:

```
if (i == j) f = g + h; else f = g - h;
```

```
      bne   $19,$20,Else    # goto Else if i ≠ j
      add   $16,$17,$18     # f = g + h (skipped if i ≠ j)
      j     Exit            # go to Exit
Else: sub   $16,$17,$18     # f = g - h (skipped if i = j)
Exit:
```

Figure 3.9 shows the form of the C code that the compiler must translate into MIPS code. The second instruction above performs the "then" part of the *if* statement and the fourth instruction performs the "else" part. When `i ≠ j`, the first instruction branches to the label `Else`, going around the "then" part. To avoid executing the fourth instruction when `i = j`, we must branch around it to the label `Exit`. This introduces another kind of branch, sometimes called an *unconditional branch*. This instruction says that the machine always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as `j`.

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks in both cases.

Trend Micro, Inc.
EX1008-8