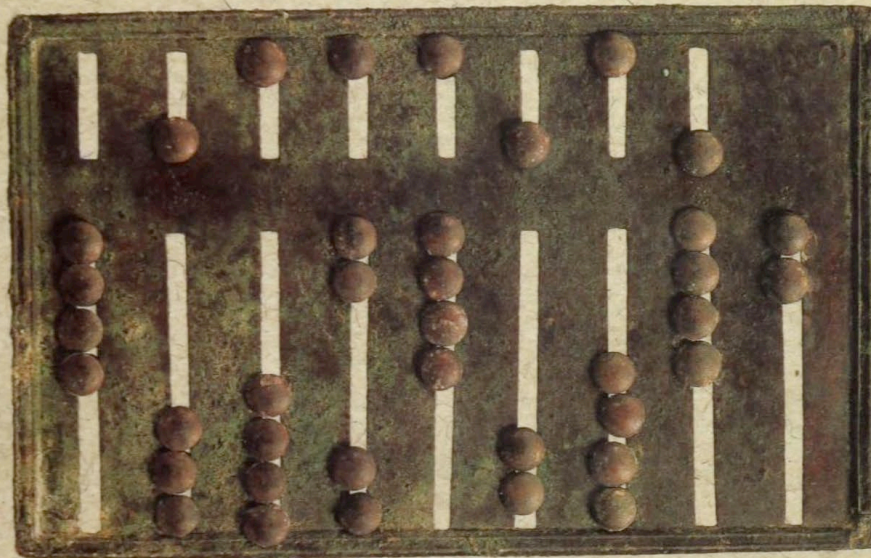


**Second
Edition**

Computer Organization & Design

THE HARDWARE / SOFTWARE INTERFACE



David A. Patterson and John L. Hennessy

Sponsoring Editor Denise Penrose
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Coordinator Jane Elliott
Text and Cover Design Ross Carron Design
Illustration Alexander Teshin Associates, with second edition modifications by Dartmouth Publishing, Inc.
Chapter Opener Illustrations Canary Studios
Copyeditor Ken DellaPenta
Composition Nancy Logan
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA

Telephone 415/392-2665
Facsimile 415/982-2665
Email mkp@mkp.com
WWW <http://www.mkp.com>
Order toll free 800/745-7323

© 1998 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

02 01 00 99 5 4

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 sent electronically to cod2bugs@mkp.com. 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.

Library of Congress Cataloging-in-Publication Data

Patterson, David A.

Computer organization and design : the hardware/software interface
/ David A. Patterson, John L. Hennessy.—2nd ed.
p. cm.

Includes bibliographical references and index.

ISBN 1-55860-428-6 (cloth).—ISBN 1-55860-491-X (paper)

1. Computer organization. 2. Computers—Design and construction.

3. Computer interfaces. I. Hennessy, John L. II. Title
QA76.9.C643H46 1997
004.2'2—dc21 97-16050

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix C, when the implementation of this finite state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

5.5

Microprogramming: Simplifying Control Design

For the control of our simple MIPS subset, a graphical representation of the finite state machine, as in Figure 5.42, is certainly adequate. We can draw such a diagram on a single page and translate it into equations (see Appendix C) without generating too many errors. Consider instead an implementation of the full MIPS instruction set, which contains over 100 instructions (see Appendix A). In one implementation, instructions take from 1 clock cycle to over 20 clock cycles. Clearly, the control function will be much more complex. Or consider an instruction set with more instructions of widely varying classes: The control unit could easily require thousands of states with hundreds of different sequences. For example, the Intel 80x86 instruction set has many more addressing mode combinations, as well as a much larger set of opcodes.

In such cases, specifying the control unit with a graphical representation will be cumbersome, since the finite state machine can contain hundreds to thousands of states, and even more arcs! The graphical representation—although useful for a small finite state machine—will not fit on a page, let alone be understandable, when it becomes very large. Programmers know this phenomenon quite well: As programs become large, additional structuring techniques (for example, procedures and modules) are needed to keep the programs comprehensible. Of course, specifying complex control functions directly as equations, without making any mistakes, becomes essentially impossible.

Can we use some of the ideas from programming to help create a method of specifying the control that will make it easier to understand as well as to design? Suppose we think of the set of control signals that must be asserted in a state as an instruction to be executed by the datapath. To avoid confusing the instructions of the MIPS instruction set with these low-level control instructions, the latter are called *microinstructions*. Each microinstruction defines the

set of datapath control signals that must be asserted in a given state. Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction.

In addition to defining which control signals must be asserted, we must also specify the sequencing—what microinstruction should be executed next? In the finite state machine shown in Figure 5.42 on page 396, the next state is determined in one of two different ways. Sometimes a single next state follows the current state unconditionally. For example, state 1 always follows state 0, and the only way to reach state 1 is via state 0. In other cases, the choice of the next state depends on the input. This is true in state 1, which has four different successor states.

When we write programs, we also have an analogous situation. Sometimes a group of instructions should be executed sequentially, and sometimes we need to branch. In programming, the default is sequential execution, while branching must be indicated explicitly. In describing the control as a program, we also assume that microinstructions written sequentially are executed in sequence, while branching must be indicated explicitly. The default sequencing mechanism can still be implemented using a structure like the one in Figure 5.43 on page 398; however, it is often more efficient to implement the default sequential state using a counter. We will see how such an implementation looks at the end of this section.

Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called *microprogramming*. The key idea is to represent the asserted values on the control lines symbolically, so that the microprogram is a representation of the microinstructions, just as assembly language is a representation of the machine instructions. In choosing a syntax for an assembly language, we usually represent the machine instructions as a series of fields (opcode, registers, and offset or immediate field); likewise, we will represent a microinstruction syntactically as a sequence of fields whose functions are related.

Defining a Microinstruction Format

The microprogram is a symbolic representation of the control that will be translated by a program to control logic. In this way, we can choose how many fields a microinstruction should have and what control signals are affected by each field. The format of the microinstruction should be chosen so as to simplify the representation, making it easier to write and understand the microprogram. For example, it is useful to have one field that controls the ALU and a set of three fields that determine the two sources for the ALU operation as well as the destination of the ALU result. In addition to readability, we would also like the microprogram format to make it difficult or impossible to write inconsistent microinstructions. A microinstruction is inconsistent if it requires that a given control signal be set to two different values. We will see an example of how this could happen shortly.