

S E C O N D E D I T I O N

Computer Organization and Design

T H E H A R D W A R E / S O F T W A R E I N T E R F A C E

John L. Hennessy

Stanford University

David A. Patterson

University of California, Berkeley

With a contribution by
James R. Larus
University of Wisconsin



Morgan Kaufmann Publishers, Inc.
San Francisco, California

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 10 9 8 7 6

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

Arrays of Logic Elements

Many of the combinational operations to be performed on data have to be done to an entire word (32 bits) of data. Thus we often want to build an array of logic elements, which we can represent simply by showing that a given operation will happen to an entire collection of inputs. For example, we saw on page B-9 what a 1-bit multiplexor looked like, but inside a machine, much of the time we want to select between a pair of *buses*. A bus is a collection of data lines that is treated together as a single logical signal. (The term *bus* is also used to indicate a shared collection of lines with multiple sources and uses, especially in Chapter 8, where I/O buses were discussed.)

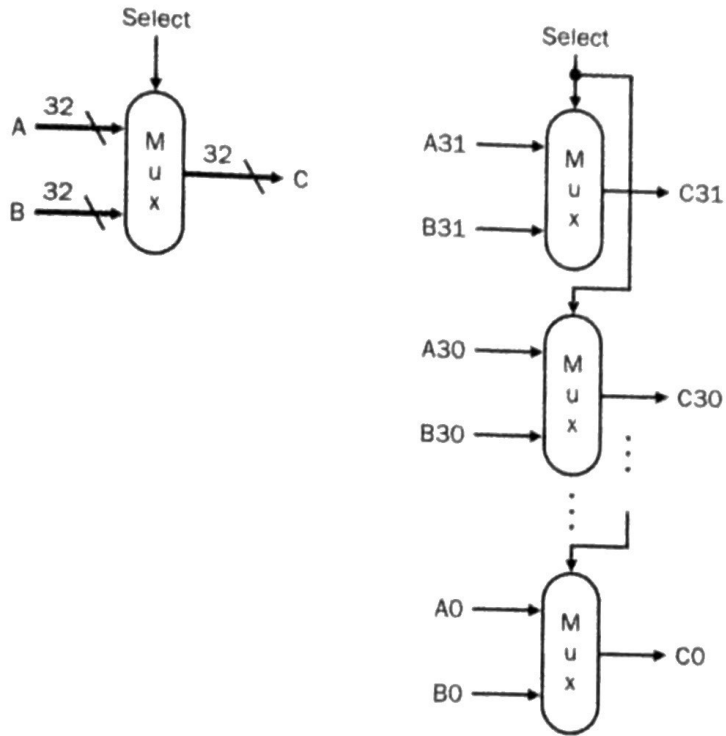
For example, in the MIPS instruction set, the result of an instruction that is written into a register can come from one of two sources. A multiplexor is used to choose which of the two buses (each 32 bits wide) will be written into the Result register. The 1-bit multiplexor, which we showed earlier, will need to be replicated 32 times. We indicate that a signal is a bus rather than a single 1-bit line by showing it with a thicker line in a figure. Most buses are 32 bits wide; those that are not are explicitly labeled with their width. When we show a logic unit whose inputs and outputs are buses, this means that the unit must be replicated a sufficient number of times to accommodate the width of the input. Figure B.8 shows how we draw a multiplexor that selects between a pair of 32-bit buses and how this expands in terms of 1-bit-wide multiplexors. Sometimes we need to construct an array of logic elements where the inputs for some elements in the array are outputs from earlier elements. For example, this is how a multibit-wide ALU is constructed. In such cases, we must explicitly show how to create wider arrays, since the individual elements of the array are no longer independent, as they are in the case of a 32-bit-wide multiplexor.

B.4

Clocks

Before we discuss memory elements and sequential logic, it is useful to discuss briefly the topic of clocks. This short section introduces the topic and is similar to the discussion found at the beginning of Chapter 5. More details on clocking and timing methodologies are presented in section B.7.

Clocks are needed in sequential logic to decide when an element that contains state should be updated. A clock is simply a free-running signal with a fixed *cycle time*; the *clock frequency* is simply the inverse of the cycle time. As shown in Figure B.9, the *clock cycle time* or *clock period* is divided into two portions: when the clock is high and when the clock is low. In this text, we use only *edge-triggered clocking*. This means that all state changes occur on a clock edge. We use an edge-triggered methodology because it is simpler to explain. Depending on the technology, it may or may not be the best choice for a clocking methodology.



a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

FIGURE B.8 A multiplexor is arrayed 32 times to perform a selection between two 32-bit inputs. Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

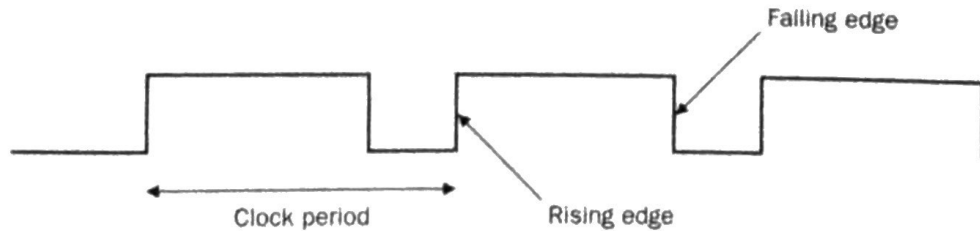


FIGURE B.9 A clock signal oscillates between high and low values. The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.

In an edge-triggered methodology, either the rising edge or the falling edge of the clock is *active* and causes state changes to occur. As we will see in the next section, the state elements in an edge-triggered design are implemented so that the contents of the state elements only change on the active clock edge. The choice of which edge is active is influenced by the implementation technology and does not affect the concepts involved in designing the logic.

The major constraint in a clocked system, also called a *synchronous system*, is that the signals that are written into state elements must be *valid* when the active clock edge occurs. A signal is valid if it is stable (i.e., not changing) and the value will not change again until the inputs change. Since combinational circuits cannot have feedback, if the inputs to a combinational logic unit are not changed, the outputs will eventually become valid. Figure B.10 shows the relationship among the state elements and the combinational logic blocks in a synchronous, sequential logic design. The state elements, whose outputs change only on the clock edge, provide valid inputs to the combinational logic block. To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize. This constraint sets a lower bound on the length of the clock period. In the rest of this appendix, as well as in Chapters 5 and 6, we usually omit the clock signal, since we are assuming that all state elements are updated on the same clock edge. Some state elements will be written on every clock edge, while others will be written only under certain conditions (such as a register being updated). In such cases, we will have an explicit write signal for that state element. The write signal must still be gated with the clock so that the update occurs only on the clock edge if the write signal is active. We will see how this is done and used in the next section.

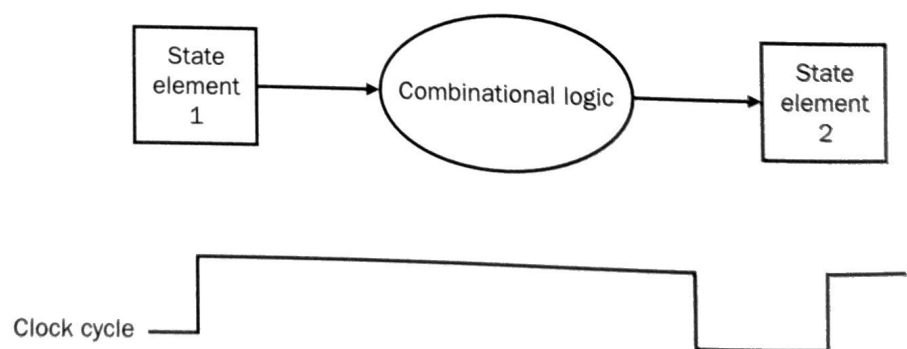


FIGURE B.10 The inputs to a combinational logic block come from a state element, and the outputs are written into a state element. The clock edge determines when the contents of the state elements are updated.

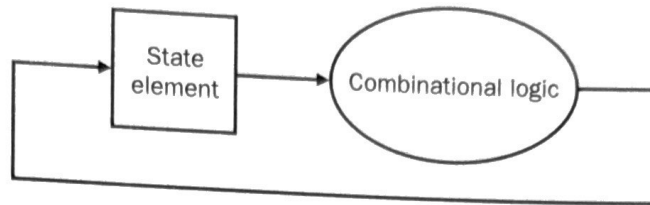


FIGURE B.11 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to undetermined data values. Of course, the clock cycle must still be long enough so that the input values are stable when the active clock edge occurs.

One other advantage of an edge-triggered methodology is that it is possible to have a state element that is used as both an input and output to the same combinational logic block, as shown in Figure B.11. In practice, care must be taken to prevent races in such situations and to ensure that the clock period is long enough; this topic is discussed further in section B.7.

Now that we have discussed how clocking is used to update state elements, we can discuss how to construct the state elements.

B.5

Memory Elements

In this section, we discuss the basic principles behind memory elements, starting with flip-flops and latches, moving on to register files, and finally to memories. All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.

The simplest type of memory elements are *unclocked*; that is, they do not have any clock input. Although we only use clocked memory elements in this text, an unclocked latch is the simplest memory element, so let's look at this circuit first. Figure B.12 shows an *S-R latch* (set-reset latch), built from a pair of NOR gates (OR gates with inverted outputs). The outputs Q and $\bar{Q}$ represent the value of the stored state and its complement. When neither S nor R are asserted, the cross-coupled NOR gates act as inverters and store the previous values of Q and $\bar{Q}$. For example, if the output, Q , is true, then the bottom inverter produces a false output (which is $\bar{Q}$), which becomes the input to the top inverter, which produces a true output, which is Q , and so on. If S is asserted then the output Q will be asserted and $\bar{Q}$ will be deasserted, while if R is asserted, then the output $\bar{Q}$ will be asserted and Q will be deasserted. When S and R are both deasserted the last values of Q and $\bar{Q}$ will continue to be stored in the