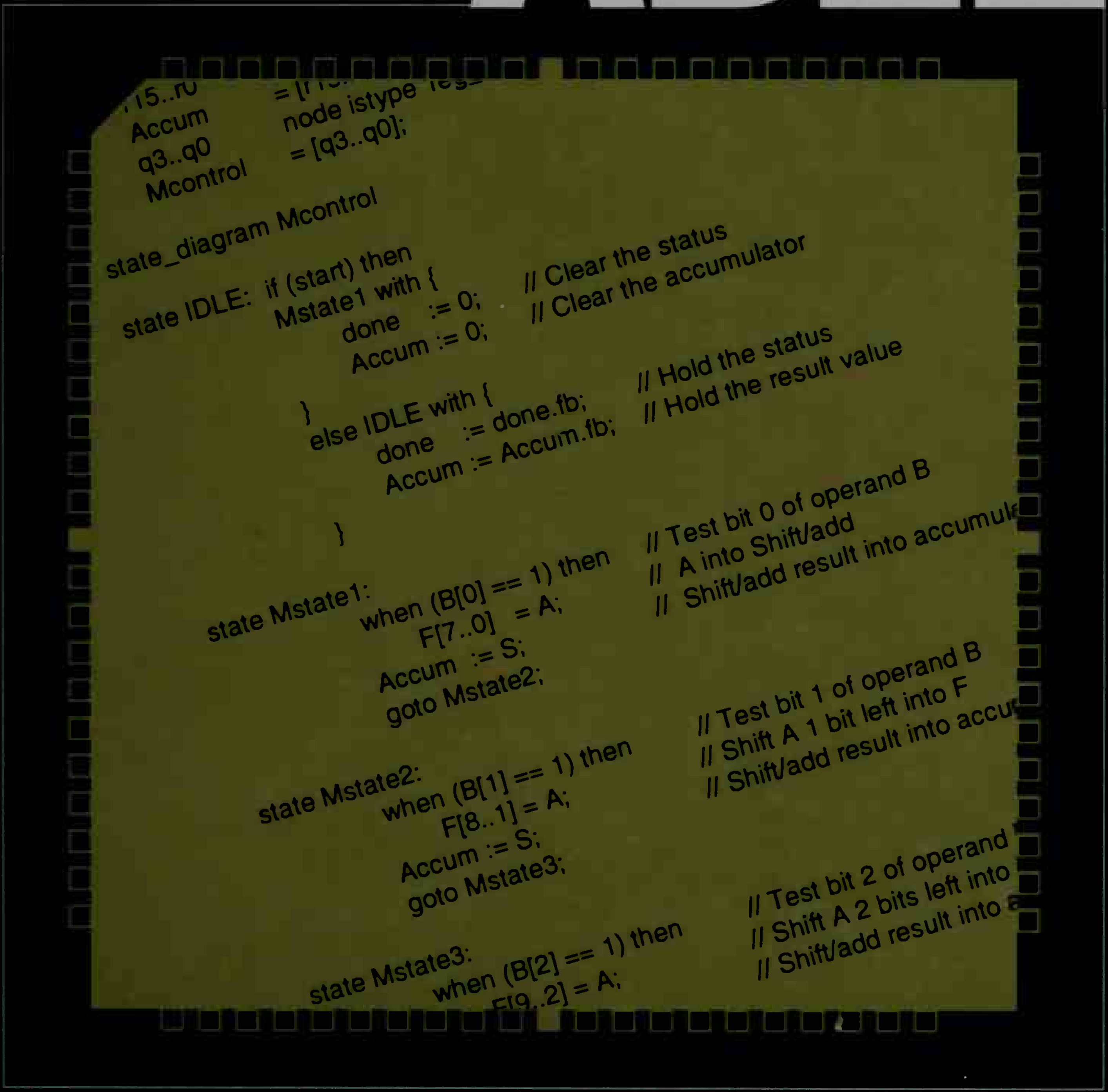
DISKETTES INCLUDED

Includes ABEL Software

DIGITAL DESIGN USING ABEL™



```
 15..r0          = [r15.
Accum            node istype reg
q3..q0           = [q3..q0];
Mcontrol

state_diagram Mcontrol

state IDLE:  if (start) then
              Mstate1 with {
                 done    := 0;      // Clear the status
                 Accum := 0;        // Clear the accumulator

              }
              else IDLE with {
                 done    := done.fb;      // Hold the status
                 Accum := Accum.fb;       // Hold the result value

              }

state Mstate1:
             when (B[0] == 1) then        // Test bit 0 of operand B
                 F[7..0]  = A;            //  A into Shift/add
                 Accum  := S;             //  Shift/add result into accumul
                 goto Mstate2;

state Mstate2:
             when (B[1] == 1) then        // Test bit 1 of operand B
                 F[8..1] = A;             // Shift A 1 bit left into F
                 Accum := S;              // Shift/add result into accu
                 goto Mstate3;

state Mstate3:
             when (B[2] == 1) then        // Test bit 2 of operand
                 F[9..2] = A;             // Shift A 2 bits left into
                                          // Shift/add result into a
```

DAVID PELLERIN / MICHAEL HOLLEY

Digital Design Using ABEL

David Pellerin
and
Michael Holley

PTR Prentice Hall
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data
Pellerin, David
Digital design using ABEL / David Pellerin, Michael Holley
p. cm.
Includes index.
ISBN 0-13-605874-4
1. Programmable logic devices--Design--Data processing. 2. Logic design--Data processing 3. ABEL (Computer program language)
I. Holley, Michael. II. Title
TK7872.L64P39 1994
621.39'5--dc20 93-39558
CIP

Editorial/production supervision: *Camille Trentacoste*
Cover design: *Wanda Lubelska*
Manufacturing manager: *Alexis Heydt*
Acquisitions editor: *Karen Gettman*

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

0-13-605874-4

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface, vii

Chapter 1 **Introduction, 1**

1.1 DESIGN ENTRY USING ABEL, 1

Chapter 2 **Combinational Circuits, 9**

2.1 LOGIC CIRCUITS AND LOGIC FUNCTIONS, 9

2.2 BOOLEAN ALGEBRA, 14

2.3 USING BOOLEAN ALGEBRA IN ABEL, 21

2.4 REFERENCES, 22

Chapter 3 **Logic Minimization, 23**

3.1 TWO-LEVEL MINIMIZATION, 23

3.2 ON-SETS, OFF-SETS, AND DC-SETS, 40

3.3 MULTIPLE-OUTPUT MINIMIZATION, 43

3.4 MULTILEVEL OPTIMIZATION, 44

3.5 OPTIMIZING WITH EXCLUSIVE-ORS, 47

3.6 REFERENCES, 52

Chapter 4 **Sequential Circuits, 55**

4.1 SYNCHRONOUS SEQUENTIAL CIRCUITS, 55

4.2 STATE MACHINE LOGIC MINIMIZATION, 67

4.3 METASTABILITY, 68

4.4 ASYNCHRONOUS STATE MACHINES, 70

4.5 REFERENCES, 76

Chapter 5 **Using ABEL, 77**

5.1 LANGUAGE OVERVIEW, 77

5.2 USING ABEL EQUATIONS, 78

5.3 USING TRUTH TABLES, 81

5.4 USING STATE DIAGRAMS, 85

5.5 REFERENCES, 88

Chapter 6 **Basic TTL Logic Functions, 91**

6.1 BUFFERS, 91

6.2 LATCHES, 94

6.3 REGISTERS, 98

6.4 SHIFT REGISTERS, 98

6.5 MULTIPLEXERS, 104

6.6 DEMULTIPLEXERS, 104

6.7 REFERENCES, 108

Chapter 7 **Counters, 109**

7.1 T FLIP-FLOP COUNTERS, 109

7.2 D FLIP-FLOP COUNTERS, 113

7.3 LOADABLE COUNTERS, 114

7.4 COUNTER RESET SCHEMES, 115

7.5 POLARITY CONSIDERATIONS, 117

7.6 SR FLIP-FLOP COUNTERS, 119

7.7 UP AND DOWN COUNTERS, 119

7.8 ARBITRARY LENGTH COUNTERS, 122

7.9 CHAINED COUNTERS, 127

7.10 WAVEFORM GENERATOR, 127

7.11 REFERENCES, 131

Chapter 8 **Decoders and Comparators, 133**

8.1 ADDRESS DECODERS, 133

8.2 COMPARATORS, 135

8.3 REFERENCES, 145

Chapter 9 **Arithmetic Functions, 147**

9.1 PARITY DETECTION, 147

9.2 ADDERS, 150

9.3 HIGH-LEVEL ADDER EQUATIONS, 152

9.4 FAST CARRY (LOOK-AHEAD) ADDERS, 153

9.5 CHAINED ADDERS, 153

9.6 SUBTRACTERS, 155

9.7 FIBONACCI SEQUENCE GENERATOR, 155

9.8 BINARY MULTIPLICATION, 158

9.9 REFERENCES, 163

Chapter 10 **Code Converters, 165**

10.1 BCD TO SEVEN-SEGMENT DECODER, 165

10.2 GRAY-CODE CONVERTERS, 167

10.3 CRC GENERATORS, 167

10.4 REFERENCES, 175

Chapter 11 **Describing State Machines, 177**

11.1 STATE TRANSITION TABLES, 177

11.2 USING STATE DIAGRAMS, 180

11.3 POWER-UP AND ILLEGAL STATES, 181

11.4 ADDING STATE BITS TO SAVE LOGIC, 185

11.5 ONE-HOT STATE ENCODING, 188

11.6 STATE MACHINE PARTITIONING, 188

11.7 ASYNCHRONOUS STATE MACHINES, 198

11.8 REFERENCES, 201

Chapter 12 **A Video Frame Grabber, 203**

12.1 GENERAL DESIGN REQUIREMENTS, 203

12.2 THE VIDEO FRAME GRABBER SYSTEM, 207

12.3 FRAME GRABBER CONTROL SOFTWARE, 221

12.4 REFERENCES, 224

Appendix A **The ABEL Language, 225**

A.1 LANGUAGE OVERVIEW, 225

A.2 DECLARATIONS, 228

A.3 EXPRESSIONS, 236

A.4 EQUATIONS, 250

A.5 STATE DIAGRAMS, 256

A.6 TRUTH TABLES, 262

A.7 TEST VECTORS, 265

A.8 MACROS, 266

A.9 DIRECTIVES, 268

A.10 REFERENCES, 278

Appendix B **The ABEL Software, 279**

Appendix C **Mach 215 Special Device Information, 283**

C.1 MACH 215 DEVICE ARCHITECTURE, 283

C.2 DESIGN STRATEGIES, 288

C.3 DESIGN EXAMPLES, 293

C.4 REFERENCES, 306

Appendix D **Supplementary Listings for Chapter 12, 307**

VFG.C, 307

VFG2BMP.C, 309

Index, 315

Preface

ABEL (Advanced Boolean Expression Language) was introduced by Data I/O Corporation in 1984 and is now used by tens of thousands of digital circuit designers worldwide. The ABEL language was designed to allow small to medium-sized circuits (ranging in size from a few logic gates up to circuits composed of many thousands of gates) to be described and implemented in programmable logic devices (PLDs).

PLDs have become extremely common in modern digital systems, and the increasing popularity of these devices has led to an explosion of new device architectures. In addition, the densities of the newest devices have made it impractical to attempt a complex PLD-based design without design entry software. ABEL allows you to quickly and efficiently create large designs and to use one design description method for virtually all PLDs available.

Digital Design Using ABEL leads you through the basics of HDL-based digital design and provides dozens of examples of how ABEL can be used for digital applications. One primary reason for ABEL's popularity is its simplicity. While more complex and universal hardware description languages (such as VHDL and Verilog) may become more widely used design languages at some point in the future, these languages are primarily intended for circuit modeling, rather than for general-purpose logic design. ABEL, on the other hand, was designed specifically for the purpose of describing digital circuits for implementation in programmable logic. This specific purpose has made it possible to keep the language relatively simple and easy to use and makes ABEL a good language for learning concepts of HDL-based design.

The book begins with a discussion of the fundamentals of digital design, with special emphasis placed on techniques that are appropriate for users of ABEL. Chapters 1 through 4 cover topics such as Boolean algebra, logic minimization, and sequential circuit design and help the reader to better understand how traditional digital design techniques are applied, as well as shedding light on the processing and optimizations that are performed by the ABEL design software. Chapter 5 introduces the ABEL language and shows by example how the various forms of description (equations, truth tables, and state diagrams) are used to describe circuits at a high level.

In the second half of the book, beginning with Chapter 6, we provide examples of logic circuits and complete designs that have been described using the ABEL language. These chapters describe a variety of common digital circuits, which range from simple decoders and comparators to more complex sequential circuits such as counters and shift registers. You can use these common circuits as building blocks for larger applications. Armed with the information presented in these chapters, it will be possible for you to describe circuits ranging from simple combinational functions to complex state machine applications consisting of hundreds, or even thousands, of equivalent TTL gates. In the final chapter we describe a complex application that has been developed using ABEL.

Appendix A serves as a complete ABEL language reference and includes a wealth of information useful for new ABEL users. This appendix also contains additional information (including information not found in the standard ABEL documentation) useful to those who already have some experience with the ABEL language.

The software included with *Digital Design Using ABEL* is a device-limited subset of ABEL version 5.0. The software includes all features necessary to enter, compile, simulate, and fit designs using a variety of widely available programmable logic devices. These devices range from simple PAL-type devices like the 16L8, 16R4, 16R6, and 16R8 to more complex PLDs with configurable macrocell logic, including the PEEL 18CV8, Altera EP320 EPLD and AMD Mach 215 complex PLD. With this software it is possible for you to create, optimize, and simulate logic circuits that would be difficult or impossible to optimize using traditional methods of design.

Digital Design Using ABEL incorporates a great deal of material from an earlier book by the authors, *Practical Design Using Programmable Logic* (Prentice Hall, 1991). Readers who are interested in learning more about programmable logic devices, their history, architectures, and applications, are encouraged to consult *Practical Design Using Programmable Logic.*

This book would not have been possible without the support of many people. We would especially like to thank our colleagues at Data I/O Corporation and the many ABEL users who have contributed, through their questions and comments, to the information presented here. We would also like to thank Mark Sasten for his careful review of the manuscript.

This book would not be complete without the ABEL software. Bob LaTurner burned the midnight oil to create the customized version of ABEL included with the book, and his efforts are greatly appreciated. Special thanks also go to Karen Gettman at Prentice Hall for her support and encouragement.

David Pellerin
Michael Holley

1 Introduction

ABEL (Advanced Boolean Expression Language) is a design language and set of supporting software programs that allow complex logic designs to be entered, compiled, optimized, simulated, and then implemented in programmable logic. ABEL is a device-independent language, meaning that a design written in ABEL does not have to explicitly state what device or technology will be used for implementation. The language does, however, include many features and constructs that give you control over a variety of device-specific circuit elements. In this book we concentrate on the device-independent aspects of ABEL and point out device-specific features or design requirements as needed.

1.1 DESIGN ENTRY USING ABEL

ABEL provides three different design representations that you can combine as needed to completely specify a design. These representations are equations, truth tables, and state diagrams. ABEL also provides a test vector language that can be used to describe circuit stimulus and expected outputs for simulation. A sample ABEL source file is shown in Figure 1.1. This source file describes an eight-state counter using equations. (This and other counters are described in Chapter 7.)

Equations are most useful when the design to be described has some underlying pattern or regularity. Multiplexers, shift registers, and counters are all examples of circuits that have these attributes. ABEL

```
module count256
title '8-bit counter'
/////////////////////////////////////////////////////
// 8-bit counter with hold and synchronous reset.  //
// Appropriate for most PLD architectures.         //
/////////////////////////////////////////////////////

    clock,reset,hold  pin;
    q7..q0            pin istype 'reg';

    Count = [q7..q0];

equations

    Count.clk = clock;

    Count := !hold & (Count.fb + 1) & !reset
            # hold & Count.fb & !reset;

test_vectors([clock,reset,hold]->Count)
             [ .c. ,  1  , .x.]->  0;
             [ .c. ,  0  ,  0 ]->  1;
             [ .c. ,  0  ,  0 ]->  2;
             [ .c. ,  0  ,  0 ]->  3;
             [ .c. ,  0  ,  0 ]->  4;
             [ .c. ,  0  ,  1 ]->  4;
             [ .c. ,  0  ,  1 ]->  4;
             [ .c. ,  0  ,  0 ]->  5;
             [ .c. ,  0  ,  0 ]->  6;
             [ .c. ,  0  ,  0 ]->  7;
             [ .c. ,  0  ,  0 ]->  8;
             [ .c. ,  0  ,  0 ]->  9;
             [ .c. ,  0  ,  0 ]-> 10;
end
```

Figure 1.1 ABEL source file for 8-bit counter

raises the level of design abstraction for circuits such as these by incorporating high-level equation features such as sets, arithmetic operators, and relational operators. These features allow far more abstract forms of expression than are possible using simple sum-of-products Boolean equations.

Truth tables are most useful for designs that have no underlying pattern or order. A typical example of such a circuit is a decoder for a seven-segment display such as the one shown in Figure 1.2. (This design is described in more detail in Chapter 5.) The truth table is a natural way to describe partially specified functions in which there are don't-care conditions. Truth tables are also a convenient way in which to describe the behavior of sequential circuits that contain a large number of similar

```
module BCD7
title 'BCD to 7-segment display driver'
/////////////////////////////////////////////////////////
// Seven-segment display driver with active-low        //
// outputs. Segments:     -a-                          //
//                      f|   |b                        //
//                        -g-                          //
//                      e|   |c                        //
//                        -d-                          //
/////////////////////////////////////////////////////////

    D3..D0          pin;                     "BCD input
    a,b,c,d,e,f,g   pin istype 'dc,com';     "Segment outputs
    OE              pin;                     "Output enable

    BCD     = [D3..D0];
    LED     = [a,b,c,d,e,f,g];
    ON,OFF  = 0,1;                         "Inverted sense

equations

    LED.oe = !OE;           "Define output enable

truth_table(BCD->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
             0 ->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
             1 ->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
             2 ->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
             3 ->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
             4 ->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
             5 ->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
             6 ->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
             7 ->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
             8 ->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
             9 ->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];

test_vectors
      ([OE,BCD]->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
       [ 0, 0 ]->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
       [ 0, 1 ]->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
       [ 0, 2 ]->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
       [ 0, 3 ]->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
       [ 0, 4 ]->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
       [ 0, 5 ]->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
       [ 0, 6 ]->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
       [ 0, 7 ]->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
       [ 0, 8 ]->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
       [ 0, 9 ]->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];
       [ 1, 5 ]->[ .z., .z., .z., .z., .z., .z., .z.];

end
```

Figure 1.2 Seven-segment display driver design file

```
state_diagram BCD

    State Zero:     C4 = 0;
                    If Cin Then One   Else Zero;

    State One:      C4 = 0;
                    If Cin Then Three Else Two;

    State Two:      C4 = 0;
                    If Cin Then Five  Else Four;

    State Three:    C4 = 0;
                    If Cin Then Seven Else Six;

    State Four:     C4 = 0;
                    If Cin Then Nine  Else Eight;

    State Five:     C4 = 1;
                    If Cin Then One   Else Zero;

    State Six:      C4 = 1;
                    If Cin Then Three Else Two;

    State Seven:    C4 = 1;
                    If Cin Then Five  Else Four;

    State Eight:    C4 = 1;
                    If Cin Then Seven Else Six;

    State Nine:     C4 = 1;
                    If Cin Then Nine  Else Eight;
```

Figure 1.3 ABEL state diagram language

state transitions. Truth tables for these applications will be explored in Chapter 11.

ABEL's state diagram language can be used to describe the behavior of finite state machines (FSMs). The choice of whether to use state diagrams, equations, or truth tables for descriptions of these circuits is largely a matter of personal taste. State diagrams tend to be more lengthy than equivalent truth table descriptions, but are usually more readable if there are a large number of states. Figure 1.3 is an excerpt from a serial BCD (binary coded decimal) decoder presented in Chapter 10.

```
Simulate ABEL 5.03e  Date: Fri Sep  10 18:10:27 1993
Fuse file: 'fib1.tt1'  Vector file: 'fib1.tmv'  Part: 'PLA'

       C C
       1 1   A A A A A A A A B B B B B B B B S S S S S S S S
       k r   7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

V0001  C 1   L L L L L L L L L L L L L L L L L L L L L L L L
V0002  C 0   L L L L L L L H L L L L L L L L L L L L L L L H
V0003  C 0   L L L L L L L L L L L L L L L H L L L L L L L H
V0004  C 0   L L L L L L L H L L L L L L L H L L L L L L H L
V0005  C 0   L L L L L L L H L L L L L L H L L L L L L L H H
V0006  C 0   L L L L L L H L L L L L L L H H L L L L L H L H
V0007  C 0   L L L L L L H H L L L L L H L H L L L L H L L L
V0008  C 0   L L L L L H L H L L L L H L L L L L L L H H L H
V0009  C 0   L L L L H L L L L L L L H H L H L L L H L H L H
V0010  C 0   L L L L H H L H L L L H L H L H L L H L L L H L
V0011  C 0   L L L H L H L H L L H L L L H L L L H H L H H H
V0012  C 0   L L H L L L H L L L H H L H H H L H L H H L L H
V0013  C 0   L L H H L H H H L H L H H L L H H L L H L L L L
V0014  C 0   L H L H H L L H H L L H L L L L H H H L H L L H
V0015  C 0   H L L H L L L L H H H L H L L H L H H H H L L H
V0016  C 0   L L L L L L L L L L L L L L L L L L L L L L L L
V0017  C 0   L L L L L L L L L L L L L L L L L L L L L L L L
V0018  C 0   L L L L L L L H L L L L L L L L L L L L L L L H
V0019  C 0   L L L L L L L L L L L L L L L H L L L L L L L H
V0020  C 0   L L L L L L L H L L L L L L L H L L L L L L H L
20 out of 20 vectors passed.
```

Figure 1.4 Sample ABEL simulation results

Test Vectors and Simulation

Test vectors are lists of values representing inputs and corresponding outputs of the design while under test. In ABEL, test vectors are entered into source files as an integral part of the design. The ABEL test vector language is virtually identical to the ABEL truth table language and is used to describe test stimulus and expected outputs for simulation. The ABEL logic simulator that is provided with this book can be used to check the function of a design before moving into the implementation phase. Figure 1.4 shows a sample output from the ABEL simulator.

Logic Synthesis

Logic synthesis is the process of converting a circuit description into a form appropriate for hardware implementation and optimizing that description to create a physical circuit that meets specific constraints. The constraints that drive logic synthesis may be weighted and may include such things as the total circuit size, operating speed, power usage, or testability. Device-specific synthesis constraints may include such things as primitive libraries, fan-in and fan-out restrictions, and device resource constraints. The actual device mapping process may include logic partitioning, resource allocation, placement and routing, and other functions.

Logic minimization and multilevel logic optimization are specific logic synthesis features that are covered in this book. The software provided with this book includes logic minimization (based on the Espresso optimization software developed at the University of California at Berkeley), but does not include the more advanced logic synthesis features required to implement designs in the most advanced complex PLDs and FPGAs (field programmable gate arrays).

When high-level design methods are used, effective logic synthesis techniques are the key to a successful and efficient final circuit. It is logic synthesis that allows designs to be described at a higher level of abstraction. Since the goal of logic synthesis is to translate a circuit representation into a form that meets the specified constraints of the design, it's important to decide what constraints are most important for each design.

For most designs being implemented with programmable devices, the most direct constraints are related to the restricted architectures of the devices. Automated logic synthesis tools that help to optimize combinational circuits for various global cost and operational constraints have existed for some time. Automated tools that help with logic synthesis for constrained architectures, however, have only recently appeared. These device-specific tools, called *fitters*, are the key to efficient implementation of device-independent designs.

Logic synthesis features available in today's design tools are certainly useful for that part of the design process in which logic synthesis can benefit. It's important to realize, however, that the currently available synthesis tools, no matter how highly automated or tightly integrated, cannot be used without an awareness of their limitations. Specifically, you must always be aware of how basic design decisions can affect the final circuit produced by the tools. Automated logic synthesis tools can only help to optimize a logic design as you have specified it. They cannot explore design-level alternatives and make intelligent decisions about those alternatives. In later chapters, we'll explore some of these design-level optimizations and alternatives.

Implementation Considerations

All potential implementation technologies (TTL, PLD, FPGA, or ASIC) have their strengths and weaknesses, and it is important to keep these strengths and weaknesses in mind when creating a new design. Since most ABEL designs are eventually implemented in PLDs and FPGAs, these are the technologies that we will concentrate on.

PLDs and FPGAs are devices with constrained architectures. While the specific constraints may differ from one device family to another, the existence of these constraints leads to limitations in the size and configuration of the circuits that can be implemented in them. While automated design tools can help in the conversion of abstract, device-independent design concepts into working circuits, the designer must still understand the basic limitations of the target technology. By showing how the most common constraints effect the implementation of relatively simple circuits, we'll be able to demonstrate some of the many design trade-offs that are required for larger designs and other technologies.

2 Combinational Circuits

A solid understanding of digital logic fundamentals is critical to the understanding of ABEL and the high-level design capabilities that it provides. In this chapter, we'll review these fundamentals and examine how the techniques of digital logic design can be used to help us design at a higher level. From there we'll move quickly into advanced topics that are particularly important for ABEL users to understand.

The terminology of digital logic can be confusing. Therefore, a major goal of this chapter is to define the language of logic design as used in this book. Even if you are a seasoned logic designer, bear with us during the early parts of this discussion so that we can make sure that we are speaking a common language. This will be more important in Chapter 3, as we discuss specific logic optimizations that are performed by the ABEL software, as well as those that must be performed manually.

2.1 LOGIC CIRCUITS AND LOGIC FUNCTIONS

First, we will try at all times to distinguish between real circuits and purely functional representations of these circuits. In the following discussions, we will consider a logic function to be an abstract concept, whereas a logic circuit is a collection of physical circuit elements and interconnections (typically wires or metal traces). Logic circuits have real-world attributes, such as size, speed, and critical timing relationships. This distinction is important to us because, when designing with ABEL, we are designing at a higher functional level, rather than at a real implementation level. As we'll see, though, the form of the implementation and the constraints of the target architecture must frequently be considered even when designing at a high level.

Basic Logic Gates

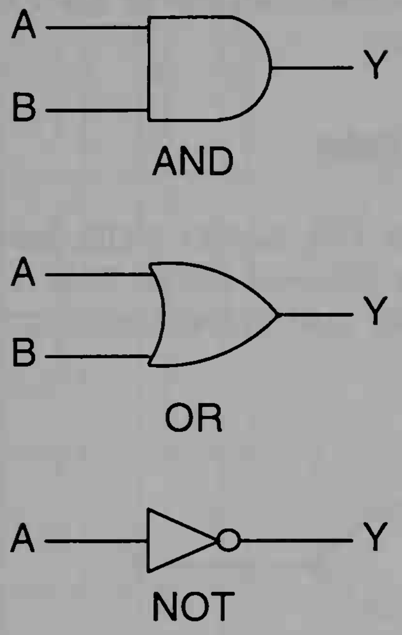
Logic gates are the basic building blocks of digital logic circuits. A logic gate may be thought of as a decision-making element. A logic gate has one or more inputs (each of which can be either *true* or *false* at any given time) and a single output that produces either a true or false value based on the values of the gate inputs. To simplify later discussions, we may refer to the input and output values of these gates as either logic level 1 (true) or logic level 0 (false). The numerals 0 and 1, when used in this context, are not numbers; they are logic values.

The inputs and outputs of logic gates are called *signals*. In an actual hardware implementation, a signal corresponds to a wire or metal channel that carries current from one circuit element to another. For the moment, we can assume that all signals are, at any given time, either 0 or 1, with no ambiguous signal values. When one or more logic gates are connected together and the input and output signals are given some significance (usually indicated by specific signal names), they form a logic circuit.

In describing the operation of logic gates and logic circuits, it is common to use a form of representation known as the *truth table*. A truth table simply associates combinations of inputs with resulting outputs. Truth tables can be used to fully specify a logic function (as in the following examples) or to partially specify it. We'll use fully specified truth tables to define the basic logic gates. Advanced applications of fully and partially specified truth tables will be covered later.

The logic gates of most interest to us for Boolean logic manipulations are the AND, OR, and NOT (inverter) gates. The most common symbolic representations of these gates is shown in Figure 2.1, along with truth tables describing their operations.

The AND, OR, and NOT gates are analogous to the words "and," "or," and "not" in the English language. We could describe a logic function using the english language. For example, we might say "The function *Y* shall be true when the input signal *A* is true and the input signal *B* is not true, or when the input signal *C* is true." If we wanted to build a



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	Y
0	1
1	0

Figure 2.1 Basic gates

circuit that implemented this function, it might look like the circuit shown in Figure 2.2.

This logic circuit is what is known as a combinational logic circuit. This means that, for any set of input values (as shown in a fully specified truth table) there is one and only one possible circuit output value, regardless of the previous state of the circuit. By gluing together an appropriate collection of ANDs, ORs, and NOTs, it is possible to implement any combinational logic function.

There are, of course, other types of logic gates. Designers who are used to constructing circuits out of standard TTL gates or to designing circuits using "bubble logic" techniques are often more comfortable with NAND or NOR gates. These gates are widely used because they normally require less transistor circuitry for implementation than do ANDs and ORs. For our purposes, however, it is more convenient to consider NAND

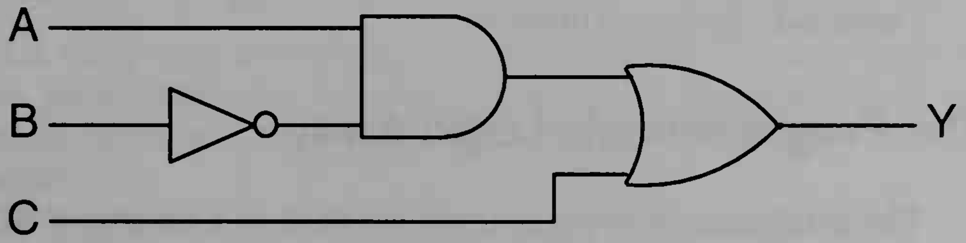
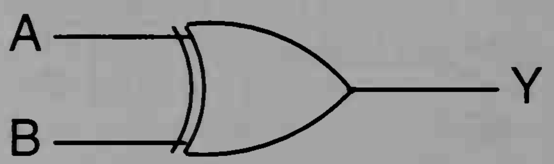


Figure 2.2 A simple logic circuit

and NOR gates to be combinations of AND and NOT gates or OR and NOT gates, respectively.

The Exclusive-OR Gate

In addition to the AND, OR, NAND, NOR, and NOT gates, another type of gate is of particular interest to us. This is the *exclusive-OR gate*, or *XOR*. The XOR symbol and corresponding truth table are shown in Figure 2.3.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.3 Exclusive-OR (XOR) gate

Like NAND and NOR gates, the XOR can be constructed from the basic AND and OR gates, as shown in Figure 2.4. We'll discuss how the XOR gate can be used to advantage later in this chapter.

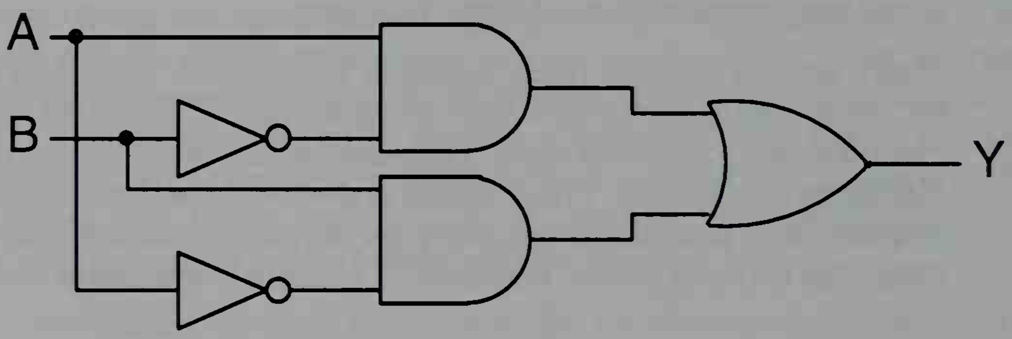


Figure 2.4 Exclusive-OR circuit

The Programmable Logic Array

The programmable logic array, or *PLA*, is a matrix of NOT, AND, and OR gates arranged as shown in Figure 2.5. The diagram is simplified by combining all the inputs to each AND and OR gate into a single line.

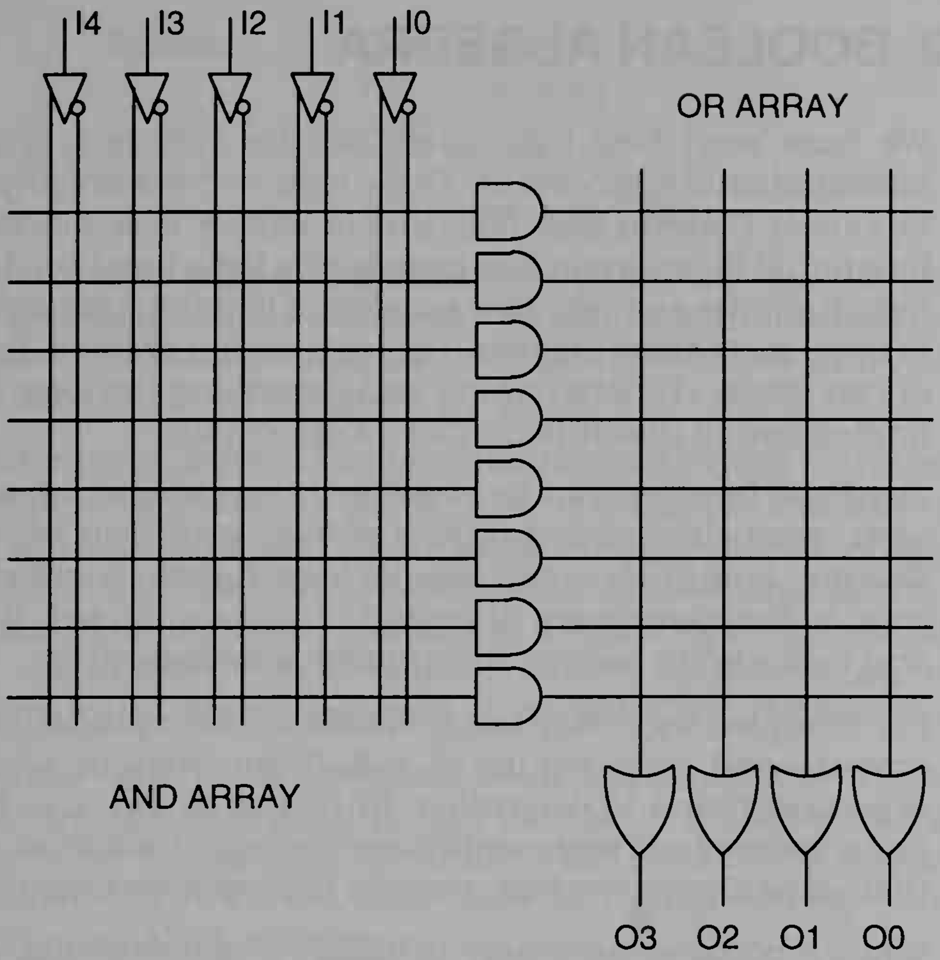


Figure 2.5 Programmable logic array

The actual number of signals feeding each gate can be determined by counting the number of intersections on the line.

Each intersection of a vertical and horizontal line represents a potential interconnect point. There are two arrays of possible interconnects in a PLA. The upper array is the AND array, while the lower is the OR array.

The simple PLA shown in the figure has five inputs, each of which is available to the AND array either directly (its *true* value) or through an inverter (its complement). The AND gate outputs are fed into the OR array, so any AND gate output can be used as an input to any OR gate.

The PLA structure is important to us because it is the basis for nearly all the programmable devices currently in use, and it is the basic structure assumed by ABEL and used as an internal design representation.

2.2 BOOLEAN ALGEBRA

We have seen how logic gates can be arranged to create a simple combinational logic circuit. These logic circuits are physical implementations of *Boolean logic functions* or simply *logic functions*. Every combinational logic circuit has exactly one logic function, but for any given logic function an unlimited number of logic circuits can be constructed to implement that function. The logic design process, then, is composed of two steps: (1) determining and describing the logic function and (2) implementing that function as a logic circuit.

Any logic function can be expressed in a number of ways. As we have seen, symbolic representations of logic gates can be used to describe circuits, and in fact most simple logic functions are described in this way. In the case where discrete TTL gates are used, the logic function and logic circuit may be determined simultaneously.

For complex logic functions, however, the optimal logic circuit isn't obvious and may require a significant amount of calculation and experimentation to determine. In this situation, it is beneficial to use more conceptual representations for logic functions, representations that allow alternative logic circuits to be quickly identified and selected.

When it becomes necessary to modify a logic function, experiment with alternative circuits, or manipulate the form of the circuit, these conceptual forms of representation are usually more helpful. We have seen how truth tables, and even English language statements, can be used to describe the operation of a digital logic function in a conceptual way. Another form of representation, *Boolean algebra*, has been used for many years to describe and manipulate logic functions.

Boolean algebra is a system of mathematics that allows us to manipulate logic. As before, we use the value 1 to indicate the true logical value and a 0 to indicate the false logical value. We'll use true/false and 1/0 values interchangeably in the remaining discussions.

Boolean Equations

At the heart of Boolean algebra are the Boolean operators. These operators correspond to the AND, OR, XOR, and NOT logic gates presented earlier. There are many ways of representing the AND, OR, NOT, and XOR Boolean operators, and a variety of different representations are used in digital design languages and formal texts. In ABEL and in this book, we use the following symbols for Boolean operations:

Symbol	Description
!	NOT (inversion)
&	AND
#	OR
$	XOR (exclusive-OR)

In most formal texts on logic design, the AND operator is omitted from Boolean equations. In this book, we will at all times include the AND operator in equations in order to remain consistent with ABEL-related discussions in other chapters.

Operator precedence (*order of evaluation* or *binding*) is the order shown in the preceding list. (In ABEL, however, the OR and XOR operations have equal precedence.) As in standard algebra, parentheses may be used to change the order of operations. In Boolean algebra, a variable is a named entity (corresponding to a signal in a logic circuit) that can have one of two possible values: 1 or 0. A Boolean *expression* is a grammatically correct sequence of variables and Boolean operators.

A Boolean equation is composed of an output variable that is assigned a Boolean logic function through the use of the assignment operator, which is the equals sign. In ABEL, each equation is terminated by a semicolon. The AND, OR, XOR, and NOT gates can be represented by the following ABEL equations:

```
Y_AND = A & B;
Y_OR = A # B;
Y_XOR = A $ B;
Y_NOT = !A;
```

To help clarify these concepts, consider the following Boolean equation:

```
Y = A & B # C & !B;
```

This Boolean equation describes a logic function of three input variables ($Y = f(A, B, C)$). The function can be described by the following English language statement: "The output variable *Y* shall be true if the input variables *A* and *B* are both true, or if the input variable *C* is true and the input variable *B* is false."

The NOT operator is a unary operator, which means it operates on a single variable or subexpression. The NOT operator creates the complement (inverted) value of a variable or subexpression. For Boolean algebra purposes, it is often most convenient to consider a true variable and the complement of that variable as two separate entities. When this is done, we refer to the inputs as literals. In the Boolean equation we just presented, for example, the variable *B* appears as both a true and

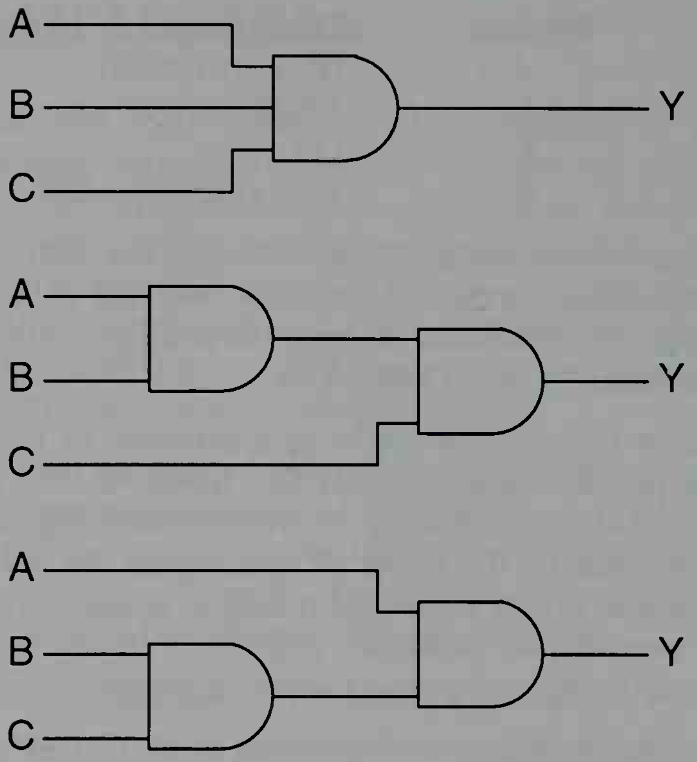


Figure 2.6 Three possible implementations of the same logic function

complemented variable. This means that the equation contains four distinct literals (*A*, *B*, *!B*, and *C*).

The AND, OR, and XOR operators are binary operators, operating on two variables or subexpressions. Actually, it is sometimes more convenient to consider the AND and OR operators to be n-ary operators (operators that can operate on more than two variables) since the actual logic gates that correspond to these operators may have more than two inputs. This is allowed by the associative laws for these operators, as we'll see in a moment.

Boolean algebra is a convenient method with which to describe logic functions. It is never possible, however, to completely specify a logic circuit with Boolean algebra, since there are many possible circuit interpretations of a Boolean equation. This is shown in Figure 2.6. Each of the three circuits shown will have subtly different operating characteristics, due to signal propagation and gate switching delays.

Formal Rules for Boolean Operations

A number of operations are used in the manipulation of Boolean equations. We will present the formal rules of the operations most useful to logic designers, although complete descriptions and proofs of these are beyond the scope of this book. The interested reader is referred to any one of a number of books on the subject of logic design for further study. Some of these texts are listed in the references at the end of this chapter.

The following is a list of the operations that are of greatest interest to us. We have added some exclusive-OR operations to the standard operations commonly presented. We will assume the validity of these additional operations without formal proof.

Standard Operations

1. Commutative laws:

   (a) A # B = B # A
   (b) A & B = B & A

2. Identities:

   (a) A # 0 = A
   (b) A # 1 = 1
   (c) A & 0 = 0
   (d) A & 1 = A

3. Complement identities:

   (a) A # !A = 1
   (b) A & !A = 0

4. Involution:

   (a) !!A = A

5. Indempotence:

   (a) A # A = A
   (b) A & A = A

6. Distributive laws:

   (a) A # (B & C) = (A # B) & (A # C)
   (b) A & (B # C) = (A & B) # (A & C)

7. Associative laws:

   (a) A # (B # C) = (A # B) # C
   (b) A & (B & C) = (A & B) & C

8. Absorption:

   (a) A # (A & B) = A
   (b) A & (A # B) = A
   (c) A # (!A & B) = A # B
   (d) A & (!A # B) = A & B

9. Unity:

   (a) A & B # !A & B = B
   (b) (A # B) & (!A # B) = B

Exclusive-OR Operations

10. Associative law:

    (a) A $ (B $ C) = (A $ B) $ C

11. Distributive law:

    (a) A & (B $ C) = A & B $ A & C

12. Commutative law:

    (a) A $ B = B $ A

13. XOR identities:

    (a) A $ 1 = !A
    (b) A $ 0 = A
    (c) A $ A = 0
    (d) A $!A = 1

14. Rules of XOR complement:

    (a) !(A $ B) = A $!B
    (b) !(A $ B) = !A $ B
    (c) A $ B = !A $!B

In addition to these rules, two theorems are widely used in Boolean equation manipulations and logic minimization.

DeMorgan's Theorem

DeMorgan's theorem is used to find the complement of a Boolean expression. This is particularly useful for programmable logic applications:

(a) !(A # B) = !A & !B
(b) !(A & B) = !A # !B

DeMorgan's theorem can be generalized as follows: The complement of any Boolean expression can be determined by replacing each OR operator with an AND operator (while preserving the order of evaluation), replacing each AND operator with an OR operator, and replacing each literal with its complement. For example, the following two relationships are valid:

(c) !(A # (B & C)) = !A & (!B # !C)
(d) !((A # !B) & (C & A)) = (!A & B) # (!C # !A)

Shannon's Expansion Theorems

Shannon's theorems are used to isolate (factor out) one or more variables in a logic function. Shannon's expansion theorems may be described by the notation:

(a) f(A,B,C,..) = A & f(1,B,C,..) # !A & f(0,B,C,..)
(b) f(A,B,C,..) = (A # f(0,B,C,..)) & (!A # f(1,B,C,..))

where *f(A,B,C,...)* represents any multiple-variable Boolean logic function. Shannon's theorems are particularly useful during multilevel logic minimization.

Standard Forms for Boolean Logic

A number of general forms may be used to express a Boolean logic function. An understanding of these forms is important for effective use of logic minimization techniques and tools.

We refer to any AND operation as a product. Any Boolean expression that contains one or more literals operated on by one or more AND operators is referred to as product term. For example, the expression

```
!A & B & !C & D
```

qualifies as a product term, while the expression

```
!A # B & !C
```

does not, since it includes a binary operator other than AND.

A product term expression is distinguished by the fact that only one set of input conditions will result in a true evaluation of the expression.

Logical Sum

OR operators are referred to as sums. Any Boolean expression that contains one or more literals operated on by one or more OR operators is a sum. The expression

```
!A # B # !C # D
```

is an example of a sum.

Sum of Products

When two or more product terms are operated on by one or more OR operators, the form of the resulting expression is *sum of products.* If we add an assignment operator and an output variable, we have a *sum of products Boolean equation*, as in the following ABEL equation:

```
Y = A & !B & C # A & !B & !C # !A & !B & C;
```

Expressed as logic gates, this expression could be composed of one 3-input OR gate fed by three 3-input AND gates. As Figure 2.7 shows, a sum-of-products form requires two levels of logic gates (discounting the NOT gates required for the complemented input literals) for implementation. For this reason sum-of-products Boolean logic functions are often referred to as two-level logic functions.

The sum of products form is the basis for most logic optimization methods, and maps directly into the PLA structure presented earlier.

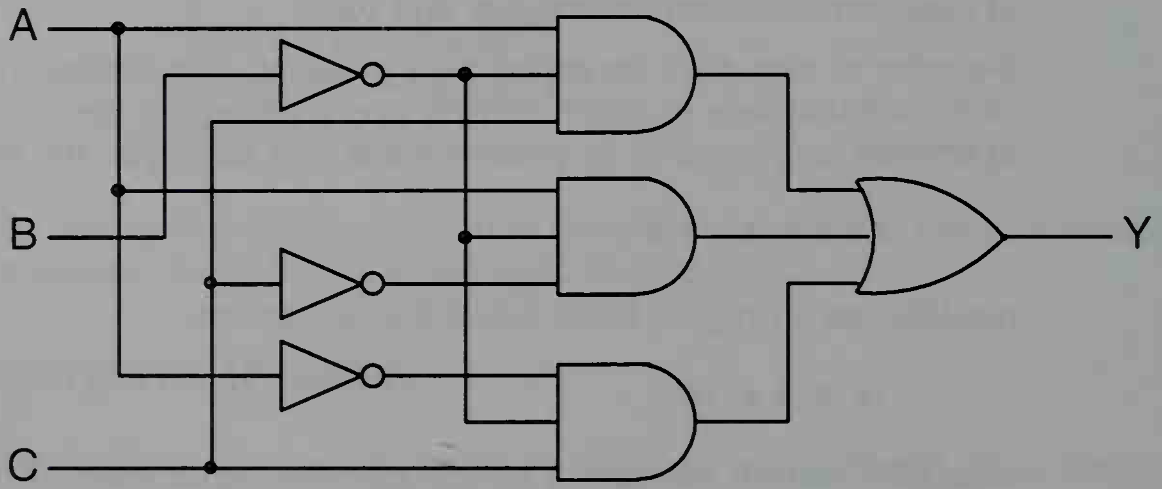


Figure 2.7 Sum-of-products two-level logic

By utilizing the laws and theorems presented earlier, it is possible to express any logic function, regardless of complexity, in sum-of-products form.

2.3 USING BOOLEAN ALGEBRA IN ABEL

Using the basic laws and theorems presented as tools, we can now begin to experiment with various techniques that are useful for ABEL users. Many of these manipulations are performed automatically by the ABEL software, while others must be performed by the ABEL user. Chapter 3 describes in detail which optimizations are performed by the software provided with this book.

Before moving on, though, let's examine some of the reasons why these manipulations are often necessary. The primary motivations for Boolean logic manipulations are to reduce circuit size or improve circuit speed. All the common implementation technologies (simple and complex PLDs, FPGAs, and gate arrays) are constrained in some way. Most require that logic functions be implemented in specific forms (usually sum of products or some derivative of sum of products), and all have limited resources. The limiting resource is usually either logic gates or signal interconnections (routing).

Traditional PLDs, for example, are limited in the number of product terms that can be used for each output, so product term reduction is a primary goal of optimization for these devices. If a logic function can't be reduced sufficiently for a PLD implementation, it will be necessary to perform multilevel optimizations to spread the circuitry for that function across multiple levels of logic.

Another common limitation of PLDs is the number of inputs and outputs available in each device. If a design can't be implemented into a PLD due to I/O constraints, it is necessary to either change the design or choose a different implementation. Choosing a different implementation can involve using a different type of device, more than one device for the function, or an entirely different implementation technology.

Speed isn't generally much of an issue when you are dealing with a single PLD, since in most cases the delay times from inputs to outputs are fixed and predictable, and PLDs are available that are fast enough for most applications. For large designs implemented in FPGAs, or designs that utilize many PLDs, the delay of signals due to multiple levels of logic can wreak havoc on a system. For this reason more advanced multilevel logic optimizations may be needed for large designs.

2.4 REFERENCES

Breeding, Kenneth J., *Digital Design Fundamentals*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Mano, M. Morris, *Digital Design*, Prentice Hall, Englewood Cliffs, NJ, 1984.

Unger, Steven J., *The Essence of Logic Circuits*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Wakerly, John F., *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, NJ, 1990.

3

Logic Minimization

Logic minimization is the process of reducing the amount of circuitry required to implement a logic function. As we saw in Chapter 2, the optimal form of a logic circuit depends to a large extent on the architecture into which the logic function is to be implemented. There are dozens of methods for logic minimization and a wide variety of computer-based tools that utilize these methods. Although we can't hope to describe all these methods in this chapter, we will describe some of the more popular minimization methods and define some concepts common to all these methods.

Users of computer-based logic synthesis tools are often bewildered by the descriptions of algorithms and logic forms and the strange language used to describe the effectiveness of various techniques. The result, all too often, is that the users of these tools never really understand how to get the most benefit from them. In this chapter we'll go over some common logic minimization techniques and clear up some of the confusion over terminology.

3.1 TWO-LEVEL MINIMIZATION

To start with, consider a Boolean logic function with three variables *A*, *B*, and *C*. Each variable can be expressed as either its true or complemented value. If we enumerate all combinations of these three variables using the AND operator, we find eight possibilities, as follows:

```
!A & !B & !C
!A & !B &  C
!A &  B & !C
!A &  B &  C
 A & !B & !C
 A & !B &  C
 A &  B & !C
 A &  B &  C.
```

Each of these eight expressions is referred to as a minterm, or canonical product. For any logic function of n variables, there are 2^n possible minterms. These Boolean expressions are called minterms because, for each expression, there is only one set of input values that will result in the value 1. For example, the minterm *!A & !B & C* will only evaluate to 1 for one set of inputs (*A*, *B*, and *C* equal 0, 0, and 1, respectively). The importance of minterms lies in the fact that any logic function, regardless of complexity, can be expressed as a sum of one or more of its minterms.

Maxterms

A *maxterm* is similar to a minterm, but is related to OR operations, rather than AND operations. The eight possible maxterms for three input variables are

```
!A # !B # !C
!A # !B #  C
!A #  B # !C
!A #  B #  C
 A # !B # !C
 A # !B #  C
 A #  B # !C
 A #  B #  C.
```

These expressions are called maxterms because they will evaluate to a 1 for all but one possible set of input values.

Minsums

A sum of products expression that consists of a sum of minterms is referred to as a minsum. This form is important for many logic minimization techniques and is sometimes referred to as the *canonical sum-of-products* or disjunctive normal form. For example, the 3-input sum-of-products expression

```
!A & B & C # A & !B & C # A & B & !C
```

is a minsum, since all the product terms in the expression are minterms, whereas the expression

```
!A & B & C # A & !B & C # B & !C
```

is not a minsum, since the product term *B & !C* is not a minterm (the variable *A* is not specified in the third product term).

Determining the Minsum of a Logic Function

Any arbitrarily complex logic function can be converted to sum-of-products form. Furthermore, any logic function can be converted to its minsum form. To demonstrate how the minsum form of a logic function can be determined, consider the following expression:

```
A & !B & !C # A & B
```

To get this Boolean function into minsum form, we'll use the postulates and laws presented in Chapter 2. First, we know that the minsum form will require three input variables for each product term in the expression, so we use the identity postulate to get the intermediate form:

```
A & !B & !C # A & B & 1
```

Next, the complement postulate allows us to replace the 1 with the expression *C # !C*, resulting in

```
A & !B & !C # A & B & (C # !C)
```

Finally, by utilizing the distributive law we obtain the minsum expression

```
A & !B & !C # A & B & C # A & B & !C
```

Other Standard Forms

There are other standard forms in which Boolean logic functions can be expressed, including the *product-of-sums canonical form* (referred to as the maxproduct form), but since the vast majority of devices are based on the sum of products form, and most logic minimization techniques are also based on sum-of-products, we won't dwell on these other forms in this book. Again, the interested reader is referred to one of the many books written on the subject of digital logic design for further information.

Logic Minimization Methods

The various Boolean logic manipulations that we have shown can be used to minimize logic. For large Boolean expressions, however, the most efficient approach to take using these manipulations typically isn't obvious, and finding the minimal solution can be extremely time consuming.

A common method for systematically determining the minimal sum-of-products expression for a given logic function was first described by Maurice Karnaugh in 1953. This method uses a symbolic representation of the logic function. The representation, called a Karnaugh map, graphically depicts the function in a modified form of truth table and allows simple, organized methods to be used for minimization.

To demonstrate how a Karnaugh map can be used, we will minimize the following logic function:

```
A & B # !A & B # A & !B
```

A two-variable Karnaugh map for this function is shown in Figure 3.1.

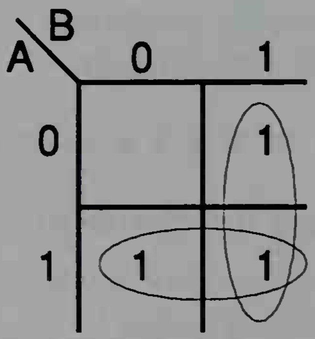


Figure 3.1 Two-variable Karnaugh map

The Karnaugh map (which we will from now on refer to as a *K-map*) is composed of a number of boxes. Each box, or *cell*, represents one possible minterm for the function. Since this function has only two variables, there are 2^2 possible minterms, as described earlier. Each box is identified by the values indicated on the top and sides of the box; a 1 indicates the true asserted variable, whereas a 0 indicates the

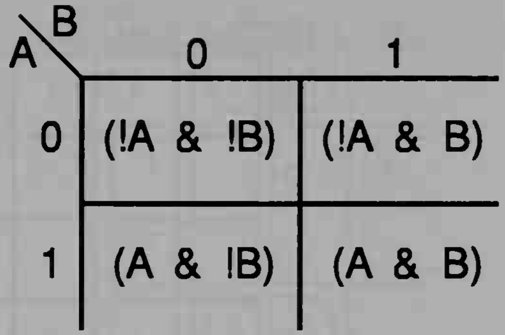


Figure 3.2 Mapping minterms onto a 2-variable K-map

complement of that variable. In Figure 3.2, the possible minterms for a two-input Boolean function are shown in their corresponding K-map boxes.

If the minterm associated with a particular box actually exists in the minsum form of the Boolean expression, then a 1 is written in that box. The boxes so marked are referred to as *1-cells*. Cells not marked are called *0-cells*.

To minimize a function using a K-map, we utilize the following relationship developed from our earlier Boolean identities:

A & f() # !A & f() = f()

The variable *f()* represents any arbitrary Boolean function. To generalize this and apply it to the K-map: if there are two product terms and one is ANDed with a variable (*A*) while the other is ANDed with that variable's complement (*!A*), the product terms can be combined and the variable eliminated if the product terms are otherwise identical.

This situation is identified on the K-map by pairs of horizontally or vertically adjacent 1-cells. As Figure 3.1 shows, the pairs of 1-cells are grouped. Each group of two corresponding minterms can be combined into a single product term by elimination of the differing variable. For this expression, the two groupings result in the simpler sum *A # B*.

The K-map technique is a method for determining the minimal sum-of-products representation of a logic function from its minsum form. Most logic minimization algorithms, in fact, first determine the minsum form of a sum of products expression before beginning the minimization process.

If you examine the K-map layout, you will notice that moving one box either vertically or horizontally always results in a single bit change; in this case, either A or B changes, but never both. This is obviously the

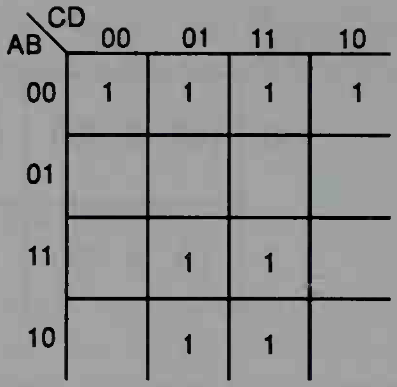


Figure 3.3 Four-variable K-map

case for the simple two-variable K-map, but what about K-maps for larger numbers of variables?

Figure 3.3 shows how a K-map for a four-variable Boolean function is written. This K-map is composed of 16 boxes, but is still two dimensional. Therefore, each box needs to represent the value of more than two variables as indicated in the numbering. To maintain the requirement of single bit changes between horizontally and vertically adjacent boxes, the horizontal and vertical axis values are numbered using a gray code (also called a reflected code). As in the two-variable K-map, each box represents a unique minterm. The rightmost box on the lower row, for example, represents the minterm *A & !B & C & !D*.

To reduce the four-variable minsum expression indicated in the figure, we first identify and circle all adjacent 1-cells. This is shown in Figure 3.4. Notice that, in the lower center of the K-map, there are four 1-cells grouped together in a square pattern. If these four were to be grouped as pairs, they could be combined in a number of ways. Which combination is appropriate? Actually, this decision isn't necessary. Whenever

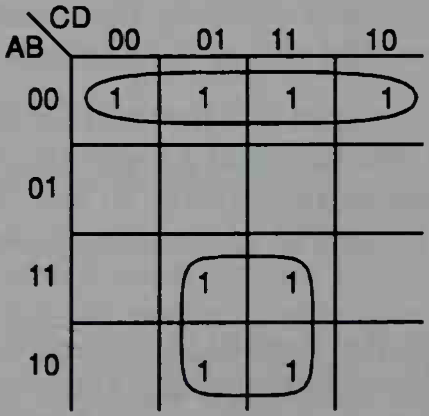


Figure 3.4 Minimizing a logic function with a four-variable K-map

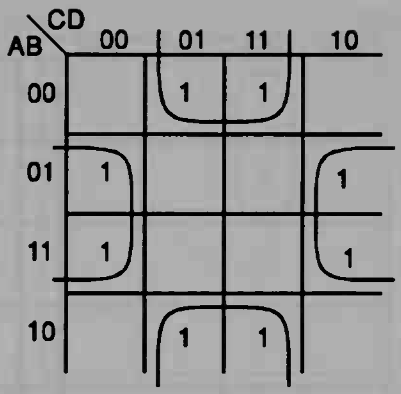


Figure 3.5 K-map for eight-minterm function

two pairs of 1-cells are adjacent in this manner, it's an indication that all four minterms can be combined into a single product term.

Similarly, the four 1-cells that form an encircled rectangle on the upper row can be combined, resulting in a significant reduction of logic. The completely minimized expression is

```
!A & !B # A & D
```

In this example, the variable *C* was completely eliminated.

Next, consider the expression and K-map of Figure 3.5. In this K-map, the eight 1-cells have been grouped into two squares. This is possible because the K-map edges are logically connected; it is convenient to think of the K-map in terms of a cylinder in both the horizontal and vertical directions. The variables *A* and *C* are both eliminated, and the minimized form of this expression is simply

```
B & !D # !B & D.
```

Using K-maps for Larger Functions

There are many more applications of K-map techniques that are of academic interest, but these are beyond the scope of this book. It is interesting to note however, that K-maps like the one shown in Figure 3.6 can be constructed to determine the minimized sum-of-products representation of expressions with more than four variables.

Since the size of the K-map increases exponentially with the increased numbers of variables, this method quickly becomes impractical for large designs. This fact and the availability of computer-based algorithms for logic minimization have for the most part eliminated the need for tedious manual K-map minimization. The K-map is useful, however, for graphi-

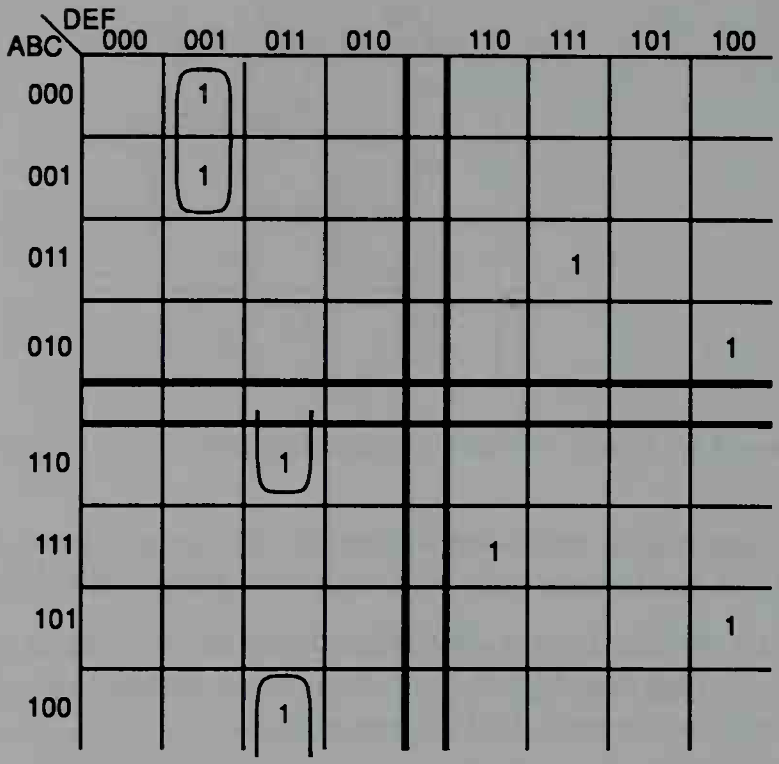


Figure 3.6 Six-variable K-map

cally demonstrating logic minimization concepts and gives important insights into the workings of computer-based logic minimization algorithms.

Cubes

Another graphic representation that is useful for describing logic minimization concepts is the cube. Consider again the case where two minterms of a function differ only in the value of one variable, as in

```
!A & B & C # !A & B & !C
```

As we saw in the K-map discussion, the two minterms in this minsum can be represented by sequences of ones and zeroes that correspond to true and complemented input variables. Using this representation in another way, we can assign the minterms the values *011* and *010*. A function of *n* variables has a total of 2^n possible minterms. If we imagine that each variable corresponds to a dimension in space, we can represent the entire set of minterms as an *n*-dimensional cube.

Figure 3.7 shows a two-dimensional cube, or 2-cube, represented by four points connected by four line segments. This cube corresponds

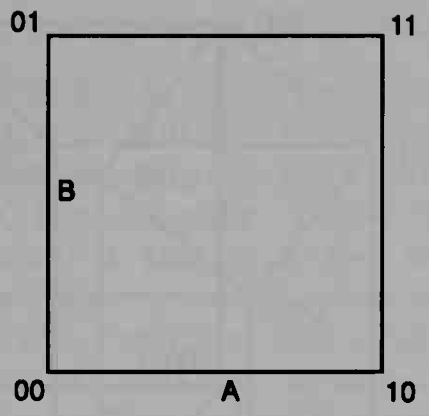


Figure 3.7 Cube representation for two variables (2-cube)

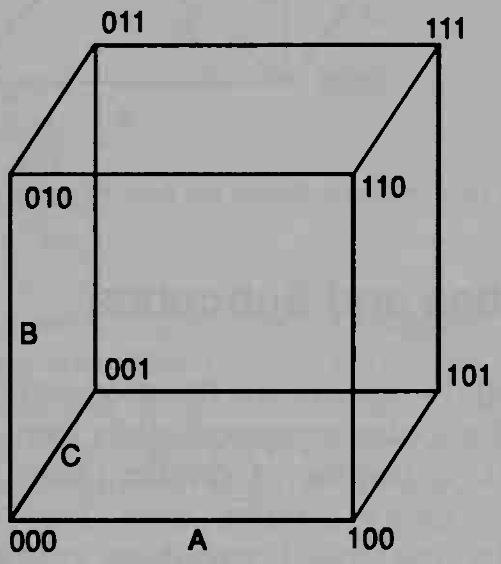


Figure 3.8 Cube representation for three variables (3-cube)

directly to its K-map representation; each point on the cube represents one box on a four-variable K-map. Movement along a horizontal line segment correspond to changes in the first variable, while movement in the vertical direction corresponds to changes in the second variable. Each point on the two-dimensional cube is labeled with the corresponding value of the input variables.

Figure 3.8 shows a three-dimensional cube, or 3-cube. The third dimension of this cube is drawn as diagonal line segments. Motion along these line segments corresponds to changes in the third variable. Notice that, no matter what line segment you are on, movement from one point to another always results in a change in only one input variable.

Figure 3.9 shows a four-dimensional cube (sometimes called a hypercube). This, of course, is a 4-cube.

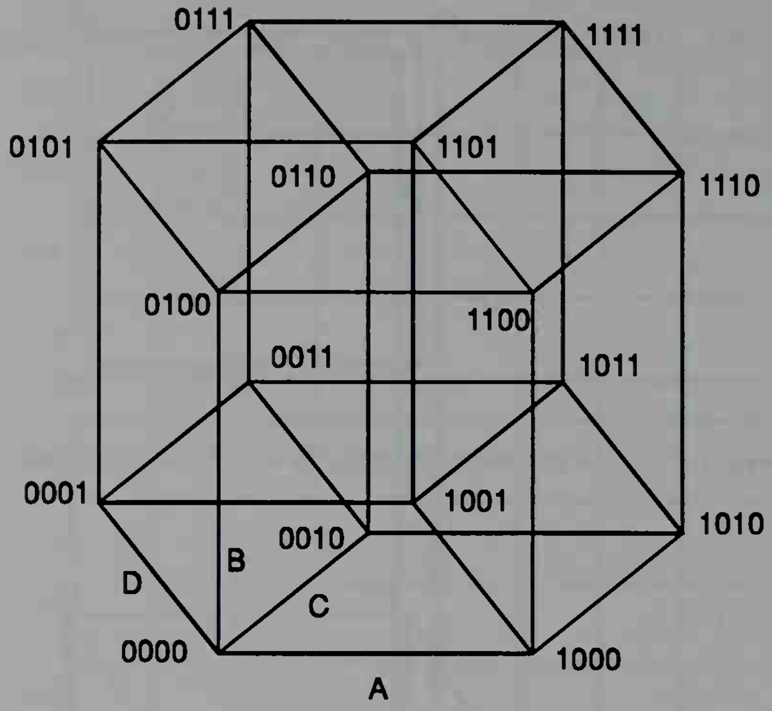


Figure 3.9 Cube representation for four variables (hypercube)

Finding Cubes and Subcubes

Staying with the 3-cube for the moment, let's examine some of the ways in which the cube representation can be used. The 3-cube has, as components, a number of smaller cubes. Each face of the cube, in fact, is a 2-cube. Every 3-cube, then, has six smaller *subcubes* that are identified by the input variables represented by the line segments associated with the subcube. In Figure 3.10, the indicated subcube is

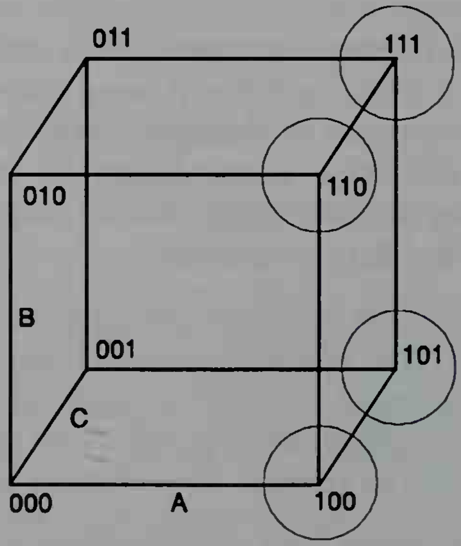


Figure 3.10 Identifying subcubes on a 3-cube

defined by the 3-cube corners with the values *100*, *101*, *111*, and *110*.

Notice that the first variable in these four points remains unchanged, while at least one of the second and third variables changes from one point to the next. The second and third variables are therefore the variables that define the subcube. These variables are said to be unspecialized in relation to that subcube, while the first variable is specialized. If we specialize one of the two variables of the resulting 2-cube (or specialize two variables of the 3-cube), we end up with a 1-cube. We can even go as far as specializing all the variables of a cube, in which case we find ourselves with 2^n 0-cubes, where *n* is the number of input variables (a 0-cube simply corresponds to one point on a cube). If we map the minterms of a logic function onto a cube, we can quickly recognize subcubes of the function. Consider the following 3-variable minsum function:

```
  !A & B & !C
# !A & B & C
# !A & !B & C
# !A & !B & !C
# A & !B & !C
```

Figure 3.11 shows how this function is mapped onto a 3-cube. Four of the minterms of this function form a complete 2-cube. This 2-cube is defined by specialization of the variable *A*, so we can immediately recognize that these four minterms can be replaced by a single product term that consists of nothing but *!A*. In addition, the 1-cube that is

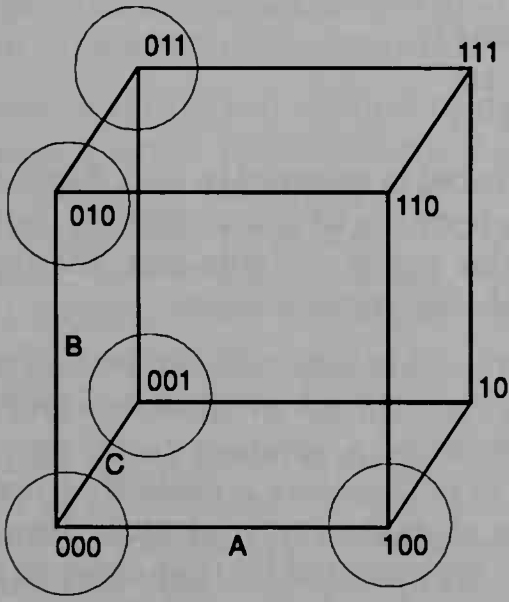


Figure 3.11 Mapping a 3-variable function onto a cube

formed by the minterms 100 and 000 can be reduced to the simpler product term *!B & !C*, since *A* is unspecialized for that subcube. The minimized function is then

```
!A # !B & !C
```

The K-map presented earlier is simply a convenient representation of a cube and a method for quickly identifying subcubes.

Tabular Representation of Sum of Products

In the K-map and cube representations, we use the values 0 and 1 to represent the complement and true values of input variables. This helps to simplify the identification and combination of cubes and subcubes within the function, but requires that the function be expressed in its minsum form.

Many logic minimization algorithms utilize another representation that doesn't require that the function be expressed in minsum form. Like the K-map representation, complement and true variables are indicated by a 0 or 1, respectively, and another symbol, the dash, is used to indicate that a variable of the function isn't used (is unspecialized) for a specific product term. To represent the four-variable function

```
A & !B & C & D # B & !C # A & D
```

in tabular form, for example, we write

```
1011
-10-
1--1
```

The dash symbol is referred to as a don't-care input condition since, for the product term in which a dash is found, we don't care what value that particular input variable has; the product term will evaluate to 1 regardless of that input's value.

As you may have already realized, this form is convenient for recognizing or obtaining such things as minterms and cubes. A minterm is indicated whenever there is a product term row that contains no dashes. A minsum form is therefore a table that contains no dashes whatsoever. The number of dashes in a product term indicates the dimensions of the subcube represented by that entry in the table: two dashes indicate a 2-cube, one dash indicates a 1-cube, and no dash indicates a 0-cube, which is, as we said, a minterm.

To convert the preceding table to minsum form, you replace each row of the table with however many rows are required to eliminate the dashes. For example, the last row becomes four rows as follows:

```
(1)    1--1   10-1
              11-1

(2)    10-1   1001
              1011
       11-1   1101
              1111
```

Prime Implicants

In a sum of products expression, each product term is called an implicant of the logic function. If any product term evaluates to a 1, it implies that the entire function will evaluate to a 1, regardless of the value of the other product terms. Consider the following minsum expression:

```
  !A & B & C & D
# A & B & C & D
# A & B & !C & D
# !A & B & !C & D
```

Each of the four minterms in this expression is an implicant of the logic function represented by the expression since, if any of these minterms evaluates to a 1, the entire expression will also evaluate to a 1. These four minterms aren't the only implicants of the function, however.

When the expression is simplified, we find that the first two minterms can be combined into a single product term *B & C & D* by elimination of the variable *A*. Similarly, the third and fourth minterms can be combined to form the product term *B & !C & D*, also by elimination of the variable *A*. These two smaller product terms are therefore also implicants of the function, since any input condition that results in either of these product terms evaluating to 1 will result in the entire expression evaluating to 1 as well.

The implicant product terms *B & C & D* and *B & !C & D* can, of course, be simplified further by eliminating the variable *C*. The resulting product term *B & D* is what is called a *prime implicant* of this logic function. Since this logic function has only one prime implicant, that single product term represents the minimal form for the logic function.

To summarize, a prime implicant is any implicant of a logic function that is not implied by any other subfunction. A logic function may have any number of prime implicants.

Minimal Cover

The set of prime implicants for a given logic function is unique, since it is determined from a unique set of minterms. It isn't always necessary, however, to use all the prime implicants for a function to obtain the minimal sum of products form. This is true because one prime implicant for a function may cover a number of minterms, and overlaps can exist; a single minterm can be covered by more than one prime implicant.

The goal of total sum of products logic minimization is to determine a minimal subset of prime implicants that will cover all the minterms of the function. This subset of prime implicants is called a *minimal cover*. The important thing to understand about prime implicants is that the fully minimized form of a logic function will consist of nothing but prime implicants, but may not require all the prime implicants for the function.

Essential Prime Implicant

The search for the elusive minimal cover of prime implicants can be hastened if we first attempt to determine which prime implicants are required in a fully minimized sum-of-products representation of the function. These prime implicants are known as *essential prime implicants* and can't be eliminated from the function by any means. For example, the three-variable function

```
!A & !C # !A & B # A & C # A & !B
```

is composed entirely of prime implicants. This is not, however, the minimal sum of products form for this function. Two of the prime implicants, *!A & !C* and *A & !B*, are essential prime implicants and must be maintained. The remaining two prime implicants are not essential and can be replaced by the single prime implicant *B & C*, resulting in the function

```
!A & !C # B & C # A & !B.
```

The Quine-McCluskey Procedure

The methods of logic minimization presented thus far are useful to help understand the minimization process, but are not generally applicable to large designs with more than five or six variables. In addition, these methods are difficult to implement on a computer, due to their reliance on pattern recognition.

The Quine-McCluskey procedure is a tabular method of logic minimization that can be performed manually or implemented on a computer. This method is composed of two algorithms. The first algorithm determines all the prime implicants for a given logic function, and the second selects from this set of prime implicants a subset that provides a minimal full cover for the function. The Quine-McCluskey procedure has been largely replaced by more efficient minimization methods, but it is nonetheless valuable for developing an understanding of logic minimization algorithms in general.

Determining Prime Implicants

As in the K-map method, the unity theorem (*A* & *B* # !*A* & *B* = *B*) is used. A simplistic way to describe this process is to say that the algorithm first determines the complete set of minterms for the function and then utilizes the unity theorem repeatedly for all possible pairs of product terms to obtain all the prime implicants of the function. In this process, each pair of minterms is examined to determine if the two minterms differ by exactly one position. Pairs so identified are combined into a single product term having one fewer literal than the previous two. After all minterm pairs have been examined, the process is repeated for the resulting terms. This process is continued until we are left with the set of prime implicants. A prime implicant is identified by the fact that it can't be compared in this way with any other term.

This determination of prime implicants can be very time consuming, particularly for functions with a large number of variables. The Quine-McCluskey method uses the tabular representation described earlier to speed this process.

We'll use an example to illustrate the algorithm. Consider a four-input logic function

$$f(w,x,y,z)$$

To perform a tabular comparison, the minterms for this function are listed in order of the number of ones in the tabular assignment, as

w x y z	
0 0 0 0	!w & !x & !y & !z
0 0 1 0	!w & !x & y & !z
0 1 0 0	!w & x & !y & !z
1 0 0 0	w & !x & !y & !z
0 1 0 1	!w & x & !y & z
1 0 1 0	w & !x & y & !z
1 1 0 0	w & x & !y & !z
1 0 1 1	w & !x & y & z
1 1 0 1	w & x & !y & z
1 1 1 1	w & x & y & z

Figure 3.12 Minterms for a 4-input logic function

shown in Figure 3.12. This ordering assists in the identification of paired minterms.

Figure 3.13 shows the steps used in the process of searching for prime implicants. The second list is determined by combining those minterms that differ by one position, as described previously. In the second list, the dash (don't-care) replaces the removed variable. The two terms that are combined to create the entry in the second list are marked with an asterisk in the first list, indicating that they can't be prime implicants.

```
LIST 1          LIST 2          LIST 3

0000 *          00-0 *          -0-0
0010 *          0-00 *          --00
0100 *          -000 *          -10-
1000 *          ------
------          -010 *
0101 *          010- *
1010 *          -100 *
1100 *          10-0 *
------          1-00 *
1011 *          ------
1101 *          -101 *
1111 *          101-
                110- *
                ------
                1-11
                11-1
```

Figure 3.13 Finding prime implicants by tabular comparison

Having marked an entry in the first list doesn't complete the comparison process for that entry, however; each entry in the list must be compared with every other entry to eliminate all possible nonprime implicants. This means that there are n^2 comparisons required for the first list alone, where n is the number of minterms in the list. In practice, the ordering of the list (by the number of ones in each product term) means that not all pairs need to be examined, speeding the comparison process.

When all the entries in the first list have been so compared, the process is repeated for the second list to create the third list. Note that, during the comparisons for the second and subsequent lists, candidate pairs must have dashes in the same position. It isn't possible, for example, to compare the terms *1-00* and *-101*.

When all the entries in the second list have been compared, three entries are left that are not marked. These entries are prime implicants. Attempts to compare the third list produce no combinations, so we are left with the three prime implicants from the second list plus the three in the third for a total of six prime implicants for the function. This technique for finding prime implicants has general applicability beyond the Quine-McCluskey method and is frequently used.

Determining a Minimal Cover

The second step in the Quine-McCluskey method is the determination of a minimal cover for the function, using the prime implicants found as a result of the first step. This is done by first creating a covering table as shown in Figure 3.14. The covering table shows all the minterms of the function and which minterms are covered by which prime impli-

	0000	0010	0100	0101	1000	1010	1011	1100	1101	1111
1-1-						X	X			
1-11							X			X
11-1									X	X
-0-0	X	X			X	X				
--00	X		X		X			X		
-10-			X	X				X	X	

Figure 3.14 Covering table

	1011	1111
101-	X	
1-11	X	X
11-1		X

Figure 3.15 Reduced covering table

cants. Obviously, it will speed the process if this covering table is constructed from the comparison information determined during the first step. The rows of the table list the prime implicants identified, while the columns represent the minterms of the function.

When the covering table has been completed, we can easily identify the essential prime implicants. These are the implicants (in this case the terms *-0-0* and *-10-*) that are the only covers for one or more minterms. In the covering table, the essential prime implicants are entered in bold-face.

Since we know that the essential prime implicants must appear in the minimal cover, we can simplify the covering table by removing the rows corresponding to these essential prime implicants and all the columns corresponding to minterms covered by them. This simplification leaves us with the reduced covering table shown in Figure 3.15.

This reduced table shows that the prime implicant *1-11* covers both of the remaining minterms, so this prime implicant, combined with the two essential prime implicants previously identified, comprises the minimal cover for the function. The minimal cover, expressed in equation form, is then

```
w & y & z # !x & !z # x & !y
```

3.2 ON-SETS, OFF-SETS, AND DC-SETS

When a logic function is described, it can be expressed in one of many ways. When we describe a function using the form of Boolean equations we presented earlier, we are describing the set of input conditions that will cause the expression to evaluate true. We call this set of input conditions the *on-set*. All other input conditions will result in the expression evaluating to false. This set of implied conditions is called the *off-set*. When we describe a function by specifying only its on-set (as when using Boolean equations), we must assume that all conditions not covered in the description comprise the off-set. Similarly, if a function

```
truth_table([A,B,C] -> [Y])
            [0,0,0] -> [0];
            [0,0,1] -> [1];
            [0,1,1] -> [1];
            [1,1,1] -> [1];
            [1,1,0] -> [0];
            [1,0,0] -> [0];
```

Figure 3.16 Incompletely specified truth table

is described only by its off-set, then it must be assumed that all unspecified conditions form the on-set.

The truth table representation is used to describe a logic function by specifying both its on-set (indicated by ones) and off-set (indicated by zeroes). For many (perhaps most) logic designs, certain input conditions will never be encountered by the circuitry being described. Also, some input conditions may only occur at times when the output of the circuit will be unused. This information about the design can be used to advantage when minimizing the logic for that circuitry. If we know that, for certain input conditions, the output may be either true or false with no effect on the operation of the system, we can reduce the amount of circuitry required to implement the function. These input conditions are collectively referred to as the don't-care set, or *dc-set*. This is best illustrated in a truth table. Figure 3.16 shows an ABEL truth table representation of a three-variable logic function.

This truth table is only partially complete; the value of the output variable *Y* is missing for the input conditions, where *A*, *B*, and *C* are *111* or *001*. This truth table is what we call an incompletely specified truth table.

The missing conditions are implied by omission, and are the don't-care conditions for *Y*. The truth table rows that specify a 1 for *Y* are conditions belonging to the on-set, while the remaining rows are the conditions

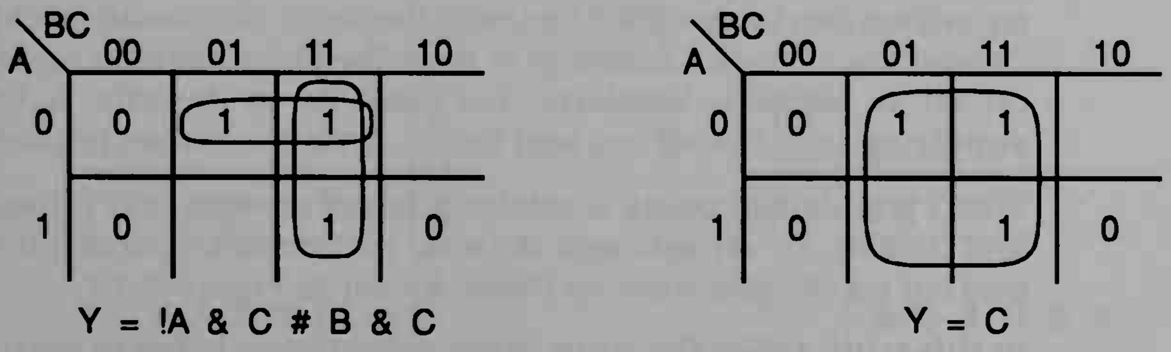


Figure 3.17 Two forms of a partially specified logic function

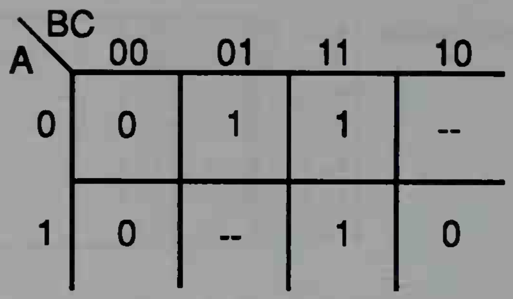


Figure 3.18 Indicating don't-cares on a K-map

that are members of the off-set. The two unspecified conditions form the dc-set for this function. There are four possible nonredundant implementations for this function; two are shown in Figure 3.17. As this example demonstrates, the intelligent use of don't-cares can have a dramatic affect on the size of the circuit.

Determining the minimal implementation of a function containing don't-care conditions is simply an extension of the principles outlined earlier. To represent don't-care input conditions on a K-map, we place a dash (-) symbol on the cell corresponding to that input condition, as shown in Figure 3.18.

The K-map can be processed as though the dash symbol represented either a 1 or 0, whichever results in the minimal cover. For this K-map, we can see that replacing the dash in the lower row with a 1 will result in a reduction of logic, whereas replacing it with a 0 will not.

For many purposes, it's useful to think of the don't-care value as an actual circuit condition. This is a common approach used in circuit simulators. This means that each variable now has three possible values: true, false, and don't-care. The truth table representation allows us to specify a function by supplying the on-set and off-set, with the dc-set being implied by omission from the table. When describing a function with equations, it's usually more convenient to describe the function by supplying its on-set and dc-set. (In ABEL, the dc-set equations can be specified by using the don't-care assignment operators ?= and ?:=.) When a function is described by its on-set and dc-set, the off-set is implied. Similarly, it's possible to describe a function by supplying only the off-set and dc-set, with the on-set implied.

When you design using a combination of on-sets and off-sets, on-sets and dc-sets, or off-sets and dc-sets, you must be careful not to create overlap conditions such as those shown in Figure 3.19.

In this truth table, the same input condition is listed in both the on-set and off-set for *Y*. This is easy to detect in a truth table (or K-map), but is not so easy to detect if on-sets and off-sets are being expressed in

```
truth_table([A,B,C] -> [Y])
            [1,1,0] -> [0];
            [1,0,0] -> [0];
            [0,0,1] -> [0];
            [0,1,0] -> [1];
            [0,1,1] -> [0];
            [0,1,0] -> [0];   "Error here
            [1,1,1] -> [1];
            [1,0,1] -> [1];
```

Figure 3.19 Overlapping truth table conditions

some other form (notably Boolean equations or state transitions). This is a common area of confusion for users of computer-based logic minimization tools. (The ABEL compiler will, in most cases, detect overlaps such as this and report an error.)

3.3 MULTIPLE-OUTPUT MINIMIZATION

When targeting a design to a simple PLD, the preceding techniques are adequate, since there is no sharing of logic from one device output to another. In PLA-type devices, however, there is opportunity for improvement. The total amount of circuitry required to implement a multiple-output function can be reduced if all outputs of the function are evaluated together.

Figure 3.20 shows two three-variable functions expressed as K-maps. Each function has two prime implicants as shown. The K-maps, when overlaid, show that one prime implicant is common to both functions. Considered together, then, it is possible to implement both of these functions using a total of three product terms.

The prime implicant shared by the two functions is called a multiple-

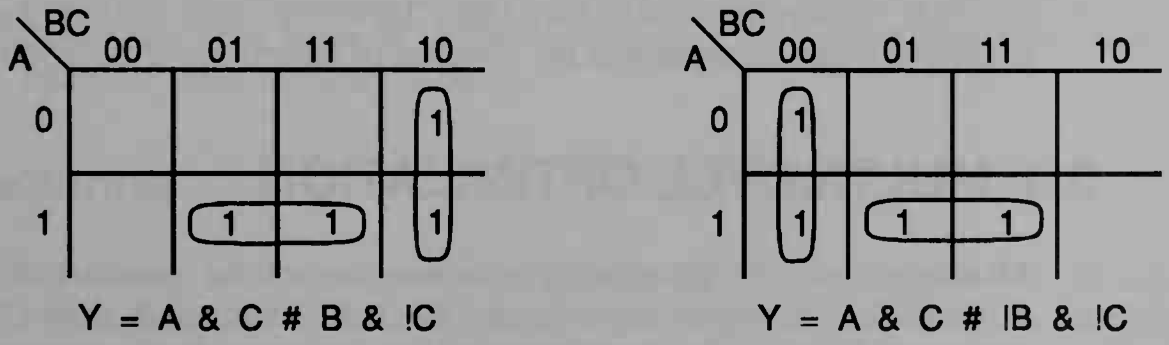
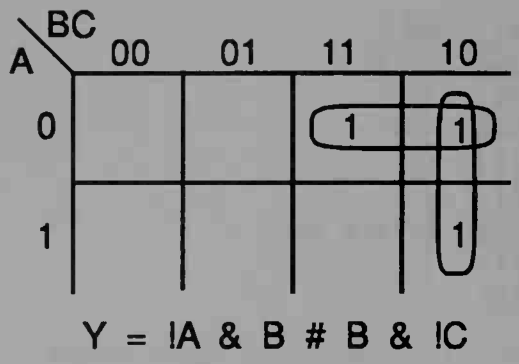


Figure 3.20 Two functions with a shared prime implicant



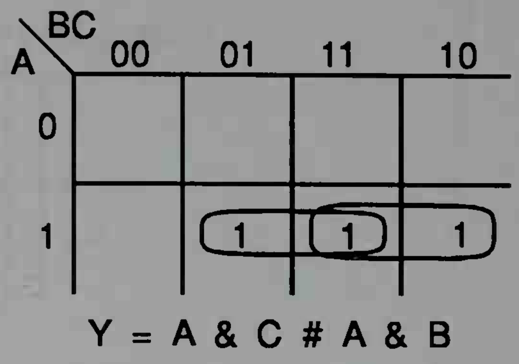


Figure 3.21 Two independent functions

output prime implicant. The two prime implicants that are not shared are called single-output prime implicants. In this example, the minimal form for the multiple-output function is the same form that results from the outputs being minimized individually. This is not, however, always the case. Consider the example functions shown in Figure 3.21.

If we minimize these two functions separately, the results are the groupings indicated, which correspond to the minimized equations

```
Y1 = !A & B # B & !C;
Y2 = A & C # A & B;.
```

There are no common prime implicants in these functions. If, however, we identify those 1-cells that are common to both K-maps and group in a manner that isolates those cells, we can obtain a set of implicants (not necessarily prime implicants) that will provide a minimal cover for the combined functions. Figure 3.22 shows such a grouping and the corresponding Boolean equations for the two functions.

Notice that the two functions now require a total of three product terms, compared to the original four. Notice also that the product term *A & B & !C* is not a prime implicant of either function and is therefore not a single-output prime implicant. It is a multiple-output prime implicant for this multiple-output function, however, since there is no other multiple-output implicant for this set of functions that implies it.

3.4 MULTILEVEL OPTIMIZATION

All the techniques described previously have been used to minimize sum of products Boolean equations. We have assumed that the sum of products form is the most optimal (in terms of gate requirements). In the real world, however, many constraints beyond the product term

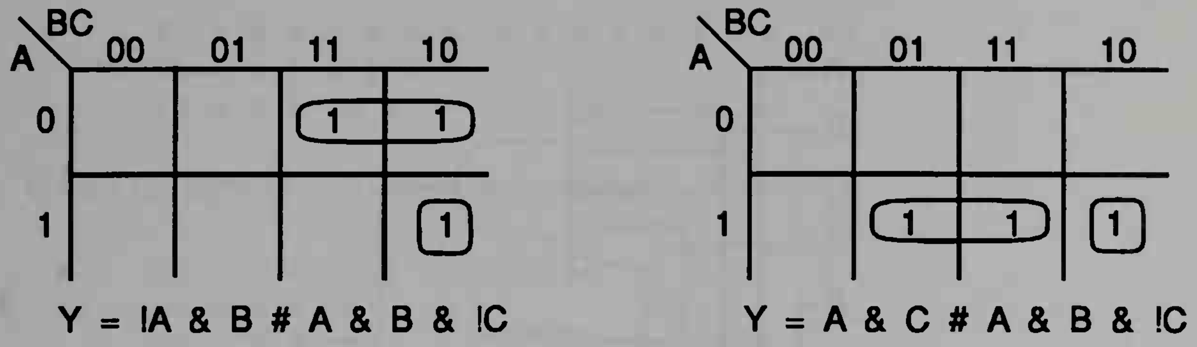


Figure 3.22 Isolating shared implicant

count must be considered. These include the problems of fan-out, constrained numbers of inputs to gates or sum of product logic blocks, and problems of timing hazards. For these reasons, it is often necessary to implement a design using multiple levels of logic. An example of this is found in the following sum-of-products equation:

```
Y = !A & !B & !C & !D & !G
    # !A & !B & F
    # !A & !B & D & E & !G;
```

This seven-variable function might be implemented as shown in the circuit of figure 3.23.

If, however, the constraints of the target implementation include a requirement that all gates have four or fewer input variables, some changes will be required. Figure 3.24 shows how the function can be implemented in a multilevel form to meet the four-input gate criteria. In a multilevel circuit, some or all of the signal paths experience a delay of more than two gates.

Actually, many possible multilevel circuits could have been constructed to implement the function within the constraints specified. The circuit shown takes advantage of the fact that there is a term (*!A & !B*) that is common to all three product terms of the original sum-of-products function. This term is isolated by a process called *factoring*, and the isolated term is called a factor.

Factoring

Factoring is the primary technique for converting two-level representations into multilevel forms. Factoring is useful for those situations where an equation is too large (too many inputs or too many product terms) to fit into the logic dedicated to one PLD output or if other

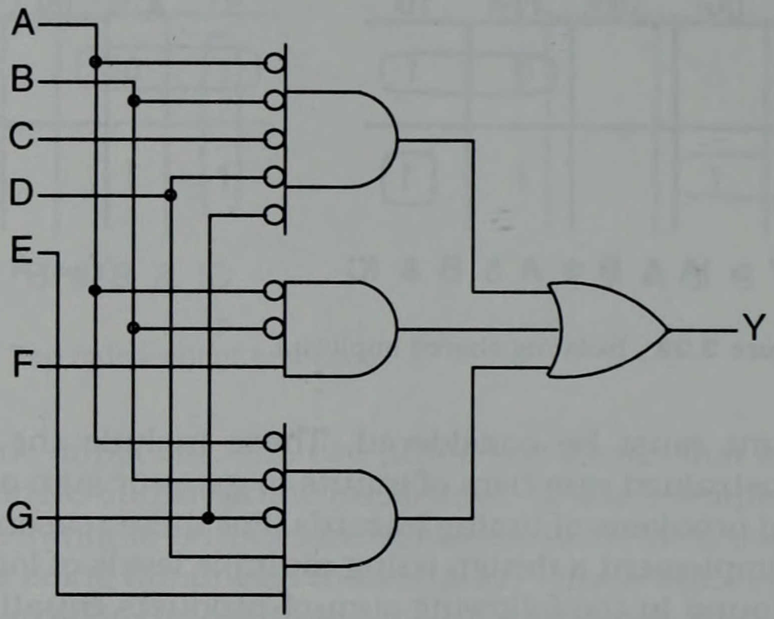


Figure 3.23 2-level form of a 7-variable function

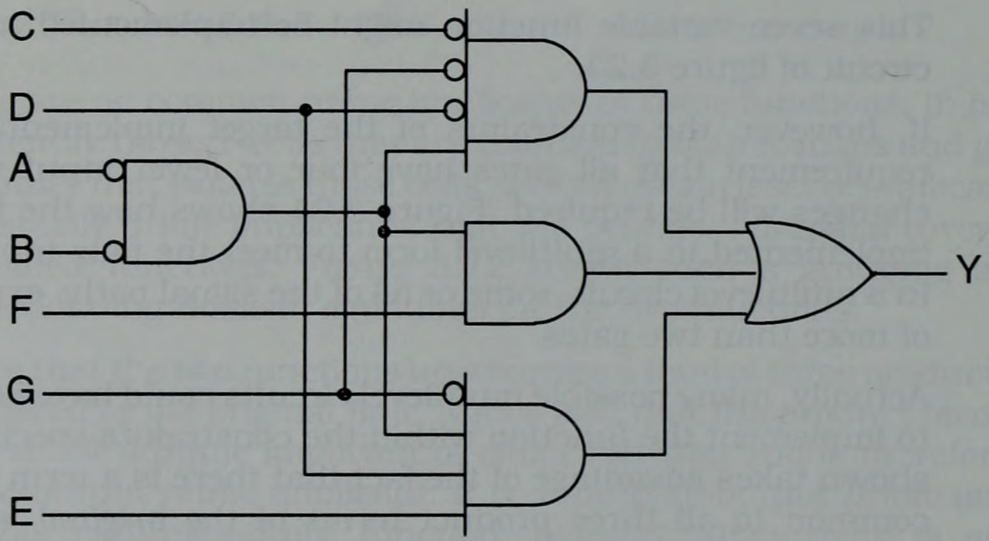


Figure 3.24 Multilevel form of 7-variable function

electrical constraints (such as fan-out restrictions) prevent the use of two-level logic.

In factoring, we break up a two-level sum of products equation into multiple levels of sum of products equations by using intermediate variables. For example, the sum of products equations

```
Y0 = A & B # A & !C # D & B # D & !C;
Y1 = D & B # D & !C;
Y2 = E & !B & C;
```

could be factored into the equations

```
Y0 = A & f0 # D & f0;
Y1 = D & f0;
Y2 = E & !f0;
f0 = B # !C;
```

for a savings of one product term. The variable *f0* is the *intermediate factor variable*. In this example, a reduction in the total number of product terms was achieved, but this isn't always the case. Often, the constraints that drive the need for factoring result in a larger set of equations. Factoring, then, can be used as a tool for multilevel logic minimization or as a tool for meeting other constraints that may actually result in larger circuits.

It may seem that factoring is of no use for simple devices like PLDs, since they are usually inherently two-level in their design. Although it is true that most PLDs are essentially sum-of-products logic devices, the intermediate signals (factors) can often be allocated to unused combinational outputs of the PLD and fed back into the sum-of-products array as an input. Many of the newer PLDs feature folded arrays or buried nodes that allow multilevel logic to be mapped into the devices with no waste of I/O pins.

The ABEL compiler will do a rudimentary factoring (in the form of intermediate carry generation) if the @CARRY directive is specified for large set expressions such as adders, counters, and comparators. More comprehensive factoring is typically performed by device fitters, since multilevel optimizations are usually better performed when the software has knowledge of the target device architecture.

3.5 OPTIMIZING WITH EXCLUSIVE-ORS

Exclusive-OR gates, or XORs, are found in many of devices, and these gates can be used to advantage in a number of ways. When we add an XOR gate to the usual sum-of-products structure, we have a structure that is referred to as an XOR of sum of products. This form is shown in Figure 3.25.

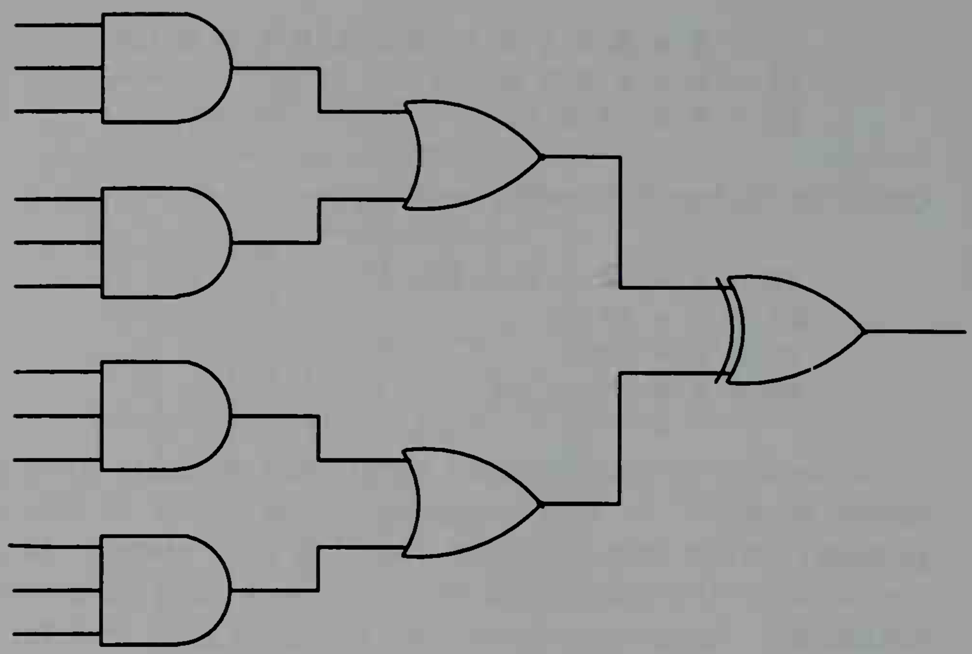


Figure 3.25 XOR of sum of products

Standard Forms For XOR Functions

One form of Boolean equation that utilizes the XOR-of-sums-of-products structure is called the Reed-Muller form, in which all literals of the Boolean equation are asserted true, as in the equation

```
X = (A & B # C & D) $ (D # E & F);
```

Like the sum of products form, any logic function can be expressed in Reed-Muller form. Since no input inverters are required, the Reed-Muller form requires the same number of gate delays (in terms of gate count from input to output) that the sum of products form requires, while requiring a smaller total number of logic gates for most functions.

Another form is called the generalized Reed-Muller, or GRM, form. In the GRM form, there may be a mixture of true and complement literals, but no variable of the function can appear in the equation in both its true and complement form. If both the true and complement literals for a variable appear in the equation, the equation is said to be in a mixed GRM form. Mixed GRM is the form used in PLDs that have XOR gates fed by sum of products arrays.

Optimizing for Constrained XORs

There are many different implementations of XORs in programmable devices, all of which have certain constraints. Some of these implementations are asymmetrical, and some allow term sharing between the OR gates. We can exploit the behavior of these built-in XORs for efficient polarity control of complex equations. Consider the following XOR equation:

```
X = (A # C & B # C & D) $ (E # F);
```

If we wish to implement this equation in a programmable device such as the 20X8 PAL (which has an XOR arrangement similar to that just shown), it will be necessary to make some modifications to the equation since, as the equation is written, it is inappropriate for the structure of the device.

First, it will be necessary to complement both sides of the equation, since the 20X8 features inverters on its outputs. This is easily done with an XOR equation if we utilize the XOR complement identities presented earlier. According to those rules, it is possible to complement the entire equation simply by complementing one sum of products input to the XOR. For the preceding equation, either of the following equations might be appropriate:

```
!X = (A # C & B # C & D) $ (!E & !F);
```

or the alternative form

```
!X = (!A & !B & !D # !A & !C) $ (E # F);
```

In the first equation, the right side of the XOR operator was complemented (*E # F* became *!E & !F*), while in the second equation, the left side of the XOR was complemented (*A # C & B # C & D* became *!A & !B & !D # !A & !C*).

Complementing the inputs to the XOR gates was accomplished through the use of DeMorgan's theorem. Notice that both forms of the complemented equation use a similar amount of product terms and literals and therefore require the same amount of circuitry. The first equation, however, cannot be implemented in the 20X8 device, since in the 20X8 only two product terms are available on each side of the fixed XOR gate. The first equation given requires three product terms for one of the XOR inputs. For this implementation, the second form of complemented equation is the preferred form.

It is important to understand that, in the absence of a specific target device type, there is no right answer when making these sorts of equation optimizations. The first of the two equations, while not being appropriate for the 20X8 architecture, would be in the correct form for other types of devices (the Cypress 330, for example, which has many product terms allocated to one input of the XOR gate and only one product term allocated to the other).

The XOR complement identities can also be used to help minimize logic in situations where no equation inversion is desired. Consider the equation

```
!X = (A # C & B # C & D) $ (E # F);
```

This is the same equation we just presented, but with the output already complemented. As noted, this equation cannot be directly mapped into the 20X8 device because there are too many product terms on the left side of the equation. We can modify this equation, however, so that it will fit. All that is required is that we complement both sides of the XOR operator, in accordance with the XOR complement identities. The resulting equation

```
!X = (!A & !B & !D # !A & !C) $ (!E & !F);
```

will fit into the 20X8 device.

Using XOR Factoring

A lesser known application of XOR gates is a technique called *XOR factoring*. This method can be used to reduce the number of product terms needed to implement a design in an XOR device. Consider the following sum-of-products equation:

```
Y = A
    # X0 & X1 & X3 & X4 & X5 & X6 & X7
    # X2 & X3 & X4 & X5 & X6 & X7
    # !X0 & !X7
    # !X1 & !X7
    # !X2 & !X7
    # !X3 & !X7
    # !X4 & !X7
    # !X5 & !X7
    # !X6 & !X7;
```

As written, this equation consumes ten product terms. Because of this, it is not possible to implement this function in a PLD with only eight product terms per output. It is possible, however, to implement this equation in just four product terms, using XOR factoring.

The technique of XOR factoring is based on the XOR identities presented earlier. These identities allow us to XOR any Boolean expression with a second expression and then XOR the resulting larger expression with the second expression again, ending finally with the original expression. For example, the equation

```
X = A & B;
```

is functionally equivalent to the equation

```
X = B $ (B $ (A & B));
```

In effect, the two XORs cancel each other.

By identifying certain commonly used subexpressions in an equation and factoring these out with an XOR, we can reduce the number of product terms required for the primary expression in the equation. In many cases, this can dramatically reduce the total number of product terms required. Using the previous equation for *Y* as an example, we first modify the sum of products equation as follows:

```
Y = (!A & X7)
  $ ((!A & X7)
     $ (A
        # X0 & X1 & X3 & X4 & X5 & X6 & X7
        # X2 & X3 & X4 & X5 & X6 & X7
        # !X0 & !X7
        # !X1 & !X7
        # !X2 & !X7
        # !X3 & !X7
        # !X4 & !X7
        # !X5 & !X7
        # !X6 & !X7);
```

For this equation, we have chosen the term *!A & X7* for a factor. When we convert the equation to mixed GRM form (preserving *!A & X7* as one operand of the XOR) and minimize the two sum-of-products XOR inputs, we get an equation of the form

```
Y = (!A & X7)
     $ ((!A & X0 & X1 & X3 & X4 & X5 & X6 & X7
       # !A & X2 & X3 & X4 & X5 & X6 & X7
       # !A & X0 & X1 & X2 & X3 & X4 & X5 & X6));
```

This equation now requires only four product terms in addition to the XOR gate.

How did we decide on the XOR factor *!A & X7*? Partly through trial and error. We chose a variety of different candidate factor terms based on which literals were observed most frequently in the equation, and experimented to determine which would result in the most savings of logic. We deliberately chose a single-term XOR factor that was relatively simple. There is no reason, however, why an XOR factor can't be a complex expression that results in an even split of the equation into two similarly sized components. As you might imagine, then, the number of possible XOR factors is enormous, including subexpressions of all sizes.

The choice of XOR factor is affected to a great extent by the architecture of the device into which the design is being implemented. Clearly, if one of the OR gates that feeds the XOR has more product terms available than the other, this needs to be taken into consideration when selecting a factor term.

Exclusive-OR optimizations (such as XOR factoring) are not performed by the ABEL compiler or logic optimizer supplied with this book. These optimizations are performed only by device-specific fitting software, since XOR structures vary widely from one device to the next.

3.6 REFERENCES

Brayton, Robert K., Hachtel, Gary D., McMullen, Curtis T., Sangiovanni-Vincentelli, Alberto L., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Hingham, MA, 1984.

Breeding, Kenneth J., *Digital Design Fundamentals*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Mano, M. Morris, *Digital Design*, Prentice Hall, Englewood Cliffs, NJ, 1984.

Pellerin, David and Holley, Michael, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Unger, Steven J., *The Essence of Logic Circuits*, Prentice Hall, Englewood Cliffs, NJ, 1989.

John F. Wakerly, *Digital Design Priciples and Practices*, Prentice Hall, Englewood Cliffs, NJ, 1990

4 Sequential Circuits

Chapters 2 and 3 focused on the design and optimization of purely combinational logic. Combinational logic is distinguished by the fact that it contains no memory elements and has no feedback loops. A combinational circuit is so named because, for any given input combination, the output (or outputs) will produce a known logical result regardless of the circuit's previous condition. The behavior of sequential logic circuits, on the other hand, depends not only on the inputs to the circuit, but also on the fed-back state of the circuit. The current state (as observed on the circuit's outputs) is provided back into the circuit as one or more inputs by means of feedback.

A typical model of a sequential circuit is shown in Figure 4.1. There are two fundamental parts to a sequential circuit such as this. The first is the combinational logic that decodes the current state of the circuit as it is fed back. The second is the feedback loop and associated delay element. The delay element may be nothing more that the propagation delay of the gates in the combinational logic or as complex as a clocked memory element (a flip-flop, for example.)

4.1 SYNCHRONOUS SEQUENTIAL CIRCUITS

A synchronous *sequential circuit* features some form of periodically clocked memory element to ensure proper synchronization of circuit outputs. Such a circuit operates in regular cycles of sufficient duration that the propagation delays within the combinational logic portion do

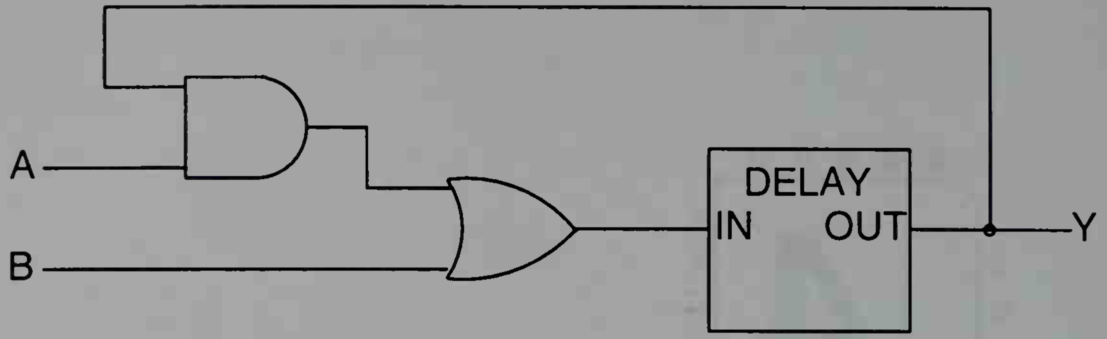


Figure 4.1 Typical model of a sequential circuit

not affect the operation of the circuit as a whole. In each cycle of operation there exists, simultaneously, information about the current state of the circuit and the next state. For this reason, these circuits are often called state machines. State machines can also be constructed that are self-timed, and do not rely on periodically clocked memory elements. These asynchronous state machines will be described later in this chapter.

A simple state machine is illustrated in Figure 4.2. This circuit is composed of combinational logic that decodes the previous state of the machine (as stored in the clocked memory element) and the state machine inputs to determine the next state. The next state is fed to the memory element and, when the memory element is clocked, the next state becomes the current state.

This simple state machine has only one fed-back signal and one corresponding memory element. Since there only two possible values for the signal and memory element, this state machine has only two possible states. The number of states possible in a state machine is 2^n, where n is the number of memory elements, or state bits, in the machine.

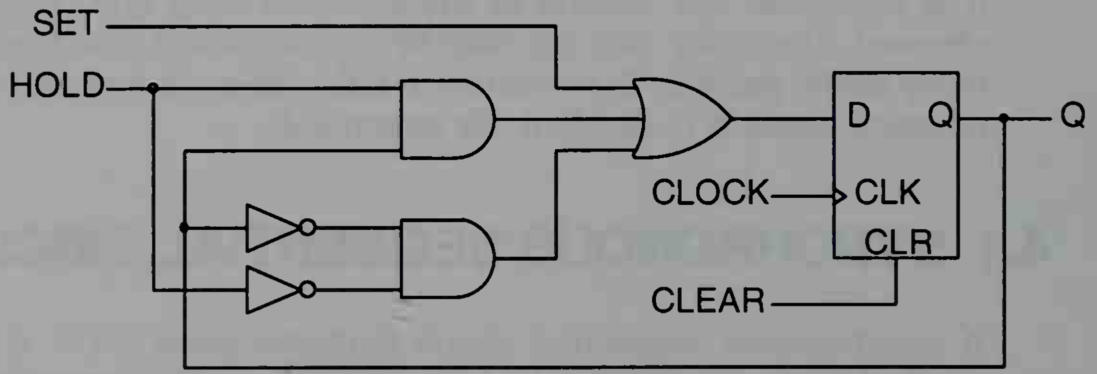


Figure 4.2 Simple state machine circuit

Synchronous Memory Elements

A variety of different types of memory elements can be used to store the current state of a synchronous sequential circuit. These memory elements include the basic flip-flop and latches familiar to most designers. The most common of these memory elements used in simple PLDs and FPGAs (the most likely target technologies for ABEL designs) is the edge-triggered D flip-flop shown in Figure 4.3.

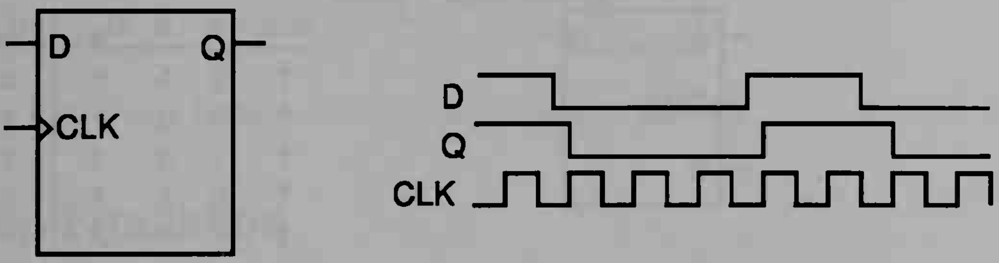


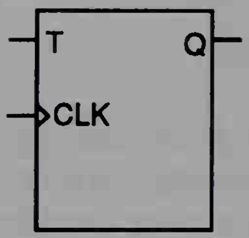
Figure 4.3 Edge-triggered D flip-flop

As the timing diagram indicates, the D flip-flop stores the value applied to its *D* (data) input whenever the *CLK* input transitions from a low to a high. The amount of time required to load the *D* input, during which that input must be stable, is called the setup and hold time. Once a value has been loaded into the D flip-flop, the *Q* output of the flip-flop will remain at that value until the next rising clock transition is observed on the *CLK* input.

A distinguishing characteristic of a clocked D flip-flop is its inherent default state. When no information is presented to the flip-flop during a clock, it will always return to the low state. This fact can be used to advantage when designing state machines, with some caveats that we'll explore later.

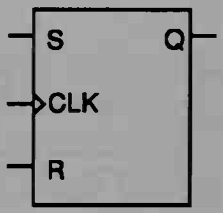
The clocked T flip-flop, shown in Figure 4.4, differs from the D flip-flop in that the *Q* output of the flip-flop doesn't reflect the value observed on the flip-flop input. Instead, a true (high) input to the *T* input during the rising clock transition will cause the *Q* output to toggle (reverse its value). This behavior is useful for counter applications.

The clocked SR flip-flop features two inputs, labeled *S* and *R*. This flip-flop is illustrated in Figure 4.5. The truth table shown in the figure defines the operation of the flip-flop. The *S* (set) input is used to load a true (high) value into the flip-flop during rising clock transitions, whereas the *R* (reset) is used to clear the flip-flop during rising clock transitions. If both the *S* and *R* inputs are asserted true during the rising clock transition, the result observed on the *Q* output is indeterminate.



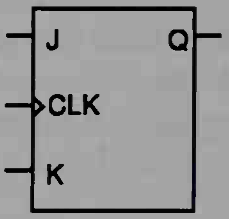
CLK	T	Q_{N-1}	Q_N
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0

Figure 4.4 Clocked T flip-flop



CLK	S	R	Q_{N-1}	Q_N
↑	0	0	0	0
↑	0	0	1	1
↑	1	0	X	1
↑	0	1	X	0
↑	1	1	X	?

Figure 4.5 Clocked SR flip-flop



CLK	J	K	Q_{N-1}	Q_N
↑	0	0	0	0
↑	0	0	1	1
↑	1	0	X	1
↑	0	1	X	0
↑	1	1	1	0
↑	1	1	0	1

Figure 4.6 Clocked JK flip-flop

The SR flip-flop does not have an inherent default state. Unspecified circuit conditions will result in the flip-flop remaining in the previous state.

The clocked JK flip-flop is shown in Figure 4.6. This flip-flop has two data inputs, labeled *J* and *K*. The JK flip-flop operates the same as an SR flip-flop, with the exception of the case when both inputs are true. In this case, the flip-flop toggles. This is shown in the truth table.

The value of the JK flip-flop lies in the fact that, by adding a simple interconnection or inverter gate between the *J* and *K* inputs, it can be made to function as a D or T flip-flop. This is shown in Figure 4.7.

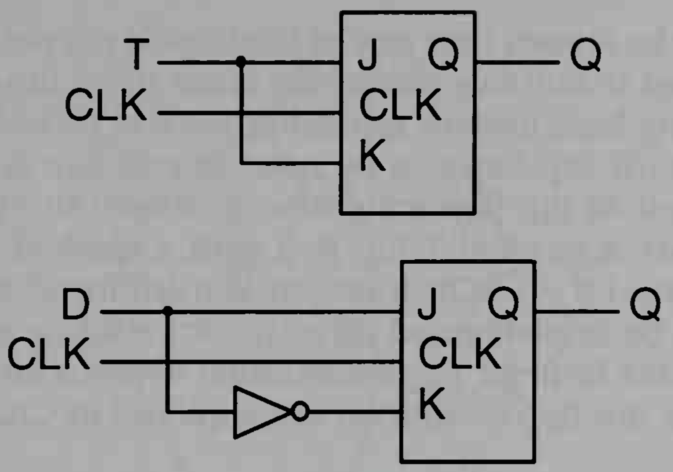


Figure 4.7 T and D flip-flop emulation using a JK flip-flop

Flip-flop Emulation

The technique of flip-flop emulation can be used to advantage when designing sequential circuits. Flip-flop emulation will frequently allow a sequential design to be implemented in a device that features flip-flops not of the type required for a particular function. XOR gates are often used for these emulations.

The type of flip-flop that is most appropriate for a design is not always available in the device being used for implementation. Devices that feature configurable flip-flops (D/JK or D/T, for example) can simplify the job of designing state machines, at the expense of device complexity. Even without configurable flip-flops, the use of flip-flop emulation techniques can help simplify the determination of sequential logic.

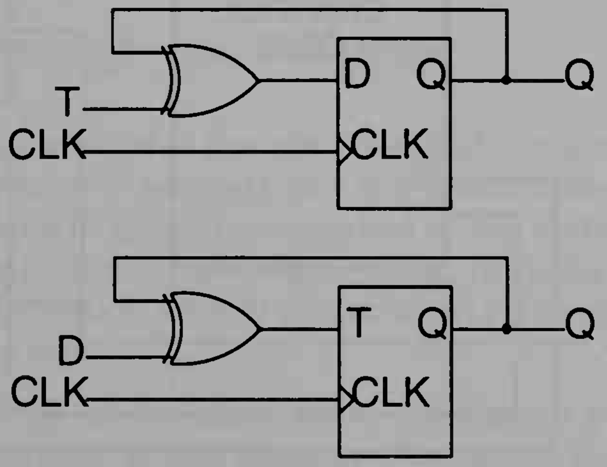


Figure 4.8 D-to-T and T-to-D flip-flop emulation using an XOR gate

It can be shown that any of the flop-flop types described previously can be used to emulate any of the other three flip-flops if appropriate input forming logic and/or outut feedback is provided. We have seen already how a JK flip-flop can be used to emulate D and T flip-flops. Another example of flip-flop emulation is shown in Figure 4.8: it is possible to emulate a clocked T flip-flop with a clocked D flip-flop (the reverse is also true) if an XOR function is introduced into the circuit. This XOR could be implemented directly, or could be expanded into the sum of products form for implementation without an XOR gate. Specific examples of flip-flop emulation are explored in Chapter 7.

Determining State Machine Logic

A four-state machine is shown as a block diagram in Figure 4.9. We haven't shown the circuitry for this machine. Instead, we'll show two representations of the machine's operation. The first (Figure 4.10) is a representation called a state graph or bubble diagram. The state graph contains circles corresponding to the four states of the machine. Each state is identified by a binary value that is the state value for that state. Each arrow corresponds to a transition from one state to the next and is annotated by the set of input conditions that will cause that transition.

Figure 4.11 shows another representation of the state machine: the state table. A state table is nothing more than a truth table segmented horizontally into three (or four) sections. The first two sections corre-

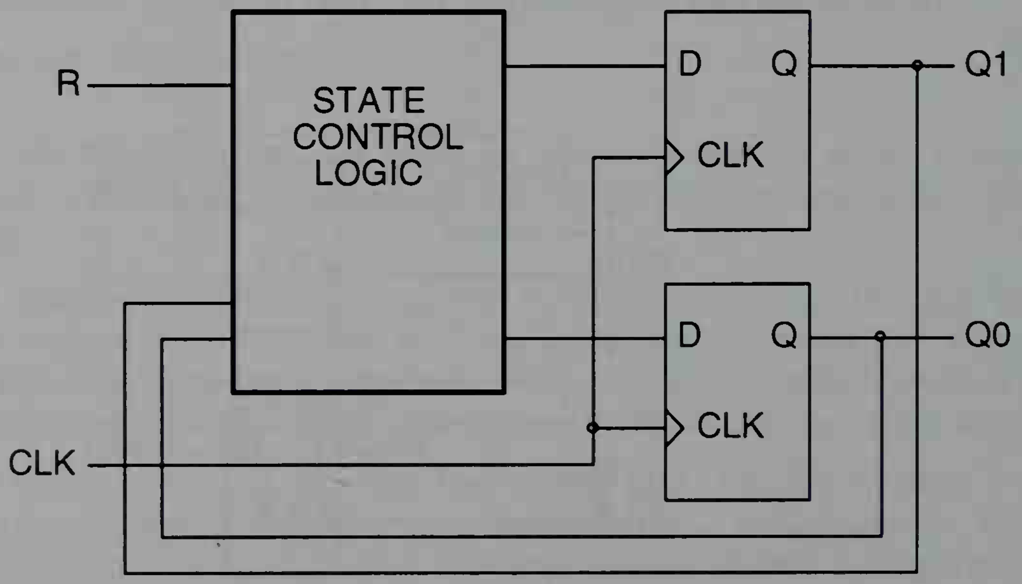


Figure 4.9 Four-state state machine block diagram

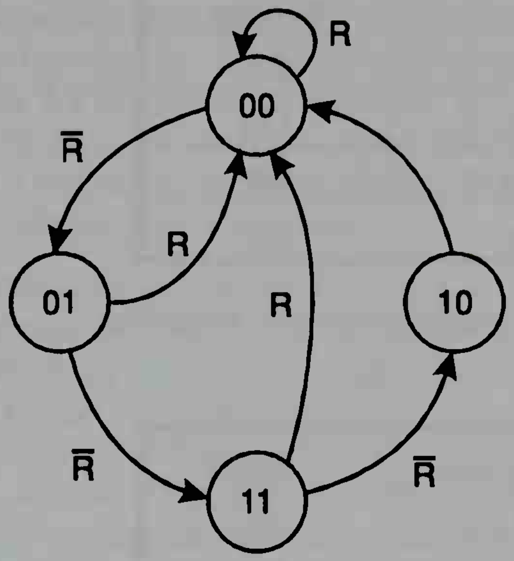


Figure 4.10 State graph of a four-state state machine

Current State	Inputs	Next State
Q1 Q0	R	Q1 Q0
0 0	1	0 0
0 0	0	0 1
0 1	1	0 0
0 1	0	1 1
1 1	1	0 0
1 1	0	1 0
1 0	X	0 0

Figure 4.11 State table for four-state state machine

spond to the current state of the machine and the state machine inputs, respectively. These two sections are the truth table inputs. The third section corresponds to the values applied to the state register and is the output of the truth table representing the next state of the machine. The fourth section (which we will utilize later) represents circuit outputs that are not state bits.

Each horizontal row of the state table represents one state transition for the state machine. If you compare the state graph with the state table, you will see that there is one state table row entry for each transition arrow in the graph.

Transition	**D Input**
0 to 0	Off
0 to 1	On
1 to 0	Off
1 to 1	On

Figure 4.12 Transition table for D flip-flop

Transition	**T Input**
0 to 0	Off
0 to 1	On
1 to 0	On
1 to 1	Off

Figure 4.13 Transition table for T flip-flop

Transition	**J Input**	**K Input**
0 to 0	Off	
0 to 1	On	
1 to 0		On
1 to 1		Off

Figure 4.14 Transition table for JK flip-flop

Transition	**S Input**	**R Input**
0 to 0	Off	
0 to 1	On	Off
1 to 0	Off	On
1 to 1		Off

Figure 4.15 Transition table for SR flip-flop

The state table is a convenient form for evaluating state transitions and determining transition logic. The determination of transition logic is done independently for each bit of the state register. To determine the logic for each state bit, we analyze each transition and convert the transition data to a set of on-set and off-set equations that are appropriate for the type of flip-flop being used.

We begin by converting each row of the state table into one product term that represents the current state and state machine inputs for that row. These product terms are called the condition product terms. The condition product terms are then applied to the flip-flop inputs according to the type of transition required (low to high, high to low, or hold). Transition tables specify the types of equations required for each possible transition in a specific flip-flop. There is one flip-flop transition table for each type of flip-flop. Flip-flop transition tables for the D, SR, JK, and T flip-flops are shown in Figures 4.12 through 4.15.

Armed with our flip-flop transition tables, we can now generate logic for each bit of the state machine. Let's begin by mapping the condition product terms for the state machine into on-set and off-set equations for D flip-flops. To do this, we refer to the flip-flop transition table for D flip-flops (Figure 4.12). This table shows us which transitions (in terms of bit changes) will require on-sets or off-sets. Beginning with Q1, we can see from the table that a change from state 0 to 1 or from 1 to 1 will require an on-set equation, whereas a change from state 0 to 0 or 1 to 0 will require an off-set equation. Using this information, we can map the condition product terms for each transition into on-sets and off-sets. The resulting on-sets and off-sets can be mapped directly into a K-map (with on-sets corresponding to 1-cells and off-sets corresponding to 0-cells) or written as equations. Both forms are shown in Figure 4.16.

The K-map can be used to minimize the logic, using the methods presented earlier. As you can see from the K-map representation, this state machine is completely specified; there are no undefined states,

R \ Q0 Q1	00	01	11	10
0	1	0	0	1
1	0	0	0	0

```
Q0 = !R & !Q0 & !Q1
   # !R & Q0 & !Q1
```

R \ Q0 Q1	00	01	11	10
0	0	0	1	1
1	0	0	0	0

```
Q1 = !R & Q0 & !Q1
   # !R & Q0 & Q1
```

Figure 4.16 K-map and Boolean representations for state machine transitions

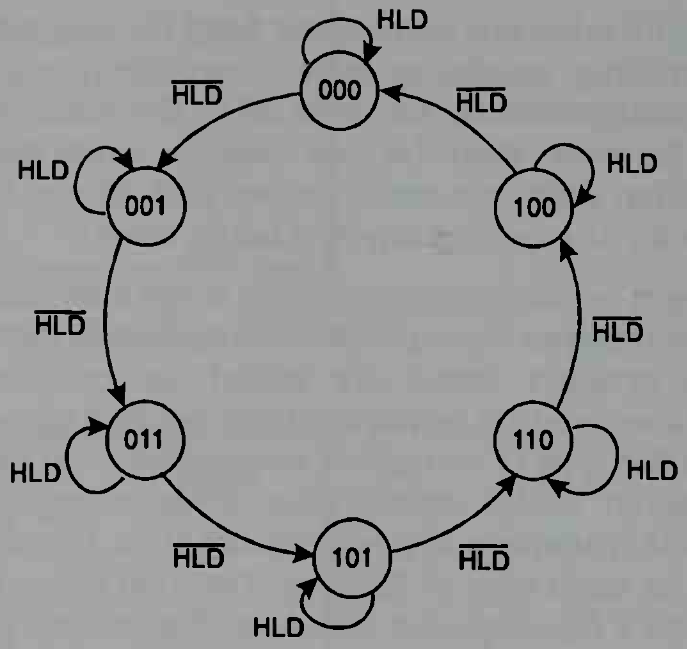


Figure 4.17 State graph for a more complex machine

and all transitions are accounted for. Now let's generate the logic for a more complex state machine: one intended for JK flip-flops and with undefined states and unspecified transitions. A state graph for this state machine is shown in Figure 4.17.

To implement this state machine in JK flip-flops, we apply the flip-flop transition table for JK flip-flops to the condition product terms. Once again, we use the bit changes (from current states to next states) to map into the flip-flop transition table. The transition table for JK flip-flops has entries for both the *J* and *K* inputs to the flip-flops. As you can see from the table, the two inputs of a JK flip-flop require the same number of condition product terms as a D type. This is because the JK flip-flop retains its data until cleared and can exploit don't-care conditions during toggle operations (transitions from 0 to 1 or 1 to 0).

The on-sets and off-sets for this state machine are shown in K-map form in Figure 4.18. The indicated groupings, utilizing don't-care minimization, result in the following optimized equations for the *J* and *K* inputs to the state register flip-flops:

```
Q2.J = Q1 & !HLD;
Q2.K = !Q1 & !Q0 & !HLD;
Q1.J = Q0 & !HLD;
Q1.K = Q1 & !HLD;
Q0.J = !Q2 & !HLD;
Q0.K = Q2 & !HLD;
```

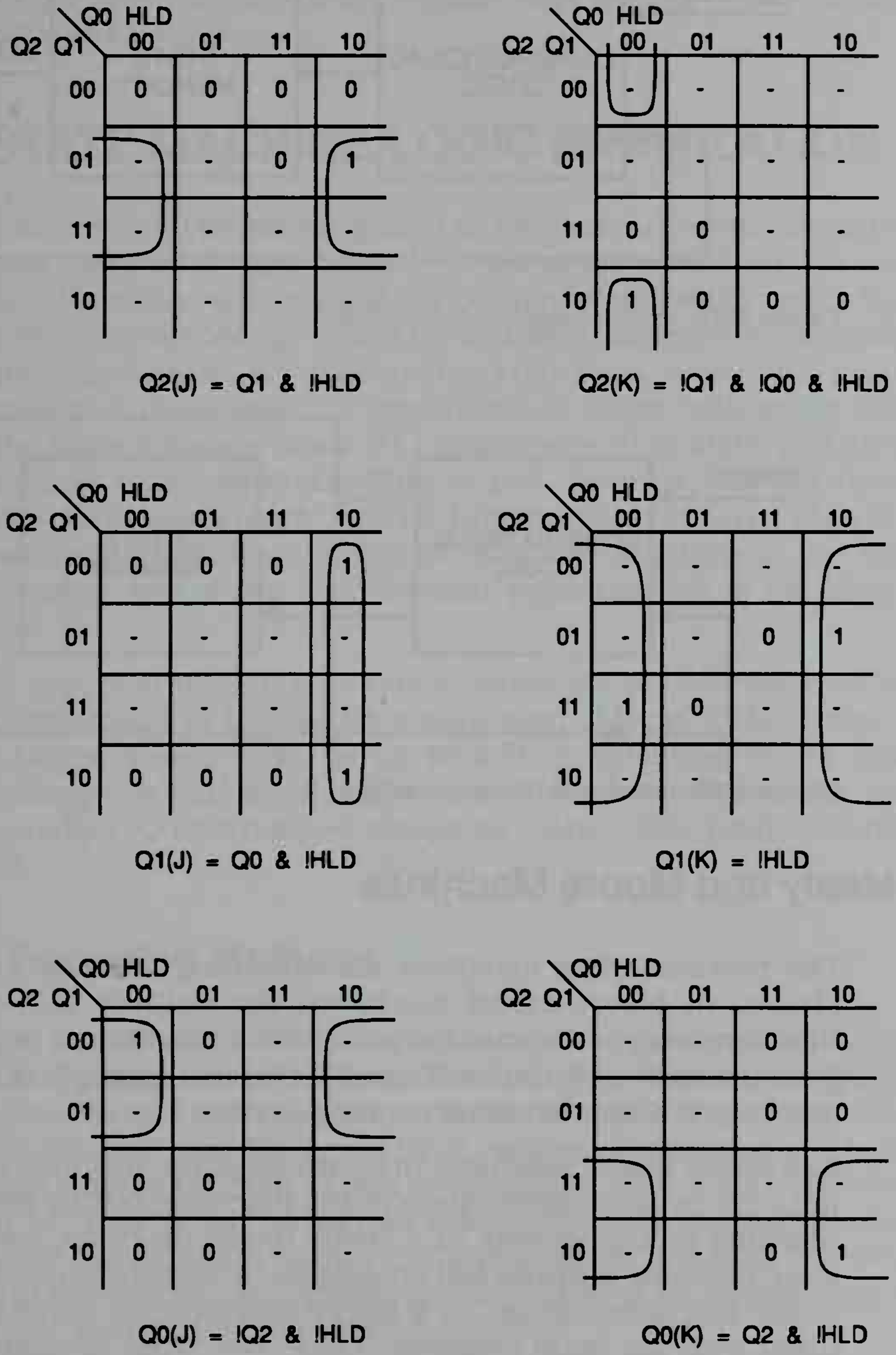


Figure 4.18 Transition logic for JK flip-flop state machine

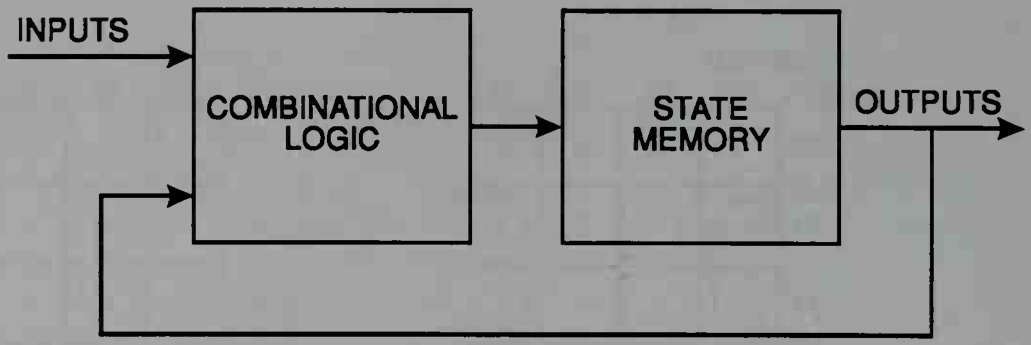


Figure 4.19 Moore model state machine

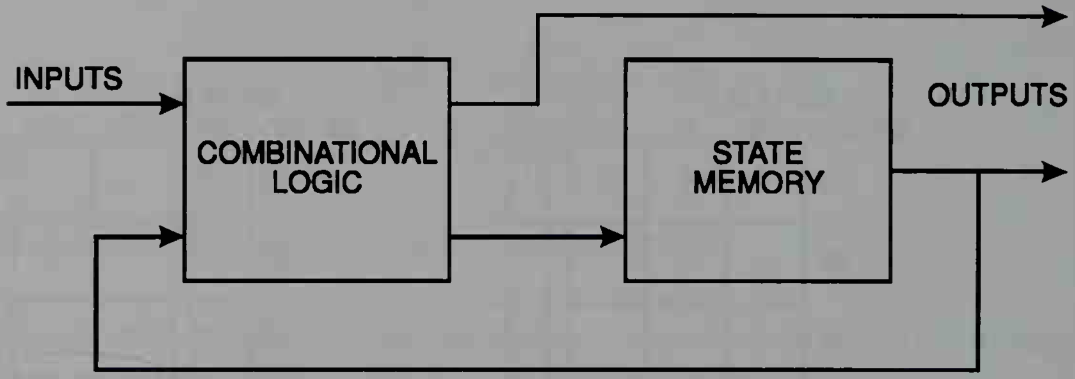


Figure 4.20 Mealy model state machine

Mealy and Moore Machines

The previous state machines are examples of *Moore model* state machines. In Moore model machines, the outputs of the machine are functions only of the current state of the machine or of inputs that are synchronized with the machine. The typical example of a Moore model machine is a synchronous counter circuit.

In a Mealy model machine, the outputs of the machine are determined not only by the current state of the machine, but by decoding of state machine inputs as well. In a Mealy model machine, some or all of the state machine outputs will change state asynchronously as the inputs to the machine change. In a Moore machine, all outputs are synchronized with the state registers. These two types of state machines are diagrammed in Figures 4.19 and 4.20.

The model of machine you are constructing can have an impact on the type of device in which the design can be implemented. For example, if you are designing a Mealy model state machine, you probably aren't concerned with the actual state values used for each state of the machine and may not even require that the state bits be accessible outside the device.

4.2 STATE MACHINE LOGIC MINIMIZATION

The amount of circuitry required to implement a state machine can be greatly reduced by the careful choice of bit values for the machine's states. How do you determine the optimal state assignment? In general, you should number the states for as few bit changes as possible for the transitions defined. If you are using D flip-flops, you should also attempt to minimize the number of transitions to states with many values of 1 in the state registers (state *011* costs more than state *100*, in terms of transition logic required). This is not, however, always the optimal assignment, particularly for PLD implementations. In a PLD, it is often necessary to optimize a single bit of a state machine at the expense of other state bits if the logic for that particular bit is too large for the device.

As a practical matter, it is rarely necessary to determine an encoding that will result in the absolute minimum of logic. If the design fits into the target device, there is no benefit in spending hours trying new encodings. In PLD implementations of state machines, the constraint is usually I/O and register resources rather than product term limitations.

State Encoding Methods

When minimization of state machines is required, one of a number of basic approaches can be taken. The first approach is to analyze the state transitions and renumber the states to minimize logic requirements for problem state bits. The brute force version of this approach is to simply assign unique random values to the states until a set of values results in a fit. This is not as bad as it sounds; a near minimal encoding is usually encountered within ten attempts, so selecting the best result out of ten random encodings may be just as good as spending the time to fully analyze the machine.

If it becomes necessary to analyze the state machine to determine a near-minimal encoding, the following steps should be taken. First, decide on a default state for transitions out of unspecified states. If D flip-flops are being used, this should be the encoding in which all state registers are 0. If T or JK flip-flops are being used, you must either provide a method for determining when unspecified (presumably illegal) states have been entered and resetting to a default state or insulate the state machine from any asynchronous inputs that could cause a false transition to an illegal state.

After determining the default state, begin assigning state values to the other states in an order determined by the number of transitions into each state. Beginning with the state having the most transitions in from other states, assign an encoding that requires the least logic: for D flip-flops, this will be an encoding that has as few ones as possible (state 0001, for example) whereas for T flip-flops, an encoding that results in the fewest bit changes is best. This latter case implies that the source states must be encoded at the same time as the destination state. This is why determining a near-optimal encoding for T flip-flops can be more tedious than for D flip-flops.

If one or more states in the machine share common transition logic to the same destination state, further reductions in logic can be achieved by numbering these source states with similar encodings (encodings that share as many state bits as possible.) For both D and T state registers, the use of don't-cares in the minimization process (as described earlier) can have a dramatic impact on the transition logic required. Using don't-cares, however, can cause problems in D and T state machines that do not have any provision for escaping from illegal states.

An alternative approach to state machine logic minimization is to reduce the input forming logic (the combinational logic feeding the state registers) by assigning a larger number of state registers to the state machine. The extreme case of this approach is *one-hot* encoding, in which a unique register is assigned to each state in the machine. One-hot encodings are most valuable for state machines that are to be implemented in FPGA-type architectures, since these architectures provide a relatively large number of registers.

4.3 METASTABILITY

Metastability is a condition that plagues synchronous designs that are required to interface with asynchronous input signals. In normal operation, digital storage elements such as flip-flops are said to be bistable; this means that they are stable in only two possible states (high or low).

A metastable state is an unpredictable, symmetrically balanced state of operation that a clocked flip-flop can enter if the input to that flip-flop changes during the setup and hold period of the clock transition. Once in this state, the flip-flop will require an unknown amount of time to *relax* to a stable (but unpredictable) state.

With an increasing number of systems being developed that rely on asynchronous circuit interfaces, the problem of metastability is becoming more widespread. This problem frequently exhibits itself when asynchronous signals are used as inputs to synchronous state machines. Typical applications in which metastability appears include arbiter circuits, bus interfaces, synchronizers, and controllers.

Device manufacturers have attempted to respond to the metastability problem by designing the flip-flops in their devices to be metastable hardened, meaning they will relax to a steady state quickly after a metastable event. Such claims should be examined carefully, however, particularly if high clock speeds are required and the application requires a high level of reliability. Even with quick recoveries from metastable events, the problem of false transitions to illegal states must be considered in the design of a state machine that includes asynchronous interfaces.

If a high level of metastable protection is required, you can utilize additional flip-flops to insulate the state machine from asynchronous events, as shown in Figure 4.21. This circuit, a bus synchronizer, has two additional D flip-flops that are used to insulate the primary storage element from metastable triggering input conditions. Note, however, that it is not possible to completely eliminate metastable events. The best a designer can hope for is the reduction of the probability of such events occurring to the point where the system can be reliably operated. Determining how far to go in protecting a given system from metastable events is therefore a function of the number of metastable-vulnerable components in the system, the expected frequency of occurrances of

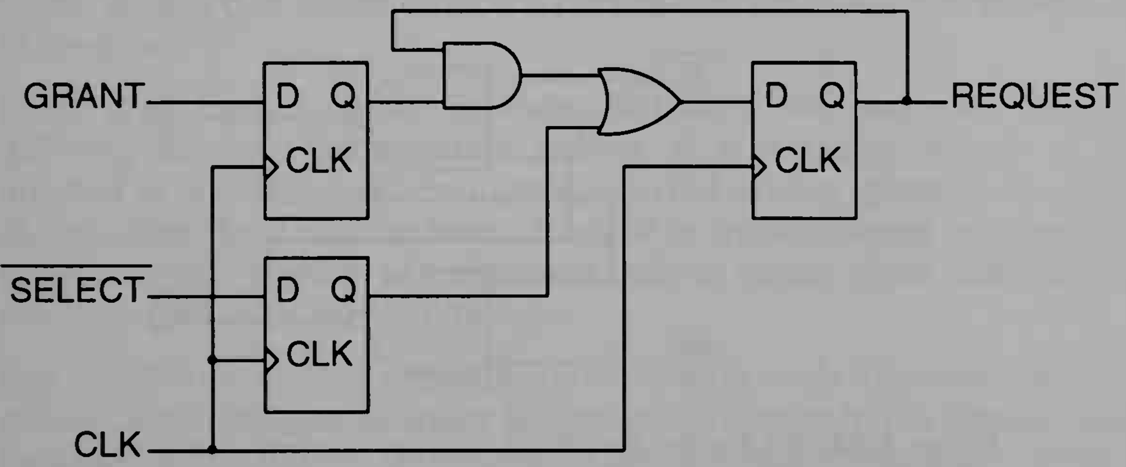


Figure 4.21 Metastable insulated synchronizer

metastable-triggering inputs, the volumes to be produced, and the life cycle of the product.

To help in these calculations, many device manufacturers publish statistics on the metastable behavior of their devices. Statistics can be used to quantify the metastable behavior of specific device types, and these statistics, if available, should be utilized to help determine the suitability of a particular implementation for asynchronous interface or arbitration applications. If a very high level of reliability is required, these statistics must be carefully applied along with the other factors discussed previously.

4.4 ASYNCHRONOUS STATE MACHINES

When no globally synchronized memory element is used in a sequential circuit or when the circuits inputs are not synchronized with the rest of the circuit, the circuit is said to be asynchronous.

Asynchronous state machines are circuits that do not rely on an external clock to provide sequencing and separation of transition events. The simplest example of an asynchronous state machine is the *set-reset (SR) flip-flop* shown in Figure 4.22.

The SR flip-flop has two states that can be observed on its outputs: high and low. As the function table indicates, the SR flip-flop must be protected from situations in which both of its inputs (*S* and *R*) are high. This is because the resulting next state when both inputs are asserted is dependent on the exact timing of the input signals.

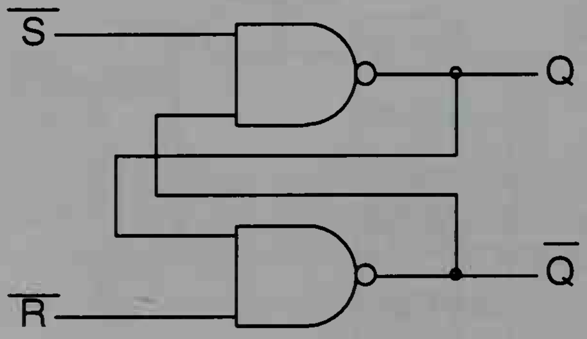


Figure 4.22 SR flip-flop circuit

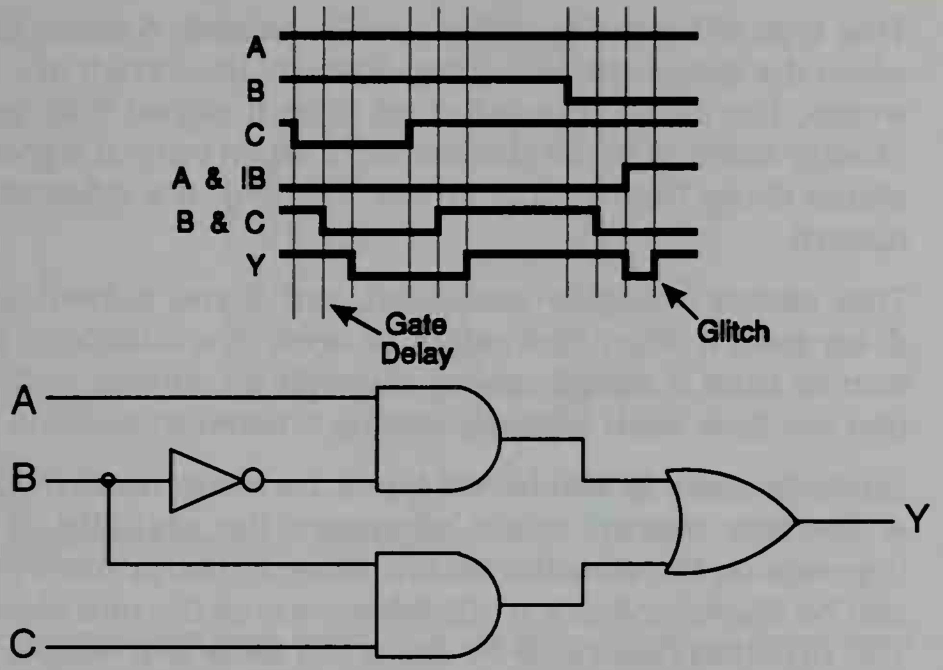


Figure 4.23 Circuit with static hazard

Hazards

Asynchronous state machines must be carefully designed to avoid hazards. *Hazards* are unpredictable conditions that can exist within logic circuits and lead to erroneous circuit operations. A major problem with hazards is in their often insidious nature. A design that works reliably when implemented in one manufacturer's device may not work at all in a seemingly identical device obtained from another manufacturer. This is due to the different timing characteristics of various types of devices.

PLDs produced by different manufacturers, for example, often have differing internal propagation delays. If a working circuit is implemented in a PLD but relies on this internal timing, glitches are likely to occur when the design is later changed or implemented in a new device or technology. A *glitch* is a spurious output signal often seen as a spike when viewed with an oscilloscope.

Figure 4.23 illustrates a circuit that contains such a hazard. The output of this circuit will glitch under the situation shown in the timing diagram included in the figure. This is due to the unequal propagation delay from the input signal *B* to the two AND gates.

This type of hazard is called a static hazard. A static hazard is indicated when the initial and resulting values of the circuit are the same. In other words, the glitch occurs on an output signal that wouldn't otherwise change state. When a glitch occurs on an output signal that is changing states (from high to low or low to high), it's referred to as a dynamic hazard.

This circuit is highly simplified, but if you substitute for the inverter delay path a delay that might be seen in a multilevel logic implementation or from a signal routed through a complex path in an FPGA, you can see how such hazards can be created in designs of all sizes.

Hazards come in two broad types: function hazards and logic hazards. A *function hazard* exists whenever the stability of a circuit output depends on the simultaneous change in two or more circuit inputs. This can be illustrated in a truth table such as the one shown in Figure 4.24. The function described by the truth table is a simple XOR.

In the sequence of circuit stimuli shown in the timing diagram of Figure 4.25, there is a period of time during which the two circuit inputs are both changing state. During this time the output of the function can glitch.

```
module hazard

A,B pin;
Y   pin istype 'com';

truth_table ([A,B] -> [Y])
             [0,0] -> [0];
             [0,1] -> [1];
             [1,0] -> [1];
             [1,1] -> [0];
end
```

Figure 4.24 Truth table with function hazard

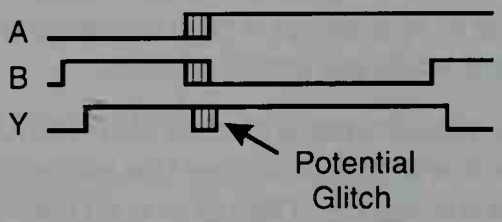


Figure 4.25 Condition under which a glitch can occur

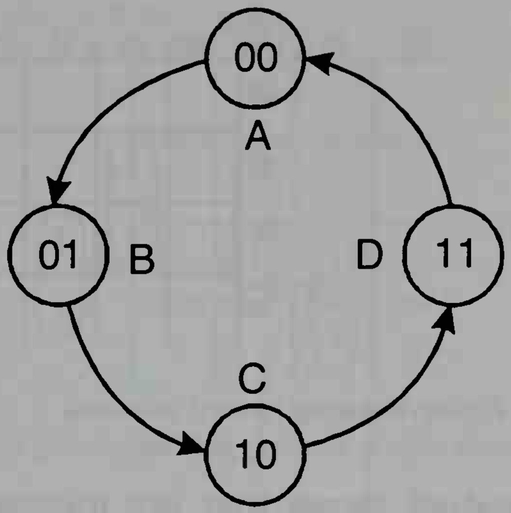


Figure 4.26 State machine with hazard

As you might imagine from this simple example, function hazards are found everywhere in asynchronous systems. The challenge to the design engineer is to identify the hazards that will adversely affect the reliability of the circuit as a whole and to make whatever circuit modifications are necessary to protect the system. How do you protect your system from such hazards? In most cases you can't remove the hazards, so you must instead either avoid them or mask them.

An example of hazard avoidance is found in state value assignment. Consider an asynchronous state machine with the four states shown in Figure 4.26. The current state of the machine is indicated by the two state bits whose values for each state are indicated in the diagram. The transition from state *B* to state *C*, for example, creates the same glitch potential that we saw in the timing diagram of Figure 4.25. If the change in state of the two state bits doesn't occur at precisely the same time, the machine will briefly appear to be in state *A* or *D*. If these states result in other illegal transitions or glitches, the machine will become hopelessly lost. To avoid this hazard, we can simply swap the state values for states *C* and *D* so that, for any state machine transition, only one bit change is required.

There are other classes of hazards that don't require that two or more inputs change at the same time. These hazards are called logic hazards because they are related to the physical characteristics of the actual logic circuitry. Logic hazards can be identified in simple circuits by examining a K-map representation of the logic.

Figure 4.27 illustrates a design, in K-map form, that may contain a logic hazard. The logic hazard is related to the *C* input, which is the single input variable that is specialized in the two product terms of the function (refer to Chapter 3 for a complete description of these terms).

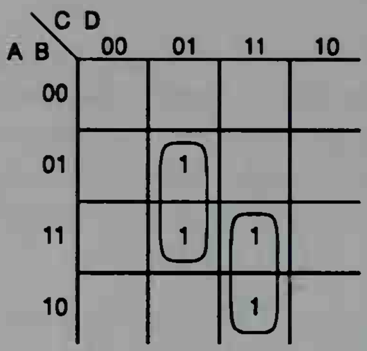


Figure 4.27 K-map showing hazard potential

The two product terms that are indicated by the K-map groupings shown are

```
B & !C & D # A & C & D
```

Since the output of both of these product terms relies on *C*, the period during which *C* changes state can create a glitch under the circumstances illustrated in Figure 4.28.

To mask this glitch, it's necessary to provide additional circuitry that will hold the output high during the change in value of *Y*. The appropriate circuitry can be quickly identified from the K-map, as shown by the new groupings in Figure 4.29.

The additional product term (*A* & *B* & *D*) identified in the K-map serves to protect the output from the glitch. The modified ABEL equations are

```
Y = B & !C & D # A & C & D # A & B & D;
```

When redundant product terms are introduced into the design, it's important to turn off any logic reduction performed by the ABEL compiler. In ABEL this is done by specifying the 'retain' attribute in signal declarations (see Appendix A.)

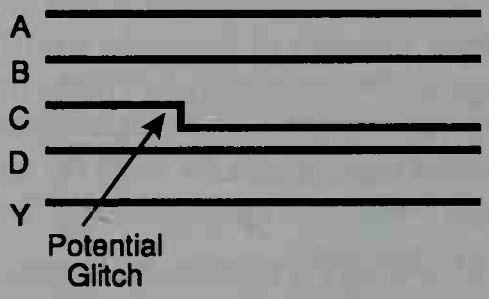


Figure 4.28 Timing diagram showing logic hazard

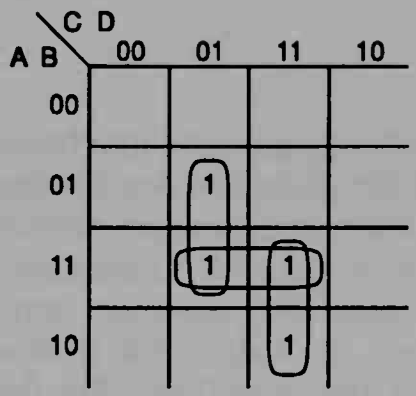


Figure 4.29 Identifying glitch masking product with a K-map

Asynchronous State Encoding and Transition Logic

To design a reliable asynchronous state machine, hazards must be avoided in all state transition logic, and each state in the machine must be self-stabilizing. Assigning state values for asynchronous state machines requires you to analyze each transition and ensure that each change from a source to a destination state results in only a single bit change and that the inputs to the machine are either hazard free or hazard protected through the addition of redundant product terms. For states that include branches to two or more destination states or in situations where a single bit change is impossible, it may be necessary to add one or more hazard states to cover potentially unintended transitions and to recover to one of the valid destination states.

Outputs of asynchronous state machines often require that delay circuitry be added to the state machine in the form of additional states. Since the time that a self-timed asynchronous state machine may spend in a particular state can be extremely short (possibly as short as the t_{PD} of the target device) it may be necessary to hold the outputs at the desired signal level through one or more delay states to meet minimum pulse width requirements. These delay states must be treated like any other state in the machine, with single bit change transitions or hazard states provided.

If automated tools are used for logic minimization, it may be necessary to disable the optimization of state machine transition and output equations, or to add hazard protection logic (redundant product terms) after logic minimization.

Input Timing

Just as in synchronous state machines, asynchronous machines must be protected from potential destabilizing input conditions. Although it is sometimes believed that asynchronous state machines are inherently better equipped to deal with asynchronous inputs, this is not actually true in all cases. Synchronous state machines constructed using clocked flip-flops are vulnerable to destabilizing inputs only during the setup and hold times of these flip-flops. At all other times, spurious inputs are ignored. Asynchronous state machines, on the other hand, are vulnerable to destabilizing inputs at all times; however, the input situations that cause problems for the two types of state machines are different. For the synchronous state machine, a single change in the value of any one input during the setup and hold time is a potential problem that can cause a metastable situation. For asynchronous state machines, the problem is inputs that change values faster than the t_{PD} of the state machine itself or situations in which two or more inputs change value within the t_{PD} time period.

4.5 REFERENCES

Chirlian, Paul M., *Digital Circuits with Microprocessor Applications*, Matrix Publishers, Beaverton, OR, 1982.

Hollaar, Lee A., *Direct Implementation of Asynchronous Control Units*, IEEE Transactions On Computers (December, 1982) pp. 1133-41.

Pellerin, David B. and Holley, Michael, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Treseler, Michael, *Designing State Machine Controllers Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1992.

Unger, Steven J., *The Essence of Logic Circuits*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Wakerly, John F., *Digital Design Principles and Practices, Prentice Hall, Englewood Cliffs, NJ, 1990.*

5

Using ABEL

ABEL (Advanced Boolean Expression Language) is a language optimized for, but not limited to, the description of circuits to be implemented in PLDs and FPGAs. Version 1.0 of the language was introduced by Data I/O Corporation in 1984. This first version was intended solely for the description of PLD-based logic circuits, and the simplicity of the original language reflected the relative simplicity of the programmable devices available at that time. As time went on and devices became more complex, ABEL was updated accordingly. In version 4.0 the language was significantly overhauled, allowing it to be used for general-purpose logic design independent of a specific target device or technology. Version 5.0 added new language features that make ABEL more appropriate for describing very large designs.

Appendix A of this book contains more complete ABEL language information and should be referred to when new designs are being developed. In this chapter we introduce the ABEL language and show how equations, truth tables, and state diagrams (the three basic logic description methods provided in ABEL) are written and processed.

5.1 LANGUAGE OVERVIEW

Hardware design using ABEL is similar in many respects to software design using languages such as Pascal or C. ABEL designs are entered with a text editor and are then compiled into an internal form that may be merged with other design elements, optimized, and executed (via simulation) before implementation in hardware.

ABEL differs from software programming languages in that it is used to describe functions that are inherently parallel. All statements in an ABEL design may be thought of as being executed at the same time. This is particularly important to realize when describing sequential circuits. The sequential operation of a circuit is never a function of the order in which ABEL language statements describing that circuit are entered.

ABEL, like other HDLs, provides different textual entry formats that may be combined as needed to meet the specific requirements of the design. The description methods available in ABEL are: high-level equations, truth tables, and state diagrams. Within a single ABEL module, one or more of these three description methods are used to completely specify the desired circuit.

5.2 USING ABEL EQUATIONS

Before launching into the more interesting circuits described in the later chapters of this book, it's important for us to understand how high-level equations are used in ABEL design descriptions. In ABEL, the complex expressions available for high-level equations are also available for use with other design description methods (such as state descriptions). For this reason it's important to understand how high-level equations relate to lower-level Boolean equations.

First, consider the design of a simple 12-input, 4-output multiplexer. A digital multiplexer such as this is used to select one or more inputs from a larger set of inputs and to route these signals to a corresponding number of outputs. The selection of inputs is made by providing information to the multiplexer's data selection inputs. At the least, a multiplexer requires *n* data selection inputs in order to select between 2^n possible signal routings.

Our multiplexer selects one of three sets of four inputs (*a0-a3*, *b0-b3*, *c0-c3*) and routes the signals to the outputs (*y0-y3*) as indicated by the values appearing on the select input lines (*s0* and *s1*). The possible values for *s1* and *s0* are shown next, along with the resulting signal routings:

Select Input	Data Outputs
s1 s0	y3 - y0
0 1	a3 - a0
1 0	b3 - b0
1 1	c3 - c0
0 0	(all low)

To simplify the design of the multiplexer, we have grouped the various inputs and outputs into sets. Figure 5.1 shows the declarations and equation portions of an ABEL source file that completely describe the function of our multiplexer.

In the design file, all the input and output signals are declared using PIN declarations, and the declared signals are grouped into sets through the use of constant declarations. This grouping of signals into sets simplifies the subsequent description of the circuit.

The equations section follows the declarations section, and contains the actual description of the multiplexer circuit in the form of a high-level equation. In the equation, the relational operator == is used to provide a comparator function for the *Select* input set. Each line of the equation corresponds to one of the possible multiplexer selections.

```
module mux

    a3..a0     PIN;
    b3..b0     PIN;
    c3..c0     PIN;
    s1,s0      PIN;
    y3..y0     PIN ISTYPE 'com';

    A       = [a3..a0];
    B       = [b3..b0];
    C       = [c3..c0];
    Y       = [y3..y0];
    Select  = [s1,s0];

equations

    Y = (Select == 1) & A
      # (Select == 2) & B
      # (Select == 3) & C;
end
```

Figure 5.1 ABEL equations and declarations

When processed by the ABEL language compiler, the multiplexer equation is converted into four separate sum-of-products Boolean equations, one for each output (*y0* through *y3*). These equations can then be implemented into hardware directly (as a sum-of-products logic array) or can be further processed by logic minimization routines to reduce the number of product terms used.

For a better understanding of the design techniques to be described later, it's important to understand how a high-level equation such as this is converted into sum-of-products Boolean equations. We'll show how this is done by going through the conversion process manually. (For efficiency reasons, the actual sequence of operations performed by the language compiler is different and requires fewer steps, but the resulting circuit is the same.)

First, the declarations for *A*, *B*, *C*, *Select*, and *Y* are substituted into the equation to form

```
[y3,y2,y1,y0] = ([s1,s0] == 1) & [a3,a2,a1,a0]
              # ([s1,s0] == 2) & [b3,b2,b1,b0]
              # ([s1,s0] == 3) & [c3,c2,c1,c0];
```

Next, the numeric constants are expanded to match the set widths of the expressions in which they are found. For this equation, the expansion of numeric constants results in the equation

```
[y3,y2,y1,y0] = ([s1,s0] == [0,1]) & [a3,a2,a1,a0]
              # ([s1,s0] == [1,0]) & [b3,b2,b1,b0]
              # ([s1,s0] == [1,1]) & [c3,c2,c1,c0];
```

Now that the relational expression has been normalized in terms of set widths, the relational operators can be converted to Boolean operators. The rules for conversion of the == operator result in the equation

```
[y3,y2,y1,y0] = (!s1 & s0) & [a3,a2,a1,a0]
              # (s1 & !s0) & [b3,b2,b1,b0]
              # (s1 & s0) & [c3,c2,c1,c0];
```

The next phase of the conversion is the distribution of AND operators into the sets. This conversion results in

```
[y3,y2,y1,y0] = [a3 & !s1 & s0,
                 a2 & !s1 & s0,
                 a1 & !s1 & s0,
                 a0 & !s1 & s0]
              # [b3 & s1 & !s0,
                 b2 & s1 & !s0,
                 b1 & s1 & !s0,
                 b0 & s1 & !s0]
              # [c3 & s1 & s0,
                 c2 & s1 & s0,
                 c1 & s1 & s0,
                 c0 & s1 & s0];
```

The equation is now in a form that can be separated easily into individual equations for each of the four outputs:

```
y3 = a3 & !s1 & s0,
   # b3 & s1 & !s0,
   # c3 & s1 & s0;

y2 = a2 & !s1 & s0,
   # b2 & s1 & !s0,
   # c2 & s1 & s0;

y1 = a1 & !s1 & s0,
   # b1 & s1 & !s0,
   # c1 & s1 & s0;

y0 = a0 & !s1 & s0,
   # b0 & s1 & !s0,
   # c0 & s1 & s0;
```

These equations, which are now in a sum of products form, can be mapped directly into the sum-of-products logic array of a simple PLD or implemented as a part of a larger FPGA. If necessary, the Boolean equations can be processed by logic minimization modules. For this design, further minimization is unnecessary.

5.3 USING TRUTH TABLES

Truth tables are useful for describing circuits such as decoders in which the relationship of inputs to outputs doesn't follow a regular pattern. They can also be used effectively for describing state machines.

A truth table is composed of a header that specifies the ordering of the input and output entries in the table and truth table entries that specify

```
truth_table([D3,D2,D1,D0] -> [ a, b, c, d, e, f, g])
            [ 0, 0, 0, 0] -> [ 1, 1, 1, 1, 1, 1, 0];
            [ 0, 0, 0, 1] -> [ 0, 1, 1, 0, 0, 0, 0];
            [ 0, 0, 1, 0] -> [ 1, 1, 0, 1, 1, 0, 1];
            [ 0, 0, 1, 1] -> [ 1, 1, 1, 1, 0, 0, 1];
            [ 0, 1, 0, 0] -> [ 0, 1, 1, 0, 0, 1, 1];
            [ 0, 1, 0, 1] -> [ 1, 0, 1, 1, 0, 1, 1];
            [ 0, 1, 1, 0] -> [ 1, 0, 1, 1, 1, 1, 1];
            [ 0, 1, 1, 1] -> [ 1, 1, 1, 0, 0, 0, 0];
            [ 1, 0, 0, 0] -> [ 1, 1, 1, 1, 1, 1, 1];
            [ 1, 0, 0, 1] -> [ 1, 1, 1, 1, 0, 1, 1];
```

Figure 5.2 ABEL truth table syntax

input to output relationships. A sample ABEL truth table is shown in Figure 5.2. This truth table describes a logic function that converts binary coded decimal numbers into the appropriate set of outputs to drive a seven-segment decoder.

As with Boolean equations, truth tables can be written more concisely if set notation is used. Figure 5.3 shows a complete ABEL source file, with set notation simplifications, for the seven-segment display driver.

This design is an example of a truth table that is incompletely specified; not all possible combinations of inputs are listed on the left side of the truth table. What sort of circuit is created when an incompletely specified truth table is processed? This depends on the level of logic minimization used.

The unspecified entries in a truth table can be evaluated as don't-cares and used to help minimize the circuitry required. In ABEL, the optimization of don't-cares in truth tables is performed if the 'dc' attribute has been specified for each truth table output. If the 'dc' attribute has not been specified, the missing truth table entries will be placed in the off-set for the function along with other truth table entries with outputs of 0. If the 'neg' signal attribute has been specified, the missing entries will be placed in the on-set of the function so that the design is optimized for a negative polarity implementation. Appendix A describes the 'dc' and 'neg' attributes in more detail.

Using Truth Tables for State Machine Design

Truth tables are also well suited for describing state machines, particularly those that contain a large number of similar transitions. Truth tables written for the purpose of state machine specification are often referred to as state tables. To demonstrate how a state machine is expressed using truth tables, consider the state graph illustrated in

```
module BCD7
title 'BCD to 7-segment display driver'
/////////////////////////////////////////////////////////
// Seven-segment display driver with active-low      //
// outputs. Segments:     -a-                        //
//                      f|   |b                      //
//                        -g-                        //
//                      e|   |c                      //
//                        -d-                        //
/////////////////////////////////////////////////////////

    D3..D0          pin;                    "BCD input
    a,b,c,d,e,f,g   pin istype 'dc,com';    "Segment outputs
    OE              pin;                    "Output enable

    BCD     = [D3..D0];
    LED     = [a,b,c,d,e,f,g];
    ON,OFF  = 0,1;      "Inverted sense for common anode LEDs

equations

    LED.oe = !OE;           "Define output enable

truth_table(BCD->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
                0 ->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
                1 ->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
                2 ->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
                3 ->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
                4 ->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
                5 ->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
                6 ->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
                7 ->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
                8 ->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
                9 ->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];

test_vectors
      ([OE,BCD]->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
       [ 0, 0 ]->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
       [ 0, 1 ]->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
       [ 0, 2 ]->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
       [ 0, 3 ]->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
       [ 0, 4 ]->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
       [ 0, 5 ]->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
       [ 0, 6 ]->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
       [ 0, 7 ]->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
       [ 0, 8 ]->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
       [ 0, 9 ]->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];
       [ 1, 5 ]->[ .Z., .Z., .Z., .Z., .Z., .Z., .Z.];

end
```

Figure 5.3 Seven-segment display decoder design file

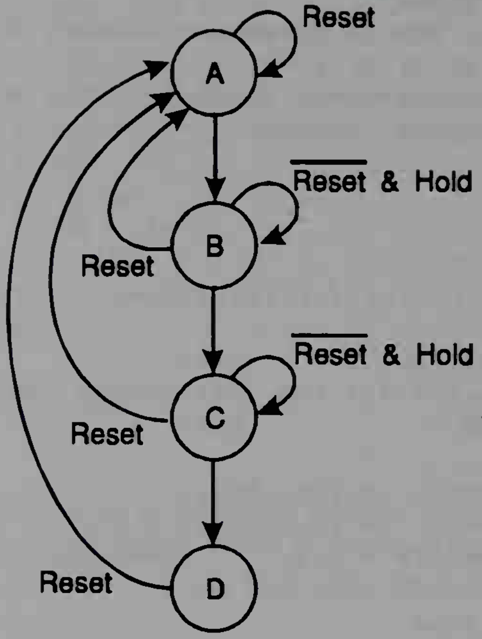

Reset
A
$\overline{\text{Reset}}$ & Hold
B
Reset
$\overline{\text{Reset}}$ & Hold
C
Reset
Reset
D

Figure 5.4 Four-state Moore-model state machine

Figure 5.4. The state machine illustrated by the state graph has four states (represented by the bubbles) and nine possible state transitions (represented by the arrows). This Moore-model state machine has no outputs other than the state register itself, which could consist of a pair of D flip-flops.

To describe this simple state machine in a truth table, all possible transition conditions are listed along with their resulting next states. The ABEL design file is shown in Figure 5.5.

In the design file, the four states of the machine are given the values of zero through three, and the symbolic names of *A*, *B*, *C*, and *D*, respectively. The state values decode to the binary values 00, 01, 10, and 11, respectively.

Next, a truth table is used to specify the state machine's operation. The first line of the truth table specifies that the machine is to return to state *A* unconditionally when the *Reset* input is asserted. The second line specifies that when *Reset* is false the machine should advance to state *B* (without regard to the value of the *Hold* input). The third through sixth entries describe the behavior of the state machine in states *B* and *C*. In these states, an asserted *Hold* input results in the machine holding in its current state, while a false *Hold* results in the machine advancing to the next state. The final line of the truth table specifies that the machine should return unconditionally to state *A* after one clock cycle in state *D*.

```
module TT2
title 'Truth table state machine example'
////////////////////////////////////////////////////////////
// 4-state state machine described using a truth table.   //
// Each line of the truth table represents one state      //
// transition. Multiple transitions (such as the reset)   //
// can be combined in a single table entry with the .X.   //
// no-connect (don't-care input) special constant.        //
////////////////////////////////////////////////////////////
    Clk,OE          pin;                "Clock, enable inputs
    Hold,Reset      pin;                "Control inputs
    Q1,Q0           pin istype 'reg';   "State bits

    A = 0; B = 1; C = 2; D = 3;         "State values

Equations
    [Q1,Q0].clk = Clk;
    [Q1,Q0].oe  = !OE;

Truth_table ([Reset,Hold,[Q1,Q0]] :> [Q1,Q0])
             [  1  , .X.,  .X.  ] :>   A ;
             [  0  , .X.,   A   ] :>   B ;
             [  0  ,  1 ,   B   ] :>   B ;
             [  0  ,  0 ,   B   ] :>   C ;
             [  0  ,  1 ,   C   ] :>   C ;
             [  0  ,  0 ,   C   ] :>   D ;
             [ .X. , .X.,   D   ] :>   A ;

Test_Vectors ([Clk,OE,Reset,Hold] -> [Q1,Q0])
              [.C., 0,  1  ,  0 ] ->   A;
              [.C., 0,  1  ,  1 ] ->   A;
              [.C., 0,  0  ,  1 ] ->   B;
              [.C., 0,  0  ,  1 ] ->   B;
              [.C., 0,  0  ,  0 ] ->   C;
              [.C., 0,  0  ,  1 ] ->   C;
              [.C., 0,  0  ,  0 ] ->   D;
              [.C., 0,  0  ,  0 ] ->   A;
              [.C., 0,  0  ,  0 ] ->   B;
              [.C., 0,  0  ,  0 ] ->   C;
              [.C., 0,  1  ,  0 ] ->   A;
End
```

Figure 5.5 Four-state state machine design file

5.4 USING STATE DIAGRAMS

A state diagram is a relatively simple method of describing the operation of complex state machines. Any state machine that can be described in a state diagram can also be described using a truth table (or equations, for that matter.) The key difference between truth tables and state diagrams is that when you describe a state machine using a truth table

```
module SM2
title 'Truth table state machine example'
//////////////////////////////////////////////////////////////
// 4-state state machine described using a state diagram. //
//////////////////////////////////////////////////////////////
    Clk,OE          pin;                "Clock, enable inputs
    Hold,Reset      pin;                "Control inputs
    Q1,Q0           pin istype 'reg';   "State bits

    A = 0; B = 1; C = 2; D = 3;         "State values

Equations
    [Q1,Q0].clk = Clk;
    [Q1,Q0].oe  = !OE;

State_Diagram [Q1,Q0]

    State A: If !Reset Then B Else A ;

    State B: Case    Reset            : A;
                    !Reset &  Hold    : B;
                    !Reset & !Hold    : C;
             Endcase;

    State C: If Reset Then A
             Else If Hold Then C
             Else D;

    State D: Goto A;

Test_Vectors ([Clk,OE,Reset,Hold] -> [Q1,Q0])
               [.C., 0,  1  ,  0 ] ->    A;
               [.C., 0,  1  ,  1 ] ->    A;
               [.C., 0,  0  ,  1 ] ->    B;
               [.C., 0,  0  ,  1 ] ->    B;
               [.C., 0,  0  ,  0 ] ->    C;
               [.C., 0,  0  ,  1 ] ->    C;
               [.C., 0,  0  ,  0 ] ->    D;
               [.C., 0,  0  ,  0 ] ->    A;
               [.C., 0,  0  ,  0 ] ->    B;
               [.C., 0,  0  ,  0 ] ->    C;
               [.C., 0,  1  ,  0 ] ->    A;
End
```

Figure 5.6 ABEL state diagram syntax

you describe the machine primarily in terms of its transitions. When you use a state diagram, you describe the machine more in terms of its possible states.

Figure 5.6 shows an ABEL state diagram. This state diagram describes the same state machine that we previously described using a truth table. Notice that the state diagram contains one state description for each of the four states in the machine. The state diagram, like the truth table, has a header that specifies which signals are to be used for the state register. Unlike a truth table, other state machine inputs and outputs are not included in the state diagram header. Another difference is that the state register declared in the state machine header is flip-flop type independent; the state machine is written the same regardless of the type of flip-flops used for the state register.

Following the header is a series of state descriptions. Each state description includes a value declaration, transition information, and optional equations for circuit I/O (not used in this example).

To describe the state machine transitions, each state description contains one or more transition statements. If you compare the state description for each state to its equivalent bubble in the state graph, you can see how the state transitions are expressed. State *A* has a simple two-way branch which is described using an IF-THEN-ELSE statement. States *B* and *C* have identical transition logic, but were described using different methods to show the use of ABEL's CASE transition statement. State *D* has a single unconditional transition, represented by the GOTO statement.

Figure 5.7 shows a state graph of a simple state machine with the same four states presented previously, but with different transition logic and other state machine outputs.

The operation of this state machine begins in state *A* and remains there as long as the *Hold* input is false. When *Hold* is asserted, the machine

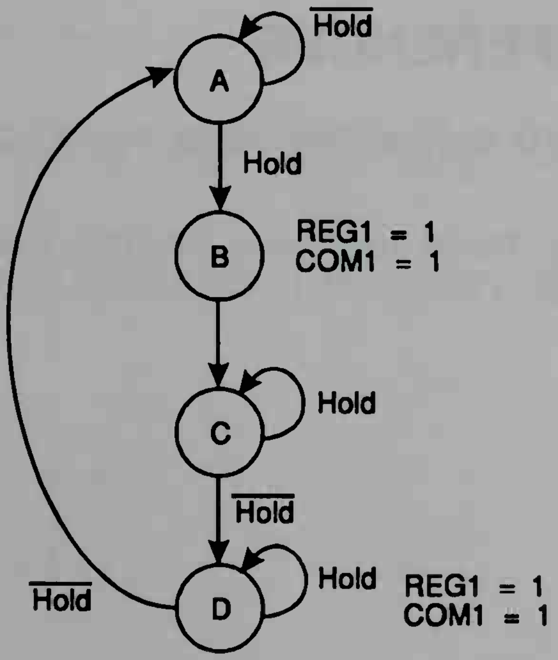


Figure 5.7 Four-state state machine

advances to state *B*, and from there to state *C*, remaining in *C* until *Hold* is again false. In states *C* and *D*, an asserted *Hold* signal results in the machine staying in its present state, while a false *Hold* results in the machine advancing to the next state (*D* and *A*, respectively). The state graph also shows two outputs and their corresponding values for each state in the machine. One output (*COM1*) is a combinational signal, while the other (*REG1*) is registered. If we map this diagram directly into an ABEL state machine description, we might write a design file such as the one shown in Figure 5.8.

As written, this state machine has a peculiar timing characteristic that needs attention: the registered output *REG1* lags the state machine by a full clock cycle. This behavior is reflected in the test vectors written in the ABEL source file. In the ninth test vector, *REG1* is still false even though the machine has advanced to state *D*. Similarly, *REG1* remains true for a full clock cycle in state *A* when the machine transitions from state *D*. This is because the input-forming logic for *REG1*'s D-type flip-flop relies on the current state as reflected in the *Q1* and *Q0* state registers. Since *REG1*, *Q1*, and *Q0* are all clocked from a common source, the current state information is not available to *REG1* at the appropriate time.

This situation is corrected by decoding the logic for *REG1* one state early. To do this, we write the equation for *REG1* in terms of state transitions, rather than in terms of current states. This is done by writing the equation in a WITH block associated with a particular transition condition. The modified state machine description is shown in Figure 5.9.

5.5 REFERENCES

Data I/O Corporation, *ABEL User's Guide*, Data I/O, Redmond, WA, 1993.

Pellerin, David and Holley, Michael, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

```
module SM3
title 'State machine output example'
//////////////////////////////////////////////////////////////////
// 4-state state machine with state outputs.                   //
//////////////////////////////////////////////////////////////////
    Clk,Hold,OE     pin;                  "Inputs
    Q1,Q0           pin istype 'reg';     "State bits
    REG1            pin istype 'reg';     "Reg output
    COM1            pin istype 'com';     "Com output

    A = 0; B = 1; C = 2; D = 3;           "State values

Equations
    [REG1,Q0,Q1].clk = Clk;
    [REG1,Q0,Q1].oe  = !OE;

State_Diagram [Q1,Q0]

    State A:
            REG1 := 0;
            COM1  = 0;
            If !Hold Then A Else B ;

    State B:
            REG1 := 1;
            COM1  = 1;
            Goto C;

    State C:
            REG1 := 0;
            COM1  = 0;
            If Hold Then C Else D;

    State D:
            REG1 := 1;
            COM1  = 1;
            If Hold Then D Else A;

Test_Vectors
        ([Clk,OE,Hold] -> [[Q1,Q0],REG1,COM1])
         [.C., 0,  0 ] -> [  .X.   , .X., .X.];
         [.C., 0,  0 ] -> [  .X.   , .X., .X.];
         [.C., 0,  0 ] -> [  .X.   , .X., .X.];
         [.C., 0,  0 ] -> [   A    ,  0 ,  0 ];
         [.C., 0,  0 ] -> [   A    ,  0 ,  0 ];
         [.C., 0,  1 ] -> [   B    ,  0 ,  1 ];
         [.C., 0,  1 ] -> [   C    ,  1 ,  0 ];
         [.C., 0,  1 ] -> [   C    ,  0 ,  0 ];
         [.C., 0,  0 ] -> [   D    ,  0 ,  1 ];
         [.C., 0,  1 ] -> [   D    ,  1 ,  1 ];
         [.C., 0,  1 ] -> [   D    ,  1 ,  1 ];
         [.C., 0,  0 ] -> [   A    ,  1 ,  0 ];
         [.C., 0,  0 ] -> [   A    ,  0 ,  0 ];
End
```

Figure 5.8 Four-state state machine with state outputs

```
module SM4
title 'State machine output example'
////////////////////////////////////////////////////////////
// 4-state state machine with state transition outputs.   //
////////////////////////////////////////////////////////////
    Clk,Hold,OE   pin;                  "Inputs
    Q1,Q0         pin istype 'reg';     "State bits
    REG1          pin istype 'reg';     "Reg output
    COM1          pin istype 'com';     "Com output
    A = 0; B = 1; C = 2; D = 3;         "State values

Equations
    [REG1,Q0,Q1].clk = Clk;
    [REG1,Q0,Q1].oe  = !OE;

State_Diagram [Q1,Q0]
    State A:
            REG1 := 0;
            COM1  = 0;
            If !Hold Then A Else B ;

    State B:
            REG1 := 1;
            COM1  = 1;
            Goto C;

    State C:
            COM1  = 0;
            If Hold Then C  With REG1 := 0;
                    Else D  With REG1 := 1;

    State D:
            REG1 := 1;
            COM1  = 1;
            If Hold Then D Else A;

Test_Vectors
       ([Clk,OE,Hold] -> [[Q1,Q0],REG1,COM1])
        [.C., 0,  0 ] -> [  .X.  , .X., .X.];
        [.C., 0,  0 ] -> [  .X.  , .X., .X.];
        [.C., 0,  0 ] -> [  .X.  , .X., .X.];
        [.C., 0,  0 ] -> [   A   ,  0 ,  0 ];
        [.C., 0,  0 ] -> [   A   ,  0 ,  0 ];
        [.C., 0,  1 ] -> [   B   ,  0 ,  1 ];
        [.C., 0,  1 ] -> [   C   ,  1 ,  0 ];
        [.C., 0,  1 ] -> [   C   ,  0 ,  0 ];
        [.C., 0,  0 ] -> [   D   ,  1 ,  1 ];
        [.C., 0,  1 ] -> [   D   ,  1 ,  1 ];
        [.C., 0,  1 ] -> [   D   ,  1 ,  1 ];
        [.C., 0,  0 ] -> [   A   ,  1 ,  0 ];
        [.C., 0,  0 ] -> [   A   ,  0 ,  0 ];
End
```

Figure 5.9 Using WITH equations for state diagram outputs

6

Basic TTL Logic Functions

In this and subsequent chapters, we'll examine how ABEL can be used to describe a variety of small- to medium-sized circuits. These basic circuits are commonly used in much larger designs, and help to illustrate a number of important implementation considerations.

Because of the widespread use of TTL functions, we will begin by taking a tour through the TTL data book and describing how to create similar building blocks using ABEL.

6.1 BUFFERS

In a logic design sense, buffers are simply wires that route a set of inputs to a set of outputs. In TTL-based designs, buffers are normally used to provide three-state output capabilities for bus interfaces or other situations in which there is more than one function that must drive the same signal (but at different times.) The 74241 device shown in Figure 6.1 is a standard TTL device that provides this function. To provide protection for TTL outputs that must drive the same bus, the 74241 includes output enable controls for the device outputs. Figure 6.2 is an example of how these same output enables are described using ABEL. The design shown uses the .OE dot extensions to describe the output

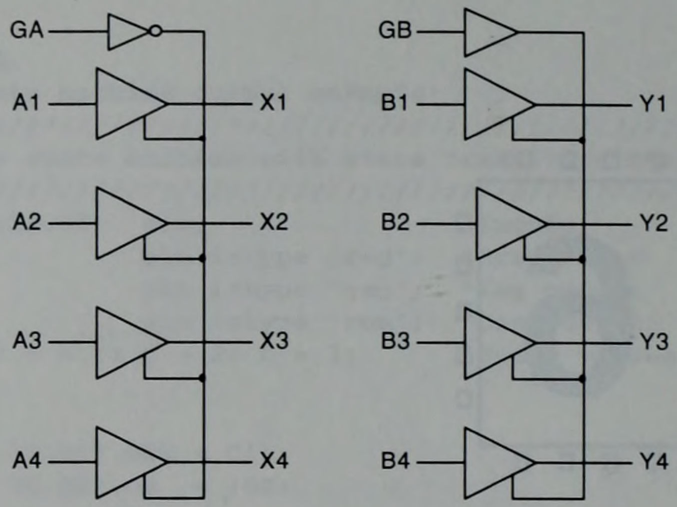


Figure 6.1 74241 Standard logic device

```
module sn74241
title 'Octal buffer and line driver with 3-state output'

    A1,A2,A3,A4      pin;
    B1,B2,B3,B4      pin;
    GA,GB            pin;
    Y1,Y2,Y3,Y4      pin istype 'com';
    X1,X2,X3,X4      pin istype 'com';

    A = [A4..A1];
    B = [B4..B1];
    X = [X4..X1];
    Y = [Y4..Y1];

equations
    X    = A;
    X.oe = !GA;      "Active Low Enable

    Y    = B;
    Y.oe = GB;       "Active High Enable

test_vectors
        ([GA, A , GB, B ] -> [ X , Y ])
         [ 1, 0 ,  0, 0 ] -> [.Z.,.Z.];
         [ 0, 0 ,  0, 0 ] -> [ 0 ,.Z.];
         [ 0, 5 ,  0, 0 ] -> [ 5 ,.Z.];
         [ 0,10 ,  1, 0 ] -> [10 , 0 ];
         [ 0,15 ,  1, 5 ] -> [15 , 5 ];
         [ 1,15 ,  1,10 ] -> [.Z.,10 ];
         [ 1, 0 ,  1,15 ] -> [.Z.,15 ];

end
```

Figure 6.2 74241 Standard logic device described with ABEL enables for outputs *X1* through *X4* and *Y1* through *Y4*. This design performs the function of the 74241 octal buffer.

To use this design in an actual device, the device chosen must include output enable features. Virtually all PLDs have outputs with TTL-compatible output enable controls, so this simple circuit is widely used (in combination with other circuitry) in PLD-based designs.

Bus transceivers are frequently designed using three-state output enables. A bus transceiver is a circuit that provides two-way interface capabilities between two separate circuits, such as between two circuits communicating on a data bus. These interfaces are often asynchronous. To provide this capability, a direction control input is used that specifies which circuit has "write privilege" to the data bus. The standard TTL device that provides this function, the 74245, is shown in schematic form in Figure 6.3. Figure 6.4 is an ABEL design description for a 74245 circuit.

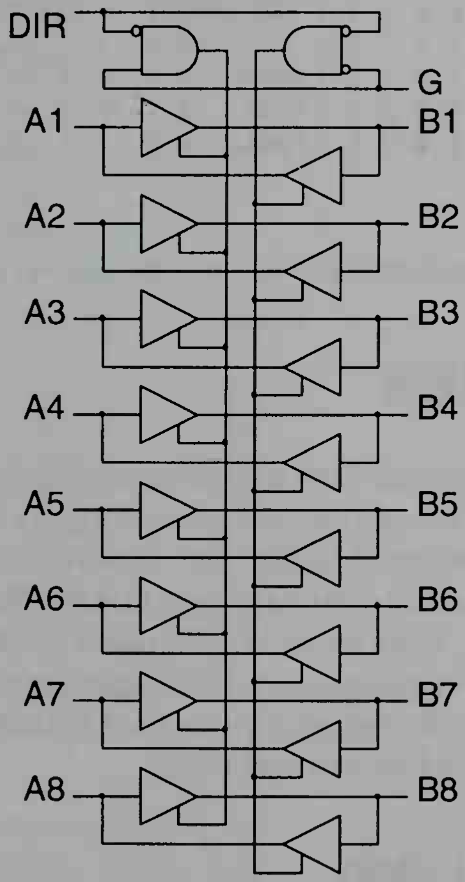


Figure 6.3 74245 Bidirectional bus transceiver

```
module sn74245
title 'Octal Bidirectional Bus Transceiver'

    G,DIR       pin;
    A1..A8      pin istype 'com';
    B1..B8      pin istype 'com';
    A = [A8..A1];
    B = [B8..B1];

equations
    A      = B;
    B      = A;
    A.oe = !DIR & !G;
    B.oe = DIR & !G;

test_vectors
        ([ G ,DIR,  A ,   B ] -> [  A ,  B ])
         [ 1 , 0 ,  0 ,   0 ] -> [ .Z., .Z.];
         [ 0 , 0 , .X.,^h00] -> [^h00, .X.];
         [ 0 , 0 , .X.,^h55] -> [^h55, .X.];
         [ 0 , 0 , .X.,^hAA] -> [^hAA, .X.];
         [ 0 , 0 , .X.,^hFF] -> [^hFF, .X.];
         [ 0 , 0 , .X.,^hF0] -> [^hF0, .X.];
         [ 0 , 1 , .X., .X.] -> [ .Z., .X.];
         [ 0 , 1 ,^h00, .X.] -> [ .X.,^h00];
         [ 0 , 1 ,^h55, .X.] -> [ .X.,^h55];
         [ 0 , 1 ,^hAA, .X.] -> [ .X.,^hAA];
end
```

Figure 6.4 74245 Bidirectional bus transceiver described with ABEL

6.2 LATCHES

A latch is a circuit that provides data storage. Latches are level sensitive and hold their current value as long as their control inputs are held at the latched state. Unlike the flip-flops described in Chapter 5, which were intended for synchronous control functions and rely on clock edges for operation, latches are often used in situations where no system clock is available. In programmable logic applications, latches are often used to provide buffering of inputs to a synchronous state machine (to meet setup and hold requirements.)

74373 Octal Latch

Figure 6.5 is a diagram of the 74373 standard logic device. This octal latch is a common TTL latch that can be described using ABEL. In the ABEL source file shown in Figure 6.6, the .LE and .OE dot extensions describe the latch enable and output enable functions for the eight

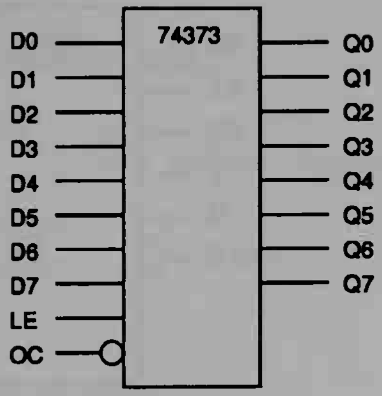


Figure 6.5 74373 Standard logic device

```
Module sn74373
Title 'Octal D-type Transparent Latch'

    OC,LE  pin;
    D7..D0 pin;
    Q7..Q0 pin istype 'reg,buffer';
    Input  = [D7..D0];
    Output = [Q7..Q0];

Equations
    Output := Input;
    Output.le = !LE;
    Output.oe = !OC;

Test_Vectors
        ([OC ,LE ,Input] -> Output)
         [ 1 ,.X., .X. ] ->  .Z.;
         [ 0 , 1 ,^h00 ] -> ^h00;
         [ 0 , 1 ,^h55 ] -> ^h55;
         [ 0 , 1 ,^hAA ] -> ^hAA;
         [ 0 , 1 ,^hFF ] -> ^hFF;
         [ 0 , 1 ,^hA5 ] -> ^hA5;
         [ 0 , 0 ,^hA5 ] -> ^hA5;
         [ 0 , 0 ,^h00 ] -> ^hA5;
         [ 0 , 1 ,^h22 ] -> ^h22;
End
```

Figure 6.6 74373 Octal latch described with ABEL

latches (signals *Q0* through *Q7*). The := assignment operator is used to describe the data inputs to the latches (inputs *D0* through *D0*). When the input *LE* is high (as shown in the test vectors), the data applied to the *D0* throuth *D7* inputs is routed directly to the outputs. When *LE* is low, the current data value is latched into *Q0* through *Q7*, which then remain in that state regardless of what is applied to the inputs. The *OE* input provides a three-state control for all eight latch outputs.

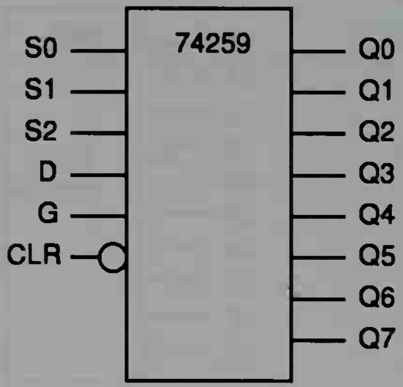


Figure 6.7 74259 Standard logic device

74259 Addressable Latch

An addressable latch such as the 74259 (shown in Figure 6.7) is a set of two or more latches that can be individually controlled through the use of a multiplexer-like data select input. Figure 6.8 is an ABEL source file describing an 8-bit addressable latch that performs the same function as a 74259. The design description uses ABEL's .LE dot extension to specify a latch function (inputs *S0* through *S2*) for signals *Q0* through *Q7*. This design could be implemented as a part of a larger circuit in any programmable logic device that features latched outputs (the Lattice 6001 device is a good example) or in an FPGA such as a Xilinx LCA.

Addressable latches have four basic modes of operation. The mode of operation is determined with the *CLR* and *G* inputs, as shown in the following table:

Clr	G	Mode
H	L	Data from input routed to selected latch
H	H	All outputs latched
L	L	one- to eight-line demultiplexer
L	H	Clear all latches

The three data select inputs (*S0* through *S2*) are used to select which of the eight latches is to be used as the destination for the data appearing on input *D*.

```
Module sn74259
Title '8-bit Addressable Latch'

    S2,S1,S0,G,D,Clr    pin;
    Q7..Q0              pin istype 'reg,buffer';

    Output = [Q7..Q0];
    Select = [S2..S0];

Equations

    when (Select == 0) then
        Output := [Q7,Q6,Q5,Q4,Q3,Q2,Q1,D ];
    when (Select == 1) then
        Output := [Q7,Q6,Q5,Q4,Q3,Q2,D ,Q0];
    when (Select == 2) then
        Output := [Q7,Q6,Q5,Q4,Q3,D ,Q1,Q0];
    when (Select == 3) then
        Output := [Q7,Q6,Q5,Q4,D ,Q2,Q1,Q0];
    when (Select == 4) then
        Output := [Q7,Q6,Q5,D ,Q3,Q2,Q1,Q0];
    when (Select == 5) then
        Output := [Q7,Q6,D ,Q4,Q3,Q2,Q1,Q0];
    when (Select == 6) then
        Output := [Q7,D ,Q5,Q4,Q3,Q2,Q1,Q0];
    when (Select == 7) then
        Output := [D ,Q6,Q5,Q4,Q3,Q2,Q1,Q0];

  [Q7..Q0].le = G;
  [Q7..Q0].ar = !Clr & G;

Test_Vectors
        ([ G ,Clr,D,Select] ->   Output)
         [ 1 , 0 ,0,  0   ] -> ^b00000000;
         [ 1 , 1 ,1,  4   ] -> ^b00000000;
         [ 0 , 1 ,1,  4   ] -> ^b00010000;
         [ 1 , 1 ,1,  4   ] -> ^b00010000;
         [.K., 1 ,1,  7   ] -> ^b10010000;
         [.K., 1 ,1,  6   ] -> ^b11010000;
         [.K., 1 ,0,  7   ] -> ^b01010000;
         [ 1 , 1 ,0,  0   ] -> ^b01010000;
         [ 1 , 0 ,0,  0   ] -> ^b00000000;
         [ 0 , 0 ,1,  0   ] -> ^b00000001;
         [ 0 , 0 ,1,  3   ] -> ^b00001001;
         [ 0 , 0 ,1,  5   ] -> ^b00101001;
         [ 0 , 0 ,1,  2   ] -> ^b00101101;
         [ 0 , 1 ,1,  2   ] -> ^b00101101;
         [ 1 , 1 ,1,  2   ] -> ^b00101101;
         [ 1 , 1 ,0,  0   ] -> ^b00101101;
End
```

Figure 6.8 74259 Addressable latch described with ABEL

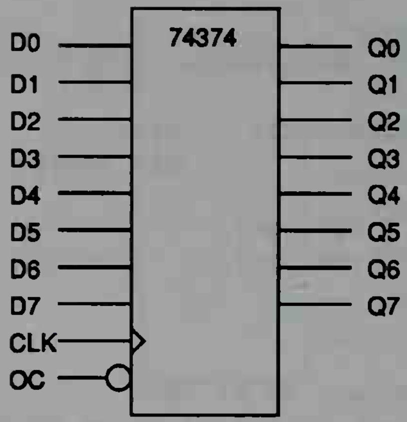


Figure 6.9 74374 Octal register

6.3 REGISTERS

The most commonly used building block for register circuits is the D-type flip-flop described in Chapter 5. These types of flip-flops are found in nearly every programmable logic device available, and the ABEL language has many features that are specialized for them.

74374 Octal Register

The 74374 device shown in figure 6.9 is nearly identical to the 74373 device presented earlier, but features D-type flip-flops instead of latches. Figure 6.10 is an ABEL description of an octal D-type register identical in operation to the 74374 standard logic device. The .CLK and .OE dot extensions describe the clock and output enable functions for the eight flip-flops represented by signals *Q0* through *Q7*. The := assignment operator is used to describe the data inputs to the flip-flops (inputs *D0* through *D7*). When a positive clock edge occurs on input *Clk* (as shown in the test vectors) the data applied to the *D0* through *D7* inputs is loaded into the flip-flops. The *OE* input provides a three-state output for all eight flip-flops.

6.4 SHIFT REGISTERS

Shift registers are data storage and conversion circuits constructed of sequences of flip-flops. In operation, a shift register collects, stores, and transfers data serially between adjacent flip-flops in the register. This shifting of data occurs once each time the shift register is clocked, as illustrated in Figure 6.11. In this example, an 8-bit value is shifted from

```
Module sn74374
Title 'Octal Edge-Triggered Flip-Flop'

    OC,Clk    pin;
    D7..D0    pin;
    Q7..Q0    pin istype 'reg,buffer';
    Input   = [D7..D0];
    Output = [Q7..Q0];

Equations

    Output     := Input;
    Output.clk = Clk;
    Output.oe  = !OC;

Test_Vectors
        ([OC ,Clk,Input] -> Output)
         [ 1 ,.X., .X. ] ->  .Z.;
         [ 0 ,.C.,^h00 ] -> ^h00;
         [ 0 ,.C.,^h55 ] -> ^h55;
         [ 0 ,.C.,^hAA ] -> ^hAA;
         [ 0 ,.C.,^hFF ] -> ^hFF;
         [ 0 ,.C.,^hA5 ] -> ^hA5;
         [ 0 , 0 ,^h00 ] -> ^hA5;
         [ 0 ,.C.,^h22 ] -> ^h22;
End
```

Figure 6.10 ABEL file for 74374 octal register

the left (most significant bit) to the right (least significant bit), with zeros being shifted into the leftmost flip-flop. Other types of shift registers may allow data to be loaded into the register serially (from the left or right side) or in parallel (all flip-flops at once.) Shift registers are useful for timing chains and for parallel-to-serial or serial-to-parallel data conversion.

This example loads a 0 into the leftmost bit of the shift register with each clock cycle. Depending on the application, a shift register may load in 0,1 or a value driven from another portion of the circuit (for serial-to-parallel data conversion, for example.) It is also possible to feed the serial data from the opposite end of the shift register, forming a circular (barrel) shifter. This operation is known as a rotate.

Shift registers can be constructed using D-type flip-flops as shown in Figure 6.12. This circuit chains eight D-type flip-flops, with the first flip-flop in the chain being fed by the incoming serial data. The flip-flops are clocked from a common source, and the parallel data is available at any time from the *Q* outputs of the eight flip-flops.

The 74164 standard logic device (shown in Figure 6.13) is a commonly used 8-bit serial-to-parallel converter that is built around such a shift

	MSB LSB
Start:	1 1 0 1 0 1 1 0
Clock 1:	0 1 1 0 1 0 1 1
Clock 2:	0 0 1 1 0 1 0 1
Clock 3:	0 0 0 1 1 0 1 0
Clock 4:	0 0 0 0 1 1 0 1
Clock 5:	0 0 0 0 0 1 1 0
Clock 6:	0 0 0 0 0 0 1 1
Clock 7:	0 0 0 0 0 0 0 1
Clock 8:	0 0 0 0 0 0 0 0

Figure 6.11 Shift register operation

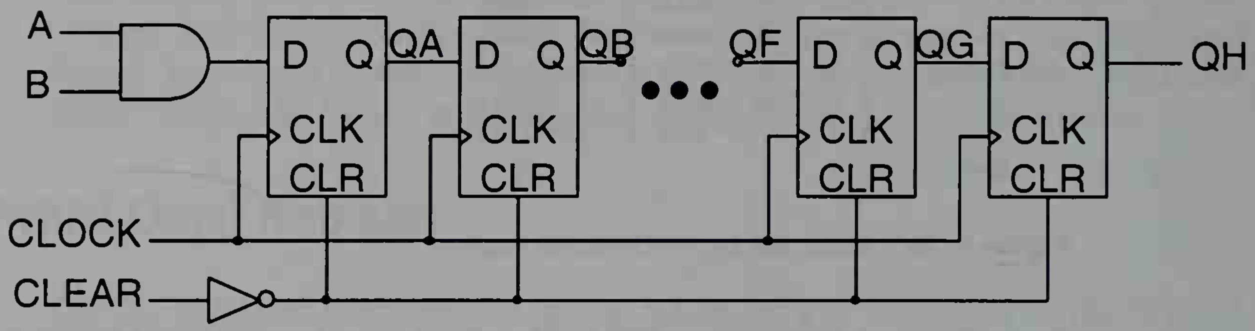


Figure 6.12 Shifter constructed using D flip-flops

register. Figure 6.14 is an example of how this standard shift register circuit can be described using ABEL. The 74164 accepts data (one bit per clock cycle) into the rightmost flip-flop in an 8-bit chain and then shifts that data left by one bit with each subsequent clock. The ABEL source file shown emulates the operation of the standard 74164 parallel-out serial shift register.

Parallel-to-serial shift registers require a means for parallel loading of data. Figure 6.15 shows a modified version of the 8-bit shift register that allows data loading for parallel-to-serial applications. In this design, the parallel load operation is performed asynchronously through the use of built-in reset and preset features assumed to be available in the target device. The *ShLd* input controls whether the data observed in the data inputs (*A* through *H*) should be loaded into the flip-flops. This design emulates a 74165 device in operation.

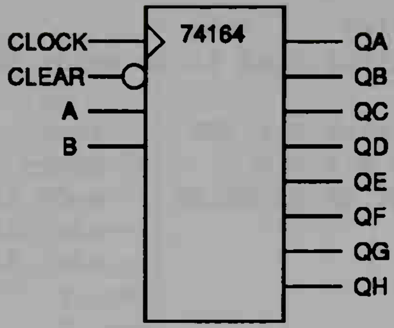


Figure 6.13 74164 Standard logic device

```
Module sn74164
Title '8-bit Parallel-Out Serial Shift Register'

    A,B                      pin;
    Clear,Clock              pin;
    QA,QB,QC,QD,QE,QF,QG,QH pin istype 'reg,buffer';

Equations

    QA := A & B;
    [QH,QG,QF,QE,QD,QC,QB] := [QG,QF,QE,QD,QC,QB,QA];

    [QH,QG,QF,QE,QD,QC,QB,QA].clk = Clock;
    [QH,QG,QF,QE,QD,QC,QB,QA].ar  = !Clear;

Test_Vectors
    ([Clock,Clear, A, B] -> [QA,QB,QC,QD,QE,QF,QG,QH])
     [  0  ,  0  , 1, 0] -> [ 0, 0, 0, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 1, 0] -> [ 0, 0, 0, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 0, 0, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 1, 1] -> [ 1, 0, 0, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 1, 1] -> [ 1, 1, 0, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 1, 1, 0, 0, 0, 0, 0];
     [ .C. ,  1  , 1, 1] -> [ 1, 0, 1, 1, 0, 0, 0, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 1, 0, 1, 1, 0, 0, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 0, 1, 0, 1, 1, 0, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 0, 0, 1, 0, 1, 1, 0];
     [ .C. ,  1  , 0, 1] -> [ 0, 0, 0, 0, 1, 0, 1, 1];
     [ .C. ,  0  , 0, 1] -> [ 0, 0, 0, 0, 0, 0, 0, 0];
End
```

Figure 6.14 ABEL design description for 74164 shift register

An alternative method of loading data into a shift register is used in the 74166 device (figure 6.16). The ABEL description for this device is shown in Figure 6.17. This design, which emulates the 74166 device, uses a multiplexer (in the form of a WHEN-THEN equation) to route the eight bits of input data to the shift register when the *ShLd* input is asserted.

```
Module sn74165
Title 'Parallel-Load 8-bit Shift Register, async load'

    ShLd,ClkInh,Clk,SER       pin;
    A,B,C,D,E,F,G,H           pin;
    QA,QB,QC,QD,QE,QF,QG      node istype 'reg,buffer';
    QH                        pin  istype 'reg,buffer';
    QH_                       pin  istype 'com';

Equations

  QH_ = !QH;

  [QH,QG,QF,QE,QD,QC,QB,QA] := [QG,QF,QE,QD,QC,QB,QA,SER];
  [QH,QG,QF,QE,QD,QC,QB,QA].clk = Clk # ClkInh;

  when (!ShLd) then
  { [QH,QG,QF,QE,QD,QC,QB,QA].ar = ![H,G,F,E,D,C,B,A];
    [QH,QG,QF,QE,QD,QC,QB,QA].ap =  [H,G,F,E,D,C,B,A];
  }

Test_Vectors
 ([Clk,ClkInh,ShLd,SER,A,B,C,D,E,F,G,H]->[QA,QB,QC,QD,QE,QF,QG,QH,QH_])
  [.C.,    1  , 0  , 0 ,1,0,1,0,1,0,1,1]->[ 1, 0, 1, 0, 1, 0, 1, 1, 0 ];
  [.C.,    1  , 1  , 0 ,1,0,1,0,1,0,1,1]->[ 1, 0, 1, 0, 1, 0, 1, 1, 0 ];
  [.C.,    0  , 1  , 0 ,0,0,0,0,0,0,0,0]->[ 0, 1, 0, 1, 0, 1, 0, 1, 0 ];
  [.C.,    0  , 1  , 0 ,0,0,0,0,0,0,0,0]->[ 0, 0, 1, 0, 1, 0, 1, 0, 1 ];
  [.C.,    0  , 1  , 1 ,0,0,0,0,0,0,0,0]->[ 1, 0, 0, 1, 0, 1, 0, 1, 0 ];
  [.C.,    0  , 1  , 1 ,0,1,1,1,1,0,0,0]->[ 1, 1, 0, 0, 1, 0, 1, 0, 1 ];
  [.C.,    0  , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 0, 0, 1, 0, 1, 0 ];
  [ 0 ,    0  , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 0, 0, 1, 0, 1, 0 ];
  [ 0 ,    1  , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 0, 1, 1, 0, 0, 1, 0, 1 ];
  [.C.,    1  , 0  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 1, 1, 1, 0, 0, 1 ];
End
```

Figure 6.15 ABEL source file for 74165 loadable shifter

This design includes an asynchronous register clear function (the *Clr* input) in addition to the parallel load function. To provide exact emulation of the 74166 device, the clock in this design has been ANDed with a clock inhibit input (ClkInh). Since most PLDs available today do not allow complex clocking such as this, the clock inhibit feature would have to be modified or removed before implementing this circuit in a simple PLD. If the design were being implemented in a more complex PLD (such as a Mach 215), the clock inhibit feature could be written as shown.

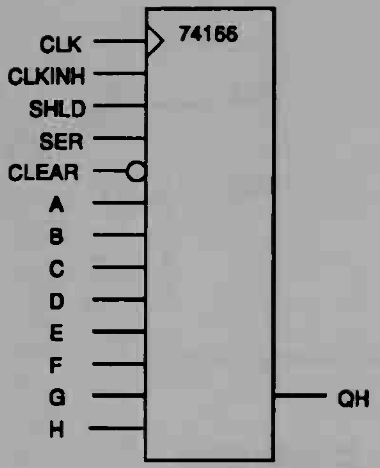


Figure 6.16 74166 Standard logic device

```
Module sn74166
Title 'Parallel-Load 8-bit Shift Register, async clear'
    ShLd,ClkInh,Clk,Ser,Clr pin;
    A,B,C,D,E,F,G,H          pin;
    QA,QB,QC,QD,QE,QF,QG     node istype 'reg,buffer';
    QH                       pin  istype 'reg,buffer';
    Clock                    node istype 'com';

Equations
    Clock = Clk # ClkInh;
    When (ShLd) Then
        [QH,QG,QF,QE,QD,QC,QB,QA] := [QG,QF,QE,QD,QC,QB,QA,Ser];   "Shift
    Else
        [QH,QG,QF,QE,QD,QC,QB,QA] := [ H, G, F, E, D, C, B, A];    "Load
    [QH,QG,QF,QE,QD,QC,QB,QA].clk = Clock;
    [QH,QG,QF,QE,QD,QC,QB,QA].ar  = !Clr;

Test_Vectors
([Clk,ClkInh,Clr,ShLd,Ser,A,B,C,D,E,F,G,H]->[QA,QB,QC,QD,QE,QF,QG,QH])
 [.C.,   1  , 1 , 0  , 0 ,1,0,1,0,1,0,1,1]->[ 1, 0, 1, 0, 1, 0, 1, 1];
 [.C.,   1  , 1 , 1  , 0 ,1,0,1,0,1,0,1,1]->[ 1, 0, 1, 0, 1, 0, 1, 1];
 [.C.,   0  , 1 , 1  , 0 ,0,0,0,0,0,0,0,0]->[ 0, 1, 0, 1, 0, 1, 0, 1];
 [.C.,   0  , 1 , 1  , 0 ,0,0,0,0,0,0,0,0]->[ 0, 0, 1, 0, 1, 0, 1, 0];
 [.C.,   0  , 1 , 1  , 1 ,0,0,0,0,0,0,0,0]->[ 1, 0, 0, 1, 0, 1, 0, 1];
 [.C.,   0  , 1 , 1  , 1 ,0,1,1,1,1,0,0,0]->[ 1, 1, 0, 0, 1, 0, 1, 0];
 [.C.,   0  , 1 , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 0, 0, 1, 0, 1];
 [ 0 ,   0  , 1 , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 0, 0, 1, 0, 1];
 [ 0 ,   1  , 1 , 1  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 0, 1, 1, 0, 0, 1, 0];
 [.C.,   1  , 1 , 0  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 0, 1, 1, 0, 0, 1, 0];
 [.C.,   0  , 1 , 0  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 1, 1, 1, 1, 1, 0, 0];
 [ 0 ,   0  , 0 , 0  , 0 ,0,1,1,1,1,1,0,0]->[ 0, 0, 0, 0, 0, 0, 0, 0];
End
```

Figure 6.17 ABEL source file for 74166 loadable shifter

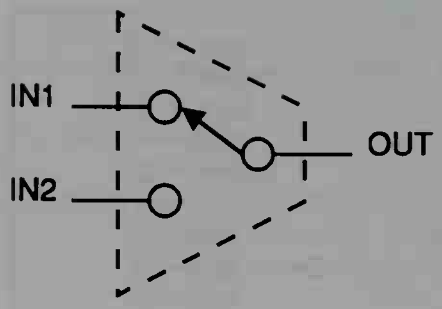


Figure 6.18 Trivial multiplexer

6.5 MULTIPLEXERS

A multiplexer is a circuit that directs data from two or more sets of inputs into one set of outputs. The simplest multiplexer can be represented by a double-pole switch, as shown in Figure 6.18. This multiplexer simply selects one of the two inputs and routes it to the output, based on the state of the switch. More complex multiplexers may have many possible inputs to be selected from and therefore require a decoding function to determine which input (or set of inputs) is desired.

Multiplexers are used often in digital circuits and can be thought of as just another type of logic gate, no less useful than ANDs and ORs. Some FPGAs (most notably the Actel devices) use multiplexers as building blocks for larger circuits. On a somewhat larger scale, multiplexers are used to save wires (or routing channels) when passing data from one portion of a circuit to another.

The 74153 standard logic device (shown in Figure 6.19) is a commonly used multiplexer device. Figure 6.20 is an ABEL description of a four-into-one multiplexer that has the same function as a 74153 standard logic device. This circuit selects one set of two inputs from the four possible input sets and routes them to the outputs. The design description uses ABEL's WHEN-THEN equation language to describe the multiplexer operation.

6.6 DEMULTIPLEXERS

Demultiplexers are circuits that perform the reverse of a multiplexer operation. A demultiplexer takes a single set of inputs and routes the data to a selected set of outputs. The only characteristic of demultiplexers that is different from that of multiplexers is that a demultiplexer must provide a value (either high or low, 1 or 0) for the outputs that are not currently selected.

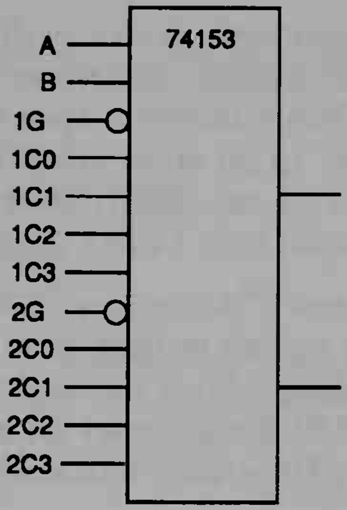


Figure 6.19 74153 Standard logic device

```
Module sn74153
Title 'Dual 4-line to 1-line data-selector/multiplexer'

    _1C0,_1C1,_1C2,_1C3      pin;
    _2C0,_2C1,_2C2,_2C3      pin;
    A,B,G                    pin;
    _1G,_2G                  pin;
    _1Y,_2Y                  pin istype 'com';
    C0 = [_2C0,_1C0];
    C1 = [_2C1,_1C1];
    C2 = [_2C2,_1C2];
    C3 = [_2C3,_1C3];
    G  = [ _2G, _1G];
    Y  = [ _2Y, _1Y];

Equations
    When (!B & !A) Then
        Y = C0 & !G;
    Else When (!B &  A) Then
        Y = C1 & !G;
    Else When ( B & !A) Then
        Y = C2 & !G;
    Else When ( B &  A) Then
        Y = C3 & !G;

Test_Vectors ([ G, B, A,C0,C1,C2,C3] -> Y)
             [ 0, 0, 0, 0, 1, 2, 3] -> 0;
             [ 0, 0, 1, 0, 1, 2, 3] -> 1;
             [ 0, 1, 0, 0, 1, 2, 3] -> 2;
             [ 0, 1, 1, 0, 1, 2, 3] -> 3;
             [ 1, 1, 1, 3, 3, 3, 3] -> 2;
             [ 2, 1, 1, 3, 3, 3, 3] -> 1;
             [ 0, 0, 0, 3, 2, 1, 0] -> 3;
             [ 0, 0, 1, 3, 2, 1, 0] -> 2;
             [ 0, 1, 0, 3, 2, 1, 0] -> 1;
             [ 0, 1, 1, 3, 2, 1, 0] -> 0;
End
```

Figure 6.20 Dual four- to one-line multiplexer design file

The 74154 standard logic device is diagrammed in figure 6.21. In this device, four select inputs decode to provide one of sixteen possible output routings. The selected output is low whenever both data inputs *G1* and *G2* are low. In all other situations the sixteen data outputs are high. Figure 6.22 is an ABEL source file describing a multiplexer identical in operation to a 74154 decoder/demultiplexer.

The 74154, like most TTL devices, is active low, meaning that the value actually observed on the output pins is reversed from what one might expect when thinking about the logic function being implemented. To reflect this, the ABEL design uses an inversion of the *Output* set to allow values to be entered in a more natural way in the truth table that follows.

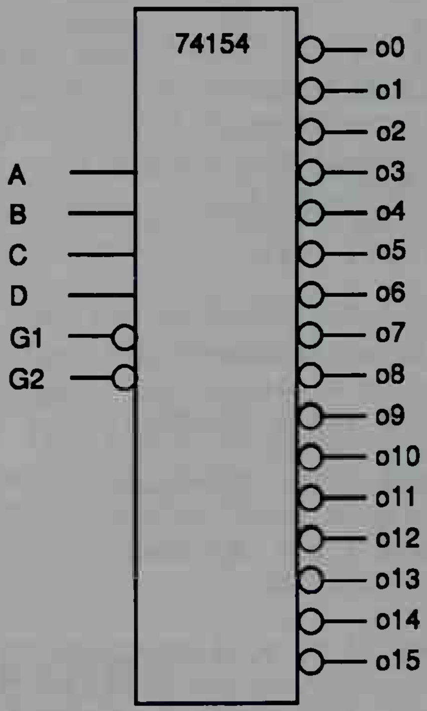


Figure 6.21 74154 Standard logic device

```
Module sn74154
Title '4-line to 16-line decoder/demultiplexer'

    A,B,C,D    pin;
    G1_,G2_    pin;
    o0..o15    pin istype 'com';

    Select = [D,C,B,A];
    Output = ![o15..o0];

Truth_Table ([G1_,G2_,Select] -> Output)
             [ 1 ,.X.,  .X. ] -> ^h0000;
             [.X., 1 ,  .X. ] -> ^h0000;
             [ 0 , 0 ,   0  ] -> ^h0001;
             [ 0 , 0 ,   1  ] -> ^h0002;
             [ 0 , 0 ,   2  ] -> ^h0004;
             [ 0 , 0 ,   3  ] -> ^h0008;
             [ 0 , 0 ,   4  ] -> ^h0010;
             [ 0 , 0 ,   5  ] -> ^h0020;
             [ 0 , 0 ,   6  ] -> ^h0040;
             [ 0 , 0 ,   7  ] -> ^h0080;
             [ 0 , 0 ,   8  ] -> ^h0100;
             [ 0 , 0 ,   9  ] -> ^h0200;
             [ 0 , 0 ,  10  ] -> ^h0400;
             [ 0 , 0 ,  11  ] -> ^h0800;
             [ 0 , 0 ,  12  ] -> ^h1000;
             [ 0 , 0 ,  13  ] -> ^h2000;
             [ 0 , 0 ,  14  ] -> ^h4000;
             [ 0 , 0 ,  15  ] -> ^h8000;

Test_Vectors ([G1_,G2_,Select] -> Output)
              [ 1 ,.X.,  .X. ] -> ^h0000;
              [.X., 1 ,  .X. ] -> ^h0000;
              [ 0 , 0 ,   0  ] -> ^h0001;
              [ 0 , 0 ,   1  ] -> ^h0002;
              [ 0 , 0 ,   2  ] -> ^h0004;
              [ 0 , 0 ,   3  ] -> ^h0008;
              [ 0 , 0 ,   4  ] -> ^h0010;
              [ 0 , 0 ,   5  ] -> ^h0020;
              [ 0 , 0 ,   6  ] -> ^h0040;
              [ 0 , 0 ,   7  ] -> ^h0080;
              [ 0 , 0 ,   8  ] -> ^h0100;
              [ 0 , 0 ,   9  ] -> ^h0200;
              [ 0 , 0 ,  10  ] -> ^h0400;
              [ 0 , 0 ,  11  ] -> ^h0800;
              [ 0 , 0 ,  12  ] -> ^h1000;
              [ 0 , 0 ,  13  ] -> ^h2000;
              [ 0 , 0 ,  14  ] -> ^h4000;
              [ 0 , 0 ,  15  ] -> ^h8000;
End
```

Figure 6.22 ABEL file for 74154 demultiplexer

6.7 REFERENCES

Greenfield, Joseph D., *Practical Digital Design Using ICs*, John Wiley & Sons, New York, 1983.

Texas Instruments, *TTL Logic Data Book*, Texas Instruments, Dallas, TX, 1988.

7

Counters

Many of the most common sequential applications are based on counters. Since most target architectures (including simple PLDs and most FPGAs) are not well optimized for counter applications, it's important to understand how a counter can be most efficiently implemented, given various constraints.

7.1 T FLIP-FLOP COUNTERS

Users of standard logic devices are probably most familiar with counters constructed out of T flip-flops. T flip-flops are well suited to counters and to many other types of synchronous state machines. Figure 7.1 shows the bit patterns that are observed on the outputs of a 4-bit up counter for the 16 states of its operation. These binary values correspond to the decimal values 0 through 15. Using flip-flops, such a counter can be implemented in a variety of ways. Users of discrete TTL devices will usually build such a counter using T flip-flops in a circuit such as the one shown in Figure 7.2. This is the simplest implementation of a counter and takes advantage of the edge-triggered clock available in each flip-flop. In a ripple counter such as this, each flip-flop is clocked at a rate of one-half the rate of the previous flip-flop. This is a reflection of the behavior seen in the bit values presented in Figure 7.1: each bit position toggles when its less-significant neighbor transitions from 1 to 0.

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Figure 7.1 Bit values of a four-bit up counter

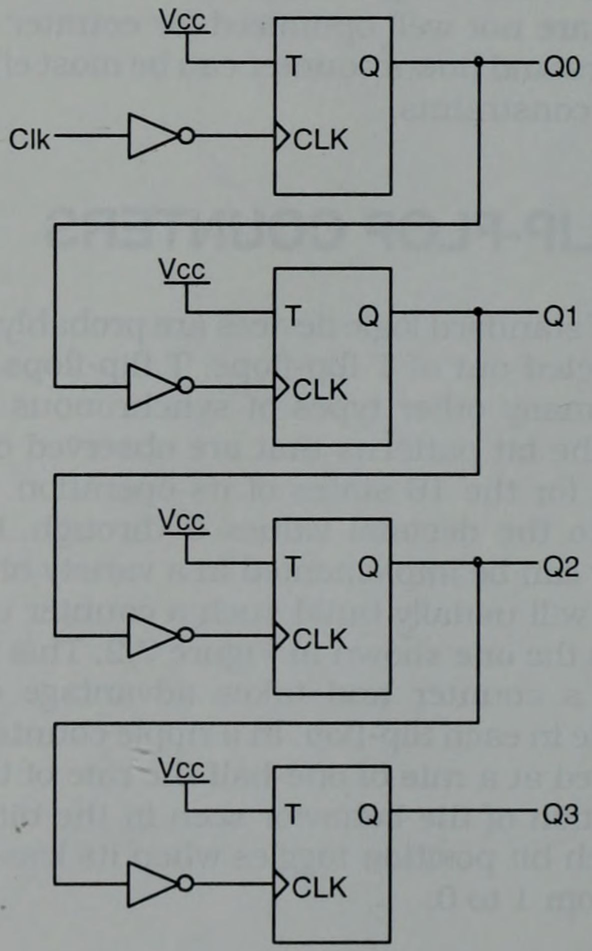


Figure 7.2 Four-bit T flip-flop counter

There are two problems with a counter such as this. First, the current state of the counter can't be determined at any point in time. This is fine if the purpose of the counter is to divide a frequency, but is not acceptable if the counter is being used to control a state machine, to cycle through memory addresses, or for some other application where the actual counter value is significant. Second, the counter can't operate at high speeds due to the large number of propagation delays. Furthermore, if we wish to use one or more PLDs to implement the counter circuit, we find that this circuit isn't appropriate for the average PLD, since most PLDs have flip-flops that are clocked from a common source (this may be a good circuit to consider, however, if the application is intended for an FPGA).

Look-ahead Counters

To implement this counter circuit in a PLD or other target architecture with constrained clock sources, we'll need to implement it using a different scheme, such as that shown in Figure 7.3.

This 8-bit counter circuit doesn't require independently clocked flip-flops or multiple levels of logic to be implemented, and can therefore be implemented with commonly clocked T flip-flops and no additional outputs. In this implementation of the counter, each succeeding bit of the counter is toggled whenever all the preceding bits are true (as seen in Figure 7.1). Thus, each of the counter's flip-flops requires just one product term with *n* inputs, where *n* is the number of less significant bits. The least significant bit of the counter is tied directly to V_{CC}. The 8-bit counter can be described with the following ABEL equations for the flip-flops' T inputs:

```
Q7.T = Q6 & Q5 & Q4 & Q3 & Q2 & Q1 & Q0;
Q6.T = Q5 & Q4 & Q3 & Q2 & Q1 & Q0;
Q5.T = Q4 & Q3 & Q2 & Q1 & Q0;
Q4.T = Q3 & Q2 & Q1 & Q0;
Q3.T = Q2 & Q1 & Q0;
Q2.T = Q1 & Q0;
Q1.T = Q0;
Q0.T = 1;
```

The T inputs to the counter's flip-flops are indicated through the use of dot extension suffixes appended to the name of the design's outputs.

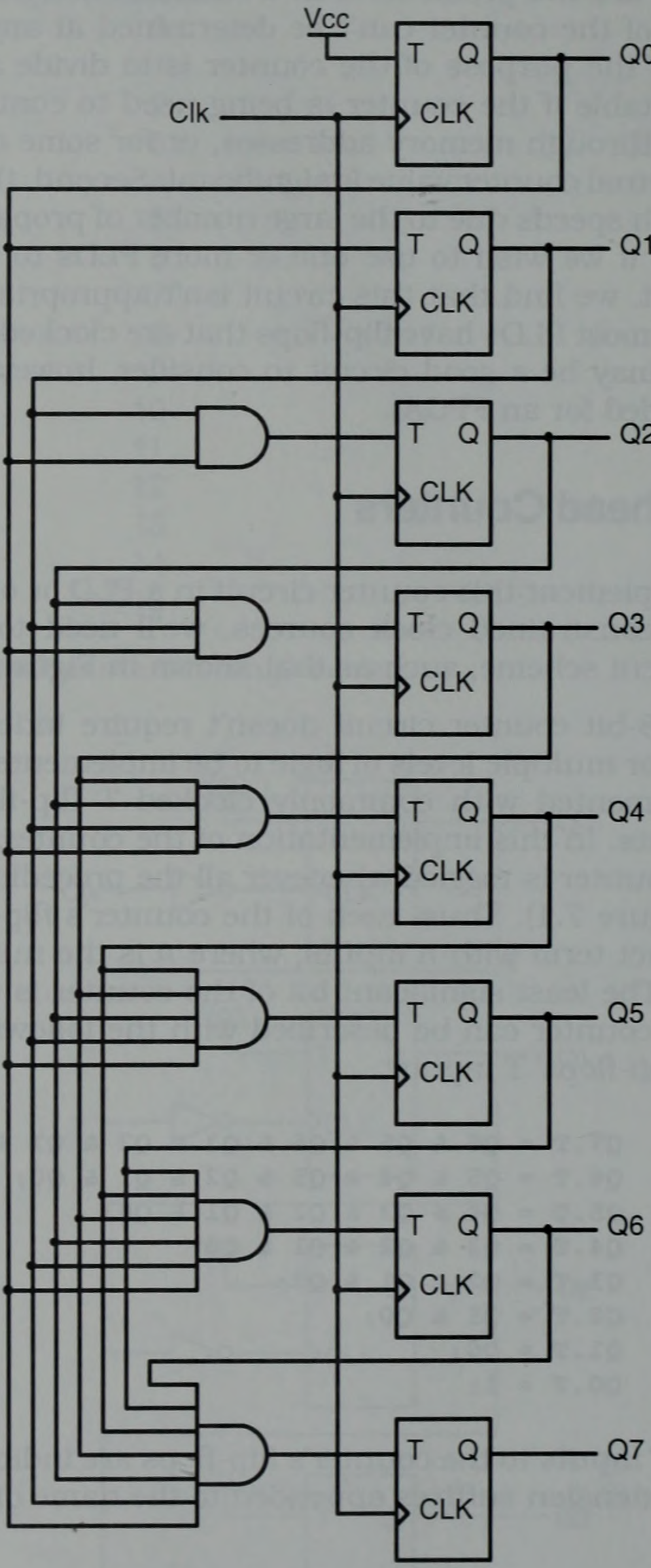


Figure 7.3 Eight-bit look-ahead carry counter

7.2 D FLIP-FLOP COUNTERS

The previous counter circuit can be implemented in a PLD with T flip-flops, but these devices are typically more expensive that the simpler PALs with D flip-flops. How can a design that is described most naturally with T flip-flops be implemented in one of the lower-cost PLDs that has only D flip-flops? The easiest way is to convert the circuit to a D flip-flop implementation by using the flip-flop emulation technique discussed in Chapter 4, where we showed that, with the simple addition of an XOR gate, a D flip-flop can be made to function as a T flip-flop, and a T flip-flop can be made to function as a D flip-flop.

To implement the eight-bit counter in a device with D flip-flops, we XOR the counter equations with the fed-back flip-flop outputs:

```
Q7 := Q7 $ Q6 & Q5 & Q4 & Q3 & Q2 & Q1 & Q0;
Q6 := Q6 $ Q5 & Q4 & Q3 & Q2 & Q1 & Q0;
Q5 := Q5 $ Q4 & Q3 & Q2 & Q1 & Q0;
Q4 := Q4 $ Q3 & Q2 & Q1 & Q0;
Q3 := Q3 $ Q2 & Q1 & Q0;
Q2 := Q2 $ Q1 & Q0;
Q1 := Q1 $ Q0;
Q0 := Q0 $ 1;
```

Notice that we haven't used any dot extensions in these equations. Since the output of a D flip-flop follows the signal values applied to the flip-flop's input, there is no distinction required between the D flip-flop input and the corresponding output, and we can describe the counter using a pin-to-pin point of view. The := assignment operator specifies to ABEL that we want registered outputs.

These equations can be implemented directly in a device with XOR gates, such as the 20X8 PAL, or implemented in a PAL device without XOR gates, such as the 16R8. In the latter case, the XOR operators are converted into sum-of-products logic. The large number of product terms required to implement an 8-bit counter (the most significant bit of an *n*-bit counter will always require at least *n* product terms) may preclude the use of a simple PAL device if additional logic is required for reset or data-loading purposes.

Using High-level Equations for Counters

In ABEL, counters for D or T flip-flops can be written using high-level equations. For example, the eight equations shown previously for D flip-flops could be replaced by the following set declaration and high-level equation:

```
Declarations

    Count = [Q7..Q0];

Equations

    Count := Count + 1;
```

When writing high-level equations such as this, it's actually more convenient to think about the counter's behavior in terms of pin-to-pin behavior. This means that a D flip-flop representation is more natural than the T flip-flop representation.

The most popular registered PLDs and FPGAs feature D flip-flops. To some extent this simplifies the job of writing equations, since the equations required to control a D flip-flop's *D* input are the same equations that you would expect to write if you wanted to describe the behavior of your design at the device outputs. For PLDs with D flip-flops, you can in most cases ignore the existence of the flip-flops when writing design equations. There are many exceptions to this, however, and later we'll cover these situations in some detail.

If we wish to implement this simple counter using T flip-flops, we can convert the design back to a T representation with another XOR, resulting in the high-level equation

```
Count.T = Count $ (Count + 1);
```

7.3 LOADABLE COUNTERS

Figure 7.4 shows a complete ABEL design that utilizes high-level equations and explicit flip-flop conversion to create a more complex up counter. This counter features synchronous hold, load, and clear inputs.

The ABEL design includes two equations: the counter equation already described and an additional equation for the clock input to the design's flip-flops. The .CLK dot extension is used to refer to the clock inputs to the design's flip-flops (*Q7* through *Q0*).

```
module DtoT
title 'Octal counter with load and clear'

    D0..D7          pin;
    Q7..Q0          pin istype 'reg_T';
    CLK, I0, I1     pin;

    Data    = [D7..D0];
    Count   = [Q7..Q0];
    Mode    = [I1,I0];
    Clear   = [ 0, 0];
    Hold    = [ 0, 1];
    Load    = [ 1, 0];
    Inc     = [ 1, 1];
    X,C     = .x.,.c.;

equations

    Count.T = ((Count.q + 1) & (Mode == Inc)
             # (Count.q     ) & (Mode == Hold)
             # (Data        ) & (Mode == Load)
             # (0           ) & (Mode == Clear))
            $  Count.q;

    Count.C = CLK;

test_vectors ([CLK,Mode ,Data] -> Count)
              [ C ,Load ,  1 ] ->    1;
              [ C ,Inc  ,  X ] ->    2;
              [ C ,Inc  ,  X ] ->    3;
              [ C ,Inc  ,  X ] ->    4;
              [ C ,Inc  ,  X ] ->    5;
              [ C ,Load ,  3 ] ->    3;
              [ C ,Inc  ,  X ] ->    4;
              [ C ,Load ,  7 ] ->    7;
              [ C ,Inc  ,  X ] ->    8;
              [ C ,Hold ,  X ] ->    8;
              [ C ,Clear,  X ] ->    0;
              [ C ,Load ,^hFE] -> ^hFE;
              [ C ,Inc  ,  X ] -> ^hFF;
              [ C ,Inc  ,  X ] -> ^h00;
end
```

Figure 7.4 D to T flip-flop conversion using XOR equation

7.4 COUNTER RESET SCHEMES

A variety of methods can be used to reset a counter to an initial value, and the target architecture is again an important consideration when choosing the most efficient scheme. If you are using a device with more

```
module clear
title 'Synchronous clear in a device with inverted outputs'

    Q7..Q0          pin istype 'reg,invert';
    Clk,Clear,OE    pin;

    Delay = [Q7..Q0];

equations

    Delay := (Delay.fb + 1) # Clear;

    Delay.clk = Clk;
    Delay.oe  = !OE;

test_vectors([Clk,Clear,OE] -> Delay)
             [.c.,   1 , 0] -> 255;      "Registers low
             [.c.,   0 , 0] ->   0;
             [.c.,   0 , 0] ->   1;
             [.c.,   0 , 0] ->   2;
             [.c.,   0 , 0] ->   3;
             [.c.,   0 , 0] ->   4;
             [.c.,   1 , 0] -> 255;
end
```

Figure 7.5 Synchronous clear with inverted output

restrictive reset and preset options, then you must consider the impact of different reset strategies.

The ABEL design files in Figures 7.5 and 7.6 show two different methods for counter initialization using a 16R8 PAL-type device. These two methods result in different reset states and require different amounts of logic. The first method doesn't require any additional product terms to be implemented, while the second method will consume one additional product term for each output of the design. This is because the first method exploits the inherent default state of the D flip-flop. Although this default flip-flop state can be used to simplify the design of a circuit requiring a reset state, the specific configuration of the target architecture will affect the operation of the resulting circuit.

As a general rule, devices such as the 16R8 that have D-type flip-flops and fixed output inversions will require one additional product term for each output if a reset to an all-low state is desired, while no additional product terms will be required for reset to the state in which all outputs are high.

```
module set
title 'Synchronous reset in a device with inverted outputs'

    Q7..Q0          pin istype 'reg,invert';
    Clk,Clear,OE    pin;

    Delay  = [Q7..Q0];

equations

    Delay := (Delay.fb + 1) & !Clear;

    Delay.clk = Clk;
    Delay.oe  = !OE;

test_vectors([Clk,Clear,OE] -> Delay)
             [.c.,   1 , 0] ->   0;
             [.c.,   0 , 0] ->   1;
             [.c.,   0 , 0] ->   2;
             [.c.,   0 , 0] ->   3;
             [.c.,   0 , 0] ->   4;
             [.c.,   1 , 0] ->   0;
end
```

Figure 7.6 D flip-flop synchronous reset for a device with inverted outputs

7.5 POLARITY CONSIDERATIONS

Because of the many different possible configurations for registered outputs, it's difficult to design efficient sequential circuits such as counters without being aware of the constraints of the selected target architecture. Differences in architectures from one type of device to another can have profound effects on the behavior of seemingly simple circuits. This is particularly true of devices featuring programmable output inversion.

First, consider the problem of register preset and reset. If your design requires the use of a flip-flop preset or reset feature, you will need to determine whether the required features exist in the device chosen and how the configuration of the device outputs will affect the operation of those features. As an example, Figure 7.7 shows an output macrocell from a 22V10 PAL device.

In a 22V10, the polarity for each registered output is controlled by selecting or bypassing an inverter located between the *Q* output of the D flip-flop and its associated output pin. This means that there may or may not be an inversion between the outputs of the flip-flops and the corresponding device output. The result is that the behavior of the

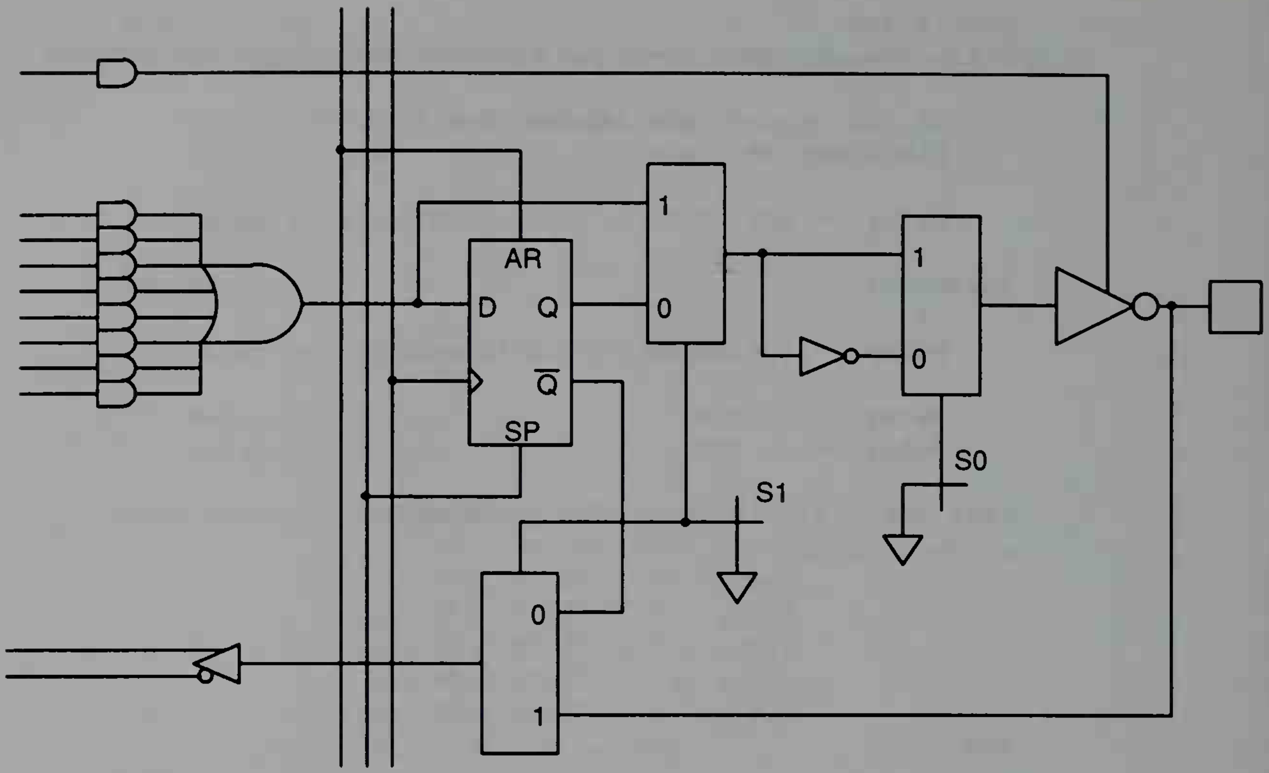


Figure 7.7 22V10 output macrocell

asynchronous reset and synchronous preset features will be different depending on whether positive or negative polarity is selected for the programmable inversion. If the reset and preset were both synchronous or both asynchronous, this difference in behavior might be accommodated by simply swapping the reset and preset logic, but this isn't possible in the 22V10. Even if such swapping is possible in your design, the nature of the preset and reset places limitations on how much benefit can be gained from this, since all the flip-flops in a 22V10 are preset and reset from common product terms.

In many devices, the programmable inverter is located before the flip-flops, on the outputs of the OR gates. This simplifies the use of programmable polarity. In all fairness, however, the 22V10's output macrocell can be used to advantage for resetting a state machine to an arbitrary default state. To design a circuit that will reset to an arbitrarily encoded default state, choose a sequence of output polarity configurations that will result in the desired state encoding on the device outputs when the global preset or reset is activated.

7.6 SR FLIP-FLOP COUNTERS

We have seen how counters can be implemented using D and T flip-flops. These two types of flip-flops are straightforward to use for counter applications and lend themselves well to high-level design description methods. Designing counters for other flip-flop types, most notably SR flip-flops, is somewhat more complex. To design a counter that utilizes SR flip-flops, we must first analyze the design requirements by once again examining the bit patterns presented previously in Figure 7.1.

When we examined these bit values to determine a T flip-flop implementation, we were attempting to determine the conditions that would indicate when each bit of the counter should toggle. This thinking reflected the behavior of the T flip-flop. The behavior of an SR flip-flop requires us to think about our counter in yet another way.

When we look at the counter's bit patterns again with SR flip-flops in mind, we are looking for those conditions under which each bit should be turned on (set) or turned off (reset). The required circuitry for each bit of the counter can be generalized: for any given clock cycle, each bit of the counter is set if it was off in the previous state and all lower-order (less significant) bits were on. Similarly, each bit is reset whenever the next higher-order (more significant) bit is turned on, or when the bit itself and all lower-order bits were previously on.

The circuit that implements this concept for a four-bit counter is shown in Figure 7.8. This circuit requires a total of five distinct product terms when implemented in a product term-sharing PLD such as the PLS105. Figure 7.9 lists the Boolean equations that implement this counter circuit. The design can also be expressed in ABEL as shown in Figure 7.10. This design description enumerates all the possible counter values and, for each counter value, sets or resets the individual counter outputs based on the required next counter value. Which design description method to use is purely a matter of personal preference.

7.7 UP AND DOWN COUNTERS

The previous counter designs were all examples of up counters — counters that increment their values with each clock cycle. In PLDs with negative polarity outputs and D flip-flops, you will find that it's usually more efficient to implement a delay or event counter circuit by using a down counter (the 16R8, for example, is a natural device for down counting, but is rather inefficient for up counting). As Figure 7.11 illustrates, converting an up counter to a down counter is a simple

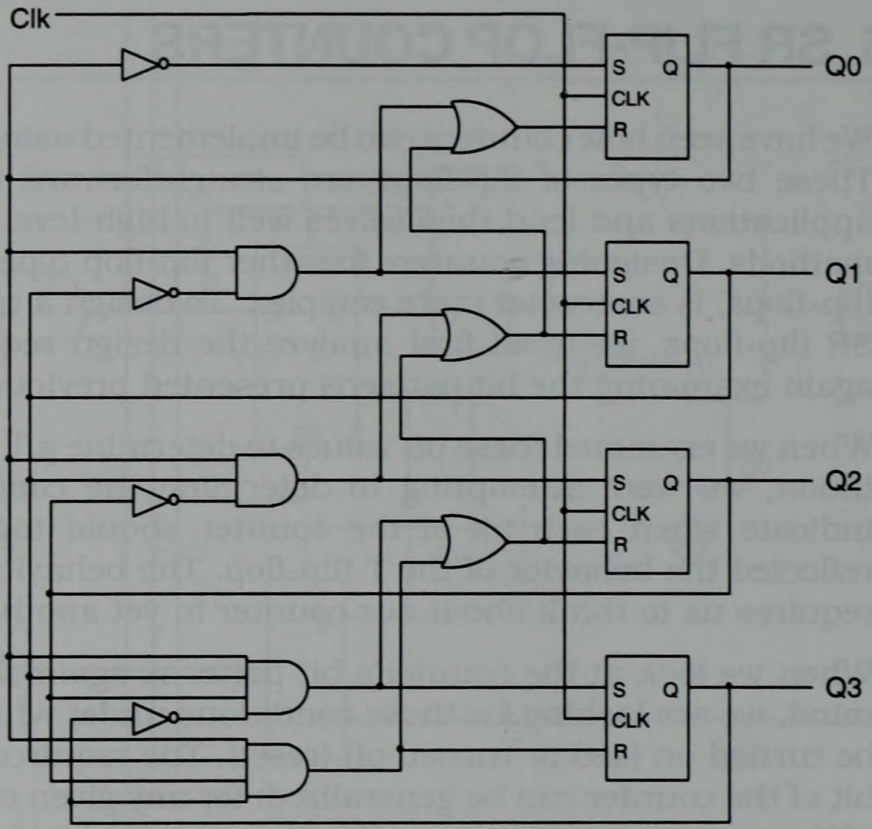


Figure 7.8 Four-bit SR flip-flop counter circuit

```
Q0.S = !Q0;
Q0.R = !Q3 & Q2 & Q1 & Q0
     # Q3 & Q2 & Q1 & Q0
     # !Q2 & Q1 & Q0
     # !Q1 & Q0;

Q1.S = !Q1 & Q0;
Q1.R = !Q3 & Q2 & Q1 & P0
     # Q3 & Q2 & Q1 & Q0
     # !Q2 & Q1 & Q0;

Q2.S = !Q2 & Q1 & Q0;
Q2.R = !Q3 & Q2 & Q1 & Q0
     # Q1 & Q2 & Q1 & Q0;

Q3.S = !Q3 & Q2 & Q1 & Q0;
Q3.R = Q3 & Q2 & Q1 & Q0;
```

Figure 7.9 SR flip-flop counter equations

```
module countsr

    Clk, PR     pin;
    Q3..Q0      pin istype 'reg_SR, buffer';
    Q           = [Q3..Q0];

equations

    Q.AP = PR;  "Asynchronous Preset
    Q.C  = Clk;

    [                Q0.S] = (Q ==  0);
    [           Q1.S,Q0.R] = (Q ==  1);
    [                Q0.S] = (Q ==  2);
    [      Q2.S,Q1.R,Q0.R] = (Q ==  3);
    [                Q0.S] = (Q ==  4);
    [           Q1.S,Q0.R] = (Q ==  5);
    [                Q0.S] = (Q ==  6);
    [Q3.S,Q2.R,Q1.R,Q0.R] = (Q ==  7);
    [                Q0.S] = (Q ==  8);
    [           Q1.S,Q0.R] = (Q ==  9);
    [                Q0.S] = (Q == 10);
    [      Q2.S,Q1.R,Q0.R] = (Q == 11);
    [                Q0.S] = (Q == 12);
    [           Q1.S,Q0.R] = (Q == 13);
    [                Q0.S] = (Q == 14);
    [Q3.R,Q2.R,Q1.R,Q0.R] = (Q == 15);

test_vectors  ([Clk,PR] ->  Q)
               [ 1 , 1] -> 15;
               [ 1 , 0] -> 15;
               [.c., 0] ->  0;
               [.c., 0] ->  1;
               [.c., 0] ->  2;
               [.c., 0] ->  3;
               [.c., 0] ->  4;
               [.c., 0] ->  5;
               [.c., 0] ->  6;
               [.c., 0] ->  7;
               [.c., 0] ->  8;
               [.c., 0] ->  9;
               [.c., 0] -> 10;
               [.c., 0] -> 11;
               [.c., 0] -> 12;
               [.c., 0] -> 13;
               [.c., 0] -> 14;
               [.c., 0] -> 15;
               [.c., 0] ->  0;
               [.c., 0] ->  1;
               [.c., 0] ->  2;
end
```

Figure 7.10 Four-bit SR counter design file

Binary	Decimal	Binary	Decimal
0000	0	1111	15
0001	1	1110	14
0010	2	1101	13
0011	3	1100	12
0100	4	1011	11
0101	5	1010	10
0110	6	1001	9
0111	7	1000	8
1000	8	0111	7
1001	9	0110	6
1010	10	0101	5
1011	11	0100	4
1100	12	0011	3
1101	13	0010	2
1110	14	0001	1
1111	15	0000	0

Figure 7.11 Comparison of up-down counter values

matter of inverting the counter circuit's outputs. The initial state then becomes the one's complement of the original initial state, and the counter sequences down, rather than up.

For further reductions in logic, you might find that inverting just some of the counter's outputs will result in the most efficient implementation. This is most easily done using target architectures that feature programmable polarity for all outputs. If your counter circuit will be feeding the inputs to other devices or design modules, you can often simply invert the active level (or *sense*) of the counter outputs and then reinvert them at the other device's inputs, as illustrated in Figure 7.12.

7.8 ARBITRARY LENGTH COUNTERS

Counters are sometimes required that are of an arbitrary length. Using high-level equations, such a counter can be easily developed. For example, if a 111-state counter is required that counts from zero to a value of 110, we could write a high-level equation of the form

```
Count := (Count + 1) & (Count <= 110) & !Clr;
```

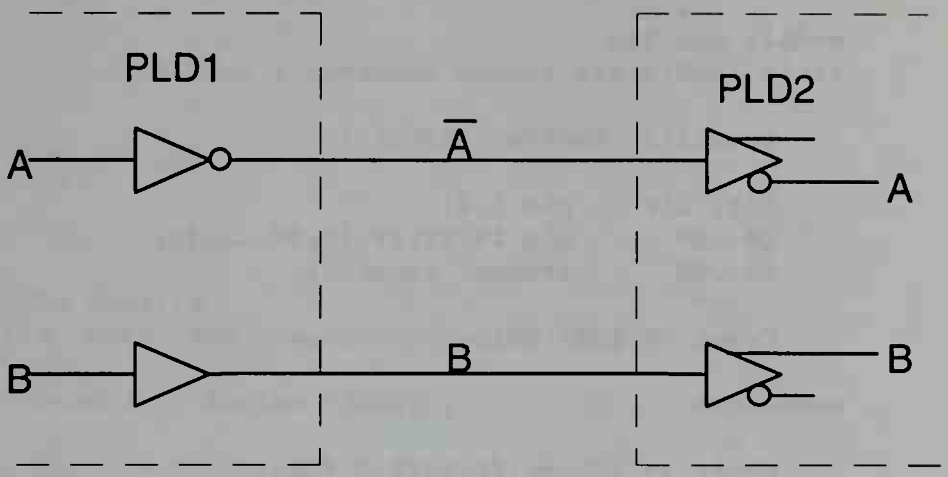


Figure 7.12 Inverting circuit active levels

This equation accurately describes a 111-state counter, but as written, it produces sum of products equations that are larger than necessary, too large, in fact, to fit in simple PAL-type devices.

When D flip-flops are used, a counter with no terminating value (a 256-state eight-bit counter, for example) will always require *n* product terms for the most significant bit of the counter, where *n* is the number of counter bits. For an arbitrary length counter, however, the size of the minimized equations depends on the specified terminating value. To get this counter to fit into typical programmable devices, some design changes are required.

One possible approach is to design the counter with a short count comparator by using a device with a synchronous reset term. The PAL 22V10 is one such device. This method is shown in the ABEL source file of figure 7.13. An equation is written for the synchronous preset term of the 22V10 that compares the counter value with the desired terminating value. This design requires that the 22V10's configurable outputs all have an inversion after the register so that when the registers are preset, the desired counter reset value (all low) will appear on the outputs. This is ensured by the use of an ISTYPE statement. In the absence of the ISTYPE 'invert' statement, ABEL would choose the output polarity that implemented the design equations most efficiently, and this polarity may or may not be the polarity required for correct operation of the counter's reset function.

If you want to use a device without a preset feature, you can implement the short count circuit by writing a short count comparator equation for an unused output and routing that output back into the counter equations. The ABEL design shown in Figure 7.14 uses this method to implement the design in a 20V8 GAL device. The architecture of the GAL

```
module cnt111a
title 'Arbitrary length counter, 0 to 110'

    count111  device 'P22V10';

    Clk, Clr    pin 1,2;
    Q6..Q0      pin 16,17,18,19,20,21,22;
    Q6..Q0      istype 'invert';

    Count   = [Q6..Q0];

equations

    Count := (Count.fb + 1) & !Clr;
    Count.sp = (Count.fb == 110);  "Synchronous preset term
    Count.clk = Clk;

test_vectors ([Clk,Clr] -> Count)
              [.c., 1 ] ->   0;
              [.c., 0 ] ->   1;
              [.c., 0 ] ->   2;
              [.c., 0 ] ->   3;
              [.c., 0 ] ->   4;
              [.c., 0 ] ->   5;
              [.c., 1 ] ->   0;

@const i=1; @repeat 107 {
              [.c., 0 ] -> @expr i;; @const i=i+1; }

              [.c., 0 ] -> 108;
              [.c., 0 ] -> 109;
              [.c., 0 ] -> 110;
              [.c., 0 ] ->   0;
              [.c., 0 ] ->   1;
              [.c., 0 ] ->   2;
end
```

Figure 7.13 111-State short counter for a 22V10 or similar device

allows seven of its eight outputs to be configured with D flip-flops while one of the outputs is combinational.

To use an even simpler device, you can route the short count comparator to a registered output and decode the counter's terminal count one state earlier. This is done in the design shown in Figure 7.15. This design implements the short counter in a 20R8 PAL that has eight D outputs and no combinational outputs.

```
module cnt111b
title 'Arbitrary length counter, 0 to 110'

     count111  device 'P20V8';

     Clk,Clr,Short  pin 1,2,15;
     Q6..Q0         pin 16..22;

     Count  = [Q6..Q0];

equations

     Count := (Count.fb + 1) & !Short;
     Short  = (Count.fb == 110) # Clr;   "Synchronous preset term
     Count.clk = Clk;

test_vectors ([Clk,Clr] -> Count)
                  [.c., 1 ] ->   0;
                  [.c., 0 ] ->   1;
                  [.c., 0 ] ->   2;
                  [.c., 0 ] ->   3;
                  [.c., 0 ] ->   4;
                  [.c., 0 ] ->   5;
                  [.c., 1 ] ->   0;

@const i=1; @repeat 107 {
                  [.c., 0 ] -> @expr i;; @const i=i+1; }

                  [.c., 0 ] -> 108;
                  [.c., 0 ] -> 109;
                  [.c., 0 ] -> 110;
                  [.c., 0 ] ->   0;
                  [.c., 0 ] ->   1;
                  [.c., 0 ] ->   2;
end
```

Figure 7.14 111-State short counter for a device with no reset feature

```
module cnt111c
title 'Arbitrary length counter, 0 to 110'

    count111  device 'P20R8';

    Clk,Clr,Short  pin 1,2,15;
    Q6..Q0         pin 16..22;

    Count  = [Q6..Q0];

equations

    Count := (Count.fb + 1) & !Short.fb;
    Short := (Count.fb == 109) # Clr;    "Synchronous preset term
    [Count,Short].clk = Clk;

test_vectors ([Clk,Clr] -> Count)
              [.c., 1 ] ->   0;
              [.c., 0 ] ->   0; "Extra cycle in 0 because of short
              [.c., 0 ] ->   1;
              [.c., 0 ] ->   2;
              [.c., 0 ] ->   3;
              [.c., 0 ] ->   4;
              [.c., 0 ] ->   5;
              [.c., 1 ] ->   6; "Clear is delayed because of short
              [.c., 0 ] ->   0;

@const i=1; @repeat 107 {
              [.c., 0 ] -> @expr i;; @const i=i+1; }

              [.c., 0 ] -> 108;
              [.c., 0 ] -> 109;
              [.c., 0 ] -> 110;
              [.c., 0 ] ->   0;
              [.c., 0 ] ->   1;
              [.c., 0 ] ->   2;
end
```

Figure 7.15 110-State counter for a simple 20R8 PAL device

7.9 CHAINED COUNTERS

If a larger counter is required (one that counts to a value greater than can be accommodated in the target architecture), multiple counters of various terminating values can be cascaded by providing carry signals between succeeding counter blocks. When breaking up a large counter in this way, the optimal size of the individual counter elements will differ depending on the target architecture.

Using @CARRY to Break up Counters

ABEL's @CARRY directive can be used to specify counter chains. @CARRY instructs the ABEL compiler to break counters into smaller segments of a specified width while preserving carry logic on automatically generated intermediate nodes.

Using @CARRY, it is possible to describe extremely large counter functions. For example, the design shown in figure 7.16 uses @CARRY to break up a large (36-bit) counter into 18 smaller 2-bit counters. The value specified in the @CARRY directive can be selected to best match the requirements of the target device architecture. For most PLDs, a value of between 2 and 8 is usually appropriate. For devices with smaller logic blocks, such as FPGAs, a smaller value (4 or less) is probably more appropriate.

7.10 WAVEFORM GENERATOR

One common application of counters is in the construction of waveform generator circuits. A simple waveform generator consists of a counter and waveform decoding logic, as shown in the block diagram of Figure 7.17.

When a repeating pattern of arbitrary waveforms such as the one shown in figure 7.18 is required, you must first determine the number of counter states that will be required to accurately produce all the desired waveform events. The counter's terminal value depends on how many events there are in the waveform, as well as when the events are to occur in relation to clock edges that occur as a result of different possible clock speeds. In our sample waveform, the pattern repeats after just twelve clock cycles, so a counter that increments (or decrements) through twelve states is sufficient.

```
module cnt36bit
title '36-bit counter described using @CARRY'

    Reset,Clock       pin;
    Hold,Load         pin;
    D35..D0           pin;
    Q35..Q0           pin istype 'reg,buffer';

    Data  = [D35..D0];
    Count = [Q35..Q0];

equations
@CARRY 2;

    Count.CLK = Clock;
    Count.CLR = Reset;

    Count := (Count.FB + 1) & !Hold & !Load
          #  Count.FB & Hold & !Load
          #  Data & Load;

test_vectors ([Clock,Reset,Hold,Load,    Data   ]-> Count)
              [ .c. ,  1 , 0 , 0 ,     .x.   ]->^h000000000;
@const i = 1;
@repeat 100 { [ .c. ,  0 , 0 , 0 ,     .x.   ]-> @expr i;;
@const i = i+1;
}
              [ .c. ,  0 , 0 , 1 ,^hFFFFFFFF9]->^hFFFFFFFF9;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFA;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFB;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFC;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFD;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFE;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^hFFFFFFFFF;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^h000000000;
              [ .c. ,  0 , 0 , 0 ,     .x.   ]->^h000000001;
              [ .c. ,  0 , 1 , 0 ,     .x.   ]->^h000000001;

end
```

Figure 7.16 Using the @CARRY directive to create an adder chain

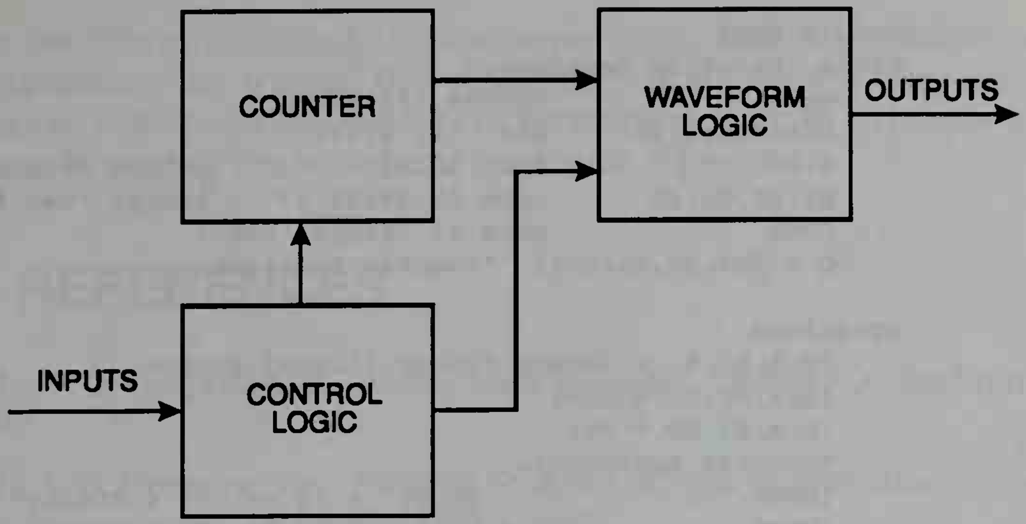


Figure 7.17 Waveform generator block diagram

Analyzing the required events, we find that signal *A* should go high after clock cycle one, and low after clock cycle three. Signal *B* goes high after cycle two, low after cycle five, and high again for one clock cycle during cycle nine.

We have described the waveform in terms of transitions, and this description maps naturally into an SR flip-flop implementation. If we target this design to a device that features SR flip-flops, we can easily implement this design. Figure 7.19 lists the ABEL design description for the waveform generator.

This design assumes buried registers (indicated by the NODE declarations) for the counter function and uses separate registers associated with output pins for the waveform decoding function. The counter segment of the design is described in a manner similar to the earlier SR flip-flop counter example. A separate combinational output is used to provide the short count function required for the 12-state counter.

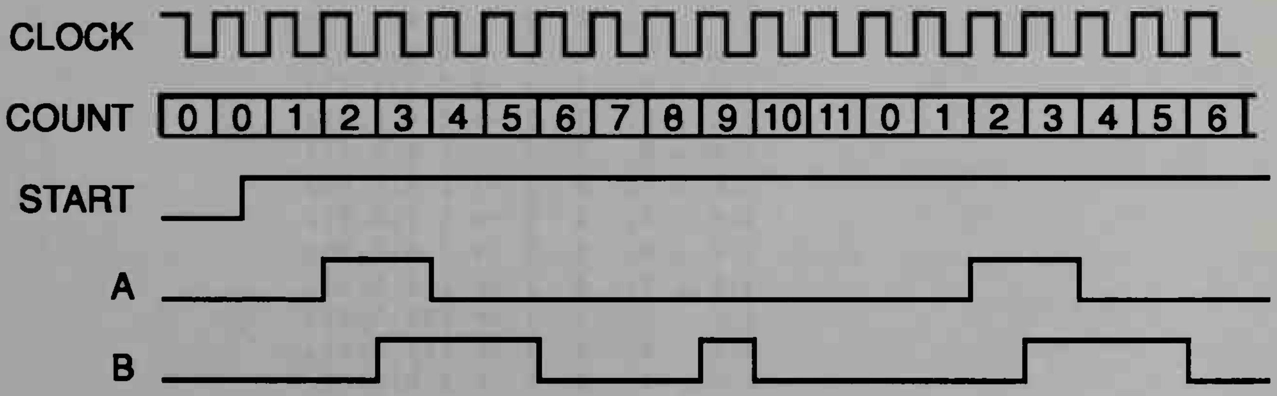


Figure 7.18 Sample waveform

```
module WAVE
title 'Waveform Generator'
    wave               device 'F105';
    Clk,Start,PR       pin   1, 8,19;
    A,B                pin  10,11           istype 'reg_SR';
    Q3,Q2,Q1,Q0        node 40,39,38,37     istype 'reg_SR';
    COMP               node 49 istype 'com';
    Q = [Q3,Q2,Q1,Q0];  "Counter Registers

equations
    [Q,A,B].R  = !COMP; "Clear Illegal states
    [Q,A,B].C  = Clk;
    [Q,A,B].AP = PR;
    "Counter Equations...
    [COMP,                 Q0.S]  = (Q ==  0) & Start; " 0 to  1
    [COMP,            Q1.S,Q0.R]  = (Q ==  1) & Start; " 1 to  2
    [COMP,                 Q0.S]  = (Q ==  2) & Start; " 2 to  3
    [COMP,       Q2.S,Q1.R,Q0.R]  = (Q ==  3) & Start; " 3 to  4
    [COMP,                 Q0.S]  = (Q ==  4) & Start; " 4 to  5
    [COMP,            Q1.S,Q0.R]  = (Q ==  5) & Start; " 5 to  6
    [COMP,                 Q0.S]  = (Q ==  6) & Start; " 6 to  7
    [COMP,Q3.S,Q2.R,Q1.R,Q0.R]    = (Q ==  7) & Start; " 7 to  8
    [COMP,                 Q0.S]  = (Q ==  8) & Start; " 8 to  9
    [COMP,            Q1.S,Q0.R]  = (Q ==  9) & Start; " 9 to 10
    [COMP,                 Q0.S]  = (Q == 10) & Start; "10 to 11
    [COMP,Q3.R,Q2.R,Q1.R,Q0.R]    = (Q == 11) & Start; "11 to  0
    "Output Waveform equations...
    A.S  = (Q==1) & Start;
    A.R  = (Q==3) & Start;
    B.S  = (Q==2) & Start;
    B.R  = (Q==5) & Start;
    B.S  = (Q==8) & Start;
    B.R  = (Q==9) & Start;

test_vectors   ([Clk,PR,Start] -> [ Q,A,B])
                [ 0 , 0,  0  ] -> [.X.,.X.,.X.];
                [ 1 , 1,  0  ] -> [15,1,1]; " Preset high
                [ 1 , 0,  0  ] -> [15,1,1]; " Preset low
                [.C., 0,  0  ] -> [ 0,0,0];
                [.C., 0,  0  ] -> [ 0,0,0];
                [.C., 0,  0  ] -> [ 0,0,0];
                [.C., 0,  1  ] -> [ 1,0,0];
                [.C., 0,  1  ] -> [ 2,1,0];
                [.C., 0,  1  ] -> [ 3,1,1];
                [.C., 0,  1  ] -> [ 4,0,1];
                [.C., 0,  1  ] -> [ 5,0,1];
                [.C., 0,  1  ] -> [ 6,0,0];
                [.C., 0,  1  ] -> [ 7,0,0];
                [.C., 0,  1  ] -> [ 8,0,0];
                [.C., 0,  1  ] -> [ 9,0,1];
                [.C., 0,  1  ] -> [10,0,0];
                [.C., 0,  1  ] -> [11,0,0];
                [.C., 0,  0  ] -> [ 0,0,0];
end
```

Figure 7.19 Waveform generator design file

The waveform decoding is done separately, and uses simple equality comparisons to trigger the required output events. Separating the counter logic from the waveform generation logic in this way makes it easier to modify the waveform generator if needed.

7.11 REFERENCES

Data I/O Corporation, *ABEL User's Guide*, Data I/O, Redmond, WA, 1993.

Data I/O Corporation, *Polarity in ABEL-FPGA* (application note), Data I/O Corporation, Redmond, WA, 1992.

Pellerin, David and Michael Holley, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

8

Decoders and Comparators

8.1 ADDRESS DECODERS

Address decoders are perhaps the most common application for small PLDs and are an important part of many larger circuits. A decoder is a circuit that translates *n* binary inputs into one of up to 2^n outputs. (Code converter circuits that translate binary input data into a pattern of multiple outputs are also sometimes referred to as decoders.) If the decoding function results in the same output for multiple input combinations (either by design or due to don't-care input combinations), the number of outputs required will be less than 2^n.

An address decoder is a specialized decoder that performs a magnitude comparison function to determine the proper decoding of its inputs into a relatively small number of outputs. The typical use for such a circuit is in enabling different sections of memory based on the memory address observed on an address bus.

The ABEL source file shown in Figure 8.1 describes an address decoder that monitors the most significant seven bits of a 16-bit address bus and enables one of four different blocks of memory based on the range of the address.

In this design, the declarations for the outputs (*SRAM*, *PORT*, *UART*, and *PROM*) are made with an active-low indication (the ! operator applied

```
module decode
title 'Address decoder example'

    !SRAM, !PORT,
    !UART, !PROM    pin istype 'com';
    A15..A9         pin;

    H,L,X = 1,0,.x.;

    Address = [A15..A9,X, X,X,X,X, X,X,X,X];

equations

    SRAM = (Address < ^h8000);
    PORT = (Address >= ^h8000) & (Address < ^h81FF);
    UART = (Address >= ^h8200) & (Address < ^h83FF);
    PROM = (Address >= ^h8400);

test_vectors ([Address]->[!SRAM, !PORT, !UART, !PROM])
                [ ^h0000]->[  L ,    H ,    H ,    H ];
                [ ^h1000]->[  L ,    H ,    H ,    H ];
                [ ^h4000]->[  L ,    H ,    H ,    H ];
                [ ^h7000]->[  L ,    H ,    H ,    H ];
                [ ^h7FFF]->[  L ,    H ,    H ,    H ];
                [ ^h8000]->[  H ,    L ,    H ,    H ];
                [ ^h81FF]->[  H ,    L ,    H ,    H ];
                [ ^h8200]->[  H ,    H ,    L ,    H ];
                [ ^h83FF]->[  H ,    H ,    L ,    H ];
                [ ^h8400]->[  H ,    H ,    H ,    L ];
                [ ^hC000]->[  H ,    H ,    H ,    L ];
                [ ^hFFFF]->[  H ,    H ,    H ,    L ];
end
```

Figure 8.1 Address decoder design file

to the signal name). The active-low declarations specify that these signals will be low under the conditions indicated by the design equations, rather than high. The reason for this is that the memory chips and I/O circuitry that are being controlled with this decoder have active-low enable inputs.

This design will fit comfortably into a simple PLD such as the 16L8 PAL device. It would also be possible to use a 16H8 PAL (or a simple PROM for that matter) since the minimized Boolean equations that result from the description will, with the appropriate equation polarity, fit into devices that have either inverting or noninverting outputs. Although this active-low design uses fewer product terms when implemented with inverting outputs, an active-low circuit doesn't necessarily require an inverted output implementation.

Another thing of interest about this ABEL description is the use of a 16-bit set declaration (the six address line inputs padded to 16 bits with no-connects). The use of a 16-bit set allows the decoder to be specified in terms of actual memory address boundaries. Using no-connects in this way can vastly simplify the description of more complex circuits.

8.2 COMPARATORS

The address decoder described earlier used a comparator function to decode a set of inputs. Comparators of various types are extremely common in digital circuits and can be constructed in a variety of ways. There are two basic types of comparators: *identity comparators* and *magnitude comparators*.

Identity Comparators

An identity comparator is a circuit that compares the values observed on a set of inputs against either a fixed pattern or another set of inputs. When the comparison is made against a fixed set of values, the circuit is a decoder and is a simple AND operation, as shown in Figure 8.2.

When two sets of inputs are compared, the logic is somewhat more complex, but is easily derived if you consider that such an "equal to" comparison function is equivalent to an XNOR operation between each bit of the input sets. Correspondingly, a "not equal to" function is equivalent to an XOR. This is shown in Figure 8.3.

The amount of circuitry required to implement an equality comparator increases in a linear fashion with an increase in the size of the input

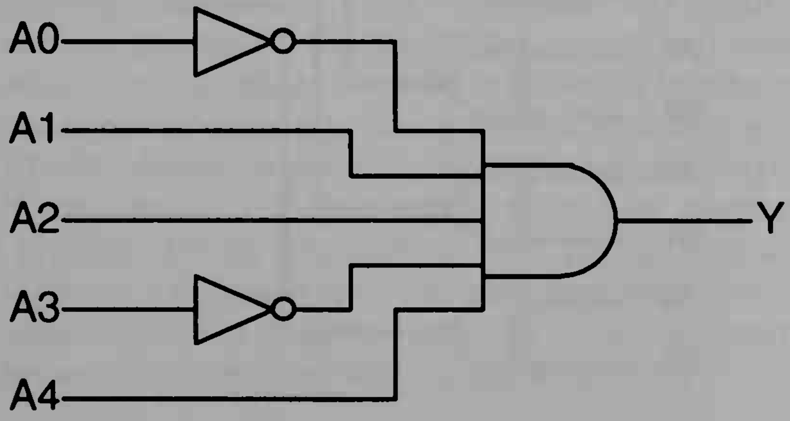


Figure 8.2 Fixed-value identity comparator circuit

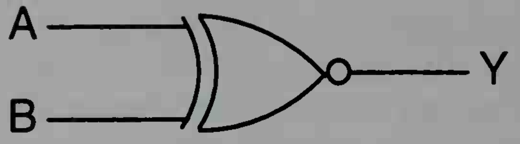


Figure 8.3 Equality comparison of two inputs

sets if XNOR (or XOR) gates are used. Figure 8.4 shows a circuit that implements an 8-bit equality comparator using XNOR gates. If the comparator is implemented in sum of products, however, the increase in size will be exponential.

The eight XNOR gates shown in the circuit provides a comparison function for one pair of bits for the two input sets. The ABEL design

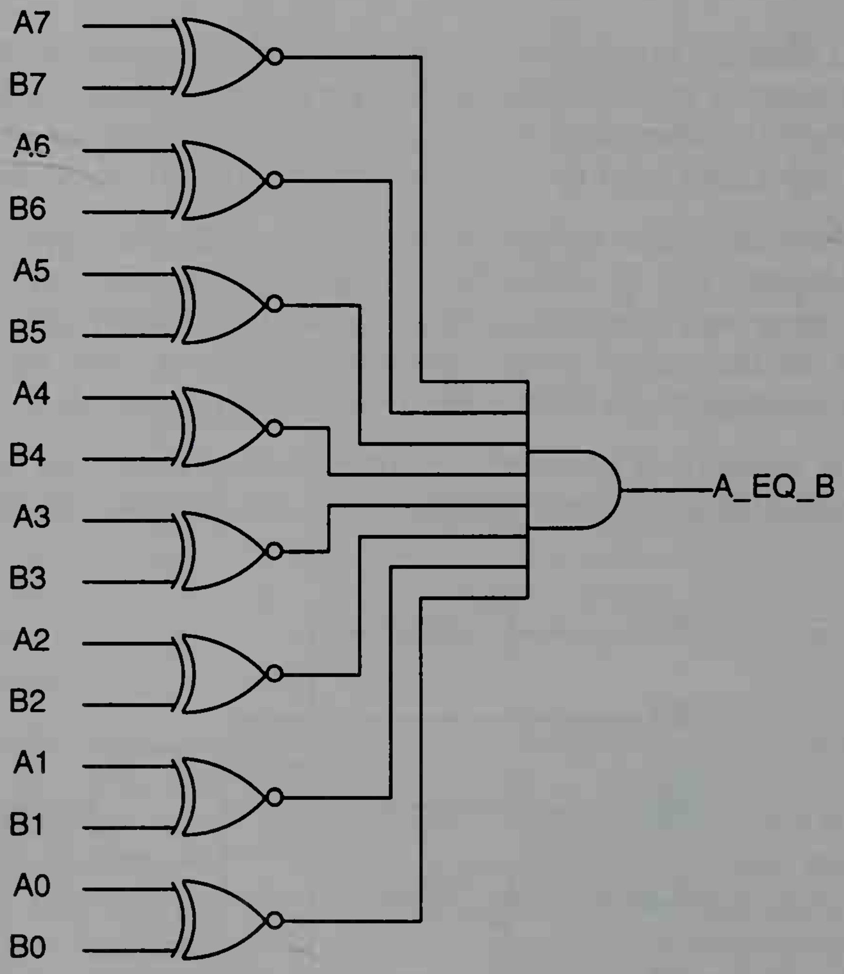


Figure 8.4 Eight-bit equality comparator circuit

```
module equal8
title '8-bit identity comparator'

    a7..a0     pin;
    b7..b0     pin;

    A_EQ_B     pin istype 'com';

    A = [a7..a0];
    B = [b7..b0];

equations

    A_EQ_B = (A == B);

test_vectors ([  A ,  B ] -> A_EQ_B)
              [^h00,^h00] ->    1;
              [^h0F,^h0F] ->    1;
              [^h1E,^hE0] ->    0;
              [^h00,^hF0] ->    0;
              [^h10,^h11] ->    0;
              [^h33,^h33] ->    1;
end
```

Figure 8.5 Eight-bit equality comparator design file

shown in Figure 8.5 implements this same circuit using a high-level equation.

To implement this design in a device that does not have XOR gates, the design will have to be flattened to convert the == operation into sum-of-products Boolean equations. The sizes of the resulting Boolean equations (shown in Figure 8.6) are too large for one output of a simple PLD, so a more complex PLD must be used unless equations are somehow split into smaller units.

When an equality comparator that compares two input sets is implemented in sum of products, the resulting circuit will always require at least $2n$ product terms, where n is the number of bits in each input set. This assumes that the circuit is being implemented in negative polarity (using a PLD with either negative or programmable output polarity, for example). If the circuit is being implemented in positive polarity the number of product terms required is 2^n. This is a general rule when designing identity comparators: equality comparators are most efficiently implemented with negative polarity, whereas inequality comparators are better implemented with positive polarity.

To reduce the product term requirements of this circuit, we need to make some changes. The simplest way to make the circuit more compact is to utilize additional outputs and implement the design in multilevel

```
!A_EQ_B =  A0 & !B0
         # !A0 &  B0
         #  A1 & !B1
         # !A1 &  B1
         #  A2 & !B2
         # !A2 &  B2
         #  A3 & !B3
         # !A3 &  B3
         #  A4 & !B4
         # !A4 &  B4
         #  A5 & !B5
         # !A5 &  B5
         #  A6 & !B6
         # !A6 &  B6
         #  A7 & !B7
         # !A8 &  B7;
```

Figure 8.6 Minimized logic for eight-bit comparator

logic. Figure 8.7 shows an 8-bit identity comparator implemented as two 4-bit comparators ANDed together. The corresponding ABEL design is shown in Figure 8.8.

In this version of the comparator, two 4-bit comparators are assigned to outputs named *TEMP1* and *TEMP2*. These outputs are then fed back and ANDed together to achieve the desired function for *A_EQ_B*.

Using @CARRY for Large Comparators

An alternative way to build large multilevel comparators is to use ABEL's @CARRY directive. This directive instructs the compiler to construct circuits such as comparators and adders using a chain of smaller circuits. To describe a very large comparator using @CARRY, we describe the comparator at a high level and specify the width of the smaller chained comparators in the argument to the @CARRY directive. Figure 8.9 illustrates how @CARRY can be used to describe a very large (36-bit) equality comparator. A value of 2 is used in the @CARRY directive, indicating that the comparator is to be constructed out of chained 2-bit comparators. Note that this is not necessarily the best implementation of a large comparator; using a balanced tree rather than a chain is usually more efficient.

Magnitude Comparators

When designing a magnitude comparator (such as an address decoder) that compares a set of inputs against one or more fixed values to

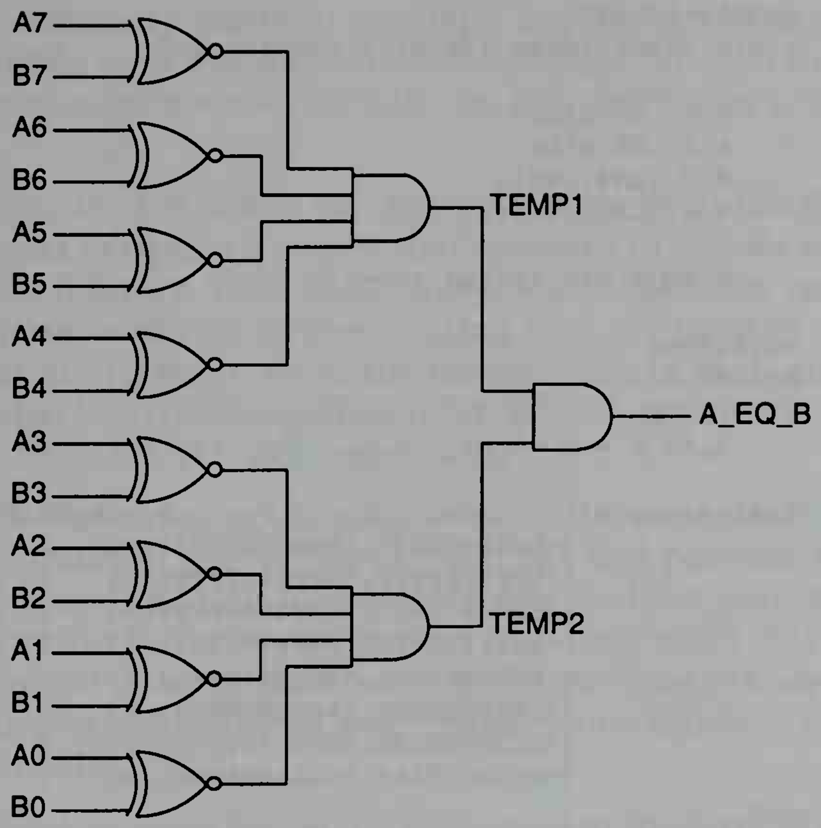


Figure 8.7 Eight-bit comparator tree

```
module equal8a
title 'Multilevel 8-bit identity comparator'

    a7..a0      pin;
    b7..b0      pin;
    A_EQ_B      pin istype 'com';
    TEMP1       node istype 'com';
    TEMP2       node istype 'com';
    A           = [a7..a0];
    B           = [b7..b0];

equations
    TEMP1   = ([a7..a4] == [b7..b4]);
    TEMP2   = ([a3..a0] == [b3..b0]);
    A_EQ_B = TEMP1 & TEMP2;

test_vectors ([  A ,  B ]-> A_EQ_B)
               [^h00,^h00]->    1;
               [^h0F,^h0F]->    1;
               [^h1E,^hE0]->    0;
               [^h00,^hF0]->    0;
               [^h10,^h11]->    0;
               [^h33,^h33]->    1;
end
```

Figure 8.8 Multilevel equality comparator described using equations

```
module comp36
title 'Very large (36-bit) comparator'

    a35..a0 pin;
    b35..b0 pin;
    A = [a35..a0];
    B = [b35..b0];

    A_EQ_B pin istype 'com';

equations
@CARRY 2;

    A_EQ_B = A == B;

test_vectors([     A      ,      B      ]->[A_EQ_B])
             [^h000000001,^h000000001]->[   1  ];
             [^hFFFFFFFFF,^hFFFFFFFFF]->[   1  ];
             [^hE045C231D,^h358B98AF9]->[   0  ];
             [^h1F76BA8F9,^h98FB5CA23]->[   0  ];
             [^h03B45CA11,^h03B45CA11]->[   1  ];
             [^hEEEEEEEE1,^hEEEEEEEE2]->[   0  ];
             [^hC3C3C3C3C,^hC3C3C3C3C]->[   1  ];
             [^h555555555,^hAAAAAAAAA]->[   0  ];
end
```

Figure 8.9 Chained 36-bit comparator described using @CARRY directive

determine a relative result (greater than or less than), the amount of circuitry required is less obvious. This is because the binary pattern of values to be compared determines the resulting circuit size. If you need to estimate the size of such a comparator, you can count the number of ones and zeros in the binary form of the fixed value. At most, each one in the pattern will require one product term if implemented with positive polarity, while the negative polarity form may require a product term for each zero. The maximum number of product terms required when you have a choice of output polarity is therefore $n/2$, where n is the number of bits in the input set.

Even more troublesome (although more predictable) are comparator circuits that compare two arbitrary sets of inputs to determine which is greater. A magnitude comparator such as this, if implemented in two-level logic, will always require at least 2^n product terms to implement, where n is the number of bits in each input set. Consider, for example, the following ABEL equation:

```
A_GT_B = [a7..a0] > [b7..b0];
```

This harmless looking equation will generate 255 product terms (383 product terms if implemented in a PLD with inverting outputs) and would be extremely wasteful (perhaps impossible) to implement in a programmable device or any other technology. With an understanding of how comparators work, though, we can determine a more efficient solution.

A comparator circuit of this sort is very regular in its construction, and this allows us to design a circuit that operates in a more procedural, or algorithmic, manner. We'll show how the algorithm for comparing two sets of inputs works by demonstrating how it's used to compare two sets of four inputs each. First, the bit positions of each number (*A* and *B*) are labeled from most significant bit to least significant bit as *A3*, *A2*, *A1*, *A0* and *B3*, *B2*, *B1*, *B0*, respectively.

To compare these two inputs sets, we scan the two sets, beginning with the least significant pair of bits (*A0* and *B0*) and perform a comparison operation to each bit pair in sequence. For each bit pair, if the A_n bit is greater than the B_n bit, we know that the value of set *A* is greater than the value of set *B* up to that point. If the two bits are equal, then the previous (less significant) bit pair comparisons must be used to determine the results.

The algorithm for each bit can be expressed in the form of the single-bit comparator circuit with carry-in shown in Figure 8.10. The information about the previous bit in a multiple-bit comparison is provided to the comparator circuit through the *C_IN* signal.

Any number of these single-bit comparators can be chained to create a multiple-bit ripple comparator. Figure 8.11 shows an eight-bit magnitude comparator circuit implemented in this way. This circuit could be implemented in a PLD if a device was used that had sufficient input and output pins available or had the capability for multilevel logic. This representation might be ideal for implementation in an FPGA, since an FPGA is composed of many internal logic modules of limited size.

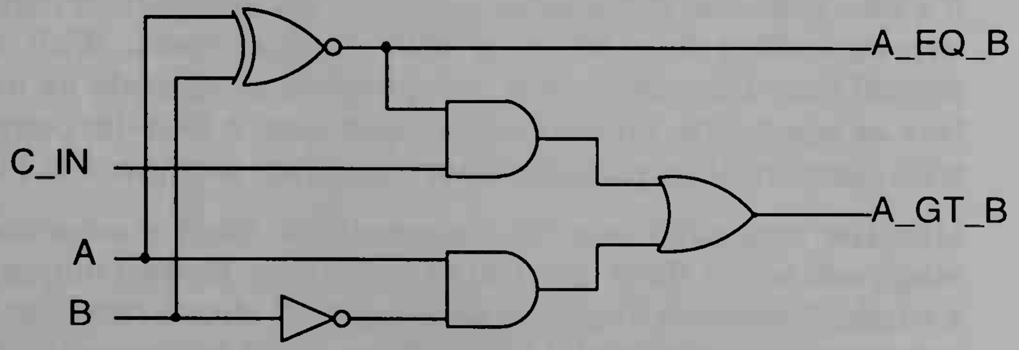


Figure 8.10 Single-bit magnitude comparator with carry-in

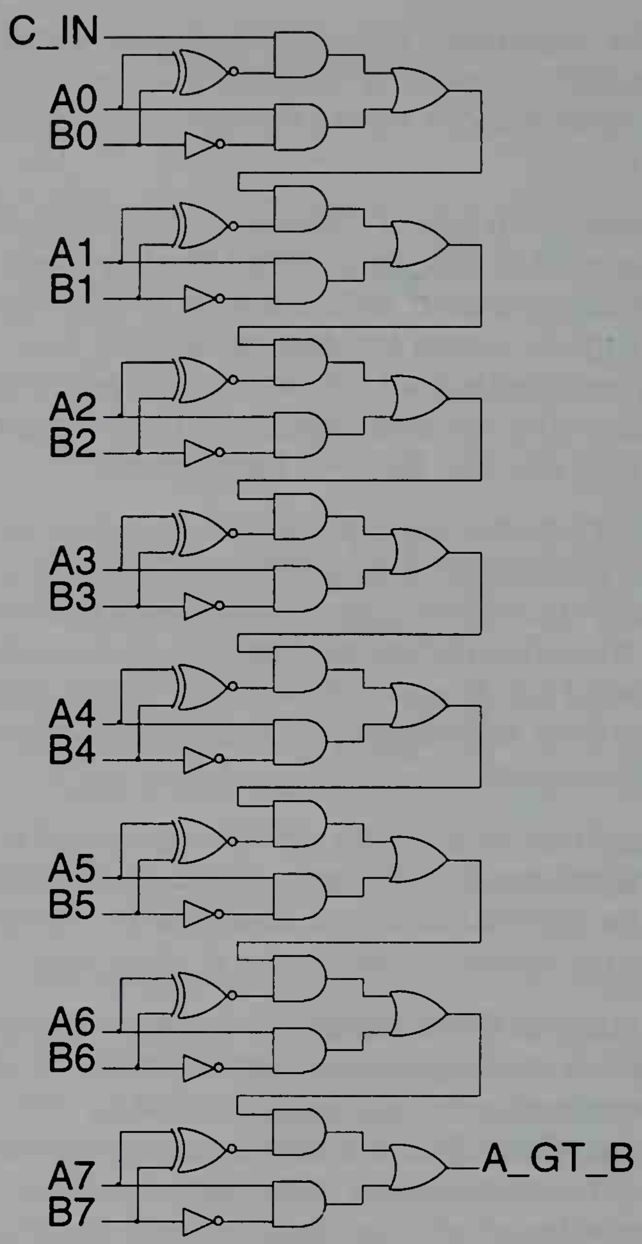


Figure 8.11 Eight-bit magnitude comparator chain

It's also possible to construct an eight-bit magnitude comparator out of two cascading four-bit magnitude comparators. With a properly designed four-bit comparator, it's possible to cascade as many comparators as needed to compare large input sets. A four-bit comparator circuit with carry-in that provides this capability is shown in Figure 8.12.

Like the cascaded one-bit comparators, each succeeding comparator stage will add a fixed amount of delay time to the comparator circuit as a whole. The block diagram in Figure 8.13 shows two four-bit magnitude comparators cascaded to create an eight-bit magnitude comparator. This circuit will operate faster than the ripple comparator shown in

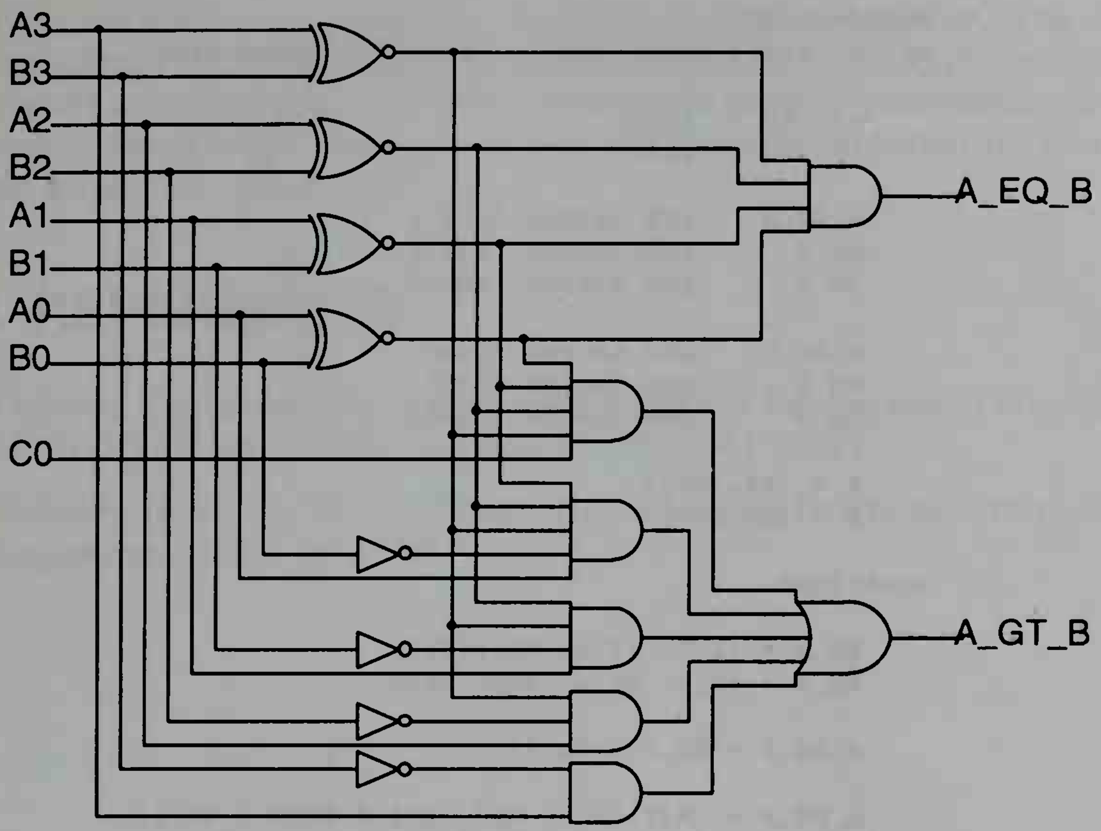


Figure 8.12 Four-bit magnitude comparator with carry-in

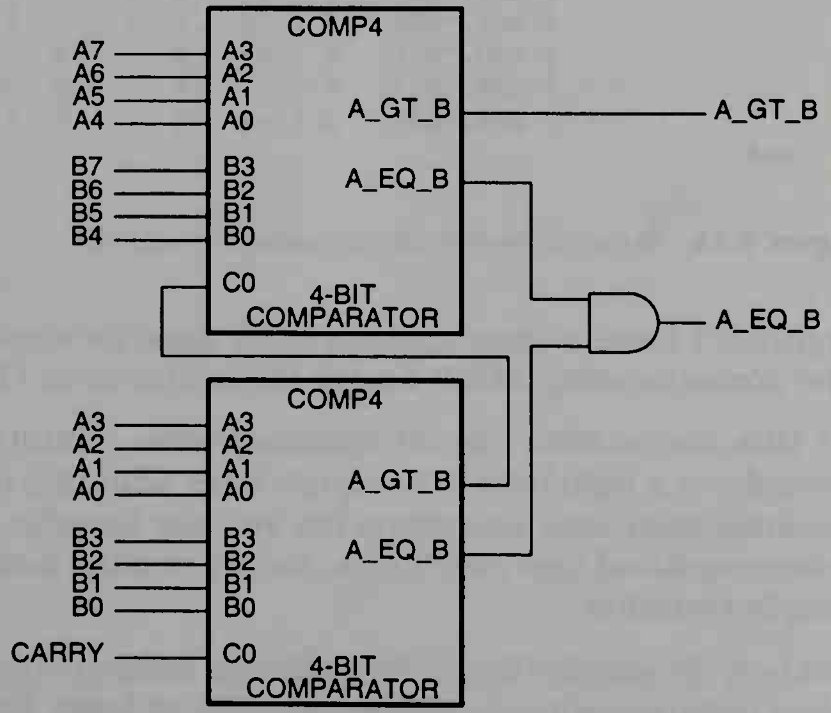


Figure 8.13 Cascading two four-bit comparators to create an eight-bit comparator

```
module mag8
title '8-bit magnitude and equality comparator'

    a7..a0     pin;
    b7..b0     pin;

    A_GT_B     pin istype 'com';
    GT_0       pin istype 'com';
    GT_1       pin istype 'com';

    A_EQ_B     pin istype 'com';
    EQ_1       pin istype 'com';
    EQ_2       pin istype 'com';

    A = [a7..a0];
    B = [b7..b0];

equations

    EQ_2 = (A[7..4] == B[7..4]);
    EQ_1 = (A[3..0] == B[3..0]);

    A_EQ_B = EQ_2 & EQ_1;

    A_GT_B = (A[7..4] > B[7..4]) # EQ_2 & GT_1;
    GT_1 = (A[3..0] > B[3..0]) # EQ_1 & GT_0;

test_vectors ([  A ,   B ,GT_0]->[A_GT_B,A_EQ_B])
              [^h00,^h00,  0 ]->[  0  ,   1  ];
              [^h0F,^h0F,  0 ]->[  0  ,   1  ];
              [^hFE,^hE0,  0 ]->[  1  ,   0  ];
              [^h00,^hF0,  0 ]->[  0  ,   0  ];
              [^h20,^h11,  0 ]->[  1  ,   0  ];
              [^h34,^h33,  0 ]->[  1  ,   0  ];
              [^h55,^h55,  1 ]->[  1  ,   1  ];
end
```

Figure 8.14 Eight-bit magnitude comparator design file

Figure 8.11 and is more appropriately sized for implementation in PLDs. The corresponding ABEL design file is shown in Figure 8.14.

As this comparator circuit demonstrates, circuits that are simple to describe at a high level can require large amounts of logic to implement. Because high-level equations are so easy to write, we tend to forget or misunderstand how much logic can be created from a single, seemingly simple equation.

Not only do simple design descriptions (such as the equation $F = A > B$) often produce unmanageable amounts of logic, but they also may not result in the optimum implementation of a circuit due to design trade-offs that must be made to adapt a design to the restrictive requirements of a particular implementation technology. The implementation of the eight-bit magnitude comparator as a ripple comparator and as a four-level logic circuit (two levels of sum-of-products equations) demonstrates how the constraints of the target architecture can have an impact on the design.

8.3 REFERENCES

Pellerin, David, and Michael Holley, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Wakerly, John F., *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, NJ, 1990.

9 Arithmetic Functions

9.1 PARITY DETECTION

Parity detectors are used to check for errors in circuits that store and process binary data. When a parity scheme is used, the binary data are modified with an error-correcting code that is chosen to guarantee that the number of 1 bits in the code is either always even (*even parity*) or always odd (*odd parity*). In memory circuits, for example, an additional bit (called the parity bit) is appended to the each data word (which is typically eight bits) to create the desired parity. If any one of the bits in the data word is read incorrectly, then the parity will be incorrect and the memory interface circuitry can either generate an error condition or attempt to correct the error.

Detection of odd parity can be accomplished by cascading exclusive-OR gates as shown in Figure 9.1. In this circuit, seven exclusive-OR gates are cascaded to produce the result. If higher speeds are required, the alternative arrangement of XORs shown in Figure 9.2 can be used. In this circuit, XOR gates are arranged in a tree structure. For extremely high-speed circuits, the multiple levels of XOR gates can be replaced by as few as two levels of sum-of-products logic. For an eight-bit parity generator, however, the amount of logic required may be prohibitive.

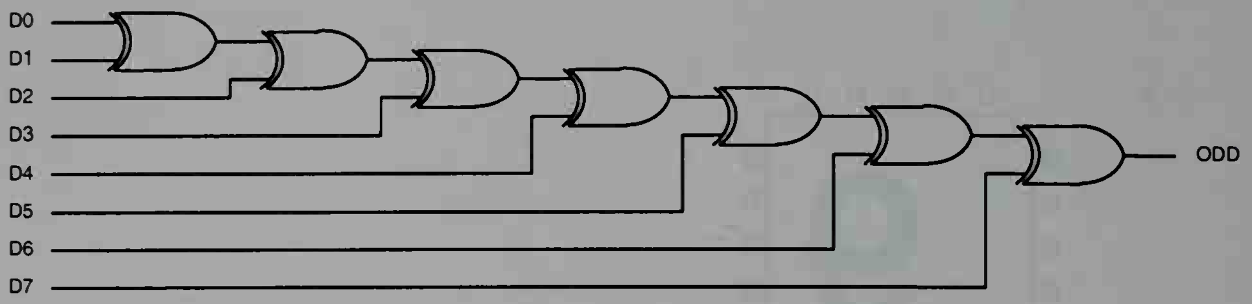


Figure 9.1 Cascading XORs for parity generation

Figures 9.3 and 9.4 show two ways to describe an eight-bit parity circuit using ABEL. In the first version, XOR operations are assigned to combinational nodes that are then fed into other XOR operations, creating the tree structure shown previously. In the second version, some of these combinational nodes have been converted into ABEL constant declarations so that they will be flattened into sum-of-products logic. In addition, the second version of the design does not assume the existence of XOR gates in the target architecture, so the 'xor' attribute is removed from the signal declarations for the outputs. The result is that the second version of the parity generater will require more logic per output to implement, but will have fewer levels of logic and will operate faster. If all the XOR operations were flattened into a single equation for *Odd*, the total number of product terms required would be impractical for all but the fastest circuits.

By experimenting with different combinations of combinational nodes

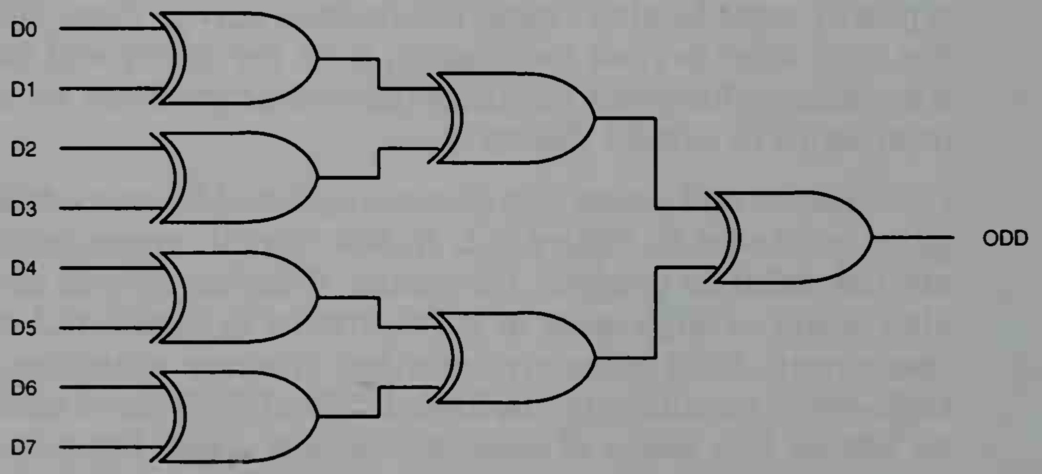


Figure 9.2 Parity generation using a tree of XORs

```
module parity1
title '8-bit parity generator';

    D7..D0    pin;    "Data
    Odd       pin istype 'com,xor';
    X1..X6    node istype 'com,xor';

equations
    X1 = D0 $ D1;
    X2 = D2 $ D3;
    X3 = D4 $ D5;
    X4 = D6 $ D7;
    X5 = X1 $ X2;
    X6 = X3 $ X4;
    Odd = X5 $ X6;

test_vectors([D7,D6,D5,D4,D3,D2,D1,D0]->[Odd])
             [ 0, 1, 0, 1, 0, 0, 1, 1]->[ 0 ];
             [ 1, 1, 0, 0, 1, 0, 1, 1]->[ 1 ];
             [ 0, 1, 1, 0, 1, 1, 0, 0]->[ 0 ];
             [ 1, 1, 1, 0, 1, 1, 0, 0]->[ 1 ];
             [ 1, 1, 1, 1, 1, 1, 1, 1]->[ 0 ];
             [ 0, 0, 0, 0, 0, 0, 0, 0]->[ 0 ];
end
```

Figure 9.3 Eight-bit parity generator design file

```
module parity2
title '8-bit fast parity generator';

    D7..D0     pin;    "Data
    Odd        pin istype 'com';
    X5,X6      node istype 'com';

    X1 = D0 $ D1;
    X2 = D2 $ D3;
    X3 = D4 $ D5;
    X4 = D6 $ D7;

equations
    X5 = X1 $ X2;
    X6 = X3 $ X4;
    Odd = X5 $ X6;

test_vectors([D7,D6,D5,D4,D3,D2,D1,D0]->[Odd])
             [ 0, 1, 0, 1, 0, 0, 1, 1]->[ 0 ];
             [ 1, 1, 0, 0, 1, 0, 1, 1]->[ 1 ];
             [ 0, 1, 1, 0, 1, 1, 0, 0]->[ 0 ];
             [ 1, 1, 1, 0, 1, 1, 0, 0]->[ 1 ];
             [ 1, 1, 1, 1, 1, 1, 1, 1]->[ 0 ];
             [ 0, 0, 0, 0, 0, 0, 0, 0]->[ 0 ];
end
```

Figure 9.4 Design file for a faster eight-bit parity generator

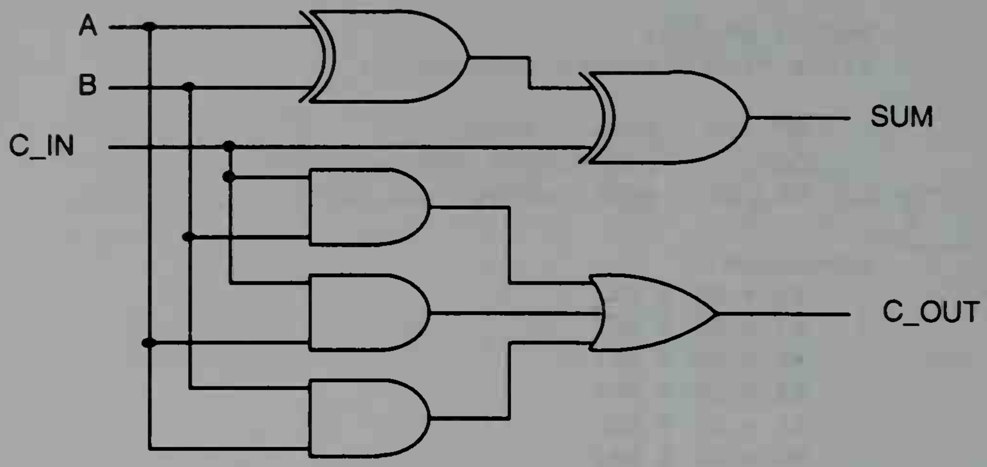


Figure 9.5 Basic adder circuit (full adder)

and constant declarations, it's possible to balance the need for circuit speed (depth of logic) with the size constraints of the target implementation.

9.2 ADDERS

Another situation in which the target architecture must be considered is the construction of an adder. Adders can be constructed in many ways, depending on the width of the adder's operands and the speed of operation required. The simplest full adder accepts two bits and a carry as input and produces a sum and carry result. This circuit is shown in Figure 9.5.

The adder circuit shown can be generalized to an arbitrary number of bits. For any adder circuit, the sum of the add operation consists of two n-bit inputs and one n-bit output, plus a carry. If more than one adder is going to be connected to form an adder chain, each adder in the chain (with the exception of the first) must have a carry-in bit as shown in

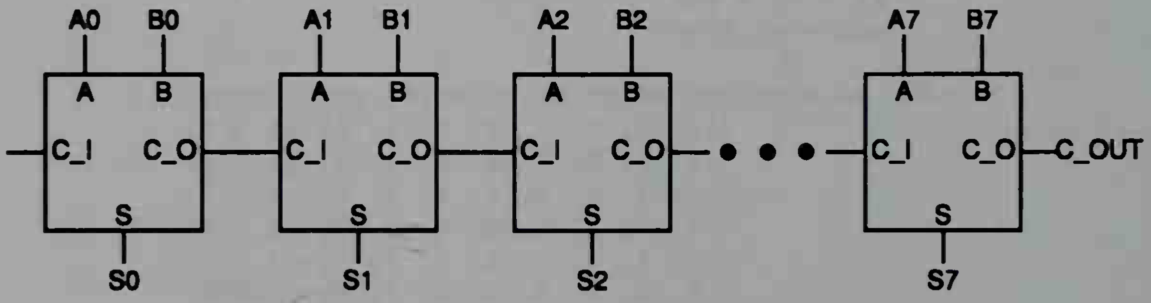


Figure 9.6 Adder chain

```
module add8a
title '8-bit adder constructed of eight 1-bit adders'

    a7..a0   pin;                    "operand 1
    b7..b0   pin;                    "operand 2
    c7..c1   node istype 'com';      "Adder carry bits
    s7..s0   pin istype 'com';      "Sum bits
    c_out    pin istype 'com';      "Carry out

equations

    s0 = a0 $ b0;                    "Half adder for bit 0
    c1 = a0 & b0;

    s1 = a1 $ b1 $ c1;               "Full adder for bit 1
    c2 = a1 & b1 # (a1 $ b1) & c1;

    s2 = a2 $ b2 $ c2;               "Full adder for bit 2
    c3 = a2 & b2 # (a2 $ b2) & c2;

    s3 = a3 $ b3 $ c3;               "Full adder for bit 3
    c4 = a3 & b3 # (a3 $ b3) & c3;

    s4 = a4 $ b4 $ c4;               "Full adder for bit 4
    c5 = a4 & b4 # (a4 $ b4) & c4;

    s5 = a5 $ b5 $ c5;               "Full adder for bit 5
    c6 = a5 & b5 # (a5 $ b5) & c5;

    s6 = a6 $ b6 $ c6;               "Full adder for bit 6
    c7 = a6 & b6 # (a6 $ b6) & c6;

    s7 = a7 $ b7 $ c7;               "Full adder for bit 7
    c_out = a7 & b7 # (a7 $ b7) & c7;

test_vectors([[a7..a0],[b7..b0]]->[[c_out,s7..s0]])
             [    1  ,     1  ]->[       2     ];
             [   24  ,   112  ]->[     136     ];
             [   99  ,    13  ]->[     112     ];
             [   82  ,   109  ]->[     191     ];
             [  100  ,   100  ]->[     200     ];
             [  255  ,   255  ]->[     510     ];  "Test carry
end
```

Figure 9.7 Eight-bit adder chain described with ABEL

Figure 9.6.

Figure 9.7 is an eight-bit adder chain described using ABEL. The least-significant bit in the chain (output *S0*) is a half-adder, so no carry-in is provided for this adder circuit. Bits *S1* through *S7* are described using full adders. The carry signals between adders are represented by signals *C1* through *C7*. A carry-out signal for the entire circuit is provided by the output pin *C_OUT*.

9.3 HIGH-LEVEL ADDER EQUATIONS

In ABEL, adders can be described using high-level operators and set notation. Figure 9.8 describes the same chain of one-bit adders rewritten using high-level add operators and set notation. In each stage of the adder, sets are used to collect and add the two operands (the addend and augend) and the carry-in from the previous bit. In the least

```
module add8b
title '8-bit adder constructed of eight 1-bit adders'

    a7..a0   pin;                  "operand 1
    b7..b0   pin;                  "operand 2
    c7..c1   node istype 'com';    "Adder carry bits
    s7..s0   pin istype 'com';     "Sum bits
    c_out    pin istype 'com';     "Carry out

equations

  [ c1   ,s0] = [.x.,a0] + [.x.,b0] + [ 0 , 0];
  [ c2   ,s1] = [.x.,a1] + [.x.,b1] + [ 0 ,c1];
  [ c3   ,s2] = [.x.,a2] + [.x.,b2] + [ 0 ,c2];
  [ c4   ,s3] = [.x.,a3] + [.x.,b3] + [ 0 ,c3];
  [ c5   ,s4] = [.x.,a4] + [.x.,b4] + [ 0 ,c4];
  [ c6   ,s5] = [.x.,a5] + [.x.,b5] + [ 0 ,c5];
  [ c7   ,s6] = [.x.,a6] + [.x.,b6] + [ 0 ,c6];
  [c_out,s7] = [.x.,a7] + [.x.,b7] + [ 0 ,c7];

test_vectors([[a7..a0],[b7..b0]]->[[c_out,s7..s0]])
             [    1  ,     1  ]->[       2     ];
             [   24  ,   112  ]->[     136     ];
             [   99  ,    13  ]->[     112     ];
             [  100  ,   100  ]->[     200     ];
             [  255  ,   255  ]->[     510     ];  "Test carry

end
```

Figure 9.8 Using arithmetic equations to describe an adder chain

significant bit of the adder, the carry-in is specified as zero. No-connect special constants are used to pad all sets to a common width of two bits. This is necessary because the ABEL compiler enforces set width matching. In the sets containing the carry-in bits, zeros are provided for padding instead of no-connects.

The advantage of using sets and high-level operators will become more obvious when we discuss adder chains constructed of wider adder circuits.

9.4 FAST CARRY (LOOK-AHEAD) ADDERS

Like the comparator circuits described in Chapter 7, adders can be tailored to the constraints of a specific implementation. The ripple adders presented in Section 9.3 are compact, but may be too slow for many applications. The look-ahead carry strategy can be employed to balance speed with size. In adders, however, the size of the circuitry generated from a full carry look-ahead implementation can be astonishing. For example, the ABEL equation

```
[s6..s0] = [.x.,a5..a0] + [.x.,b5..b0];
```

translates to over 700 product terms. The practical limit for such adders is four or five bits.

9.5 CHAINED ADDERS

If you need to operate on more than four or five bits, you will need to chain together two or more smaller adders to accomplish the desired result. Figure 9.9 shows a 16-bit adder described as a chain of eight two-bit adders. In this design, each two-bit adder is described with a high-level equation that adds two bits of data and one bit of carry from the preceding adder.

Using @CARRY to Describe Chained Adders

The @CARRY directive described in Chapter 8 can be used to create large adder chains. When @CARRY is specified, carry logic is preserved between successive adders in the chain, resulting in a multilevel implementation similar to the logic generated from the previous exam-

```
module addchain
title '16-bit chained adder'

    a15..a0     pin;                        "Operand 1
    b15..b0     pin;                        "Operand 2
    s15..s0     pin istype 'com';           "Sum
    c7..c1      node istype 'com';          "Adder carry bits
    c_out       pin istype 'com';           "Carry out

equations

 [   c1,s1,s0] = [.x., a1, a0] + [.x., b1, b0] + [0,0, 0];
 [   c2,s3,s2] = [.x., a3, a2] + [.x., b3, b2] + [0,0,c1];
 [   c3,s3,s2] = [.x., a5, a4] + [.x., b5, b4] + [0,0,c2];
 [   c4,s3,s2] = [.x., a7, a6] + [.x., b7, b6] + [0,0,c3];
 [   c5,s3,s2] = [.x., a9, a8] + [.x., b9, b8] + [0,0,c4];
 [   c6,s3,s2] = [.x.,a11,a10] + [.x.,b11,b10] + [0,0,c5];
 [   c7,s3,s2] = [.x.,a13,a12] + [.x.,b13,b12] + [0,0,c6];
 [c_out,s3,s2] = [.x.,a15,a14] + [.x.,b15,b14] + [0,0,c7];

test_vectors([[a15..a0],[b15..b0]]->[c_out,s15..s0])
             [      1  ,      1  ]->[        2   ];
             [  17245  ,   3563  ]->[    20808   ];
             [  14234  ,  14234  ]->[    28468   ];
             [    255  ,      0  ]->[      255   ];
             [  32767  ,      1  ]->[    32768   ];
end
```

Figure 9.9 Sixteen-bit adder chain composed of eight two-bit adders

```
module addchain
title '16-bit chained adder using @CARRY'

    a15..a0     pin;                        "Operand 1
    b15..b0     pin;                        "Operand 2
    s15..s0     pin istype 'com'            "Sum
    c_out       pin istype 'com';           "Carry out

equations

@carry 2;   "Generate adder chain
  [c_out,s15..s0] = [.x.,a15..a0] + [.x.,b15..b0];

test_vectors([[a15..a0],[b15..b0]]->[c_out,s15..s0])
             [      1  ,      1  ]->[        2   ];
             [  17245  ,   3563  ]->[    20808   ];
             [    255  ,      0  ]->[      255   ];
             [  32767  ,      1  ]->[    32768   ];
end
```

Figure 9.10 Using the @CARRY directive to create an adder chain

ple. Figure 9.10 is an ABEL design file that describes the same 16-bit adder using a single arithmetic equation and the @CARRY directive.

9.6 SUBTRACTERS

Digital subtraction of two numbers can be performed by adding the ones-complement of the second number (the subtrahend) to the first number (the minuend). If this method is implemented directly in hardware, a chain of full adders is required. A more efficienent method is more commonly used. In this method, 1-bit circuits known as full subtracters are chained. Each of these circuits accepts three single-bit inputs (the minuend, subtrahend, and a borrow flag) and produces a two-bit output (the difference and borrow-out flag).

A full subtracter circuit is similar to a full adder circuit. A chained subtracter, like a chained adder, is constructed of single-bit units that have carry-in and carry-out signals. In a subtracter, the carry signal represents the borrow operation necessary for subtraction.

The Boolean equations that represent a full subtracter circuit are

$$\begin{aligned} D &= M \ \$ \ S \ \$ \ B_{in} \\ B_{out} &= !M \,\&\, S \ \# \ !M \,\&\, B_{in} \ \# \ S \,\&\, B_{in} \end{aligned}$$

In these equations, *M* and *S* represent the minuend and subtrahend, *D* represents the single-bit difference, B_{in} represents the borrow-in from the less significant subtracter in the chain, and B_{out} represents the borrow-out.

Figure 9.11 is an eight-bit subtracter circuit described using ABEL. This circuit accepts two eight-bit values and subtracts the second from the first. No borrow-in is provided for this circuit, so the subtracter for bit zero (output *d0*) is a half-subtracter and has no borrow-in. The most significant bit (output *d7*) produces a borrow-out that is used to indicate that the result of subtraction is negative.

9.7 FIBONACCI SEQUENCE GENERATOR

A Fibonacci sequence is a sequence of numbers, beginning with 1, 1, 2, 3... in which every number in the sequence is the sum of the previous two numbers. To construct a circuit that generates an *n*-bit Fibonacci sequence, two *n*-bit registers *A* and *B* are required to store the last two values of the sequence and add them to produce the next value. To initialize the circuit, the *A* and *B* registers must be loaded with values

```
module sub8a
title '8-bit subtracter constructed of eight 1-bit subtracters'

    a7..a0    pin;                  "operand 1 (minuend)
    s7..s0    pin;                  "operand 2 (subtrahend)
    b7..b1    node istype 'com';    "Borrow bits
    d7..d0    pin istype 'com';     "Difference bits
    neg       pin istype 'com';     "Borrow-out (negative flag)

equations
    d0 = a0 $ s0;                  "Half subtracter for bit 0
    b1 = !a0 & s0;
    d1 = a1 $ s1 $ b1;             "Full subtracter for bit 1
    b2 = !a1 & s1 # !a1 & b1 # s1 & b1;
    d2 = a2 $ s2 $ b2;             "Full subtracter for bit 2
    b3 = !a2 & s2 # !a2 & b2 # s2 & b2;
    d3 = a3 $ s3 $ b3;             "Full subtracter for bit 3
    b4 = !a3 & s3 # !a3 & b3 # s3 & b3;
    d4 = a4 $ s4 $ b4;             "Full subtracter for bit 4
    b5 = !a4 & s4 # !a4 & b4 # s4 & b4;
    d5 = a5 $ s5 $ b5;             "Full subtracter for bit 5
    b6 = !a5 & s5 # !a5 & b5 # s5 & b5;
    d6 = a6 $ s6 $ b6;             "Full subtracter for bit 6
    b7 = !a6 & s6 # !a6 & b6 # s6 & b6;
    d7 = a7 $ s7 $ b7;             "Full subtracter for bit 7
    neg = !a7 & s7 # !a7 & b7 # s7 & b7;

test_vectors([[a7..a0],[s7..s0]]->[[d7..d0],neg])
            [    1  ,     1  ]->[    0   , 0 ];
            [  200  ,    67  ]->[  133   , 0 ];
            [   18  ,     4  ]->[   14   , 0 ];
            [  145  ,    15  ]->[  130   , 0 ];
            [    1  ,     2  ]->[  255   , 1 ];
            [   55  ,   100  ]->[  211   , 1 ];

end
```

Figure 9.11 Eight-bit subtractor chain design description

of 0 and 1, respectively. Subsequent cycles of the circuit must move the calculated next value into the *B* register while moving the value stored in the *B* register to the *A* register. In this implementation, the *A* and *B* registers form a two-deep first-in first-out (FIFO) stack.

The ABEL source file shown in Figure 9.12 describes the Fibonacci generator circuit. The eight-bit registers are declared as signals *A7* through *A0* and *B7* through *B0*, respectively, and the signals that carry the calculated next value are declared as signals *S7* through *S0*. Since an eight-bit adder circuit would be impractical to implement as a full look-ahead circuit, carry bits are declared as signals *Cout* and *C7* through *C1*. No carry-in is required for bit zero of the adder circuit. A carry-out from the MSB of the adder is required, however, to detect that

```
module fib1
title 'Fibonacci sequence generator'
    Clk,Clr      pin;
    A7..A0       node istype 'reg,buffer';
    B7..B0       pin istype 'reg,buffer';
    S7..S0       pin istype 'com';
    Cout,C7..C1  node istype 'com';
    Zero         node istype 'com';
    Restart      node istype 'reg';
    A            = [A7..A0];
    B            = [B7..B0];
    S            = [S7..S0];
    Carry        = [C7..C1,0];
    CarryOut     = [Cout,C7..C1];
    CarryIn      = [C7..C1,0];

equations
    S        = A $ B $ CarryIn;             "Chained adder
    CarryOut = B & A # (B # A) & CarryIn;   "Carry for adder
    Zero     = (S == 0);
    When (Zero) Then A := 1;
                Else A := B;
    B        := S;
    Restart  := Cout;                       "Restart on overflow

    [A,B].ar    = Clr # Restart;
    [A,B].clk   = Clk;
    Restart.clk = Clk;

Test_Vectors ([Clk,Clr] -> [  A,  B,  S])
              [.C., 1 ] -> [  0,  0,  0];
              [.C., 0 ] -> [  1,  0,  1];
              [.C., 0 ] -> [  0,  1,  1];
              [.C., 0 ] -> [  1,  1,  2];
              [.C., 0 ] -> [  1,  2,  3];
              [.C., 0 ] -> [  2,  3,  5];
              [.C., 0 ] -> [  3,  5,  8];
              [.C., 0 ] -> [  5,  8, 13];
              [.C., 0 ] -> [  8, 13, 21];
              [.C., 0 ] -> [ 13, 21, 34];
              [.C., 0 ] -> [ 21, 34, 55];
              [.C., 0 ] -> [ 34, 55, 89];
              [.C., 0 ] -> [ 55, 89,144];
              [.C., 0 ] -> [ 89,144,233];
              [.C., 0 ] -> [144,233,.X.];
              [.C., 0 ] -> [  0,  0,  0];
              [.C., 0 ] -> [  0,  0,  0];
              [.C., 0 ] -> [  1,  0,  1];
              [.C., 0 ] -> [  0,  1,  1];
              [.C., 0 ] -> [  1,  1,  2];
end
```

Figure 9.12 ABEL design description for Fibonacci generator

the sequence is complete. This final carry-out is designated as signal *Cout*. The adder portion of this design is described using set notation to simplify the description of the chained adder. Separate equations are written for the XOR and carry generation functions required for the adder. Zeros are used to pad the *Carry* and *CarryIn* sets to the proper width and orientation for the operation. Using sets in this ways means that the logic description portion of this design is completely independent of set width. Changing this design to a larger sequence generator (16 bits, for example) is a simple matter of adding the new signal declarations and modifying the set declarations.

9.8 BINARY MULTIPLICATION

There are many ways to implement multipliers or other advanced arithmetic circuits. Fast combinational multiplier circuits can be constructed out of arrays of combinational adders, or slower (but more compact) serial multipliers can be constructed using state machine design methods. The multiplier that we will present is a sequential circuit that performs multiplication using a modified form of the classic shift-and-add (pencil and paper) method.

8 X 8 Sequential Multiplier

Binary multiplication can be performed using sequential circuits that perform the same functions as would be performed using pencil and paper: successive additions and shifts. The traditional pencil and paper

```
Multiplicand:        27             11011
Multiplier:          19             10011
                                 -------------
                    243             11011
                    27             11011
                   -----          00000
Product:            513          00000
                                11011
                                 -------------
            Binary Product:     1000000001
```

Figure 9.13 Decimal and binary multiplication of two five-bit values

```
Multiplicand:          11011
Multiplier:            10011  ─┐       Accumulator
                 ─────────     └─→  10011 00000
                       11011        1001 011011
                      11011         100 1010001
                     00000          10 01010001
                    00000           1 001010001
                   11011             1000000001
                                    ─────────────
Binary Product:                      1000000001
```

Figure 9.14 Alternative method of binary multiplication

method is demonstrated for two five-bit operands in Figure 9.13. This example shows how two five-bit numbers (27 and 19) are multiplied to create a 10-bit result. The steps are shown in both decimal and binary notations.

Binary multiplication is simpler than decimal multiplication because there is no need to think about carry operations during the creation of the shifted multiplicands. During the operation of the multiplier, each bit of the multiplier is tested, starting from the rightmost bit. If the bit is 1, then the multiplicand is shifted the appropriate number of bits and added to a partial product stored in a result accumulator. If the multiplier's bit is 0, then no addition is performed. This algorithm is relatively easy to implement in circuitry, but is not the most efficient method possible for sequential binary multiplication. Instead, a different method is used.

In this method, the result accumulator is used not only for storage of the partial product, but for storage of the multiplier as well. This is important because, in a typical application, the two operands (the multiplier and multiplicand) must be latched into the circuit before the multiplication operation begins. If one operarand (in this case *A*) is stored in the high-order bits of the accumulator, then only one eight-bit register is needed to store input data.

The use of the accumulator as an input register for one of the operands is possible because the multiplier is only needed one bit at a time and is processed from right to left as the partial product grows. The partial product stored in the accumulator is initially only eight bits in length, and grows by one bit with each stage of the multiply operation. The example diagrammed in Figure 9.14 (the same five-bit example presented earlier) shows how the partial product is constructed in the

```
module mult8x8
title '8-bit shift-and-add binary multiplier.'
/////////////////////////////////////////////////////////
//This 8-bit multiplier is a 9-state machine.  The
//machine holds the current result value in the P
//product accumulator until the start input goes high.
//It then loads the multiplicand (B) into the
//MULTIPLICAND register (M) and the multiplier (A)
//into the left-most 8 bits of the partial product
//accumulator.  It then sequences through the eight
//shift-and add states, adding shifted multiplicands
//to the partial product, gradually replacing the
//multiplier.  Upon completion of the operation, the
//machine returns to the initial state with the status
//flag set.

    X = .x.; C = .c.;

    start   pin;                          "Begin multiply
    clock   pin;                          "Global clock
    reset   pin;                          "Global reset
    a7..a0  pin; A = [a7..a0];            "Operand A
    b7..b0  pin; B = [b7..b0];            "Operand B

    m7..m0  pin istype 'reg,buffer';
    M       = [m7..m0];
    MSHFT   = [X,m7..m0,0,0,0,0,0,0,0,0]; "Shift register
    p15..p0 pin istype 'reg,buffer';
    P       = [p15..p0];                    "Accumulator

    q3..q0  pin istype 'reg,buffer';
    SREG    = [q3..q0];                   "State register
    IDLE    = [0,0,0,0];                  "States are
    STATE1  = [0,0,0,1];                  "numbered for
    STATE2  = [0,0,1,1];                  "minimum
    STATE3  = [0,0,1,0];                  "bit change.
    STATE4  = [0,1,1,0];
    STATE5  = [0,1,0,0];
    STATE6  = [1,1,0,0];
    STATE7  = [1,0,0,0];
    STATE8  = [1,0,0,1];

equations

    [SREG,M,P].clk = clock;
    SREG.ar        = reset;

    when (start) then
        MSHFT[15..8] :=  A;           "Load A if start
    else
        MSHFT[15..8] := MSHFT[15..8];   "Else hold
```

Figure 9.15 Eight-bit sequential multiplier design description (part 1 of 3)

```
@CARRY 2;
state_diagram SREG
state IDLE:
    if (start) then STATE1 with P[15..8] := B[7..0];
    else            IDLE with P[15..0] := P[15..0];

state STATE1:
    P[15..9] := P[15..9];    "Preserve 7 bits of B in P
    P[7..0]  := (P[8] == 1) & MSHFT[15..8];
    goto STATE2;

state STATE2:
    P[15..10] := P[15..10]; "Preserve 6 bits of B in P
    P[9..0] := (P[9] == 0) & [0,p8..p0]
            # (P[9] == 1) & ([X,p8..p0] + MSHFT[16..7]);
    goto STATE3;

state STATE3:
    P[15..11] := P[15..11]; "Preserve 5 bits of B in P
    P[10..0] := (P[10] == 0) & [0,p9..p0]
             # (P[10] == 1) & ([X,p9..p0] + MSHFT[16..6]);
    goto STATE4;

state STATE4:
    P[15..12] := P[15..12]; "Preserve 4 bits of B in P
    P[11..0] := (P[11] == 0) & [0,p10..p0]
             # (P[11] == 1) & ([X,p10..p0] + MSHFT[16..5]);
    goto STATE5;

state STATE5:
    P[15..13] := P[15..13]; "Preserve 3 bits of B in P
    P[12..0] := (P[12] == 0) & [0,p11..p0]
             # (P[12] == 1) & ([X,p11..p0] + MSHFT[16..4]);
    goto STATE6;

state STATE6:
    P[15..14] := P[15..14]; "Preserve 2 bits of B in P
    P[13..0] := (P[13] == 0) & [0,p12..p0]
             # (P[13] == 1) & ([X,p12..p0] + MSHFT[16..3]);
    goto STATE7;

state STATE7:
    P[15] := P[15];  "Preserve 1 bit of B in P
    P[14..0] := (P[14] == 0) & [0,p13..p0]
             # (P[14] == 1) & ([X,p13..p0] + MSHFT[16..2]);
    goto STATE8;

state STATE8:
    P[15..0] := (P[15] == 0) & [0,p14..p0]
             # (P[15] == 1) & ([X,p14..p0] + MSHFT[16..1]);
    goto IDLE;
```

Figure 9.16 Eight-bit sequential multiplier (part 2 of 3)

```
"Test vector macro...

declarations

  test_mult     macro (op1,op2,result) {
    test_vectors
    'A=?op1, B=?op2, P=?result'
       ([clock,reset,start,  A ,  B ]->[SREG  ,    P  ])
        [  C  ,  0  ,  1   ,?op1,?op2]->[STATE1,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE2,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE3,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE4,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE5,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE6,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE7,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[STATE8,    X  ];
        [  C  ,  0  ,  0   ,  X ,  X ]->[IDLE  ,?result];
  }

test_vectors([reset]->[SREG ,    P  ])
             [  1  ]->[IDLE ,    0  ];     "Global reset

test_mult(^b11011,^b10011,^b1000000001);
test_mult(2,3,6);
test_mult(10,10,100);
test_mult(9,9,81);
test_mult(7,6,42);
test_mult(11,11,121);
test_mult(4,200,800);
test_mult(170,85,14450);
test_mult(100,179,17900);
test_mult(127,188,23876);
test_mult(199,0,0);
test_mult(0,131,0);
test_mult(255,255,65025);
test_mult(0,0,0);

end
```

Figure 9.17 Eight-bit sequential multiplier (part 3 of 3)

accumulator from right to left, replacing the five-bit multiplier value one bit at a time.

Figure 9.15 through 9.17 show how this binary multiplier can be described in ABEL. The design accepts two eight-bit inputs *A* and *B* and performs a multiplication on them using the method just described. The design is written as a nine-state state machine. In the first state, *IDLE*, the machine waits until the *start* input is asserted. It then transitions to state *STATE1* to begin the multiply sequence. In each of the eight multiply states, one bit of the multiplier (*B*) is tested to determine if the multiplicand (*A*) should be shifted and added into the partial product accumulator. The shift and add functions are described using set indexing. To allow a simple shift operation to be performed using set indexing, the multiplicand register (*M*) is padded to the right with eight zeros in a set named *MSHFT*, as follows:

```
MSHFT  = [X,m7..m0,0,0,0,0,0,0,0,0]; "Shift register
```

The shift register is also padded to the left by one no-connect so that set widths for the last adder (in *STATE8*) match and carries are properly preserved.

Using these set declarations and set indexing operations, the actual shift and add operations for each state can be simplified to a few lines of high-level equations. The shift and add function for *STATE2*, for example, is

```
P[9..0] := (P[9] == 0) & [0,p8..p0]
         # (P[9] == 1) & ([X,p8..p0] + MSHFT[16..7]);
```

This equation simply tests the second bit of the multipler to determine whether the multiplicand should be shifted and added to the partial product. If the bit is 1, then the set range specified for *MSHFT* will cause the correctly shifted multiplicand to be added to the least-significant 10 bits of the partial product without affecting the remaining (higher) bits of the accumulator (where the multiplier itself is stored.)

The test vectors for this design are written using a macro that generates the nine test vectors required to exercize the multiplier for one operation. With this macro, adding a test case becomes a simple matter of adding one line to the source file.

9.9 REFERENCES

Breeding, Kenneth J., *Digital Design Fundamentals*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Pellerin, David, and Michael Holley, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Wakerly, John F., *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, NJ, 1990.

10

Code Converters

A wide variety of codes are used for the storage and transmission of data in digital systems. Because different systems, or even differents parts of the same system, may use a different code for the same information, code converters are needed to perform data translation. Most code converters are combinational circuits that perform a decode function. Parallel code converters perform a straightforward decode function to convert data, while serial code converters (such as serial-to-parallel converters or checksum generators) perform translation of data that is time dependent.

10.1 BCD TO SEVEN-SEGMENT DECODER

The BCD to seven-segment display decoder is a widely used example of a simple combinational decoding circuit. This circuit accepts a four-bit BCD representation of a number and converts it to the correct code to drive a seven-segment display. This type of decoder uses a straightforward identity comparison (as described in Chapter 8) for each decoder output.

ABEL's truth table syntax simplifies the description of circuits such as this one that don't follow any predictable pattern or order. A truth table, as described in Chapter 5, consists of a header and one or more truth table entries that define all or part of the decoder function. The ABEL design file shown in Figure 10.1 uses a truth table to define the function of the seven-segment display decoder. The truth table is simplified by

```
module BCD7
title 'BCD to 7-segment display driver'

"Segments       -a-
"              f|   |b
"               -g-
"              e|   |c
"               -d-

    D3,D2,D1,D0     pin;                "BCD input
    a,b,c,d,e,f,g   pin istype 'com';  "Segment outputs
    OE              pin;                "Output enable

    BCD     = [D3..D0];
    ON,OFF  = 0,1;                      "Inverted sense

equations

    LED.oe = !OE;          "Define output enable

truth_table(BCD->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
             0 ->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
             1 ->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
             2 ->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
             3 ->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
             4 ->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
             5 ->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
             6 ->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
             7 ->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
             8 ->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
             9 ->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];

test_vectors([OE,BCD]->[  a ,  b ,  c ,  d ,  e ,  f ,  g ])
             [ 0, 0 ]->[ OFF, OFF, OFF, OFF, OFF, OFF,  ON];
             [ 0, 1 ]->[  ON, OFF, OFF,  ON,  ON,  ON,  ON];
             [ 0, 2 ]->[ OFF, OFF,  ON, OFF, OFF,  ON, OFF];
             [ 0, 3 ]->[ OFF, OFF, OFF, OFF,  ON,  ON, OFF];
             [ 0, 4 ]->[  ON, OFF, OFF,  ON,  ON, OFF, OFF];
             [ 0, 5 ]->[ OFF,  ON, OFF, OFF,  ON, OFF, OFF];
             [ 0, 6 ]->[ OFF,  ON, OFF, OFF, OFF, OFF, OFF];
             [ 0, 7 ]->[ OFF, OFF, OFF,  ON,  ON,  ON,  ON];
             [ 0, 8 ]->[ OFF, OFF, OFF, OFF, OFF, OFF, OFF];
             [ 0, 9 ]->[ OFF, OFF, OFF, OFF,  ON, OFF, OFF];
             [ 1, 5 ]->[ .z., .z., .z., .z., .z., .z., .z.];

end
```

Figure 10.1 BCD to seven-segment display driver design file

the use of set notation; the four BCD inputs are grouped into a set so that actual decimal values can be specified in the truth table and in the test vectors that follow.

This truth table is an example of an incompletely specified function. Since the values of ten through fifteen are unused in BCD, these values are don't-cares in the decoder circuit. This is implied by their ommission from the truth table. The 'dc' attribute applied to the seven decoder outputs (*a* through *g*) indicates to the compiler that the missing six conditions can be assigned values of either one or zero to improve optimization of the circuit. ABEL provides three attributes to control the processing of don't-cares: 'dc', 'pos' and 'neg'. These attributes can be used to force unspecified inputs values to be decoded as zero (istype 'pos'), one (istype 'neg'), or don't-care (istype 'dc'). In a display decoder such as this, the seven outputs are true don't-cares when any of the six unspecified input combinations are encountered, so 'dc' optimization is appropriate.

10.2 GRAY-CODE CONVERTERS

Gray codes are used whenever it is important to have only a single bit change between the sequential values of the code. In Chapter 4 we described how gray code can be used to avoid hazard situations in asynchronous state machines. Gray codes are a class of what are known as reflected codes, because for any n-bit code the second 2^{n-1} codes can be derived by reversing the order of the first 2^{n-1} codes and replacing the leading zero with a leading one. Figure 10.2 lists the binary and gray codes for the values 0 through 15.

It's also possible to generate an n-bit gray code value directly from the equivalent n-bit binary representation of a number. To convert binary codes to gray codes, the following method is used. With the bits numbered 0 to $n - 1$ (from right to left): each bit i of the gray-coded value is obtained by an exclusive-OR of bits i and $i + 1$ in the binary-coded value (assuming bit n to be 0). Figure 10.3 is an ABEL design that uses this method to convert 10-bit binary values to gray code.

10.3 CRC GENERATORS

Cyclic redundancy check (CRC) is used for error detection in digital data transmission and storage systems. CRC can be generated either serially or in parallel, depending on the requirements of the circuit and nature of the data. CRC involves the generation of check bits from the data

Decimal value	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Figure 10.2 Binary and gray codes

```
module graycode
title 'Convert 10-bit binary to gray code'

    B9..B0  pin;               "Binary inputs
    G9..G0  pin istype 'com';  "Gray code outputs

equations

    G0 = B0 $ B1;
    G1 = B1 $ B2;
    G2 = B2 $ B3;
    G3 = B3 $ B4;
    G4 = B4 $ B5;
    G5 = B5 $ B6;
    G6 = B6 $ B7;
    G7 = B7 $ B8;
    G8 = B8 $ B9;
    G9 = B9;

test_vectors(   [ B9..B0 ] ->   [ G7..G0 ])
               ^b0000000000 -> ^b0000000000;
               ^b0000000001 -> ^b0000000001;
               ^b0010011010 -> ^b0011010111;
               ^b0111001011 -> ^b0100101110;
               ^b1110101011 -> ^b1001111110;
               ^b1111111111 -> ^b1000000000;
end
```

Figure 10.3 Ten-bit gray code converter

stream and the inclusion of the check bits with the transmitted data. When the data are received, the check bits are regenerated and compared against those transmitted. When the check bits do not match, an error is indicated or a retransmission of data is requested.

All CRC generation schemes can be expressed in the form of a mathematical function known as the generator function or generator polynomial. A number of standard generator functions are in use. A few are listed in Figure 10.4.

CRC Generation Standard	CRC Generator Function
CRC-CCITT	$X^{16} + X^{12} + X^5 + 1$
CRC-CCITT reverse	$X^{16} + X^{11} + X^4 + 1$
CRC-16	$X^{16} + X^{15} + X^{12} + 1$
CRC-16 reverse	$X^{16} + X^{14} + X + 1$ $X^{16} + X^{15} + X^{13} + X^7 + X^4 + X^2 + X + 1$
CRC-12	$X^{12} + X^{11} + X^3 + X^2 + X + 1$
LRC-8	$X^8 + 1$ $X^8 + X^7 + X^5 + X^4 + X + 1$

Figure 10.4 Standard CRC generation functions

Serial CRC Generation

To generate CRC in a serial data stream, it's necessary to use a shift register to capture and process some number of bits *n*, where *n* is the number of CRC bits desired.

The type of shift register used in CRC generation is known as a linear feedback shift register (LFSR). An LFSR captures and processes some number of bits from the data stream, leaving the calculated CRC check bits in the shift register after the appropriate number of data bits has been read. With each shift of the LFSR, a logic function (typically an XOR) is performed on one or more of the bits in the shift register and fed back to other registers. Many possible XOR functions can be used, depending on the types of errors that are expected in the data stream. Figure 10.5 shows how a 3-bit LFSR can be used to generate CRC.

This 3-bit CRC generator XORs each incoming data bit with the fed-back contents of the last register in the LFSR before storing the bit in the first register. The previous contents of the first register are also XORed with

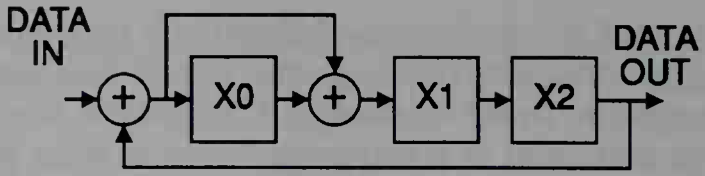


Figure 10.5 Three-bit CRC using linear feedback shift register (LFSR)

the incoming data bit before being stored in the second register. The logic for this CRC generator can be expressed with the equations

```
X0 := Din $ X2;
X1 := Din $ X0;
X2 := X1;
```

This simple CRC generator can be described with the following generator function:

$$G(X) = X^3 + X^1 + X^0$$

The CRC-CCITT Standard

The CRC-CCITT standard was developed for the IBM Synchronous Data Link Control (SDLC) communation protocol and is based on the following generator function:

$$G(X) = X^{16} + X^{12} + X^5 + 1$$

Figure 10.6 shows how the CRC-CCITT is generated. The bits are numbered from 0 to 15 in the order that they are received (or from least significant to most significant in the case of parallel data). Each bitwise calculation results in a new bit being shifted into the 0 bit with exclusive-OR operations performed for bits 5, 12 and 16. Each bit position from position 8 to position 15 has a corresponding intermediate value that is the partial checksum for that bit. These intermediate values are named *I* through *P*.

The equations required to implement this generation scheme are

```
I = X15 $ D7
J = X14 $ D6
K = X13 $ D5
L = X12 $ D4
M = X11 $ D3 $ I
N = X10 $ D2 $ J
O = X9  $ D1 $ K
P = X8  $ D0 $ L
```

			XOR							XOR					XOR
X15	X14	X13	X12	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	X0
X14	X13	X12	X11$I	X10	X9	X8	X7	X6	X5	X4$I	X3	X2	X1	X0	I=X15$D7
X13	X12	X11$I	X10$J	X9	X8	X7	X6	X5	X4$I	X3$J	X2	X1	X0	I	J=X14$D6
X12	X11$I	X10$J	X9$K	X8	X7	X6	X5	X4$I	X3$J	X2$K	X1	X0	I	J	K=X13$D5
X11$I	X10$J	X9$K	X8$L	X7	X6	X5	X4$I	X3$J	X2$K	X1$L	X0	I	J	K	L=X12$D4
X10$J	X9$K	X8$L	X7$M	X6	X5	X4$I	X3$J	X2$K	X1$L	X0$M	I	J	K	L	M=X11$D3$I
X9$K	X8$L	X7$M	X6$N	X5	X4$I	X3$J	X2$K	X1$L	X0$M	I$N	J	K	L	M	N=X10$D2$J
X8$L	X7$M	X6$N	X5$O	X4$I	X3$J	X2$K	X1$L	X0$M	I$N	J$O	K	L	M	N	O=X9$D1$K
X7$M	X6$N	X5$O	X4$P	X3$J	X2$K	X1$L	X0$M	I$N	J$O	K$P	L	M	N	O	P=X8$D0$L

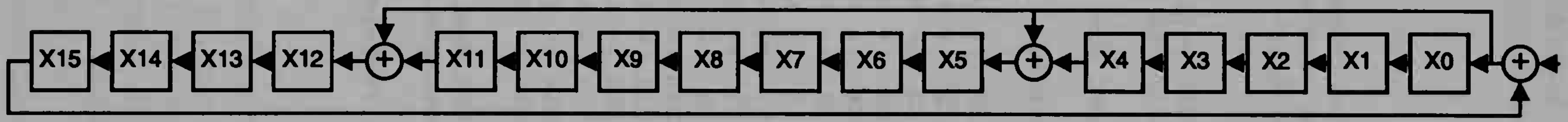


Figure 10.6 CRC-CCITT generation steps

The ABEL source file that describes a serial implementation of this CRC generator is given in Figures 10.7 and 10.8.

```
Module crc8s
Title 'Serial Cyclic Redundancy Check (CRC) Generator
using a 16-bit Linear Feedback Shift Register (LFSR)
G(X) = X16 + X12 + X5 + 1'

    Clk,Set,Din  pin;
    X15..X0      pin istype 'reg,buffer,xor';

    CRC_Sum = [X15..X0];

Equations

    X0   := Din $ X15 ;
    X1   := X0;
    X2   := X1;
    X3   := X2;
    X4   := X3;
    X5   := X4 $ Din $ X15;
    X6   := X5;
    X7   := X6;
    X8   := X7;
    X9   := X8;
    X10  := X9;
    X11  := X10;
    X12  := X11 $ Din $ X15;
    X13  := X12;
    X14  := X13;
    X15  := X14;

    CRC_Sum.Clk = Clk;
    CRC_Sum.ap  = Set;

Test_Vectors ([Clk,Set,Din] -> CRC_Sum)
              [.C., 1 , 0 ] -> ^hFFFF;

              [.C., 0 , 0 ] ->    .X.; " H = 48 = 0100 1000
              [.C., 0 , 1 ] ->    .X.;
              [.C., 0 , 0 ] ->    .X.;
              [.C., 0 , 0 ] ->    .X.;
              [.C., 0 , 1 ] ->    .X.;
              [.C., 0 , 0 ] ->    .X.;
              [.C., 0 , 0 ] ->    .X.;
              [.C., 0 , 0 ] -> ^h283C;
```

Figure 10.7 Sixteen-bit CRC generator design file (part 1 of 2)

```
                [.C., 0 , 0 ] ->    .X.; " e = 65 = 0110 0101
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 1 ] -> ^hA569;

                [.C., 0 , 0 ] ->    .X.; " l = 6C = 0110 1100
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 0 ] -> ^h2165;

                [.C., 0 , 0 ] ->    .X.; " l = 6C = 0110 1100
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 0 ] -> ^hFC69;

                [.C., 0 , 0 ] ->    .X.; " o = 6F = 0110 1111
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 0 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] ->    .X.;
                [.C., 0 , 1 ] -> ^hDADA;
End
```

Figure 10.8 Sixteen-bit CRC generator (part 2 of 2)

Parallel CRC Generation

Generation of CRC for parallel data is much simpler and is a straightfoward application of Boolean equations representing the required generator function. Figure 10.9 is an ABEL design description of a 16-bit CRC generator. In this design, the intermediate XOR functions represented by the identifiers *I* through *P* are assigned using constant declarations. This means that the logic for the CRC function will expand into sum-of-products logic. An alternative implementation would be to declare *I* through *P* as nodes and write equations for them. This would be a multilevel representation more appropriate for product-term limited devices such as LCAs.

```
Module crc8p
Title '8-bit Parallel Cyclic Redundancy Check (CRC) Generator'

    D7..D0      pin;
    X15..X0     pin istype 'reg,buffer';
    Clk,Set     pin;

    CRC_Sum     = [X15..X0];
    Data        = [D7..D0];

    I = X15 $ D7;
    J = X14 $ D6;
    K = X13 $ D5;
    L = X12 $ D4;
    M = X11 $ D3 $ I;
    N = X10 $ D2 $ J;
    O = X9  $ D1 $ K;
    P = X8  $ D0 $ L;

Equations

    X0  := P;
    X1  := O;
    X2  := N;
    X3  := M;
    X4  := L;
    X5  := K  $ P;
    X6  := J  $ O;
    X7  := I  $ N;
    X8  := X0 $ M;
    X9  := X1 $ L;
    X10 := X2 $ K;
    X11 := X3 $ J;
    X12 := X4 $ P $ I;
    X13 := X5 $ O;
    X14 := X6 $ N;
    X15 := X7 $ M;

    CRC_Sum.Clk = Clk;
    CRC_Sum.ap  = Set;

Test_Vectors ([Clk,Set,Data] -> CRC_Sum)
              [.C., 1 ,  0 ] -> ^hFFFF;
              [.C., 0 , 'H'] -> ^h283C;
              [.C., 0 , 'e'] -> ^hA569;
              [.C., 0 , 'l'] -> ^h2165;
              [.C., 0 , 'l'] -> ^hFC69;
              [.C., 0 , 'o'] -> ^hDADA;
End
```

Figure 10.9 Sixteen-bit parallel CRC-CCITT generator design file

10.4 REFERENCES

Intel, *Component Data Catalog*, Intel Corporation, Santa Clara, CA, 1978, page 4-71.

Systems Design Handbook, Monolithic Memories, Inc., Santa Clara, CA, 1983.

Vasa, Suresh, *Calculating an Error Checking Character in Software, Computer Design, May 1976, pages 190-192.*

11 Describing State Machines

State machines, as we discussed in Chapter 4, are circuits that include some form of memory to hold a current state, transition logic that determines the next state, and output logic that determines the state machine's function as seen from the outside. A number of possible methods can be used when describing a state machine in ABEL, depending on the complexity of the machine. The two methods that we will examine in this chapter are state transition tables and state diagrams.

11.1 STATE TRANSITION TABLES

ABEL's truth table language is well suited for describing state machines, particularly those that contain a large number of similar transitions. Truth tables written for the purpose of state machine specification are often referred to as state transition tables. In ABEL, a state machine written using a truth table can actually be more efficient than the same state machine described using a state diagram, because all the outputs described in a truth table are candidates for don't-care minimizations. In a state diagram, only the transition logic of the state registers is optimized for don't-cares. (The @DCSTATE directive described in Ap-

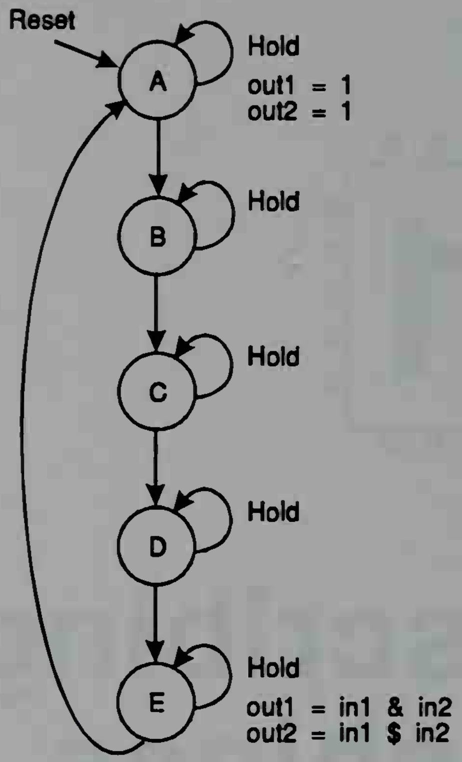


Figure 11.1 Mealy state machine with five states

pendix A does provide one method for optimizing state machine outputs, however.)

To demonstrate how a state machine is expressed using truth tables, consider the state graph illustrated in Figure 11.1. This state machine has five states and eleven possible state transitions (including holds). The machine also has two combinational (Mealy) outputs that decode the current state and the state diagram inputs.

To describe this state machine in a truth table, all possible transition conditions are listed along with their resulting next states. The ABEL design description is shown in Figure 11.2.

In the design file, the five states of the machine are given the values of zero through four and the symbolic names of *A*, *B*, *C*, *D*, and E. The state names decode to the binary values of 000,001,011,010, and 100, respectively.

Next, a truth table is used to specify the state machine's operation. Each line of the truth table represents a unique combination of inputs. The first ten entries in the table describe the transition logic for the state machine, and the last four lines describe the output logic for *out1* and

```
module state_tt
title 'Five state Mealy machine described using a state table';

    Clock,Reset   pin;
    in1,in2,hold  pin;
    Q3..Q0        pin istype 'reg,buffer,dc';
    out1,out2     pin istype 'com,dc';

    Qstate    = [Q2,Q1,Q0];
    A         = [ 0, 0, 0];
    B         = [ 0, 0, 1];
    C         = [ 0, 1, 1];
    D         = [ 0, 1, 0];
    E         = [ 1, 0, 0];

equations

    Qstate.clk = Clock;
    Qstate.ar = Reset;

truth_table
  ([in1,in2,hold,Qstate.fb]:>[Qstate]->[out1,out2])
   [.x.,.x.,  1 ,    A    ]:>[   A  ]->[  1 ,   1 ];  "Hold in A
   [.x.,.x.,  0 ,    A    ]:>[   B  ]->[  1 ,   1 ];
   [.x.,.x.,  1 ,    B    ]:>[   B  ]->[ .x., .x.];  "Hold in B
   [.x.,.x.,  0 ,    B    ]:>[   C  ]->[ .x., .x.];
   [.x.,.x.,  1 ,    C    ]:>[   C  ]->[ .x., .x.];  "Hold in C
   [.x.,.x.,  0 ,    C    ]:>[   D  ]->[ .x., .x.];
   [.x.,.x.,  1 ,    D    ]:>[   D  ]->[ .x., .x.];  "Hold in D
   [.x.,.x.,  0 ,    D    ]:>[   E  ]->[ .x., .x.];
   [.x.,.x.,  1 ,    E    ]:>[   E  ]->[ .x., .x.];  "Hold in E
   [.x.,.x.,  0 ,    E    ]:>[   A  ]->[ .x., .x.];
   [ 0 , 0 , .x.,    E    ]:>[  .x. ]->[  0 ,   0 ];  "Outputs
   [ 0 , 1 , .x.,    E    ]:>[  .x. ]->[  0 ,   1 ];  "active in
   [ 1 , 0 , .x.,    E    ]:>[  .x. ]->[  0 ,   1 ];  "state E
   [ 1 , 1 , .x.,    E    ]:>[  .x. ]->[  1 ,   0 ];

test_vectors ([Clock,Reset,in1,in2,hold]->[Qstate,out1,out2])
              [ .c. ,  1  , 0 , 0 ,  0 ]->[  A  ,  1 ,  1 ];
              [ .c. ,  0  , 0 , 0 ,  1 ]->[  A  ,  1 ,  1 ];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  B  , .x., .x.];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  C  , .x., .x.];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  D  , .x., .x.];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  E  ,  0 ,  0 ];
              [ .c. ,  0  , 0 , 1 ,  1 ]->[  E  ,  0 ,  1 ];
              [ .c. ,  0  , 1 , 0 ,  1 ]->[  E  ,  0 ,  1 ];
              [ .c. ,  0  , 1 , 1 ,  1 ]->[  E  ,  1 ,  0 ];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  A  ,  1 ,  1 ];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  B  , .x., .x.];
              [ .c. ,  0  , 0 , 0 ,  1 ]->[  B  , .x., .x.];
              [ .c. ,  0  , 0 , 0 ,  0 ]->[  C  , .x., .x.];

end
```

Figure 11.2 Five-state Mealy state machine described with ABEL

out2 while the machine is in state *E*. The two outputs of this state machine are only decoded in states *A* and *E* of the machine, so the .X. special constant is entered for *out1* and *out2* in states *B*, *C*, and *D*.

When this design is processed by the ABEL compiler, the don't-care information implied by the entries in the table for *out1* and *out2* will result in a reduction in the amount of logic required for each of the two outputs.

11.2 USING STATE DIAGRAMS

The truth table is a convenient method for describing state machines that have relatively few transitions or a larger number of transitions that are common to many states. More complex state machines, however, are often better described using alternative methods. One such method is the state diagram.

A state diagram is a relatively simple method of describing the operation of complex Mealy model and Moore model state machines. Any state machine that can be described in a state diagram can also be described using a truth table (or equations for that matter.) The key difference between truth tables and state diagrams is that when you describe a state machine using a truth table you describe the machine primarily

```
state_diagram Qstate

    state A:  out1 = 1;
              out2 = 1;
              if Hold then A
              else B;

    state B:  if Hold then B
              else C;

    state C:  if Hold then C
              else D;

    state D:  if Hold then D
              else E;

    state E:  out1 = in1 & in2;
              out2 = in1 $ in2;
              if Hold then E
              else A;
```

Figure 11.3 ABEL state diagram language

in terms of its transitions. When you use a state diagram, you describe the machine more in terms of its possible states.

Figure 11.3 shows an ABEL state diagram. This state diagram describes the same state machine that we previously described using a truth table. Notice that the state diagram contains one state description for each of the four states in the machine. The state diagram, like the truth table, has a header that specifies which signals are to be used for the state register. Unlike a truth table, other state machine inputs and outputs are not included in the state diagram header. Another difference is that the state register declared in the state machine header is flip-flop type independent; the state machine is written the same regardless of the type of flip-flops used for the state register.

A series of state descriptions follow the header. Each state description includes a value declaration, transition information, and optional equations for state machine outputs.

To describe the state machine transitions, each state description contains one or more transition statements. If you compare the state description for each state to its equivalent bubble in the state graph, you can see how the state transitions are expressed. States *A* through *E* have simple two-way branches that are described using IF-THEN-ELSE statements.

The two outputs *out1* and *out2* are defined to be high in state *A*, and as AND and XOR functions, respectively, in state *E*. This is the simplest way to describe state machine outputs in ABEL, but can have an optimization penalty. In ABEL, all equations, including those written within state diagrams, describe the on-set of a logic function. This means that all conditions not covered by equations are zero. In the truth table version of this design, the outputs were defined only in states *A* and *E*. If we want to achieve the same level of optimization for the outputs using a state diagram, we must either include don't-care equations in states B, C, and D or move the equations out of the state diagram and into a separate truth table. (The @DCSTATE directive can also be used to optimize state diagram outputs. See Appendix A for more information on this directive.) Two alternative versions of the design are shown in Figures 11.4 and 11.5.

11.3 POWER-UP AND ILLEGAL STATES

ABEL state machine descriptions are flip-flop type independent with regard to the state register. In some situations, however, the type of flip-flop being used and the architecture of the target device can affect

```
state_diagram Qstate
     state A:  out1 = 1;
               out2 = 1;
               if Hold then A
               else B;

     state B:  out1 ?= 1;
               out2 ?= 1;
               if Hold then B
               else C;

     state C:  out1 ?= 1;
               out2 ?= 1;
               if Hold then C
               else D;

     state D:  out1 ?= 1;
               out2 ?= 1;
               if Hold then D
               else E;

     state E:  out1 = in1 & in2;
               out2 = in1 $ in2;
               if Hold then E
               else A;
```

Figure 11.4 Specifying don't-cares for outputs

```
state_diagram Qstate
     state A:  if Hold then A
               else B;

     state B:  if Hold then B
               else C;

     state C:  if Hold then C
               else D;

     state D:  if Hold then D
               else E;

     state E:  if Hold then E
               else A;

truth_table([Qstate.FB,in1,in2]->[out1,out2])
           [    A    ,.x.,.x.]->[ 1 ,  1 ];
           [    E    , 0 , 0 ]->[ 0 ,  0 ];
           [    E    , 0 , 1 ]->[ 0 ,  1 ];
           [    E    , 1 , 0 ]->[ 0 ,  1 ];
           [    E    , 1 , 1 ]->[ 1 ,  0 ];
```

Figure 11.5 Decoding state machine outputs in a truth table

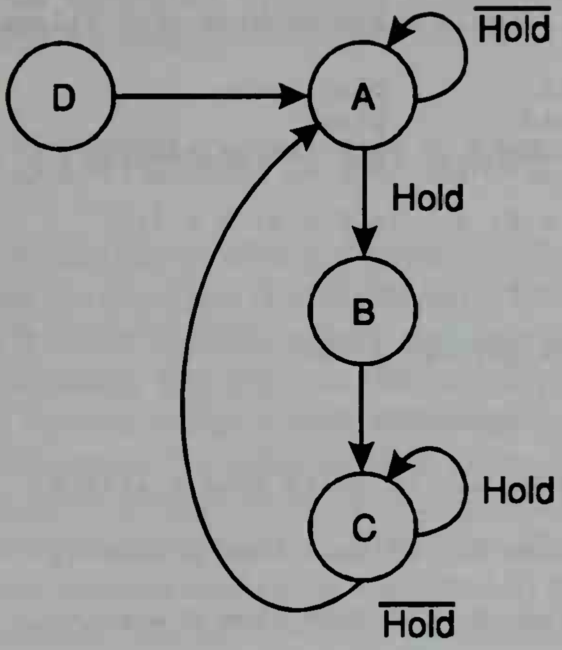


Figure 11.6 Simple machine with illegal state

how the state machine is described. Consider, for example, a modified form of our first state machine. This state machine, shown in Figure 11.6, has only three significant states. The fourth state (state *D*) is shown, but there are no transitions into this state. State *D* is therefore an illegal state.

Many (perhaps most) large state machines have a number of such illegal states. The task of the designer is to ensure that the machine is prevented from entering an illegal state or is provided with appropriate logic for escaping from the illegal state.

In a device with D-type flip-flops, the inherent default state (all registers low) can be used to advantage to protect against illegal states. If we implement this state machine with the same state values used previously (where state *D* had a value of 3, or binary 11) and use a device with inverting outputs (such as the 16R4 PAL device), we will need to provide a transition out of state *D*, since *D* represents the inherent default state (and possibly the power-up state as well) of the machine. The reset transition shown in the flow diagram (and in the ABEL design description shown in Figure 11.7) represents this method of circuit protection. For large state machines having many illegal states, however, the transition logic required to cover all illegal states may be impractical.

```
module sm5
title 'Simple state machine with illegal state recovery'

    Clk          pin;
    Hold         pin;
    Q1,Q0        pin istype 'reg';

    A = 0; B = 1; C = 2; D = 3;

equations

    [Q1,Q0].clk = Clk;

state_diagram [Q1,Q0]

    state A:  if !Hold then A else B;

    state B:  if Hold then B else C;

    state C:  if Hold then C else A;

    state D:  goto A;   "Illegal state recovery

test_vectors([Clk,Hold]->[Q1,Q0])
             [.c.,  0 ]->  .x.;  "Could power up to state D
             [.c.,  0 ]->  .x.;
             [.c.,  0 ]->   A;
             [.c.,  0 ]->   A;
             [.c.,  1 ]->   B;
             [.c.,  1 ]->   B;
             [.c.,  0 ]->   C;
             [.c.,  1 ]->   C;
             [.c.,  0 ]->   A;
end
```

Figure 11.7 ABEL description for state machine with illegal state recovery

An alternative method that is useful if the actual state values are not important is to rearrange the values so that a valid state (either *A*, *B*, or *C*) has the default value. Doing so, however, may leave open the question of whether the power-up state of the machine is accounted for, since different devices (even the same devices available from different manufacturers) may power-up to a register-low state, a register-high state, or an unpredictable state.

If the state machine is being implemented in a device with a flip-flop type other than D, particular care must be taken to ensure that all possible states are accounted for in some way or that some form of global reset is provided. Some PLA-type devices (such as the PLS105 from Signetics Corporation) feature a special product term that can be used to detect and escape from illegal states. This complement array, as it is called, can also be emulated by using a device output in a manner similar to that used for counter termination in Chapter 7.

11.4 ADDING STATE BITS TO SAVE LOGIC

We've seen many examples in which design modifications were made in order to fit into the constraints of specific target architectures. Another example for which such modifications are frequently required is the situation when too many flip-flops or device outputs are required by a design that uses classic state machine design philosophies. When a state machine is designed, it's often easiest to create the state transition circuitry first and then attach other circuitry, such as output decoding circuitry, to the state machine later. This typically results in the use of more I/O resources than are actually required to implement the circuit. Let's look at a simple example of this situation. Consider the block diagram for a decade counter and display decoder shown in Figure 11.8.

There are a number of possible ways to implement this circuit. If the actual values of the state bits are important, the circuit would probably be implemented in two pieces, with four flip-flops required for the counter state machine and seven outputs required for the display decoder circuit. This is the preferred method if you are limited in the number of available flip-flops. The ABEL source file of Figure 11.9 describes the counter/decoder using this method.

An alternative method, if the actual state of the counter isn't important, is to combine the function of the decoder with the state machine logic. In this implementation, the state machine uses seven state bits, the

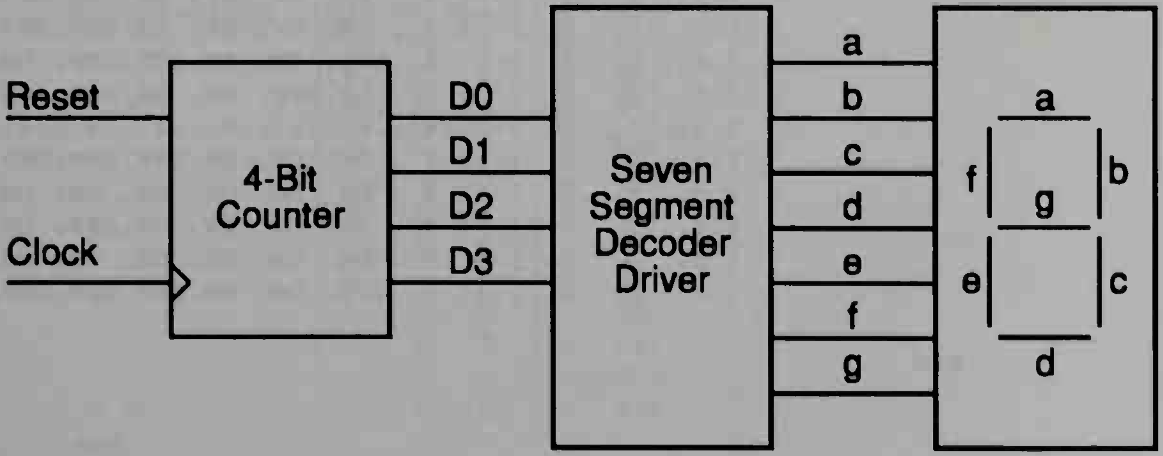


Figure 11.8 Decade counter and display decoder

```
module CNTBCD
title 'Decade counter and 7-segment decoder'

    Clk, Reset, OE      pin;
    D3..D0              pin istype 'dc,reg,invert';
    a,b,c,d,e,f,g       pin istype 'dc,com';

    bcd    = [D3..D0];
    led    = [a,b,c,d,e,f,g];

    ON,OFF = 0,1;      "Inverted sense for common anode LEDs

equations

    bcd := (bcd.fb + 1) & (bcd.fb < 9) & !Reset;

    bcd.clk = Clk;
    led.oe = !OE;

truth_table (bcd.fb -> [  a,  b,  c,  d,  e,  f,  g])
              0     -> [ ON, ON, ON, ON, ON, ON,OFF];
              1     -> [OFF, ON, ON,OFF,OFF,OFF,OFF];
              2     -> [ ON, ON,OFF, ON, ON,OFF, ON];
              3     -> [ ON, ON, ON, ON,OFF,OFF, ON];
              4     -> [OFF, ON, ON,OFF,OFF, ON, ON];
              5     -> [ ON,OFF, ON, ON,OFF, ON, ON];
              6     -> [ ON,OFF, ON, ON, ON, ON, ON];
              7     -> [ ON, ON, ON,OFF,OFF,OFF,OFF];
              8     -> [ ON, ON, ON, ON, ON, ON, ON];
              9     -> [ ON, ON, ON, ON,OFF, ON, ON];

test_vectors([Clk,OE,Reset]->[bcd,  a,   b,   c,   d,   e,   f,   g])
             [.c., 0,  1  ]->[ 0 , ON,  ON,  ON,  ON,  ON,  ON,OFF];
             [.c., 0,  0  ]->[ 1 ,OFF,  ON,  ON,OFF,OFF,OFF,OFF];
             [.c., 0,  0  ]->[ 2 , ON,  ON,OFF,  ON,  ON,OFF,  ON];
             [.c., 0,  0  ]->[ 3 , ON,  ON,  ON,  ON,OFF,OFF,  ON];
             [.c., 0,  0  ]->[ 4 ,OFF,  ON,  ON,OFF,OFF,  ON,  ON];
             [.c., 0,  0  ]->[ 5 , ON,OFF,  ON,  ON,OFF,  ON,  ON];
             [.c., 1,  0  ]->[.x.,.z.,.z.,.z.,.z.,.z.,.z.,.z.];
             [.c., 0,  0  ]->[ 7 , ON,  ON,  ON,OFF,OFF,OFF,OFF];
             [.c., 0,  0  ]->[ 8 , ON,  ON,  ON,  ON,  ON,  ON,  ON];
             [.c., 0,  0  ]->[ 9 , ON,  ON,  ON,  ON,OFF,  ON,  ON];
             [.c., 0,  0  ]->[ 0 , ON,  ON,  ON,  ON,  ON,  ON,OFF];
             [.c., 0,  0  ]->[ 1 ,OFF,  ON,  ON,OFF,OFF,OFF,OFF];

end
```

Figure 11.9 Counter and decoder described with equations and truth table

```
module CountLED
title '7-segment display decoder state machine'

    Clk,Reset,OE   pin;
    a,b,c,d,e,f,g  pin istype 'dc,reg';

    ON,OFF = 0,1;
    led    = [  a,  b,  c,  d,  e,  f,  g];
    S0     = [ ON, ON, ON, ON, ON, ON,OFF];
    S1     = [OFF, ON, ON,OFF,OFF,OFF,OFF];
    S2     = [ ON, ON,OFF, ON, ON,OFF, ON];
    S3     = [ ON, ON, ON, ON,OFF,OFF, ON];
    S4     = [OFF, ON, ON,OFF,OFF, ON, ON];
    S5     = [ ON,OFF, ON, ON,OFF, ON, ON];
    S6     = [ ON,OFF, ON, ON, ON, ON, ON];
    S7     = [ ON, ON, ON,OFF,OFF,OFF,OFF];
    S8     = [ ON, ON, ON, ON, ON, ON, ON];
    S9     = [ ON, ON, ON, ON,OFF, ON, ON];
    S15    = [OFF,OFF,OFF,OFF,OFF,OFF,OFF];

equations
    led.clk = Clk;
    led.oe = !OE;

state_diagram led
    state S0:  if Reset then S0 else s1;
    state S1:  if Reset then S0 else s2;
    state S2:  if Reset then S0 else s3;
    state S3:  if Reset then S0 else s4;
    state S4:  if Reset then S0 else s5;
    state S5:  if Reset then S0 else s6;
    state S6:  if Reset then S0 else s7;
    state S7:  if Reset then S0 else s8;
    state S8:  if Reset then S0 else s9;
    state S9:  if Reset then S0 else s0;
    state S15: goto S0;

test_vectors ([Clk,OE,Reset]-> led)
              [.c., 0,  1  ]->  S0;
              [.c., 0,  0  ]->  S1;
              [.c., 0,  0  ]->  S2;
              [.c., 0,  0  ]->  S3;
              [.c., 0,  0  ]->  S4;
              [.c., 0,  0  ]->  S5;
              [.c., 1,  0  ]-> .z.;
              [.c., 0,  0  ]->  S7;
              [.c., 0,  0  ]->  S8;
              [.c., 0,  0  ]->  S9;
              [.c., 0,  0  ]->  S0;
              [.c., 0,  0  ]->  S1;
end
```

Figure 11.10 Counter and decoder described as a state machine

value of which correspond to the ten output conditions of the decoder circuit. Figure 11.10 shows this version of the design, which can be implemented in a single small PAL device.

11.5 ONE-HOT STATE ENCODING

In a one-hot state machine, each state is represented by a unique bit, so an *n*-state machine requires *n* state bits. In a synchronous state machine, each state bit and corresponding state is represented by a flip-flop. One-hot synchronous state machines require more flip-flop resources than would otherwise be needed, but require much less input-forming logic and feedback resources than a fully encoded state machine.

In ABEL, one-hot state machines can be described using a symbolic state diagram syntax. Figure 11.11 is an example of this alternative method of description. This design is a symbolic (one-hot encoded) version of the five-state state machine presented earlier in this chapter.

Because this design description results in a one-hot implementation, the design requires a total of five flip-flips instead of the three previously required. The trade-off is a decrease in the amount of combinational logic required to decode the current state for transitions and for output logic; it's only necessary to decode one fed-back flip-flop state to detect when the machine is in a given state.

The only real drawback to a one-hot encoding strategy is that a one-hot state machine can be in more than one state at a time. While some designers actually use this fact to advantage when designing complex machines, it is not generally recommended due to the difficulty of debugging. If you do not want your one-hot state machine to land in an ambiguous state in which more than one state register is active, then you must design the transition logic so that there is no possibility of landing in an undefined state (one in which more than one flip-flop is active.) It is also important to have a reset function in a one-hot state machine, since it is unlikely that the target device will power up to a valid single-bit state.

11.6 STATE MACHINE PARTITIONING

State machine partitioning is often required when the size of the machine prohibits its being implemented in a target device due to restricted numbers of output registers or product terms. Complex state

```
module symbolic

    Hold          pin;
    clock,reset pin;
    in1,in2       pin;
    out1,out2     pin istype 'com';
    Qstate        state_register;
    A,B,C,D,E     state;

equations

    Qstate.clk = clock;

state_diagram Qstate

    async_reset A: reset;  "A is reset state

    state A:  out1 = 1;
              out2 = 1;
              if Hold then A
              else B;

    state B:  if Hold then B
              else C;

    state C:  if Hold then C
              else D;

    state D:  if Hold then D
              else E;

    state E:  out1 = in1 & in2;
              out2 = in1 $ in2;
              if Hold then E
              else A;
end
```

Figure 11.11 Symbolic state diagram language

machines with large numbers of transitions often result in large amounts of transition logic for each state bit of the machine. Even if the number of required state register flip-flops is small, the amount of input-forming logic for those flip-flops may necessitate partitioning the state machine into multiple blocks of logic. At other times, the reduction in logic achieved through state machine partitioning is sufficient to allow the state machine to be implemented in the target device when it otherwise wouldn't have fit.

The result of partitioning a state machine is usually a larger number of required state register flip-flops and unique states, but this increase in resources is usually offset by reductions in the amount of required

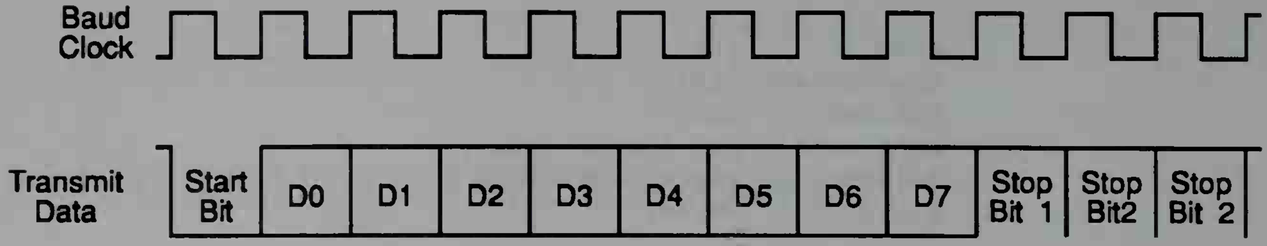


Figure 11.12 Serial transmission of eight-bit data

input-forming logic. Combining state machine partitioning with careful state assignments will result in the most efficient circuits.

To demonstrate how a state machine can be partitioned, we'll design a serial transmitter circuit that is intended for an FPGA architecture. A serial transmitter is a circuit that accepts eight bit wide parallel data and sends the data in a serial stream at a rate corresponding to the selected baud (bits per second) rate (Figure 11.12). To do this, the

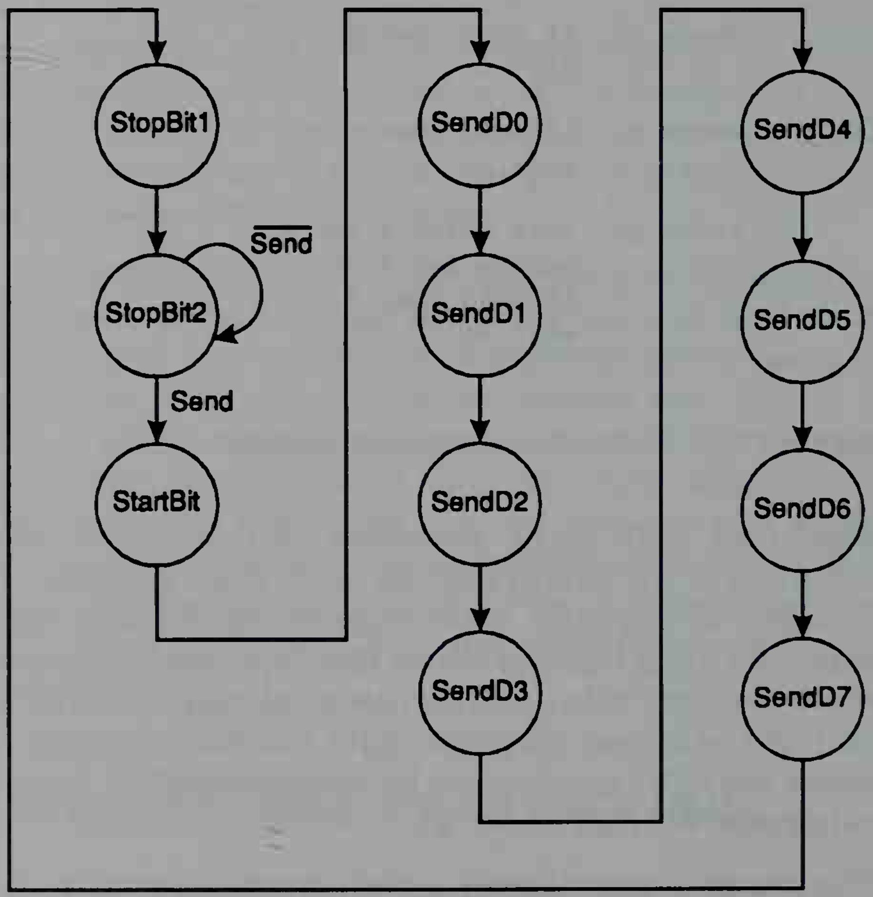


Figure 11.13 Serial transmitter state machine

transmitter (which is actually one-half of a UART circuit) must first store the eight bits of data, and then route each bit in turn to the output line. The eight bits of data are packaged with a start bit and two or more stop bits.

Figure 11.13 shows a state graph representing the operation of the serial transmitter circuit. A corresponding ABEL design description is shown in Figures 11.14, 11.15 and 11.16. As written, the design requires only four registers to represent the 16 possible states, but results in wide input-forming logic that is not efficient to implement in FPGA devices.

```
module XMIT1
title 'Serial transmitter'

// Inputs
    DI7..DI0        pin;
    Clk2,Clk1       pin;
    Clr,Send        pin;

// Outputs
    D7..D0          pin istype 'reg';
    SM3..SM0        pin istype 'reg';
    NextWord,TXD    pin istype 'com';

// Sets
    Data        = [D7..D0];
    DataShift   = [D0,D7..D1];
    Sreg        = [SM3,SM2,SM1,SM0];
    DataIn      = [DI7..DI0];
    BaudClk     = [Clk1,Clk2];

Equations
    Sreg.C = Clk1;
    Data.C = Clk2;

    Sreg.AR = Clr;
    Data.AR = Clr;

Declarations
// States
    StopBit1 =  0;
    StopBit2 =  1;
    StartBit =  2;
    SendD0   =  8;
    SendD1   =  9;
    SendD2   = 10;
    SendD3   = 11;
    SendD4   = 12;
    SendD5   = 13;
    SendD6   = 14;
    SendD7   = 15;
```

Figure 11.14 Serial transmitter design file (part 1 of 3)

```
State_Diagram Sreg
        State StopBit1: NextWord  = 1;
                        TXD       = 1;
                        Data     := DataIn;
                        goto StopBit2;

        State StopBit2: NextWord  = 0;
                        TXD       = 1;
                        Data     := DataIn;
                        if(Send) then StartBit else StopBit2;

        State StartBit: NextWord  = 0;
                        TXD       = 0;
                        Data     := DataIn;
                        goto SendD0;

        State SendD0:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD1;

        State SendD1:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD2;

        State SendD2:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD3;

        State SendD3:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD4;

        State SendD4:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD5;

        State SendD5:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD6;

        State SendD6:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto SendD7;

        State SendD7:   NextWord  = 0;
                        TXD       = D0;
                        Data     := DataShift;
                        goto StopBit1;
```

Figure 11.15 Serial transmitter design file (part 2 of 3)

```
Declarations

    C = .C.;

test_vectors
   ([BaudClk,Clr,Send,DataIn] -> [Sreg    ,NextWord,TXD])
    [   C    , 1 ,  0 ,    0  ] -> [StopBit1,    1   , 1 ];
    [   C    , 1 ,  0 ,    0  ] -> [StopBit1,    1   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [StopBit2,    0   , 1 ];
    [   C    , 0 ,  0 , ^h55  ] -> [StopBit2,    0   , 1 ];
    [   C    , 0 ,  0 , ^h55  ] -> [StopBit2,    0   , 1 ];
    [   C    , 0 ,  1 , ^h55  ] -> [StartBit,    0   , 0 ];
    [   C    , 0 ,  1 , ^h55  ] -> [SendD0  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD1  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD2  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD3  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD4  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD5  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD6  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD7  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [StopBit1,    1   , 1 ];
    [   C    , 0 ,  0 , ^h0F  ] -> [StopBit2,    0   , 1 ];
    [   C    , 0 ,  0 , ^h0F  ] -> [StopBit2,    0   , 1 ];
    [   C    , 0 ,  1 , ^h0F  ] -> [StartBit,    0   , 0 ];
    [   C    , 0 ,  1 , ^h0F  ] -> [SendD0  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD1  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD2  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD3  ,    0   , 1 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD4  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD5  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD6  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [SendD7  ,    0   , 0 ];
    [   C    , 0 ,  0 ,    0  ] -> [StopBit1,    1   , 1 ];

end
```

Figure 11.16 Serial transmitter design file (part 3 of 3)

The serial transmitter has eleven states corresponding to the one start bit, the eight data bits, and the two stop bits of the transmitted data. The machine's default state is state *StopBit1*, and the design of the transition logic ensures that the machine will always transition to the initial state *StopBit2* within ten clock cycles of power-up regardless of the power-up state. In the absence of data to send, the machine remains in state *StopBit2* awaiting an assertion of the *Send* input. In the *StartBit* state, a start bit (logic level 0) is sent to the serial output (*TXD*) while the parallel data is loaded into the eight bit wide data register. In each of the following eight states (*Send0* through *Send7*) the *D0* bit of the data is sent to *TXD* and the data register is shifted one bit. At the

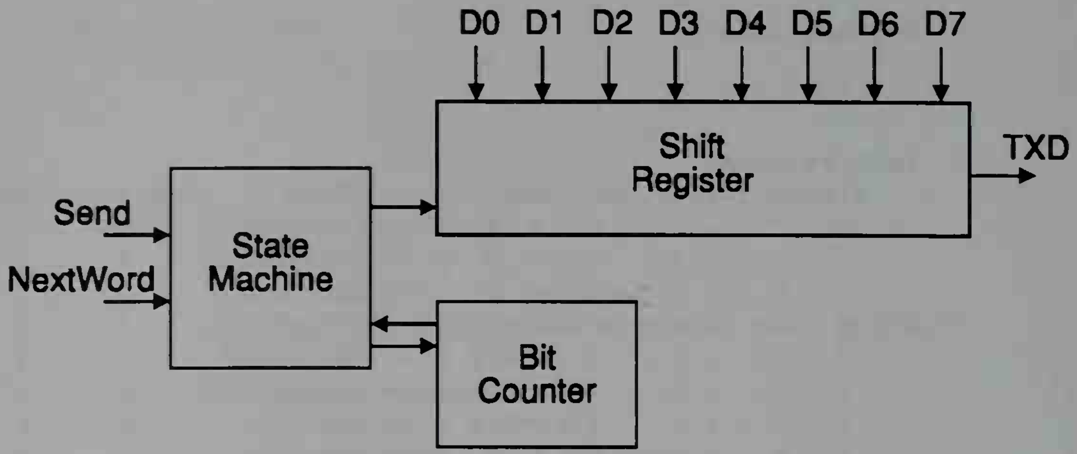


Figure 11.17 Serial transmitter functional blocks

conclusion of the data output sequence, the *NextWord* output is strobed, indicating that the next byte of parallel data can be loaded.

Written in the form shown, this state machine consumes an unnecessarily large amount of logic. Although any FPGA device is probably capable of implementing the entire machine, the wide input-forming logic that results from the design description would be very inefficient and would have unnecessary speed penalties created when the transition logic is broken into the multiple levels needed to fit in an FPGA's logic blocks. Carefully chosen state values could result in a decrease in the amount of logic required for each state bit, but the most dramatic savings in logic is obtained by increasing the number of flip-flops used for the state register and reducing the amount of transition logic. This can be done either by adding flip-flops to the state register set and reassigning the state values (in the extreme case using one-hot encoding, in which each state of the machine is represented by a unique flip-flop) or by partitioning the state machine.

This state machine can be easily partitioned into smaller functional units, resulting in an increase in the total number of states and a decrease in the total amount of logic required. Figure 11.17 illustrates the three functional parts of this design that are candidates for partitioning.

For this redesign, we've chosen to isolate the eight states that correspond to the data-shifting function, and implement these eight states as a semi independent counter. Figure 11.18 shows state graphs for the resulting two segments of the partitioned state machine. The first state machine has four states: *StopBit1*, *StopBit2*, *StartBit* and *Shift*. The *Shift* state replaces the eight shifter states of the previous state graph. During this state of the primary state machine, the counter state machine is active and cycles through its eight states. The data shifter

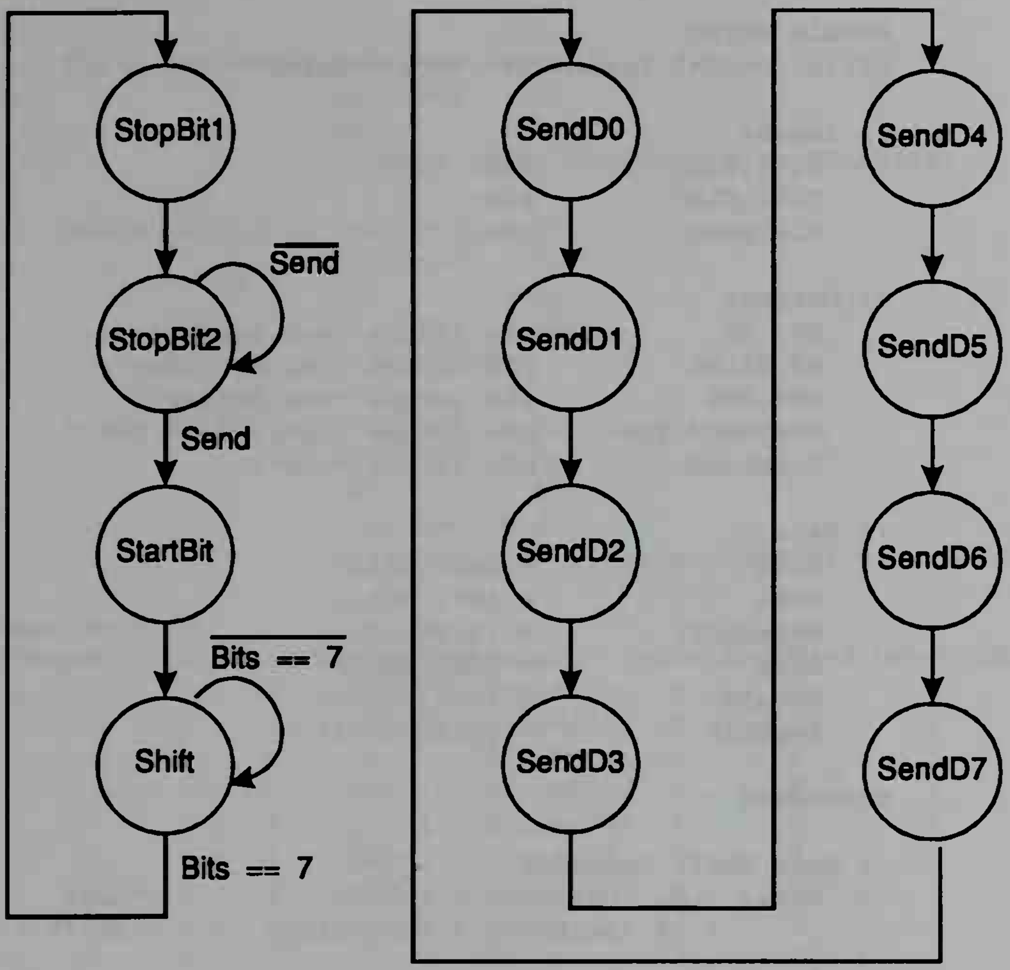


Figure 11.18 Partitioned serial transmitter state machine

is also split into a separate functional unit, further simplifying the design. The communication between the two machines is one-way; the primary state machine has complete control over the counter state machine, resetting the counter bits and monitoring their value during shift-and-transmit operations.

In the corresponding ABEL design description shown in Figures 11.19 and 11.20, the counter and shifter portions are described using high-level equations, while the remaining state machine logic is described using ABEL's state diagram language. The new design requires one additional registered output and one additional fed-back combinational output. The eight-state counter and the data shift register are both controlled from the primary state machine, which now has a total of four states. The counter value represented by *Bits* is reset in state *StartBit* (the expression *Bits.RE = !0* results in ones being applied to the reset inputs of all registers in the *Bits* set) and increments during state *Shift* until the terminal count of seven is reached. The shifter logic is

```
module XMIT2
title 'Serial transmiter with counter'

// Inputs
    DI7..DI0          pin;
    Clk2,Clk1         pin;
    Clr,Send          pin;

// Outputs
    D7..D0            pin istype 'reg,buffer';
    B2,B1,B0          pin istype 'reg_T,buffer';
    SM1,SM0           pin istype 'reg,buffer';
    NextWord,TXD      pin istype 'com';
    DataLoad          pin istype 'com';

// Sets
    Bits              = [B2..B0];
    Data              = [D7..D0];
    DataShift         = [0,D7..D1];
    Sreg              = [SM1,SM0];
    DataIn            = [DI7..DI0];
    BaudClk           = [Clk1,Clk2];

Equations

// Data Shift Register
    Data     :=  DataLoad & DataIn            "Load
              # !DataLoad & DataShift;        "Shift

// Bit Counter (T f/f)
    Bits.t   = Bits.q $ (Bits.q + 1);

    Sreg.C   = Clk1;
    Data.C   = Clk2;
    Bits.C   = Clk2;
    Sreg.AR  = Clr;
    Data.AR  = Clr;

Declarations
// States
        StopBit1       = 0;
        StopBit2       = 1;
        StartBit       = 2;
        Shift          = 3;

State_Diagram Sreg
    State StopBit1: NextWord  = 1;
                    DataLoad  = 1;
                    TXD       = 1;
                    goto StopBit2;
```

Figure 11.19 Partitioned serial transmitter design file (part 1 of 2)

```
          State StopBit2: NextWord  = 0;
                          DataLoad  = 1;
                          TXD       = 1;
                          if(Send) then StartBit else StopBit2;

          State StartBit: NextWord  = 0;
                          DataLoad  = 1;
                          TXD       = 0;
                          Bits.RE   = !0;
                          goto Shift;

          State Shift:    NextWord  = 0;
                          DataLoad  = 0;
                          TXD       = D0;
                          Bits.AR   = 0;
                          if (Bits.q == 7) then StopBit1 else Shift;

      test_vectors
      ([BaudClk,Clr,Send,DataIn]->[Sreg    ,Bits,NextWord,DataLoad,TXD])
       [  .C.  , 1 ,  0 ,   0  ]->[StopBit1, .X.,    1   ,   1    , 1 ];
       [  .C.  , 1 ,  0 ,   0  ]->[StopBit1, .X.,    1   ,   1    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[StopBit2, .X.,    0   ,   1    , 1 ];
       [  .C.  , 0 ,  0 , ^h55 ]->[StopBit2, .X.,    0   ,   1    , 1 ];
       [  .C.  , 0 ,  0 , ^h55 ]->[StopBit2, .X.,    0   ,   1    , 1 ];
       [  .C.  , 0 ,  1 , ^h55 ]->[StartBit,  0 ,    0   ,   1    , 0 ];
       [  .C.  , 0 ,  1 , ^h55 ]->[Shift   ,  0 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  1 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  2 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  3 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  4 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  5 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  6 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  7 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[StopBit1, .X.,    1   ,   1    , 1 ];
       [  .C.  , 0 ,  0 , ^h0F ]->[StopBit2, .X.,    0   ,   1    , 1 ];
       [  .C.  , 0 ,  0 , ^h0F ]->[StopBit2, .X.,    0   ,   1    , 1 ];
       [  .C.  , 0 ,  1 , ^h0F ]->[StartBit,  0 ,    0   ,   1    , 0 ];
       [  .C.  , 0 ,  1 , ^h0F ]->[Shift   ,  0 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  1 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  2 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  3 ,    0   ,   0    , 1 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  4 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  5 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  6 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[Shift   ,  7 ,    0   ,   0    , 0 ];
       [  .C.  , 0 ,  0 ,   0  ]->[StopBit1, .X.,    1   ,   1    , 1 ];
      End
```

Figure 11.20 Partitioned serial transmitter design file (part 2 of 2)

controlled by the additional fed back output signal *DataLoad*, which indicates when the input data should be loaded rather than shifted.

Whereas the original (nonpartitioned) design would have required only four state register flip-flops to represent the eleven possible states, the partitioned design requires five: three for the eight-state counter, and two for the controlling state machine. The benefit of this partitioning strategy is that the input-forming logic is much less complex, and determining an optimal state encoding for the two state machines is much simpler. As an added benefit, the counter portion can be implemented using T flip-flops, while the primary state machine can be implemented using D flip-flops.

One side benefit to this redesign is that the primary state machine now requires a maximum of only three clock cycles to stabilize after a random power-up, rather than the ten cycles required previously. This is reflected in the test vectors listed in the ABEL source file.

11.7 ASYNCHRONOUS STATE MACHINES

All the state machines described in this chapter have been synchronous and used either D- or T-type flip-flops to provide the synchronization. As described in Chapter 4, however, there is another class of state machine, the asynchronous state machine, that does not require dedicated flip-flops. While ABEL's state diagram language is primarily intended for synchronous state machines, it can be used to describe asynchronous state machines as well.

When describing an asynchronous state machine, you must observe the rules for hazard avoidance described in Chapter 4. In particular, you must ensure that the machine is self-stabilizing in each state and that transitions between each state are unambiguous and hazard free.

Figure 11.21 is an ABEL design description for a pattern detector. This pattern detector monitors the *Data* and *Valid* inputs and looks for the binary pattern of 11010 on the *Data* input. The data on the *Data* input is checked whenever the *Valid* input is strobed. Each time a correct binary value is encountered (a value that continues the sequence) the machine advances to the next state until either an incorrect bit value is encountered or five correct values have been read.

The state diagram for this state machine is written using an alternative form of state diagram, in which the state register set in the state diagram header is replaced by two sets of signals. The set of signals on the left is the current state of the machine (as observed on the inputs listed)

```
module async
title 'Asynchronous state machine example - pattern detector'

// The pattern we want to detect is 11010. The data bit
// is monitored on input Data, and the data is checked when
// the input Valid is strobed.

Q5..Q0       node istype 'com';

Data,Valid  pin;
DetectFlag  pin istype 'com';

Qstate   = [Q5..Q0];
X = .x.;

Reset    = [0,0,0,0,0,0];
Initial = [0,0,0,0,0,1];
State1a = [0,0,0,0,1,1];  "Bridge state: Initial to State1
State1   = [0,0,0,0,1,0];
State2a = [0,0,0,1,1,0];  "Bridge state: State1 to State2
State2   = [0,0,0,1,0,0];
State3a = [0,0,1,1,0,0];
State3   = [0,0,1,0,0,0];
State4a = [0,1,1,0,0,0];
State4   = [0,1,0,0,0,0];
Detecta = [1,1,0,0,0,0];
Detect   = [1,0,0,0,0,0];

state_diagram Qstate.pin->Qstate

    state Reset:   if (!Valid) then Initial
                   else Reset;

    state Initial: if (Valid) then
                       if (Data) then State1a  "1
                       else Reset;
                   else Initial;

    state State1a: if (!Valid) then State1
                   else State1a;  "Wait until strobe goes low

    state State1:  if (Valid) then
                       if (Data) then State2a  "1
                       else Reset;
                   else State1;

    state State2a: if (!Valid) then State2
                   else State2a;  "Wait until strobe goes low
```

Figure 1.21 Asynchronous state diagram (part 1 of 2)

```
        state State2:  if (Valid) then
                           if (!Data) then State3a "0
                           else Reset;
                       else State2;

        state State3a: if (!Valid) then State3
                       else State3a;    "Wait until strobe goes low

        state State3:  if (Valid) then
                           if (Data) then State4a  "1
                           else Reset;
                       else State3;

        state State4a: if (!Valid) then State4
                       else State4a;    "Wait until strobe goes low

        state State4:  if (Valid) then
                           if (!Data) then Detecta "0
                           else Reset;
                       else State4;

        state Detecta: if (!Valid) then Detect
                       else Detecta;     "Wait until strobe goes low

        state Detect:  DetectFlag = 1;
                       if (Valid) then Reset  "Wait for next bit
                       else Detect;

    test_vectors([Data,Valid]->[ Qstate,DetectFlag])
                 [  0 ,  0  ]->[  .x.  ,    .x.   ];  "Init sequence..
                 [  0 ,  0  ]->[  .x.  ,    .x.   ];
                 [  0 ,  0  ]->[  .x.  ,    .x.   ];
                 [  0 ,  0  ]->[Initial,    .x.   ];  "Ready...
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Got a 1
                 [  1 ,  0  ]->[ State1,     0    ];
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Got another 1
                 [  1 ,  0  ]->[ State2,     0    ];
                 [  1 ,  0  ]->[  .x.  ,    .x.   ];
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Oops, not a 0
                 [ .x.,  0  ]->[Initial,     0    ];  "Try again...
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Got a 1
                 [  1 ,  0  ]->[ State1,     0    ];
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Got another 1
                 [  1 ,  0  ]->[ State2,     0    ];
                 [  0 ,  1  ]->[  .x.  ,    .x.   ];  "Got a 0
                 [  0 ,  0  ]->[ State3,     0    ];
                 [  1 ,  1  ]->[  .x.  ,    .x.   ];  "Got a 1
                 [  1 ,  0  ]->[ State4,     0    ];
                 [  0 ,  1  ]->[  .x.  ,    .x.   ];  "Got a 0!
                 [  0 ,  0  ]->[ Detect,     1    ];  "Detect flag
                 [ .x.,  1  ]->[ Reset ,     0    ];  "Back to Reset
     end
```

Figure 1.22 Asynchronous state diagram (part 2 of 2)

and the signals on the right side of the header represent the next state of the machine. In this design, the input and output portions of the state diagram header are the same. This results in the asynchronous feedback needed to construct an asynchronous state machine.

To provide hazard-free transitions between states, the state machine is encoded using a modified form of one-hot encoding in which each bit represents one of the six primary states in the machine. (This is not a true one-hot encoding, however, because the machine must decode all fed-back state bits simultaneously to determine the current state.) To allow the state machine to transition from any state to the reset state and maintain the single bit change rule described in Chapter 4, *bridge states* are used between each primary state. Each time the machine advances from one primary state to another (from *State3* to *State4*, for example) it must advance through a bridge state that holds state until the *Valid* input changes from high back to low. Because the transitions out of the primary states depend on a positive edge appearing on the *Valid* input, and the transitions into the next primary state (and out of the corresponding bridge state) depend on a negative edge appearing on *Valid*, you can think of the *Valid* input as an asynchronous clock input for this circuit.

11.8 REFERENCES

Pellerin, David, and Michael Holley, *Practical Design Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Treseler, Michael, *Designing State Machine Controllers Using Programmable Logic*, Prentice Hall, Englewood Cliffs, NJ, 1992.

Unger, Steven J., *The Essence of Logic Circuits*, Prentice Hall, Englewood Cliffs, NJ, 1989.

Wakerly, John F., *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, NJ, 1990.

and the signals on the right side of the header represent the next state of the machine. In this design, the input and output portions of the state diagram header are the same. This results in the asynchronous feedback needed to construct an asynchronous state machine.

To provide hazard-free transitions between states, the state machine is encoded using a modified form of one-hot encoding in which each bit represents one of the six primary states in the machine. (This is not a true one-hot encoding, however, because the machine must decode all fed-back state bits simultaneously to determine the current state.) To allow the state machine to transition from any state to the reset state and maintain the single bit change rule described in Chapter 4, bridge states are used between each primary state. Each time the machine advances from one primary state to another (from State3 to State4, for example) it must advance through a bridge state that holds state until the Valid input changes from high back to low. Because the transitions out of the primary states depend on a positive edge appearing on the Valid input, and the transitions into the next primary state (and out of the corresponding bridge state) depend on a negative edge appearing on Valid, you can think of the Valid input as an asynchronous clock input for this circuit.

11.8 REFERENCES

Pellerin, David, and Michael Holley. *Practical Design Using Programmable Logic*. Prentice Hall, Englewood Cliffs, NJ, 1991.

Holley, Michael. *Designing State Machine Controllers Using Programmable Logic*. Prentice Hall, Englewood Cliffs, NJ, 1992.

Unger, Stephen H. *The Essence of Logic Circuits*. Prentice Hall, Englewood Cliffs, NJ, 1989.

Wakerly, John F. *Digital Design Principles and Practices*. Prentice Hall, Englewood Cliffs, NJ, 1990.

12 A Video Frame Grabber

In this final chapter, we'll bring together the information provided in earlier chapters by creating an actual application. The design we'll present here is a video frame grabber. This circuit consists of an IBM PC-compatible board that digitizes a single field of an external video image (supplied from a camera or other video source) and allows it to be displayed directly on a video monitor or accessed by software for storage and display purposes. The design utilizes PLDs for control and memory read/write functions. The frame grabber system includes a software control program that runs under Microsoft Windows.

12.1 GENERAL DESIGN REQUIREMENTS

A standard NTSC video image (our frame grabber is designed for NTSC video signals, but could easily be adapted for use with the European PAL and SECAM standards) is composed of 525 *scan lines*, as shown in Figure 12.1. Each video image, or *frame*, is actually divided into two interlaced *fields* of 262.5 lines. These fields are refreshed at a rate of 59.94 fields per second (the original RS-170 black and white standard specified a simpler 60 fields per second rate, but this was changed in the late 1950s when color was added). This results in a display rate of just under 30 frames each second.

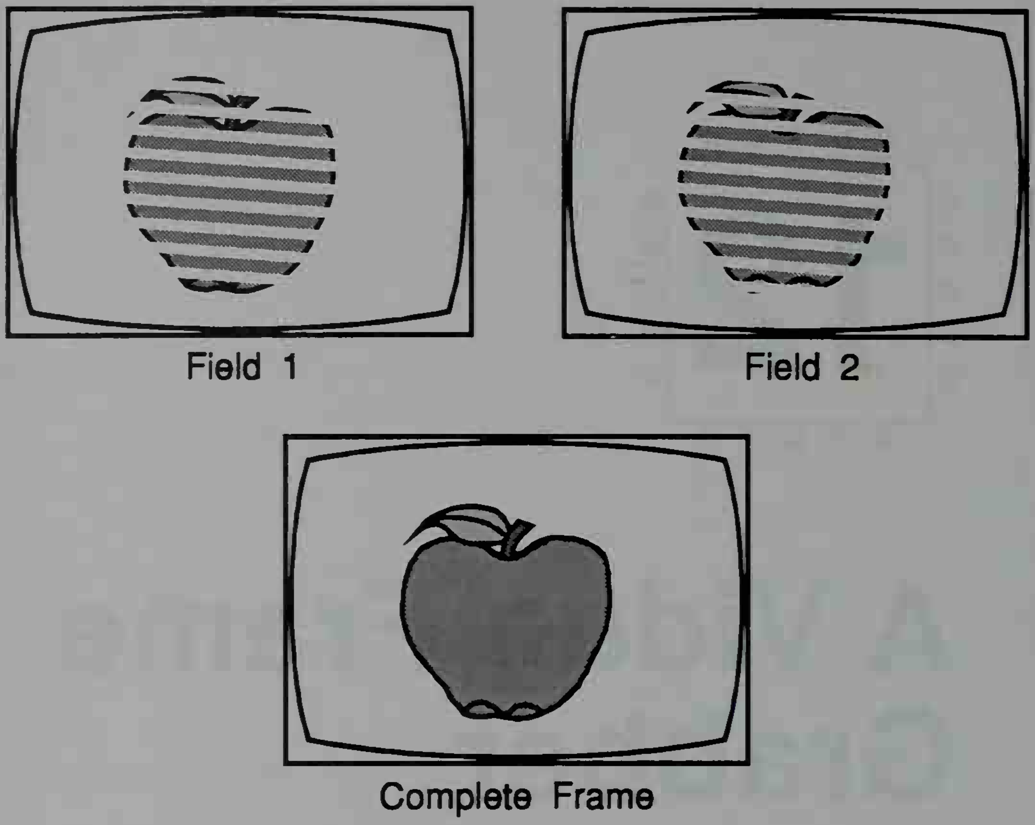


Figure 12.1 Components of a video image

To capture one frame of a live video image, high-speed circuits called *frame grabbers* are used to digitize and store hundreds of samples of video per scan line for the 525 lines of the image.

The video signals that we will be capturing are analog, so an A-D converter must be used to convert the incoming video signal to digital amplitude values. These sample values must be analyzed and stored in RAM as quickly as they arrive in order to capture a live image. To provide the A-D function, we have chosen a Samsung KSV3100A video chip. (Other chips with similar functions are available from Sony and Brooktree.) The readily available Samsung chip is a complete single-chip video system that includes an input preamp and clamping circuit, an 8-bit video A-D converter with reference, and a 10-bit D-A converter.

To sample and store a single frame of video, the frame grabber circuit must be able to recognize when a new field of the image has started and when it has finished. Fortunately, the signals that identify the critical portions of a video image are relatively easy to detect. The start of each field is identified by a 20-line vertical blanking interval (the black bar seen at the top of the picture on a poorly adjusted television or monitor). Each scan line consists of ten microseconds (ms) of horizontal synchro-

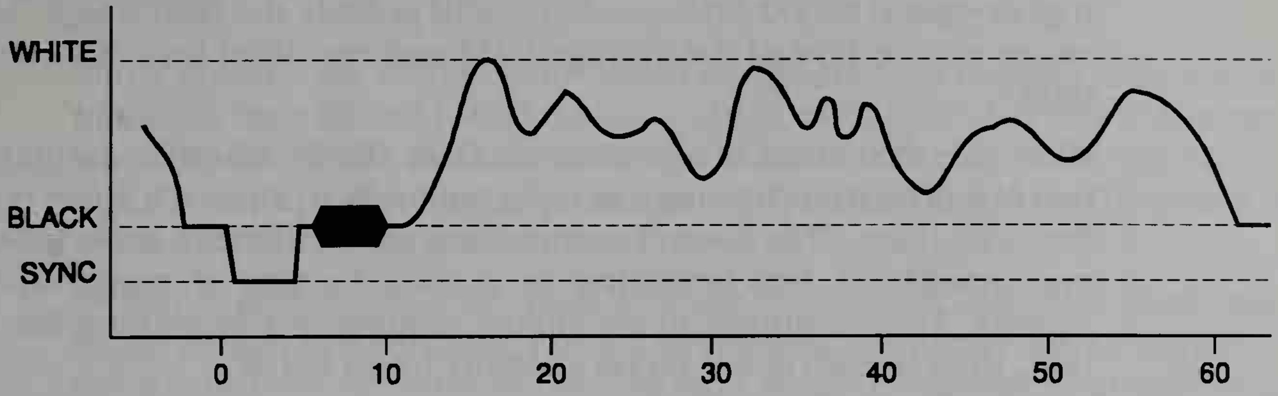


Figure 12.2 Video scan line

nization and color information (the color burst), followed by about 53 ms of video information (see Figure 12.2).

Choosing A Sample Rate

Sampling and digitizing an analog signal (of any type) can be done at a variety of rates, depending on the quality required of the reconstructed signal. If the highest frequency in the sampled signal (the Nyquist frequency) is f_h, then the minimum sampling rate at which the signal could theoretically be recovered is $2f_h$ (the *Nyquist rate*).

The NTSC color burst frequency is 3.579545 megahertz (MHz). To reproduce a color image, the video signal must be sampled at more than twice the color burst frequency. In practice, sampling the video signal at exactly twice the color burst frequency produces a poor-quality color picture, producing correct colors for, at best, half the image.

For our frame grabber, we have chosen to capture only one field of video, rather than the complete frame. This results in some loss of vertical resolution, but this loss isn't noticeable on a standard television screen. To sample at over twice the Nyquist rate, each field requires more that 119,437 samples [2 X (3,579,545 / 59.94)].

Since the digitized image must be stored in high-speed RAM, one or more appropriately sized RAM devices must be selected. The nearest standard RAM size is 128K, which provides a total of 131,072 bytes of storage. The ideal sample rate to fully use this amount of RAM would be 7.856456 MHz [131,072 X 59.94]. The nearest standard frequency oscillators available are 7.3728 MHz and 8.000 MHz. A 15.000 MHz oscillator divided by 2 will also produce a good sample rate. The higher-speed 8.000-MHz oscillator will provide the best image, but will cause an overflow of the storage RAM and resultant loss of some image data.

If we sacrifice some image data and use the 8.000-MHz oscillator, the result will be that the image is replayed too fast, since it's short by about five scan lines. This doesn't seem to be a problem, since most televisions are capable of synchronizing to a wide variety of marginal-quality signals. The advantage of the higher sample rate is better color definition; the reconstructed picture simply looks better.

The 8.000-MHz frame grabber design can improved by "faking" the missing five scan lines. One way to do this is to replay five of the existing scan lines. These lines wouldn't be seen if they were part of the vertical blanking interval. This could be done by adding an additional bit to the frame grabber's address counter and counting to a value higher than the highest RAM address. This would result in the RAM address wrapping back around to zero.

Another approach is to synchronize the saved image with a live image, using part of the live image's vertical blanking interval to fill out the missing field data. In video parlance, this is known as genlocking. To genlock the frozen image, the frame grabber circuit must monitor a live video signal and wait for its vertical blanking interval to appear. When this blanking interval is detected, the frame grabber must switch from a live display mode to a frozen-image display mode for the remainder of the field. At the completion of the (partial) field, the circuit must revert to live display mode momentarily and then repeat the process.

Choosing a Sample Size

The number of bits in each sample determines the dynamic range, or levels of intensity, that can be reproduced in the reconstructed image. These levels of intensity are also known as the *gray scale*. A four-bit sample can reproduce 16 levels of intensity, and eight bits can reproduce 256 levels. Most home video systems don't benefit from more than eight bits, but studio quality equipment will reproduce better images if ten bits are used. We have used eight bits of sample data in our frame grabber circuit, since this is the size of the sample provided by the Samsung video chip.

Selecting RAM Chips

Many options are available to circuit designers who require high-speed memory. Specialized frame memory chips exist for video frame storage (the Hitachi HM53051P and Toshiba TC521000P chips, for example), but these chips are expensive and aren't typically found on the shelves of your local electronics supply store.

Dynamic RAMS could be used, but the multiplexed address lines add to the design's complexity. For a frame buffer with no read-out capability, the refresh circuitry of DRAMs isn't needed, somewhat simplifying their use. The constant redisplay of the saved image results in an automatic refresh, since each display sequence completely cycles through all the frame grabber's memory addresses.

For this design, CMOS static RAMs were selected because of their simplicity. Byte-wide static RAMs are low in cost and readily available. The nonmultiplexed address inputs eliminate the need for the multiphased system clock that would be required if DRAMs were used.

With an 8.000-MHz sample rate, 125-nanosecond (ns) memory cycle times are required. Commonly available 100-ns RAMs are therefore sufficient for the design. The critical write cycle parameters that must be provided for in the design are *address setup and hold time* and *data setup and hold time.*

12.2 THE VIDEO FRAME GRABBER SYSTEM

Our video frame grabber system consists of a card that plugs into the motherboard of an ISA-compatible bus and allows digitized images (from a camera, VCR, or any other NTSC video source) to be displayed on a standard video monitor or stored in Windows BMP format files for later use. This board is controlled by software (written in C) that runs under Microsoft Windows. The control circuitry for the video frame grabber is implemented exclusively in PLDs.

The video frame grabber, as shown in the block diagram in Figure 12.3, monitors the digitized video signal from the A-D converter and, on command, scans and stores a single field of the live image. The frame grabber board has two RCA-type phono jacks for the video input and output signals.

When instructed by the controlling software to record an image, the frame grabber circuit must first wait for and identify the beginning of a field (indicated by the vertical blanking interval). When the blanking

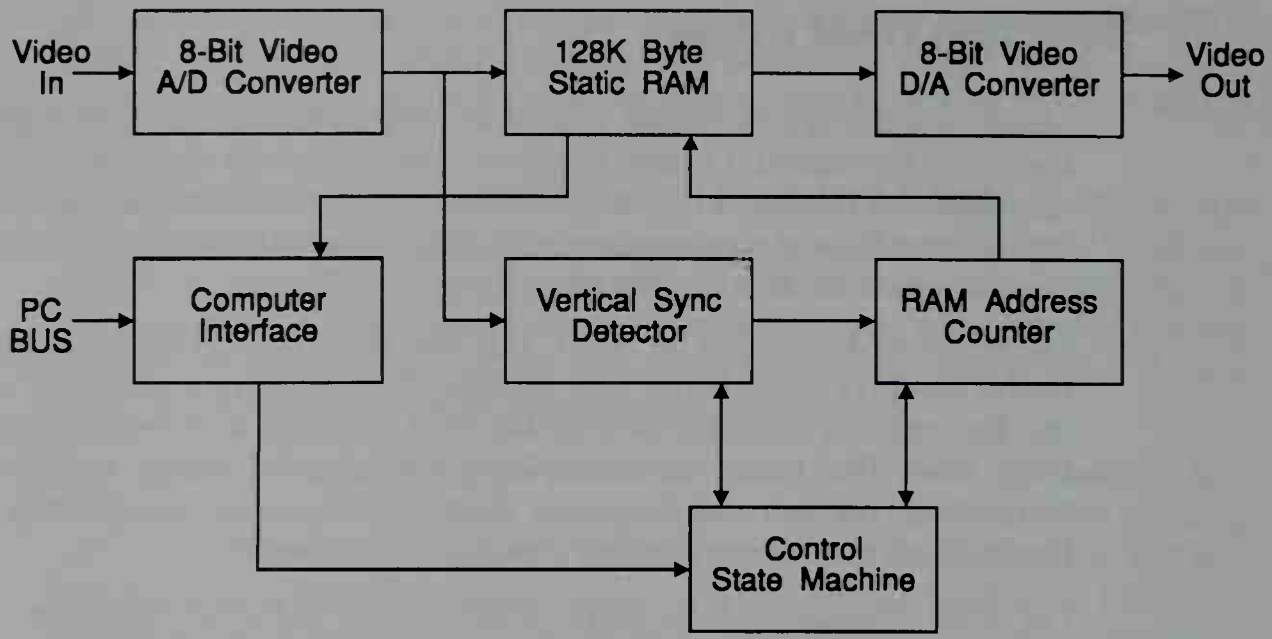


Figure 12.3 Video frame grabber block diagram

interval of the field is detected, the frame grabber begins sampling digitized scan lines and writing them to the four 32K-byte onboard static RAMs. When the image has been completely read, the contents of the RAM chips can be either sent to the D-A converter to reconstruct a frozen video image or be accessed by the controlling software for storage in a BMP file.

Implementation

The first implementation decision made about this design, after choosing the video and RAM chips, was that it would be implemented in simple 20-pin PAL-type devices. This meant that the design would have to be optimized for a limited amount of I/O resources. As the design evolved, many fundamental design changes and trade-offs were made to fit the design into the constrained architectures of this class of devices.

The frame grabber circuit was developed using the PEEL 18CV8 electrically erasable PLDs produced by ICT (International CMOS Technology) and Gould. The 18CV8 is similar in architecture to the widely used 22V10, but has only 20 pins. Architectural features of the 18CV8 devices (their programmable output polarity and banked reset terms) are used to advantage in this design. With slight design modifications, the design could fit into simpler 20-pin PAL devices: one reasonable alternative would be the GAL 16V8 from Lattice, National Semiconducter, and other manufacturers. The simpler fixed-polarity 16R8 could also be used if the active levels of some of the signals (in particular the counter signals) were inverted.

Design Method

Using the block diagram as a guide, the circuit was developed. The design was functionally partitioned, and the various modules were initially developed independently. Some of these modules required a single PLD, while others required multiple PLDs. As mentioned previously, electrically erasable devices were used during development. This allowed design changes to be made and tried out with no waste of chips and no need to wait for UV erasure.

One of the many advantages of using erasable PLDs is that test circuitry can be added with little effort during debugging. This feature came in handy when an early version of the design was moved to a system with a faster bus speed. When the prototype board was installed in the faster machine and tested, some of the scan lines that were read back from the frame grabber were offset to the left or right. Because of the many changes in the hardware and software that were made for the higher-speed system, the problem could have been anywhere. Software testing indicated that the RAM address would sometimes fail to increment or would double count.

The problem was successfully diagnosed by comparing the increment address signal with the least significant bit of the address counter chain. This comparator circuit was programmed into one of the erasable PLDs that had an available output and access to both signals. Examination of this test output with a simple logic probe confirmed the problem. The timing problem was corrected by synchronizing the control signals from the bus by means of a simple modification to one of the PLD circuits.

As design-level optimizations were identified, some circuit functions that were initially placed in one module were moved to other modules or spread between modules. The final design is implemented in a total of six PLDs. Three of the devices are used for the RAM address counter and memory chip select logic, while the remaining three are used for bus interface, field detection, and frame grabber control logic.

Control Logic Module

The ABEL source file for the *CONTROL* module (shown in Figures 12.4 and 12.5) describes the state machine that controls the frame grabber

```
Module CONTROL;
title 'Video Frame Grabber Control  Location U002
Michael Holley and David Pellerin   30 Aug 1993'
// Modes:
//   M2   0 = Read Memory    1 = Frame Grab and Playback Mode
//   M1   0 = Live Video     1 = Still Video
//   M0   0 = Hold Address   1 = Step Address

    control                           device  'P18CV8';
    Osc,M2,M1,M0,!DataEna             pin 1,2,3,4,5;
    !IOW,!IOR,VertSync                pin 6,7,8;
    ClkAD,Inc,!RAMOE,!RAMWE           pin 14,13,12,19;
    AdrClr,EnaAD,S1,VertClr           pin 15,16,17,18;
    Inc,S1,VertClr,AdrClr,EnaAD       istype 'dc,reg,buffer';

    Sreg    = [S1,VertClr,AdrClr,EnaAD,Inc];
    Live    = [ 0,   0   ,   0  ,  0  , 0 ];
    GenLock = [ 0,   1   ,   0  ,  0  , 1 ];
    Reset   = [ 0,   1   ,   0  ,  1  , 0 ];
    Wait    = [ 0,   1   ,   1  ,  1  , 0 ];
    Play    = [ 0,   1   ,   1  ,  1  , 1 ];
    Scan    = [ 1,   1   ,   0  ,  0  , 1 ];
    Record  = [ 1,   1   ,   1  ,  0  , 1 ];
    Read    = [ 1,   1   ,   1  ,  1  , 0 ];
    Step    = [ 1,   1   ,   1  ,  1  , 1 ];

Equations
        Sreg.AR  = !M1;
        Sreg.CLK = Osc;
        ClkAD    = Osc;

State_Diagram Sreg
State Live:                " Live video to monitor
        RAMWE = 0;
        RAMOE = 0;
        GOTO Scan;

State Scan:                " Look for start of field
        RAMWE = 0;
        RAMOE = 0;
        IF (VertSync) THEN Record ELSE Scan;

State Record:              " Capture one field in RAM
        RAMWE = !Osc;
        RAMOE = 0;
        IF (!VertSync) THEN Play ELSE Record;

State Play:                " Display field on monitor
        RAMWE = 0;
        RAMOE = 1;
        IF       !M2      THEN Reset
        ELSE IF  VertSync THEN Play
        ELSE                   GenLock;
```

Figure 12.4 CONTROL.ABL (part 1 of 2)

```
State GenLock:              " Wait for next vertical sync
        RAMWE = 0;
        RAMOE = 0;
        IF (VertSync) THEN Play ELSE GenLock;

State Reset:                " Clear address counter
        RAMWE = 0;
        RAMOE = 0;
        IF (M0) THEN Reset ELSE Read;

State Read:                 " PC bus can read or write RAM
        RAMWE = IOW & DataEna;
        RAMOE = IOR & DataEna;
        IF       M2   THEN Play
        ELSE IF  !M0  THEN Read
        ELSE               Step;

State Step:                 " Increment to next address
        RAMWE = 0;
        RAMOE = 0;
        GOTO Wait;

State Wait:                 " Wait for M0 to go low
        RAMWE = 0;
        RAMOE = 0;
        IF M0 THEN Wait ELSE Read;
Test_Vectors
([Osc,M2,M1,M0,VertSync,IOR,IOW,DataEna] -> [Sreg   ,RAMOE,RAMWE]);
 [.C., 1, 0, 0,    0    , 0 , 0 ,    0   ] -> [Live    ,  0  ,  0   ];
 [.C., 1, 0, 0,    0    , 0 , 0 ,    0   ] -> [Live    ,  0  ,  0   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [Scan    ,  0  ,  0   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [Scan    ,  0  ,  0   ];
 [.C., 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Record  ,  0  ,  1   ];
 [ 0 , 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Record  ,  0  ,  1   ];
 [ 1 , 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Record  ,  0  ,  0   ];
 [.C., 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Record  ,  0  ,  1   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [Play    ,  1  ,  0   ];
 [.C., 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Play    ,  1  ,  0   ];
 [.C., 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Play    ,  1  ,  0   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [GenLock,  0  ,  0   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [GenLock,  0  ,  0   ];
 [.C., 1, 1, 0,    1    , 0 , 0 ,    0   ] -> [Play    ,  1  ,  0   ];
 [.C., 0, 1, 1,    0    , 0 , 0 ,    0   ] -> [Reset   ,  0  ,  0   ];
 [.C., 0, 1, 1,    0    , 0 , 0 ,    0   ] -> [Reset   ,  0  ,  0   ];
 [.C., 0, 1, 0,    0    , 0 , 0 ,    0   ] -> [Read    ,  0  ,  0   ];
 [.C., 0, 1, 0,    0    , 0 , 0 ,    1   ] -> [Read    ,  0  ,  0   ];
 [.C., 0, 1, 0,    0    , 1 , 0 ,    1   ] -> [Read    ,  1  ,  0   ];
 [.C., 0, 1, 0,    0    , 0 , 1 ,    1   ] -> [Read    ,  0  ,  1   ];
 [.C., 0, 1, 0,    0    , 1 , 0 ,    0   ] -> [Read    ,  0  ,  0   ];
 [.C., 0, 1, 1,    0    , 0 , 0 ,    0   ] -> [Step    ,  0  ,  0   ];
 [.C., 0, 1, 1,    0    , 0 , 0 ,    0   ] -> [Wait    ,  0  ,  0   ];
 [.C., 0, 1, 1,    0    , 0 , 0 ,    0   ] -> [Wait    ,  0  ,  0   ];
 [.C., 0, 1, 0,    0    , 0 , 0 ,    0   ] -> [Read    ,  0  ,  0   ];
 [.C., 1, 1, 0,    0    , 0 , 0 ,    0   ] -> [Play    ,  1  ,  0   ];
 End
```

Figure 12.5 CONTROL.ABL (part 2 of 2)

circuit. The state machine controller has nine possible states. While these nine states could have been expressed using only four state bits, this would have meant that an additional six outputs would have been required to control the other frame grabber modules, for a total of ten outputs. Since 20-pin PAL devices have, at most, eight outputs, this wouldn't allow the state machine to be implemented in a single such device. Since four of the control outputs are registered (*VertClr*, *AdrClr*, *EnaAD*, and *Inc*), they can be combined with the state memory bits. The *RAMOE* and *RAMWE* outputs must remain asynchronous.

Using the four existing registered outputs for state bits wasn't sufficient to support the required state values (states *Play* and *Step* required the same specific values for their outputs, as did *Read* and *Wait*), so a fifth state bit output was added. The *S1* bit was used to distinguish between the ambiguous states. The total number of outputs required then became seven, which fit into a 20-pin PLD with one output left over. This extra PLD output was later used as a delay element for the video chip's clock. The various states of the frame grabber controller are shown as a state graph in Figure 12.6.

The initial state of the machine is *Live*. During this state, the image is routed directly through the video chip, and the RAM and address counters are disabled. The *VertClr* is asserted in this state, resulting in the clearing of the field detector. The state machine is reset by an external signal, *M1*, which is controlled by the PC software. The use of an explicit reset signal simplifies the design of the state machine and allows the frame grabber to be reset at any time.

The machine remains in state *Live* as long as the *M1* mode control bit remains low. When *M1* goes high, the machine advances to the *Scan* state. In this state, the field detector becomes active, and the machine loops until the *VertSync* input is asserted by the field detector, indicating that the vertical blanking interval has been detected. To save inputs to the controller, the *ATC* (address terminal count) signal is not used directly, but is instead monitored through the *VertSync* input generated in the *Field* module.

When the start of the image is indicated, the machine advances to state *Record* to begin capturing and storing video samples. The *ATC* input from the *RAMCOM* module indicates that the RAM is full. When this occurs, the machine advances to the *Play* mode.

In the *Play* mode, the A-D is disabled and the RAM output is instead fed to the D-A section of the video chip. The address counter continues to operate, so the saved image (missing five scan lines) is written to the D-A. At the conclusion of the partial field, the state machine advances to the *GenLock* state, where it waits for a vertical blanking interval to

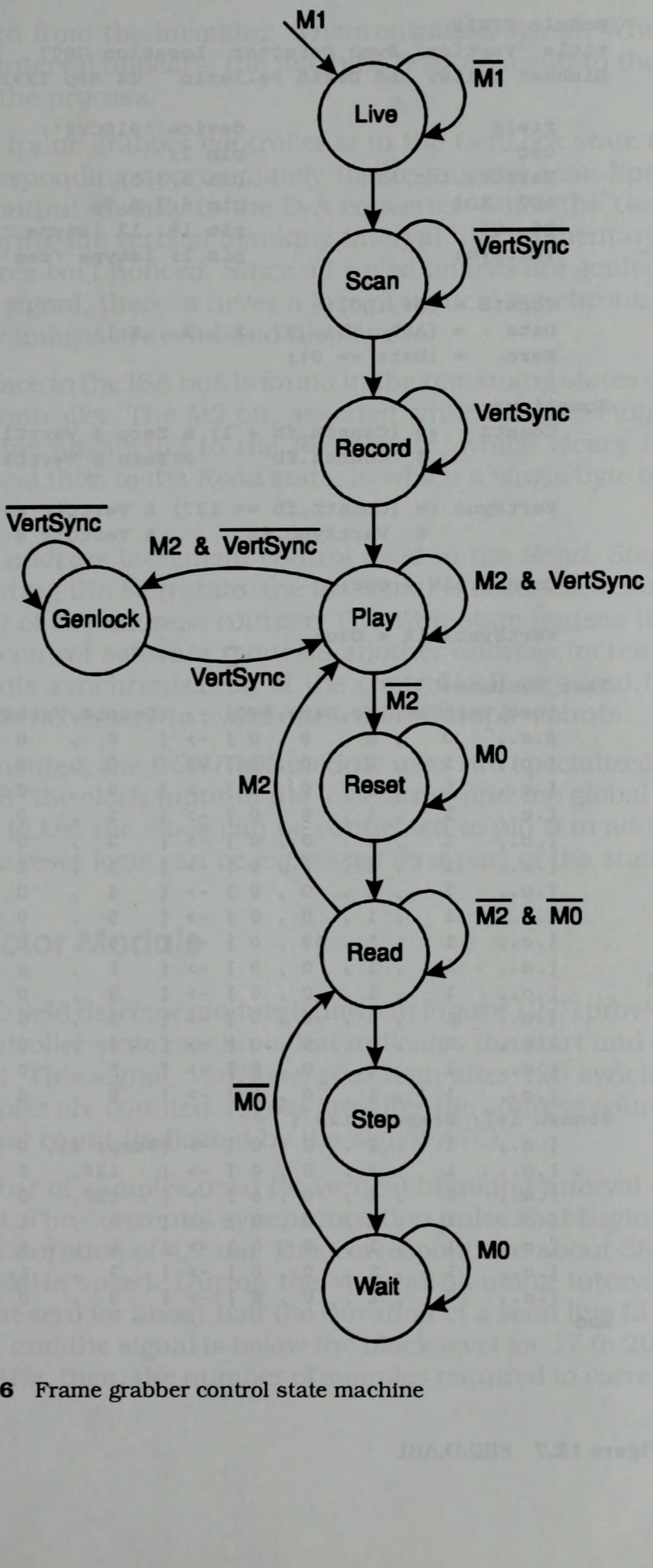


Figure 12.6 Frame grabber control state machine

```
module FIELD
title 'Vertical Sync Detector  Location U007
Michael Holley and David Pellerin   14 Sep 1993'

    field                       device 'P18CV8';
    Osc                         pin 1;
    VertClr,Inc,ATC             pin 2,3,5;
    AD7..AD4                    pin 6,7,8,9;
    Q6..Q0                      pin 18..12 istype 'reg';
    VertSync                    pin 19 istype 'reg';

    CountZ = [Q6..Q0];
    Data   = [AD7..AD4,.X.,.X.,.X.,.X.];
    Zero   = (Data == 0);

Equations
    CountZ   := (CountZ.fb + 1) & Zero & VertClr & !ATC &  Inc
              #  CountZ.fb      & Zero & VertClr & !ATC & !Inc;

    VertSync := (CountZ.fb == 127) & VertClr & !ATC
              #  VertSync.fb       & VertClr & !ATC;

    CountZ.clk = Osc;

    VertSync.clk = Osc;

Test_Vectors
   ([Osc,VertClr,Inc,Data,ATC] -> [CountZ,VertSync])
    [.c.,   0   , 0 ,  0 , 0 ] -> [   0  ,   0    ];
    [.c.,   1   , 0 ,  0 , 0 ] -> [   0  ,   0    ];
    [.c.,   1   , 0 ,  0 , 0 ] -> [   0  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   1  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   2  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   3  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   4  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   5  ,   0    ];
    [.c.,   1   , 1 , 33 , 0 ] -> [   0  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   1  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   2  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   3  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   4  ,   0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   5  ,   0    ];
    [.c.,   1   , 1 ,  0 , 1 ] -> [   0  ,   0    ];
@const i=1; @repeat 125 {
    [.c.,   1   , 1 ,  0 , 0 ] -> [@expr i;, 0 ];@const i=i+1;}
    [.c.,   1   , 1 ,  0 , 0 ] -> [  126,    0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [  127,    0    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   0  ,   1    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   1  ,   1    ];
    [.c.,   1   , 1 ,  0 , 0 ] -> [   2  ,   1    ];
    [.c.,   0   , 1 ,  0 , 0 ] -> [   0  ,   0    ];
End
```

Figure 12.7 FIELD.ABL

be detected from the incoming synchronization signal. When the next blanking interval appears, the machine returns again to the *Play* state to repeat the process.

While the frame grabber controller is in the *GenLock* state (a period of time corresponding approximately to the missing scan lines), the live image is output directly to the D-A converter. Since the *GenLock* state occurs during the vertical blanking interval, the momentary change in video source isn't noticed. Since all saved images are genlocked with a live video signal, there is never a loss of vertical synchronization, even when new images are read and displayed.

The interface to the ISA bus is found in the remaining states of the frame grabber controller. The *M2* bit, asserted while in state *Play*, results in the machine advancing to the *Reset* state, which clears the address counter, and then to the *Read* state, in which a single byte of RAM data is accessed.

M0 is the address increment control used in the *Read*, *Step*, and *Wait* states. During the *Step* state, the *Inc* signal is asserted, resulting in the advancing of the address counter. The *Wait* state pauses the machine until the control software requests another address increment via the *M0* bit. This synchronization of the control software and the address counters is necessary to avoid missed or multiple counts.

As implemented, the *CONTROL* module uses two specialized features of the 18CV8: the clock input to the logic array and the global reset term. For other PLDs, the clock can be connected to pin 9 in addition to pin 1, and the reset logic can be expressed as a part of the state diagram.

Field Detector Module

The *FIELD* field detector module (shown in Figure 12.7) provides a signal to the controller state machine that indicates the start and end of each video field. This signal, *VertSync*, goes high after 128 synchronization-level samples are counted. It goes low after the address counter reaches its terminal count (indicated by the signal *ATC*).

The number of samples used for vertical blanking interval detection is important. The horizontal synchronization pulse that begins each scan line has a duration of 4.7 ms. This corresponds to about 38 samples at our 8.00-MHz speed. During the vertical blanking interval the video signal is at zero for about half the duration of a scan line (31 ms or 248 samples), and the signal is below the black level for 17 to 20 scan lines. At 8.00-MHz, then, the number of samples required to correctly identify

```
Module Count128
Title '7-bit counter with registered carry out   Location U006
Michael Holley and David Pellerin    14 Sep 1993'

    Count128             device 'P18CV8';
    Osc,Inc,AdrClr       pin 1,3,2;
    A6..A0               pin 18..12 istype 'reg';
    CarryA               pin 19 istype 'reg';

    H,L,Z,X,C = 1, 0, .Z., .X., .C.;

    CountA    = [A6..A0];

Equations

    CountA := (CountA.fb + 1)    &  AdrClr &  Inc       " Inc
            #  CountA.fb         &  AdrClr & !Inc       " Hold
            #  0                 & !AdrClr;             " Clear

    CarryA := (CountA.fb == 126) &  AdrClr &  Inc       " Carry
            #  CarryA.fb         &  AdrClr & !Inc       " Hold
            #  0                 & !AdrClr;             " Clear

    [CountA,CarryA].clk = Osc;

test_vectors
   ([Osc,AdrClr,Inc] -> [CarryA,CountA])
    [ C ,  0   , 1 ] -> [  0   ,   0  ];
    [ C ,  1   , 1 ] -> [  0   ,   1  ];
    [ C ,  1   , 1 ] -> [  0   ,   2  ];
    [ C ,  1   , 1 ] -> [  0   ,   3  ];
    [ C ,  1   , 1 ] -> [  0   ,   4  ];
    [ C ,  1   , 1 ] -> [  0   ,   5  ];
    [ C ,  1   , 0 ] -> [  0   ,   5  ];
    [ C ,  1   , 1 ] -> [  0   ,   6  ];
    [ C ,  0   , 1 ] -> [  0   ,   0  ];
@const i=1; @repeat 123 {
    [ C ,  1   , 1 ] -> [  0   , @expr i; ]; @const i=i+1;
}
    [ C ,  1   , 1 ] -> [  0   , 124  ];
    [ C ,  1   , 1 ] -> [  0   , 125  ];
    [ C ,  1   , 1 ] -> [  0   , 126  ];
    [ C ,  1   , 0 ] -> [  0   , 126  ];
    [ C ,  1   , 1 ] -> [  1   , 127  ];
    [ C ,  1   , 0 ] -> [  1   , 127  ];
    [ C ,  1   , 1 ] -> [  0   ,   0  ];
    [ C ,  1   , 1 ] -> [  0   ,   1  ];
end
```

Figure 12.8 COUNT128.ABL

```
Module Count64
Title '6 Bit Counter with Carry Out Location U005
Michael Holley and David Pellerin   14 Sep 1993'

    Count64                     device 'P18CV8';
    Osc,Inc,AdrClr,CarryA       pin 1,3,2,9;
    A12..A7                     pin 18..13 istype 'reg';
    CarryB                      pin 19 iwtype 'com';

    H,L,Z,X,C = 1, 0, .Z., .X., .C.;

    CountB    = [A12..A7];

Equations

    CountB := (CountB.fb + 1) &  AdrClr &  Inc &  CarryA  " Inc
            #  CountB.fb      &  AdrClr &  Inc & !CarryA  " Hold
            #  CountB.fb      &  AdrClr & !Inc            " Hold
            #  0              & !AdrClr;                  " Clear

    CarryB  = (CountB.fb == 63) &  AdrClr &  Inc &  CarryA;

    CountB.clk = Osc;

test_vectors
   ([Osc,AdrClr,Inc,CarryA] -> [CarryB,CountB])
    [ C ,  0   , 1 ,   1  ] -> [  0   ,   0  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   1  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   2  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   3  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   4  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   5  ];
    [ C ,  1   , 0 ,   1  ] -> [  0   ,   5  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   6  ];
    [ C ,  0   , 1 ,   1  ] -> [  0   ,   0  ];
@const i=1; @repeat 60 {
    [ C ,  1   , 1 ,   1  ] -> [  0   ,@expr i;]; @const i=i+1;
}
    [ C ,  1   , 1 ,   1  ] -> [  0   ,  61  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,  62  ];
    [ C ,  1   , 1 ,   1  ] -> [  1   ,  63  ];
    [ C ,  1   , 0 ,   1  ] -> [  0   ,  63  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   0  ];
    [ C ,  1   , 0 ,   1  ] -> [  0   ,   0  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   1  ];
    [ C ,  1   , 1 ,   1  ] -> [  0   ,   2  ];

end
```

Figure 12.9 COUNT64.ABL

```
Module RAMCON
title 'RAM Chip Enable Control    Location U004
Michael Holley and David Pellerin  14 Sep 1993'

    ramcon                    device 'P18CV8';
    Osc,AdrClr,Inc            pin 1,2,3;
    CarryB                    pin 9;
    A14,A13,ATC               pin 18,13,12 istype 'reg';
   !CE3,!CE2,!CE1,!CE0        pin 17,16,15,14 istype 'reg';
    CE      = [CE3,CE2,CE1,CE0];
    CEshift = [CE2,CE1,CE0,CE3];
    CountC  = [A14,A13];
    CarryC  = (CountC.fb == 3) & CarryB;

Equations
    CountC := (CountC.fb + 1) &  AdrClr &  Inc &  CarryB   " Count
            #  CountC.fb      &  AdrClr &  Inc & !CarryB   " Hold
            #  CountC.fb      &  AdrClr & !Inc             " Hold
            #  0              & !AdrClr;                   " Clear

    CE     := CEshift.fb      &  AdrClr &  Inc &  CarryC   " Shift
            # CE.fb           &  AdrClr &  Inc & !CarryC   " Hold
            # CE.fb           &  AdrClr & !Inc             " Hold
            # [0,0,0,1]       & !AdrClr;                   " Clear

    ATC    := (CE.fb == ^b1000) & (CountC.fb == 3)
            & Inc & AdrClr & CarryB;

    [CountC,CE,ATC].clk = Osc;

test_vectors
([Osc,AdrClr,Inc,CarryB] -> [CountC,CE3,CE2,CE1,CE0,ATC])
 [.C.,  0   , 1 ,    1  ] -> [   0   , 0 , 0 , 0 , 1 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   1   , 0 , 0 , 0 , 1 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   2   , 0 , 0 , 0 , 1 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   3   , 0 , 0 , 0 , 1 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   0   , 0 , 0 , 1 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   1   , 0 , 0 , 1 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   2   , 0 , 0 , 1 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   3   , 0 , 0 , 1 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   0   , 0 , 1 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   1   , 0 , 1 , 0 , 0 , 0 ];
 [.C.,  1   , 0 ,    1  ] -> [   1   , 0 , 1 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   2   , 0 , 1 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   3   , 0 , 1 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   0   , 1 , 0 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    0  ] -> [   0   , 1 , 0 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   1   , 1 , 0 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   2   , 1 , 0 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   3   , 1 , 0 , 0 , 0 , 0 ];
 [.C.,  1   , 1 ,    1  ] -> [   0   , 0 , 0 , 0 , 1 , 1 ];
 [.C.,  1   , 1 ,    1  ] -> [   1   , 0 , 0 , 0 , 1 , 0 ];
End
```

Figure 12.10 RAMCON.ABL

the vertical blanking interval is between about 40 and 240 samples. We chose 128 for this design, since a 128-state counter is trivial to design.

Address Counter and RAM Control

The frame grabber, as described earlier, must sample and store 131,072 bytes of data. The data must be written into the static RAM chips, of which there are four in this design. To do this, the frame grabber requires an address counter and chip select logic. An address counter that increments beyond 256 states is impossible to construct using only the eight outputs found in a 20-pin PAL (in fact, a counter with more than 128 states is impossible to implement in most 20-pin PALs because of product term limitations), so it was necessary to split the frame grabber's large address counter into three smaller counters. Two of the counters are found in modules *COUNT128* and *COUNT64*; the third and smallest of the counters is combined with the chip select logic in the *RAMCON* module. The three ABEL source files for these modules are shown in Figures 12.8, 12.9 and 12.10.

The three counters are all described using a common format, and all have three modes of operation: increment, hold, and clear. The *AdrClr* input, when asserted, causes all counters to reset. In addition, the chip select (consisting of the outputs *CE3* through *CE0*) is reset to its initial state, where *CE0* is asserted. The counter segments will hold whenever their carry inputs (*CarryA*, *CarryB*, or *CarryC*) are asserted by the previous counter segment.

The *Inc* input is used during read and write operations when the memory addressing is being controlled from the PC-resident software. At these times, the counter must operate in a single-step mode, since the software can't operate at the speed of the frame grabber circuitry.

Chip select is provided by using a four-bit shifter in the *RAMCON* module (Figure 12.10). This eliminates the need for the most significant two bits of the address counter. If a single 128K RAM chip were used, the chip select shifter logic would be eliminated by a larger (17-bit) counter chain.

PC Interface Module

The final module of the frame grabber circuit provides the interface to and from the control software. The *PCHOST* module decodes the hexadecimal addresses 300 to 31F from the bus (this is the address range of the IBM prototyping card). Writing to addresses 310 to 317 (from the

```
Module PCHOST
title 'PC Interface and I/O address decode  Location U001
Michael Holley and David Pellerin  14 Sep 1993'

    pchost                      device 'P18CV8';
    A9,A8,A7,A6,A5,A4           pin 3,4,5,6,7,8;
    A2,A1,A0                    pin 13,12,9;
    !AEN,!IOR,!IOW              pin 2,19,1;
    M2,M1,M0                    pin 16,15,14 istype 'reg';
    !DataEna                    pin 18 istype 'com';

    Addr    = [A9..A4,.X.,.X.,.X.,.X.];
    Address = [A9..A4,.X.,A2..A0];

    CtrlPort = (AEN & (Addr == ^h310));
    DataPort = (AEN & (Addr == ^h300));

Equations

    [M2..M0]     :=  CtrlPort & [A2..A0]          "Set  M2..M0
                   # !CtrlPort & [M2..M0].fb;     "Hold M2..M0

    [M2..M0].clk = !IOW;
    [M2..M0].oe  = !OE1;

    DataEna        = IOW & DataPort & !M2.fb
                   # IOR & DataPort & !M2.fb;

test_vectors
   ([IOW,IOR,AEN,Address] -> [M2,M1,M0,DataEna])
    [.c., 0 , 1 ,  ^h310] -> [ 0, 0, 0,   0   ];
    [.c., 0 , 0 ,  ^h311] -> [ 0, 0, 0,   0   ];
    [.c., 0 , 1 ,  ^h311] -> [ 0, 0, 1,   0   ];
    [.c., 0 , 1 ,  ^h312] -> [ 0, 1, 0,   0   ];
    [.c., 0 , 1 ,  ^h313] -> [ 0, 1, 1,   0   ];
    [.c., 0 , 1 ,  ^h324] -> [ 0, 1, 1,   0   ];
    [.c., 0 , 1 ,  ^h314] -> [ 1, 0, 0,   0   ];
    [.c., 0 , 1 ,  ^h315] -> [ 1, 0, 1,   0   ];
    [.c., 0 , 1 ,  ^h316] -> [ 1, 1, 0,   0   ];
    [.c., 0 , 1 ,  ^h317] -> [ 1, 1, 1,   0   ];
    [ 0 , 1 , 1 ,  ^h300] -> [ 1, 1, 1,   0   ];
    [ 1 , 0 , 1 ,  ^h300] -> [ 1, 1, 1,   0   ];
    [.c., 0 , 1 ,  ^h310] -> [ 0, 0, 0,   0   ];
    [ 0 , 1 , 1 ,  ^h300] -> [ 0, 0, 0,   1   ];
    [ 1 , 0 , 1 ,  ^h300] -> [ 0, 0, 0,   1   ];
    [ 0 , 0 , 1 ,  ^h300] -> [ 0, 0, 0,   0   ];
    [ 0 , 1 , 0 ,  ^h300] -> [ 0, 0, 0,   0   ];
end
```

Figure 12.11 PCHOST.ABL

frame grabber control software) sets the state of the three mode control bits (*M2*, *M1*, and *M0*). Address 300 contains the byte-wide data that are read from or written to the current address of the frame grabber's RAM.

Note that there is no provision for writing to the frame grabber's memory randomly; all memory read and write operations must occur sequentially, regardless of whether the memory is being accessed by the frame grabber controller or from the interface module.

The *PCHOST* module uses pin 1 of the 18CV8 (*IOW*) both for clocking of the mode control bits (*M2* through *M0*), and as a combinational input for controlling *DataEna*. Unlike most PLDs, the 18CV8 allows the global clock pin to be fed into the logic array for applications such as this. If a device such as GAL 16V8 were used, the *IOW* would have to be split between two input pins. (In this case, pin 17 is unused and could therefore be used for this purpose). If a simpler device such as the 16R4 were used, the RAM write feature would have to be omitted. Figure 12.11 shows the ABEL source files for the *PCHOST* module.

The Complete Frame Grabber

A schematic of the complete video frame grabber board is shown in Figures 12.12 and 12.13. As implemented, the circuit requires six PLDs, three fixed-function TTL chips, the four 32K static RAMs, and the video chip. If more complex PLDs had been used, the total number of packages could have been reduced. For this version of the design, it was decided that the six low-cost PLDs were the best compromise.

12.3 FRAME GRABBER CONTROL SOFTWARE

The PC-resident frame grabber software was developed using Microsoft C; complete source listings can be found in Appendix D. The software accesses the video frame grabber at hexadecimal addresses 310 through 317. These are the addresses used to trigger the mode control bits *M0*, *M1*, and *M2*.

The video frame grabber control program reads and writes the frame grabber's onboard memory using PC memory address 300 as the data address for each image sample. The images (either live or frozen, depending on the operation selected) are sent to the frame grabber's video output jack, but are not displayed on the PC's monitor. Since all displayed images must be genlocked with a live image, the video input jack of the frame grabber board must at all times be connected to a valid

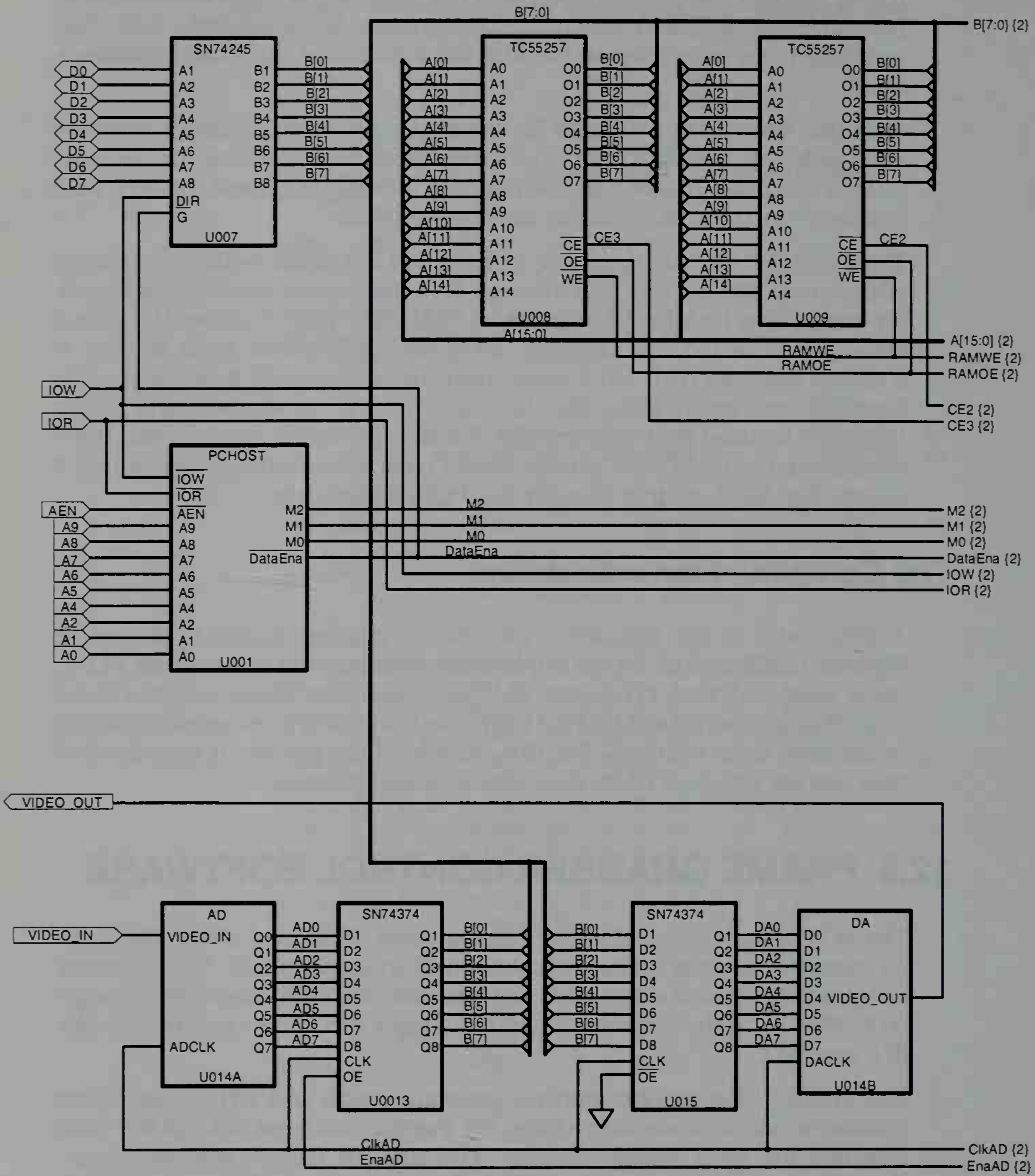


Figure 12.12 Video frame grabber schematic (part 1 of 2)

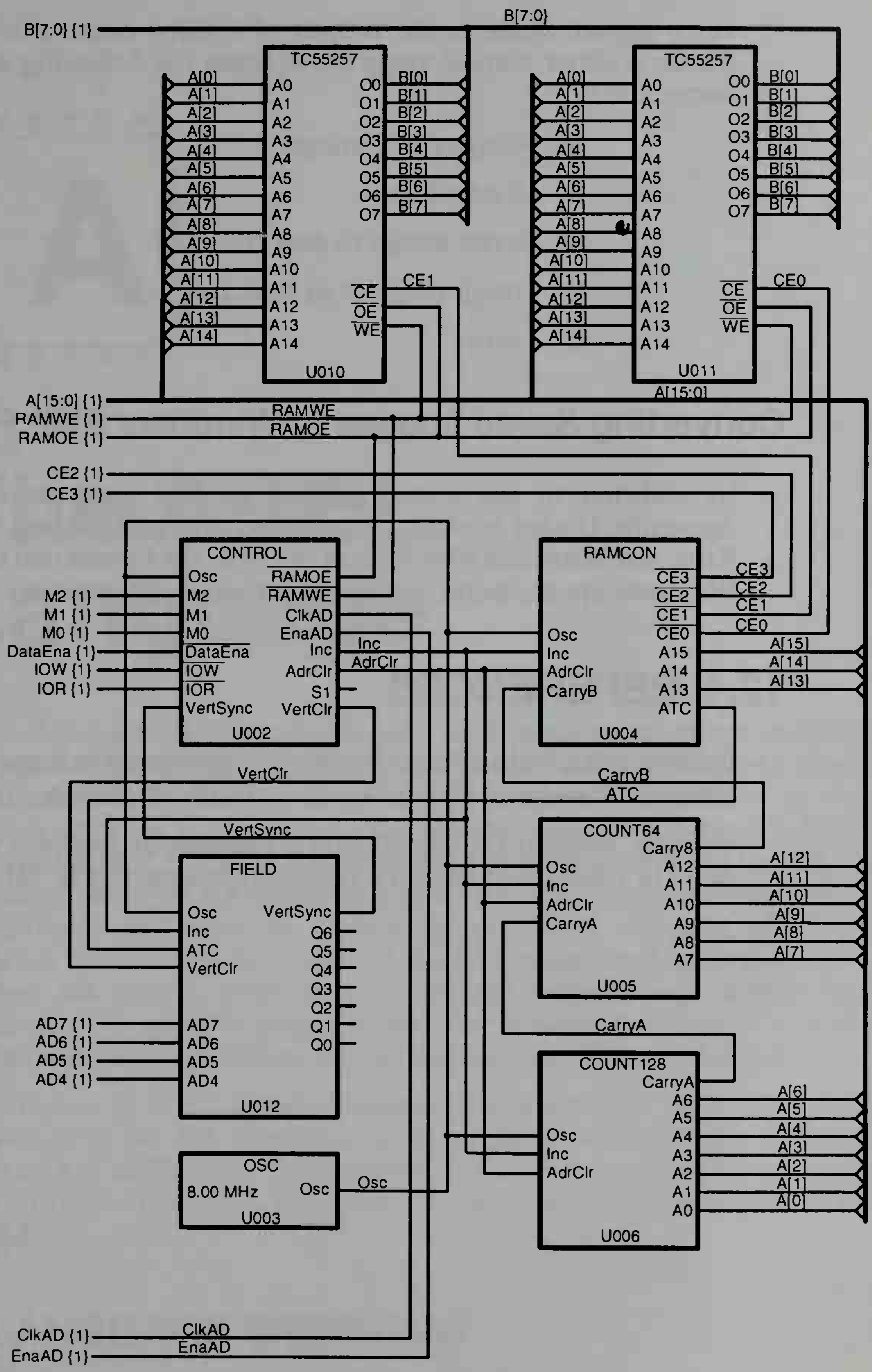


Figure 12.13 Video frame grabber schematic (part 2 of 2)

video signal, such as the output of a video camera or television. The frame grabber control program accepts the following single-character commands:

L	display live image
F	freeze image
W	write image to disk file
R	read image from disk file
Q	quit

Converting Saved Images to Windows BMP Format

In addition to the frame grabber control program described here, Appendix D also contains a program that will convert the images to a fixed-size Windows BMP format file. This file format can be read by many Windows applications, including Windows PaintBrush.

12.4 REFERENCES

Gilbert, Alfie, *Video Frame Grabber*, Programmable Logic Handbook, 4th edition, Monolithic Memories, Inc., Santa Clara, CA, 1985.

Stanley, William D., *Digital Signal Processing*, Reston Publishing Company (a Division of Prentice Hall), Englewood Cliffs, NJ, 1975.

A

The ABEL Language

ABEL (Advanced Boolean Expression Language) is a language optimized for, but not limited to, the description of circuits to be implemented in PLDs and FPGAs. Version 1.0 of the language was introduced by Data I/O Corporation in 1984. This first version was intended solely for the description of PLD-based logic circuits, and the simplicity of the original language reflected the relative simplicity of the programmable devices available at that time. As devices became more complex, ABEL was updated accordingly. In version 4 the language was significantly overhauled, allowing it to be used for general-purpose logic design independent of a specific target device or technology. Version 5 added significant new capabilities for the description of larger designs.

The version of ABEL design software that is provided with this book is version 5.0. All the language features of version 5.0 are available. (Versions of ABEL later than version 5.0 include support for hierarchy and multiple modules, but these features are not described in this book.)

A.1 LANGUAGE OVERVIEW

Hardware design using ABEL is similar in many respects to software design using languages such as Pascal or C. ABEL designs are entered with a text editor and are then compiled into an internal form that may be merged with other design elements, optimized, and executed (via simulation) before implementation in hardware.

ABEL differs from software programming languages in that it is used to describe functions that are inherently parallel. All statements in an ABEL design may be thought of as being executed at the same time. This is particularly important to realize when describing sequential circuits. The sequential operation of a circuit is never a function of the order in which ABEL language statements describing that circuit are entered.

ABEL, like other harware description languages (HDLs), provides different textual entry formats that can be combined as needed to meet the specific requirements of the design. The primary description methods available in ABEL are high-level equations, truth tables, and state diagrams. Within an ABEL module, one or more of these three description methods are used to completely specify the desired circuit.

ABEL Modules

The basic design unit in ABEL is the module. A typical PLD-based design consists of a single module, while more complex FPGA-based designs may consist of a number of modules that are combined during the device fitting process.

An ABEL module consists of three primary types of items: declarations, logic descriptions, and test vectors. To describe a design, an ABEL module must have signal (PIN or NODE) declarations and at least one logic description (equations, truth table, or state diagram). Test vectors are optional.

The declarations, logic descriptions, and test vectors can appear in any order in the ABEL source file, as long as all referenced design elements (signals, constants, macros, and so on) are declared before being used.

A single ABEL source file can contain multiple ABEL modules. In this case, the modules will be processed by the language compiler in the order encountered. Since there are no hierarchy features in the version of ABEL supplied with this book, there is no linkage between multiple ABEL modules located in the same ABEL file, and the results of processing will be the same as if the modules had been processed separately from individual source files.

A sample ABEL source file is shown in Figure A.1. This source file includes one DECLARATIONS section, one EQUATIONS section, and

```
module mux12t4
title '12 to 4 multiplexer
Michael Holley and David Pellerin'

declarations

    a3..a0     PIN;
    b3..b0     PIN;
    c3..c0     PIN;
    s1,s0      PIN;
    y3..y0     PIN ISTYPE 'com';

    A      = [a3..a0];
    B      = [b3..b0];
    C      = [c3..c0];
    Y      = [y3..y0];
    Select = [s1,s0];

    X,H,L  = .x.,!0,0;

equations

    Y = (Select == 1) & A
      # (Select == 2) & B
      # (Select == 3) & C;

test_vectors ([Select, A, B, C] -> Y)
              [  1   , 1, X, X] -> 1; "Select 1, A
              [  1   ,10, X, X] ->10;
              [  1   , 5, X, X] -> 5;

              [  2   , X, 3, X] -> 3; "Select 2, B
              [  2   , X, 7, X] -> 7;
              [  2   , X,15, X] ->15;

              [  3   , X, X, 8] -> 8; "Select 3, C
              [  3   , X, X, 9] -> 9;
              [  3   , X, X, 1] -> 1;
end
```

Figure A.1 Design file for a 12-to-4 multiplexer

one TEST_VECTORS section. The DECLARATIONS section includes pin declarations and set declarations that are used in the subsequent EQUATIONS and TEST_VECTOR sections. This sample ABEL source file describes a 12-to-4 multiplexer. The specifics of this design (and many others) are discussed in Chapters 6 through 12.

Basic Syntax

ABEL is a free-format language, meaning that the exact positioning of keywords and other source file elements is not significant to the language compiler. Spaces, tabs, newlines, and other white-space characters are ignored except when they serve to delimit the various elements of a design description. (Exceptions to this are found in the @IFB, @IFNB, @IFIDEN, and @IFNIDEN directives, which are sensitive to white-space characters.) Individual lines in the source file may not exceed 150 characters.

Keywords, identifiers, and numbers must be separated from other syntax elements by at least one white-space character (space, tab, or newline), or by a nonalphanumeric delimiter such as a comma, colon, or semicolon, or by an expression operator, right or left parenthesis or bracket, or any other nonalphanumeric character.

Keywords

Figure A.2 lists the keywords recognized by the ABEL compiler. These keywords are reserved and may not be used as identifiers in an ABEL source file. Keywords may be entered in either upper, lower, or mixed case.

Identifiers

Identifiers are used to give unique names to modules, signals, constants, macros, and dummy arguments within macros. Identifiers may be entered in upper, lower, or mixed case, but are case sensitive when processed. This means, for example, that the identifier *FRED* is not the same as the identifier *Fred*. An identifier consists of a string of alphanumeric characters that does not exceed 32 characters in length. Alphanumeric characters include the uppercase letters A through Z, lower case letters a through z, the numeric characters 0 through 9, and the character _ (the underscore character). Identifiers must not begin with a numeric character.

A.2 DECLARATIONS

Declarations are entered at the beginning of the ABEL module (after the MODULE, TITLE, and FLAGS statements and before any logic descrip-

CASE	DECLARATIONS
DEVICE	ELSE
ENABLE	ENDCASE
END	EQUATIONS
ENDWITH	FUSES
FLAGS	IF
GOTO	LIBRARY
MACRO	MODULE
NODE	OPTIONS
PIN	STATE
STATE_DIAGRAM	THEN
TITLE	TRACE
TRUTH_TABLE	WHEN
WITH	

Figure A.2 ABEL keywords

tion section) or in a DECLARATIONS section. Design elements such as signals and constants must be declared in a declarations section before they are used in a logic description.

Device Declarations

ABEL allows a target programmable logic device type to be declared in a device declaration. Device declarations are optional. If used, they must appear before any pin or node declarations are made. The following statement declares a device (an 18CV8 configurable PAL device) that will be used for a design:

```
mydevice   DEVICE    'P18CV8';
```

Declaring a device in the ABEL source file does not restrict the design to that particular device, but may imply certain attributes (such as register types and output inversions) that can restrict the ability of device fitters to map the design into alternative device types.

Pin and Node Declarations

ABEL supports two basic types of signals: pins and nodes. A pin is a signal that is accessible outside of the current module (on a device pin, for example), while a node is a signal that may be buried or removed (by collapsing) during the device fitting or synthesis process. All pins and nodes must be declared before they are used. The following statements are examples of pin and node declarations:

```
i1,i2,i3      PIN;                      "Inputs
Sync_pulse    PIN   ISTYPE 'com';       "Comb output
Buf2,Buf1     NODE  ISTYPE 'reg';       "Buried registers
q7..q0        PIN   ISTYPE 'reg';       "Use a range
D3,D2,D1,D0   PIN   2,3,4,5;            "Pin numbers
```

Pin and Node Numbers

Signal declarations may include pin numbers, as shown in the declaration for *D3* through *D0* in the preceding example. When specified, these pin numbers are written to the compiler-generated intermediate file for use by a device fitter. The pin numbers are not otherwise significant to the compiler unless a device name has also been specified for the module. If a device name has been specified, then the compiler will imply signal attributes (such as register types, or the output inversion attributes 'buffer' and 'invert') based on information stored in the ABEL device database.

Inputs and Outputs

Signal declarations in ABEL do not provide any way to explicitly state which signals are inputs, which are outputs, and which are bidirectional. This information is determined by how each signal is used in the design description. In the examples presented in this book, we follow the convention of assigning signal attributes (using an ISTYPE statement) to all signals that will be used as outputs in the design. This helps to make the design more readable, even when the signal attributes may not actually be needed.

Signal Attributes

Output signal declarations may include an ISTYPE modifier or ISTYPE statement. ISTYPEs allow signal attributes to be assigned to signals. An ISTYPE modifier is an ISTYPE keyword and attribute string that follows the PIN or NODE keyword (and pin or node numbers, if any):

```
q3..q0      PIN ISTYPE 'reg';
```

When a list of signals is specified in the declaration (as just shown), the ISTYPE modifier applies to all signals in the list.

An ISTYPE statement is simply another form of the ISTYPE modifier that can be used for signals that have already been defined in a PIN or NODE signal declaration. The previous statement, for example, could have been written as

```
q3..q0    PIN;
q3..q0    ISTYPE 'reg';
```

Multiple attributes may be combined within one ISTYPE statement by separating the attributes by commas within the single quote characters as follows:

```
MyOutput   PIN ISTYPE 'reg,buffer';
```

The following ISTYPE signal attributes are available in ABEL.

ISTYPE 'reg'

The output is registered using a level-sensitive latch or clocked D-type flip-flop and will be described using pin-to-pin logic descriptions. (The := and :> assignment operators are used in logic descriptions to describe pin-to-pin registered outputs.) Whether the design is implemented using clocked flip-flops or level-sensitive latches depends on the type of clocking specified for the design. (The .CLK dot extension described later indicates a clocked flip-flop; the .LE dot extension indicates a latch.) When an output signal of type 'reg' is processed, the corresponding equations produced by the language compiler are pin-to-pin.

ISTYPE 'reg_D'

The output is registered via a clocked D-type flip-flop and will be described using flip-flop stimulus (detailed) logic descriptions. To describe the flip-flop stimulus for this output, you must use the .D signal dot extension. When an output signal of type 'reg_D' is processed as a state diagram state register, the corresponding equations produced by the language compiler will use the .D dot extension. See also the 'invert' and 'buffer' attributes.

ISTYPE 'reg_T'

The output is registered using a clocked toggle (T flip-flop) memory element. T-type outputs must be described using flip-flop stimulus (detailed) logic descriptions. To describe the stimulus for this registered output, you must use the .T signal dot extension. When an output signal of type 'reg_T' is processed as a state diagram state register, the corresponding equations produced by the language compiler will use the .T dot extension. See also the 'invert' and 'buffer' attributes.

ISTYPE 'reg_SR'

The output is registered using a clocked set-reset (SR flip-flop) memory element. SR-type outputs must be described using flip-flop stimulus (detailed) logic descriptions. To describe the stimulus for this type of output, you must use the .S and .R dot extensions. When an output signal of type 'reg_SR' is processed as a state diagram state register, the corresponding equations produced by the language compiler will use the .S and .R dot extensions. See also the 'invert' and 'buffer' attributes.

ISTYPE 'reg_JK'

The output is registered using a clocked JK flip-flop. JK-type outputs must be described using flip-flop stimulus (detailed) logic descriptions. To describe the stimulus for a JK flip-flop, you must use the .J and .K dot extensions. When an output signal of type 'reg_JK' is processed as a state diagram state register, the corresponding equations produced by the language compiler will use the .J and .K dot extensions. See also the 'invert' and 'buffer' attributes.

ISTYPE 'reg_G'

The output is registered using a D flip-flop with a gated clock. G-type outputs must be described using flip-flop stimulus (detailed) logic descriptions. To describe the stimulus for a G-type registered output, you must use the .D and .CE (clock enable) dot extensions. When an output signal of type 'reg_G' is processed as a state diagram state register, the corresponding equations produced by the language compiler will use the .D and .CE dot extensions. See also the 'invert' and 'buffer' attributes.

ISTYPE 'com'

The output is combinational. All combinational outputs are inherently pin-to-pin. When describing the logic for a combinational output, you must use the = or -> assignment operators.

ISTYPE 'buffer'

The registered output signal is not inverted at the output pin. If 'buffer' is specified, it indicates that the signal does not have any inversion between the output of the flip-flop and the actual output pin. A 'buffer' or 'invert' attribute is usually required if you are describing an output using detailed flip-flop stimulus (using the .D, .T, .J/.K or .S/.R dot extensions) or are depending on specific reset or power-up behavior. If you do not specify 'buffer' or 'invert' for outputs that are described using detailed flip-flop stimulus, then the language compiler will produce warning messages and the results of your circuit will depend on how your design is actually implemented in hardware.

ISTYPE 'invert'

The registered output signal is inverted at the output pin. If 'invert' is specified, it indicates that the signal has an inversion between the output of the flip-flop and the actual output pin. A 'buffer' or 'invert' attribute is usually required if you are describing an output using detailed flip-flop stimulus (using the .D, .T, .J/.K or .S/.R dot extensions) or are depending on specific reset or power-up behavior. If you do not specify 'buffer' or 'invert' for outputs that are described using detailed flip-flop stimulus, then the language compiler will produce warning messages and the results of your circuit will depend on how your design is actually implemented in hardware.

ISTYPE 'pos'

The equations for this output will be optimized by the compiler for positive polarity. Note that this attribute does not indicate an active-high signal. The 'pos' and 'neg' attributes are used to control the form of the equations produced by the language compiler and are typically used when you you want to control the behavior of the circuit during don't-care conditions. If the 'pos' or 'neg' attributes are not specified, the language compiler will assume 'pos'. See also the 'neg' and 'dc' attributes.

ISTYPE 'neg'

The equations for this output will be optimized using negative polarity. This attribute does not indicate an active-low signal. When 'neg' is specified for an output, all don't-care conditions (whether implied or specified) will result in the output going high. See also the 'pos' and 'dc' attributes.

ISTYPE 'dc'

The equations for this output will be optimized using don't-cares. Don't-cares are implied whenever incompletely specified truth tables or state diagrams are written, or may be specified directly in don't-care equations (using the ?= or ?:= equation assignment operators.) See also the 'pos' and 'neg' attributes.

ISTYPE 'xor'

This attribute instructs the compiler to preserve one top-level exclusive-OR operator (if there is one available) for the indicated output. If there is no top-level XOR operator in the logic description, then this attribute will have no effect. (The language compiler will not attempt to generate XORs from a non-XOR logic description.)

Output Inversion Considerations

Many of the attributes described define implementation dependencies or assumptions in the design. The 'invert' and 'buffer' attributes, for example, are used to specify the state of the programmable output inverters that are commonly found in PLDs. Note also that there is no attribute indicating that a signal is bidirectional. Bidirectional signals are determined by how they are used in subsequent logic descriptions, not by how they are declared.

Active-low Declarations

Active-low declarations may be used for signals that are more naturally expressed using active-low (low-true) logic. To declare a signal as active low, a ! operator is placed in front of the signal name in the PIN or NODE declaration, as shown in the following example:

```
!Ready, !Init, ClockOut   PIN ISTYPE 'com';
```

This PIN declaration statement declares the combinational output signals *Ready* and *Init* as active low. Wherever these signals are specified in the design descriptions, they will automatically be inverted by the language compiler.

Since the language compiler performs the inversion during processing, there is no difference between a design that is specified using active-low declarations from one that is specified using standard declarations and inverted equations. If you need to preserve information about the active-level of a signal beyond the compilation stage, then you must use signal naming conventions to do so. (One common method is to declare active-low signals with their names prefixed by an underscore character.)

Active-low declarations affect all direct uses of the signal in the subsequent design descriptions. This includes use as an equation output, as an input or feedback, and as a state diagram state register bit. Test vectors are also affected when a signal is declared as active low. An active-low signal specified in a test vector header will result in inverted values in the subsequent test vectors entries. Active-low declarations also affect any use of the signal with pin-to-pin feedback dot extensions. Pin-to-pin feedback dot extensions are used to specify unambiguous pin or register feedback using logic values as they appear on the specified pin (taking into account any fixed pin inversion). The pin-to-pin feedback dot extensions are .PIN and .FB. Signal references that include flip-flop stimulus (detailed) dot extensions are not affected by active-level inversion. These dot extensions include .D, .T, .J, .K, .S, and .R, all reset and preset oriented dot extensions, and the .CLK, .CE, and .OE dot extensions.

Numeric Constants

Numeric constants are used in ABEL to specify integer values. All operations involving numbers in an ABEL description are processed with 128-bit unsigned integer arithmetic. The 128-bit accuracy means that numbers can have a value in the range of 0 to 2^{128} - 1. Numbers are specified in base 10 (decimal) radix unless an alternative @RADIX directive is in effect or the number is preceded by a radix modifier. The radix prefixes are ^b, ^o, ^d, and ^d.

The ^b radix prefix indicates that the number is in binary (base 2). Numbers specified using the ^b radix modifier must consist only of 1 and 0 characters.

The ^o radix prefix indicates that the number is in octal (base 8). Numbers specified using the ^o radix modifier must consist only of the characters 0 through 7.

The ^d radix prefix indicates that the number is in decimal (base 10). This is the default radix, unless an alternative radix has been specified using the @RADIX directive.

The ^h radix prefix indicates that the number is in hexadecimal (base 16). Numbers specified using the ^h radix modifier may consist of the characters 0 through 9, 'a', b', 'c', 'd', 'e', and 'f'. (The alphabetic portions of a hexadecimal number may be entered in either upper- or lowercase.) Note that, due to parsing conflicts, hexadecimal numbers that begin with an alphabet character must be prefixed with ^h even when @RADIX 16 is in effect.

Strings as Numbers

In equations, truth tables, and other logic expressions, a string of characters enclosed in single quotes can be used to represent numbers. In this case, the ASCII code of the characters is used to create a numeric value. When more than one character is concatenated to form a string, each character represents eight bits of numeric data, with the rightmost character in the string representing the least significant eight bits of the number. The following sample declarations demonstrate how strings can be used to represent numbers:

```
Str1 = 'a';      "Evaluates to 97 decimal (61 hex)
Str2 = 'fred';   "Evaluates to 1718773092 (66726564 hex)
```

Since numbers are a maximum of 128 bits in size, the maximum number of characters that can be concatenated to form a number is 16.

A.3 EXPRESSIONS

An ABEL expression consists of one or more expression elements that may be operated on by one or more expression operators. An expression element can consist of a signal, set, number, or special constant. Expressions can themselves be elements of larger expressions, with no limit to the level of nesting. Expressions constructed from these basic types are used throughout ABEL to represent combinational logic functions or arithmetic operations resulting in numeric values. Expressions of arbitrary complexity can be used anywhere in the design description where signal identifiers, sets, numbers, or special constants are required.

Signals

Signals in ABEL can be either pins or nodes and are declared in a DECLARATIONS section before their use in logic descriptions and test vectors. Wherever used, signals can be grouped into sets or operated on directly by expression operators. When signals are operated on, the rules and order of evaluation are the same as for operations performed on single-bit numeric values. The result of an expression that includes signals is always a combinational network. The following ABEL equations demonstrate some common operations using signals:

```
Y1 = !Busy;
Y2 = A & B & (C # D);
Y3 = (A != D);
```

Signal Dot Extensions

Dot extensions are added to signals to specify secondary signals associated with an output. Secondary signals include such things as clocks, resets and presets, output enables, and flip-flop data inputs. Dot extensions can also be used to remove ambiguities when specifying feedback from registered outputs. The following dot extensions are supported in ABEL.

.ACLR

Asynchronous clear (pin-to-pin). This dot extension is used to specify asynchronous clear (reset) logic for the associated output. This dot extension applies to the output pin associated with a flip-flop and, when active, causes the output pin to be set to a state of 0. If the output is inverted, a value of 1 will be loaded into the associated flip-flop to clear the output to 0.

.AP

Asynchronous preset. This dot extension is used to specify asynchronous preset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active, causes the flip-flip to be set to a state of 1. If the output is inverted, then a 0 will appear on the actual output pin.

.AR

Asynchronous reset. This dot extension is used to specify asynchronous reset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active, causes the flip-flip to be set to a state of 0. If the output is inverted, a 1 will appear on the actual output pin.

.ASET

Asynchronous set (pin-to-pin). This dot extension is used to specify asynchronous set (register preset) logic for the associated output. This dot extension applies to the output pin associated with a flip-flop and, when active, causes the output pin to be set to a state of 1. If the output is inverted, a value of 0 will be loaded into the associated flip-flop to set the output to 1.

.CE

Clock enable. This dot extension specifies a clock enable function for a gated clock D-type flip-flop. An equation written for a .CE dot extension implies an AND operation between the .CE and .CLK flip-flop inputs.

.CLK

Clock. This dot extension specifies the source of clocking signals for a flip-flop associated with the specified output signal. The presence of a .CLK dot extension indicates that a flip-flop is associated with the specified output.

.CLR

Synchronous clear (pin-to-pin). This dot extension is used to specify synchronous clear (reset) logic for the associated output. This dot extension applies to the output pin associated with a flip-flop and, when active during a clock edge, causes the output pin to be set to a state of 0. If the output is inverted, a value of 1 will be loaded into the associated flip-flop to clear the output to 0.

.D

D input to a D flip-flop or latch. This dot extension indicates the data to be loaded into the flip-flop during the next rising clock event, or into a level-sensitive latch during its transparent mode. Since this dot extensions specifies the data to be loaded into the flip-flip or latch, any inversion of the output pin (as specified with an 'invert' attribute or as a side effect of the hardware implementation) will result in a reversal of the data appearing on the actual output pin.

.FB

Pin-to-pin register feedback. This dot extension is used on the right side of equations (the R-value) to specify that the signal fed back is to come directly from the associated flip-flip. When .FB is specified, the logic fed back is normalized to the values that are observed on the actual output pin. This means that if the output of the flip-flop is inverted before being routed to the output pin, the .FB feedback will be adjusted automatically to match the pin's value. .FB is typically used in conjunction with the := assignment operator to describe registered functions in terms of pin-to-pin behavior.

.FC

Flip-flop mode control. This dot extension is highly device specific and is used to dynamically control a flip-flop's mode. Some programmable logic devices feature flip-flops that can be dynamically changed from D to JK type, and this dot extension provides direct control over this feature. This dot extension is not recommended for general-purpose logic design.

.J

J data input to a JK flip-flop. This dot extension is one of the two data inputs required for a JK flip-flop. In ABEL, all JK-type flip-flops are assumed to be clocked, so the .J and .K dot extensions must be used in conjunction with a .CLK dot extension to describe the flip-flop behavior. Since this dot extension specifies an operation to be performed directly on the flip-flip, any inversion of the flip-flop's output (as specified with an 'invert' attribute or as a side effect of the hardware implementation) will result in a reversal of the data appearing on the actual output pin.

.K

K data input to a JK flip-flop. (See .J).

.LD

Register load. This dot extension, when active, causes data applied to the flip-flop's associated output pin to be read into the flip-flop. This dot extension is device specific and should not be used unless the design is being implemented in a device that supports this feature.

.LE

Latch enable (low). This dot extension is used in place of the .CLK dot extension to indicate that the register is a level-sensitive latch. The .LE dot extension is low-true; the value applied to the latch input is read into the latch when the .LE signal is low.

.LH

Latch enable (high). This dot extension is used in place of the .CLK dot extension to indicate that the register is a level-sensitive latch. The .LH dot extension is high-true; the value applied to the latch input is read into the latch when the .LH signal is high.

.OE

Output enable. This dot extension is used to specify an output enable function for the specified signal. Output enables can be specified for combinational or registered outputs.

.PIN

Pin feedback. This dot extension is used to specify a feedback that must originate at the specified pin. If the specified signal also has an .OE (output enable) equation specified elsewhere in the design description, any feedback specified with .PIN will depend on whether or not the output is enabled. .PIN is used when describing designs that require bidirectional I/O capabilities to ensure that the feedback originates from the proper place in the circuit. If no .PIN is specified for fed-back signals on the right side of the equation (in the R-value) the origin of the feedback is ambiguous and will depend on how the design is actually implemented in hardware.

.PR

Preset (generic). This dot extension is used to specify preset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active, causes the flip-flip to be set to a state of 1. (If the output is inverted, a 0 will appear on the actual output pin.) This dot extension is generic, meaning that it does not specify whether the preset is to be synchronous or asynchronous. The ABEL simulator will treat presets specified with .PR as asynchronous, but device fitters and other synthesis software may implement the preset as synchronous depending on the resources available in the target architecture. For this reason, the .AP or .SP dot extensions are recommended instead.

.Q

Direct register feedback. This dot extension is used on the right side of equations (the R-value) to specify that the signal fed back is to come directly from the Q output of the associated flip-flip or latch. When .Q is specified, the logic fed back is normalized to the values that are observed on the actual output of the register. This means that if the output of the flip-flop or latch is inverted before being routed to the output pin the fed-back signal will be inverted from the value observed on the associated output (if any). .Q is typically used in conjunction with the .D, .T, and other data input dot extensions to describe registered functions in terms of the flip-flop contents, rather than in terms of pin-to-pin behavior.

.R

R input to an SR flip-flop. (See .S.)

.RE

Reset (generic). This dot extension is used to specify reset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active, causes the flip-flip to be set to a state of 0. (If the output is inverted, a 1 will appear on the actual output pin.) This dot extension is generic, meaning that it does not specify whether the preset is to be synchronous or asynchronous. The ABEL simulator will treat resets specified with .RE as asynchronous, but device fitters and other synthesis software may implement the reset as synchronous depending on the resources available in the target architecture. For this reason, the .AR or .SR dot extensions are recommended instead.

.S

S data input to an SR flip-flop. This dot extension is one of the two data inputs required for a SR flip-flop. In ABEL, all SR flip-flops are assumed to be clocked, so the .S and .R dot extensions must be used in conjunction with a .CLK dot extension to describe the flip-flop behavior. Since this dot extension specifies an operation to be performed directly on the flip-flip, any inversion of the flip-flop's output (as specified with an 'invert' attribute or as a side effect of the hardware implementation) will result in a reversal of the data appearing on the actual output pin.

.SET

Synchronous set (pin-to-pin). This dot extension is used to specify synchronous set (register preset) logic for the associated output. This dot extension applies to the output pin associated with a flip-flop and, when active during a clocking operation, causes the output pin to be set to a state of 1. If the output is inverted, a value of 0 will be loaded into the associated flip-flop to set the output to 1.

.SP

Synchronous preset. This dot extension is used to specify synchronous preset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active during a clocking operation, causes the flip-flip to be set to a state of 1. (If the output is inverted, a 0 will appear on the actual output pin.)

.SR

Synchronous reset. This dot extension is used to specify synchronous reset logic for the associated output. This dot extension applies directly to the flip-flop associated with the output and, when active during a clocking operation, causes the flip-flip to be set to a state of 0. (If the output is inverted, a 1 will appear on the actual output pin.)

.T

T input to a T (toggle) flip-flop. This dot extension indicates the event (either toggle or hold) to occur in the flip-flop during the next rising clock event. Since this dot extension specifies toggle and hold operations within the flip-flop, there is no direct pin-to-pin relationship between .T dot extension equations and the values appearing on the output pins. If there is an inversion between the output of the flip-flop and the actual output pin (as specified with an 'invert' attribute or as a side effect of the hardware implementation), the value appearing on the output pin will be reversed from the value in the flip-flop after the toggle or hold operation.

Sets

Sets are collections of expression elements enclosed in square brackets and separated by commas. Sets may contain other sets to an arbitrary level of nesting. Any expression element can appear as a member in a set, and different types of expression elements can be mixed to form a composite set. When a sequence of related signals (named with a common alphabetic prefix and sequential numeric suffixes) is entered in a set as members, the range operator (..) can be used as a shorthand notation. The following examples demonstrate a variety of set expressions of increasing complexity:

```
[q3,q2,q1,q0]          "Simple 4-bit bus
[q3..q0]               "Shorthand notation for a bus
[0,0,q1,q0]            "Composite: padded with zeros
[.x.,.x.,q1,q0]        "Padding with no-connects
[q3..q0] > 5           "Magnitude comparison
[[q3..q0],Select]==[^hF,A]   "Nested sets
```

Set Indexing

Any set can be indexed to obtain a subset. Set indexing can be used to select one or more elements from a set for use within an expression. When index values are specified, they specify set elements that are numbered from right to left, beginning with the number 0. The following examples demonstrate how to use set indexing:

```
declarations

  Qset   = [q7..q0];     "8-bit bus
  Qhigh  = Qset[7..4];   "High nibble, same as [q7..q4]
  Qlow   = Qset[3..0];   "Low nibble, same as [q3..q0]
```

Set indexing always produces another (usually smaller) set, even if only one set index value is specified. If you need to obtain a single-bit (nonset) value from a range operation, you can use the following relational expression:

```
MSB = (Qset[7] == 1); "1-bit result (same as q7)
```

Numbers

Numbers may be used as expression elements and are often used in combination with signals and sets to form large composite expressions. When a number is operated on in a larger expression containing sets or signals, the number is truncated or padded with zeros to match the width of the set or signal that it is being operated directly upon. Since individual signals have an inherent width of one bit, any operation between a number and a signal will result in the number being truncated to a single bit. This means that the following two expressions are exactly equivalent:

```
(Sel1 & Sel2 & Sel3) == 5 "Truncated to 1 bit
(Sel1 & Sel2 & Sel4) == 7 "Truncated to 1 bit
```

Since numbers are converted to sets when a set is involved in the expression, the following two expressions are *not* equivalent:

```
[Sel1,Sel2,Sel3] == 5   "3 bits: 1,0,1
[Sel1,Sel2,Sel3] == 7   "3 bits: 1,1,1
```

Special Constants

Special constants may be used in an expression, but if the expression appears anywhere other than in a test vector entry, the only special constant allowed is the no-connect constant (.X.). In logic descriptions, the no-connect constant is used to pad the most significant (leftmost) bits of a set so that operations can be performed on sets of otherwise unequal size. The no-connect special constant can also be used to pad the least significant (rightmost) side of a set to change the evaluation of numeric constants. Both of these uses of the no-connect special constant are shown in the following sample expressions:

```
([.x.,A2,A1,A0] + [.x.,B2,B1,B0])==[C3,C2,C1,C0]
                    "compare 4-bit result of 3-bit add

[A15..A8,.x.,.x.,.x.,.x.,.x.,.x.,.x.,.x.]== ^hE0200
                    "compare high byte of a word
```

Operator	Description
!	NOT (Boolean inversion)
&	AND
#	OR
$	XOR
!$	XNOR

Figure A.3 ABEL Boolean operators

Expression Operators

ABEL includes a rich variety of operators that can be applied in expressions. The operators are broken into three categories: Boolean, relational, and arithmetic. (The Boolean operators are referred to as logical operators in the standard ABEL documentation.) A fourth type of operator, the assignment, is discussed in Section A.4.

Expressions can be used in a wide variety of contexts within an ABEL source file. When an expression is used within a logic description, the desired result is generally a combinational network (an equation tree) that forms all or part of the combinational logic for one or more outputs. This tree representation may or may not include signals, depending on the contents of the expression. In a strictly numeric context, such as in a truth table entry, test vector, or directive, the expression is evaluated immediately to produce a numeric value. In this latter context, the expression cannot include signals, since a signal cannot be resolved into an actual value at compile time.

With the exception of the multiplication-related arithmetic operators (which cannot be applied to sets), all ABEL operators are available for use in any logic description or numeric context and can operate on any type or mix of expression elements (signals, numbers, sets or no-connects).

Boolean Operators

ABEL's Boolean operators provide all the basic operations needed for describing combinational logic functions. The ABEL Boolean operators and corresponding symbols are shown in Figure A.3.

Operator	Description
==	equality comparison
!=	inequality comparison
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Figure A.4 ABEL relational operators

Relational Operators

ABEL's relational operators allow complex logical comparison functions to be easily described. The expression elements being compared can be signals, sets, numbers, or no-connect (.X.) special constants. The result of any comparison, regardless of the width or type of the expression elements, is always a single bit. For comparisons that include signals, the result of a comparison operation is always a single output combinational network. Comparison operations that operate on numbers (or sets of numbers) always result in a numeric value of either 0 (false) or 2^{32} - 1 (the 32-bit unsigned equivalent of -1). The table of Figure A.4 lists the relational operators available in ABEL.

Arithmetic Operators

Arithmetic operators perform arithmetic operations on expression elements. Figure A.5 lists the arithmetic operators supported in ABEL.

In numeric expressions, arithmetic operations always result in a positive integer value in the range of 0 to 2^{128} - 1. In the context of a logic description, expressions containing arithmetic operations are converted to a combinational network (an equivalent expression composed of Boolean operators) according to the rules for set evaluation described later in this section. The multiplication-related operators (*, /, %, <<, and >>) are restricted to expressions containing signal, number, or special constant element types. Sets are not supported in multiplication-related operations.

Because numbers in ABEL are unsigned integers, the results of arithmetic operations are always positive integers. Negation is a twos complement operation resulting in a positive (and typically large) integer value. Subtraction consists of the addition of the second operand's twos complement to the first operand and results in the values you would

Operator	Description
-	(unary) negation (twos complement)
-	(binary) subtraction
+	addition
*	multiplication
/	division
%	modulo (remainder of division)
<<	shift left
>>	shift right

Figure A.5 ABEL arithmetic operators

expect when the first operand is greater than the second. Because numbers are integers, the result of a division operation is always an integer. The remainder of a division operator can be obtained by using the modulo (%) operator.

The shift right and shift left (<< and >>) operators result in the equivalent of a multiplication or division operation, respectively, by the power of two specified in the second operand. This arithmetic shift results in zeros being shifted in from the left (during right-shift operations) or from the right (during left-shift operations).

Operator Priorities

When writing ABEL design descriptions, it's important to keep in mind the priorities and order of operation for expressions that include more than one operator. Figure A.6 shows all the ABEL operators with a number indicating the level of each operator's priority.

Parentheses can be added to expressions to change the ordering (binding) of operators and should be used liberally in ABEL designs. Parenthesis can improve the readability of designs as well as help to avoid order of evaluation errors in complex expressions.

When more than one operator of equal priority appears in an expression that has not been placed in parentheses, the operations are evaluated from left to right. Since there are only four levels of priority in ABEL expressions, priority-related mistakes when writing logic descriptions are common. The most common errors are related to the combination of relational or arithmetic operators with Boolean operators, and these errors can be difficult to track down if the design compiles successfully but operates incorrectly. To avoid these common problems, it is recommended that all relational and arithmetic subexpressions be sur-

Priority	Operator	Description
1	-	Unary minus (negation)
1	!	NOT (invert)
2	&	Logical AND
2	>>	Arithmetic shift right
2	<<	Arithmetic shift left
2	*	Multiplication
2	/	Division
2	%	Modulo (remainder)
3	+	Addition
3	-	Subtraction
3	#	Logical OR
3	$	Exclusive-OR
3	!$	Exclusive-NOR
4	==	Equality
4	!=	Inequality
4	>	Greater than
4	<	Less than
4	>=	Greater than or equal
4	<=	Less than or equal

Figure A.6 ABEL operators and their priorities

rounded by parentheses, regardless of whether the parentheses are actually needed.

Set Evaluation

All expression operators (with the exception of the multiplication-related operators *, /, %, <<, and >>) can be applied to sets as well as to other element types. When a set is operated on, the order and manner in which the set elements are processed depends on the type of operation being performed. Figure A.7 defines the set evaluation rules for Boolean, relational, and arithmetic operators and shows how the relational and

Expression	Result
![a_k..a_1]	[a_k, !a_{k-1}, ... , !a_1]
[a_k..a_1] & [b_k..b_1]	[a_k & b_k, a_{k-1} & b_{k-1}, ... , a_1 & b_1]
[a_k..a_1] # [b_k..b_1]	a_k # b_k, a_{k-1} # b_{k-1}, ... , a_1 # b_1]
[a_k..a_1] \$ [b_k..b_1]	[a_k \$ b_k, a_{k-1} \$ b_{k-1}, ... , a_1 \$ b_1]
[a_k..a_1] !\$ [b_k..b_1]	[a_k !\$ b_k, a_{k-1} !\$ b_{k-1}, ... , a_1 !\$ b_1]
[a_k..a_1] == [b_k..b_1]	(a_k == b_k) & (a_{k-1} == b_{k-1}) & ... & (a_1 == b_1)
[a_k..a_1] != [b_k..b_1]	(a_k != b_k) # (a_{k-1} != b_{k-1}) # .. # (a_1 != b_1)
[a_k..a_1] < [b_k..b_1]	c_k with: c_n = (!a_n & (b_n # c_{n-1}) # (a_n & b_n & c_{n-1})) != 0 c_1 = 0
[a_k..a_1] > [b_k..b_1]	([b_k..b_1] < [a_k..a_1])
[a_k..a_1] <= [b_k..b_1]	!([a_k..a_1] > [b_k..b_1])
[a_k..a_1] >= [b_k..b_1]	!([a_k..a_1] < [b_k..b_1])
-[a_k..a_1]	![a_k..a_1] + 1
[a_k..a_1] + [b_k..b_1]	S with: s_n = a_n \$ b_n \$ c_{n-1} c_n = (a_n \$ b_n) # (a_n & c_{n-1}) # (b_n & c_{n-1}) c_1 = 0
[a_k..a_1] - [b_k..b_1]	[a_k..a_1] + -[b_k..b_1]

Figure A.7 ABEL set expression evaluation rules

arithmetic operators are converted to Boolean operations. In the table, *k* refers to the number of bits in the sets, while *n* is an index into the sets (ranging from 1, the least significant or rightmost bit, to *k*, the most significant or leftmost bit.) *A* and *B* are sets composed of elements a_1 through a_k and b_1 through b_k, respectively.

In many operations involving sets, the result of evaluating an expression will depend on the types of elements that are included in the expression. The result of an expression involving sets will always be another set, even if the expression involves relational operators. A relational operation performed on sets will result in a set of all zeroes or all ones, and the width of the result will depend on the context in which the relational operation was performed.

A.4 EQUATIONS

In a design description, equations can appear in an EQUATIONS section or within the state definitions of a STATE_DIAGRAM section. There can be any number of EQUATIONS sections in a single design description.

All ABEL equations consist of three basic parts: a left-hand (output) side, called the L-value, an assignment operator, and a right-hand (input) side, which is called the R-value. Equations are terminated by a semicolon. An example of an equation is

```
Out1 = (Select & [c1,c0]
     # !Select & [b1,b0]) != [1,1];
```

This equation tests the result of a multiplexer operation and assigns the result to *Out1*. The L-value of this equation is a simple expression consisting of the signal *Out1*, while the R-value of the equation is a more complex expression consisting of signals, sets, numbers, and a variety of Boolean and relational operators. The combinational assignment operator = indicates that this equation describes a combinational function whose output is *Out1*.

The R-value of an equation can be any combination of signals, sets, numbers, and the no-connect special constant (.X.), and any syntactically correct combination of Boolean, relational and arithmetic operators. As in other places where such expressions are allowed, parentheses can be used to specify the ordering of complex operations or simply to improve the readability of the expression. (The preceding example uses parentheses to enclose the OR operation, but these parentheses are not actually needed since the OR operator has a higher priority than the != relational operator.)

The structure of the L-value of an equation is more limited. L-value expressions may include signals or no-connect special constants, but may not include numbers. Sets consisting of signals or no-connects are also valid in the L-value. The only operator that is allowed in an L-value is the ! Boolean operator. The ! operator can be used to invert the L-value, which is functionally equivalent to inverting the R-value. This is demonstrated by the following two equations, which are functionally identical:

```
IO_SELECT  = !(Address == ^hC800);
!IO_SELECT = (Address == ^hC800);
```

When an equation is specified (or represented internally by the language compiler) with an inverted L-value, the equation is said to have *negative polarity*. Equations that do not have inverted L-values are positive polarity. Which form to use when describing a design is purely a matter of style and convenience. (An alternative way of describing an active-low function such as the one shown is to use an active-low declaration. Active-low declarations were described in Section A.2.)

Since the signal *Output1* appears in the L-value of an equation, it is identified by the language compiler as an output of the design. The signals *Select*, *C1*, *C0*, *B1*, and *B0*, since they appear in the R-value, are determined to be inputs. If a signal appears in both the L-value and the R-value, the compiler will determine that the signal is either bidirectional, or a fed-back output. (The language compiler makes no distinction between bidirectional signals and feedback loops. That determination is a device fitting task that must be performed by device-specific synthesis software.)

In ABEL equations involving sets on both sides of the equation, the L-value and R-value must have the same width or an error will occur during compilation. In the preceding example, both sides of the equation evaluate with a width of one, because relational operators always produce a one-bit value.

When an equation has a set on the left side of the equation, but a signal (or other single-bit result) on the right side, each element of the L-value set is assigned the R-value. For example, the equation

```
[Output1, Output2] = A & B;
```

is equivalent to the two equations

```
Output1 = A & B;
Output2 = A & B;
```

When an equation assigns a numeric value (or numeric expression) to a set, the numeric value is converted to a set of the appropriate size (either by padding from the left with zeros or truncating the most significant bits) before the assignment is made. For example, the equation

```
[Output1, Output2] = 1;
```

is equivalent to the two equations

```
    Output1 = 0;
    Output2 = 1;
```

and the equation

```
    [Output1,Output2] = 14;
```

is equivalent to the two equations

```
    Output1 = 1;
    Output2 = 0;
```

In this example, the binary representation of 14 (^b1110) was truncated to two bits (^b10) before the assignment was made.

WHEN-THEN-ELSE Equations

In addition to the equation format described previously, ABEL also provides an alternative form for expressing conditional logic. The WHEN-THEN keywords can be used to write equations like the following:

```
    WHEN Select == [0,0] THEN
                         DataOut = DataA;
    WHEN Select == [0,1] THEN
                         DataOut = DataB;
    WHEN Select == [1,0] THEN
                         DataOut = DataC;
    WHEN Select == [1,1] THEN
                         DataOut = [0,0,0,0];
```

When processed, the expressions between the WHEN and THEN keywords (the conditional expression) are ANDed with the equations following the THEN keywords to form complete equations, one for each WHEN-THEN statement. If you wish to guarantee that a series of WHEN-THEN conditions is mutually exclusive, you can use an ELSE clause to chain together two or more WHEN-THEN statements or to provide a terminating (default) conditional. The following series of statements uses chained WHEN-THEN and ELSE statements to specify a mutually exclusive function:

```
WHEN Mode == Inactive THEN
                              Stat == F_OFF;
ELSE WHEN Data > ^h00FF THEN
                              Stat == F_INC;
ELSE
                              Stat == F_HLD;
```

When processed by the language compiler, each ELSE clause results in a progressively more complex equation being generated for the associated equation. In this example, the chain of WHEN-THEN-ELSE statements will result in an equation for *Status* that is functionally equivalent to

```
Stat = (Mode==Inactive) & F_OFF
       # !(Mode==Inactive) & (Data > ^h00FF) & F_INC
       # !(Mode==Inactive) & !(Data > ^h00FF) & F_HLD;
```

WHEN-THEN statements that operate on more than one output can take advantage of ABEL's equation block construct that allows groups of equations to be combined in a single block of text. Equation blocks are enclosed in curly braces ({ and }) and can be used in place of a single equation anywhere in an ABEL source file. The following example uses an equation block to enclose three equations in a single WHEN-THEN statement:

```
WHEN (Mode == Load) THEN
{      RESULT := DataIn;
       RESULT.OE = 0;
       Status = Setup;
}
```

Multiple Equations for the Same Signal

When more than one equation is written for the same output, the equations must be combined by the language compiler to form a single equation. This is done by combining (through the use of an OR operation) all equations with noninverted L-values into one group of equations and combining all equations with inverted L-values into another group of equations. The negative equation is then converted to positive polarity and the two equations are combined into a signal equation with an OR operation. For example, if the equations

```
Output2 = A1;
Output2 = A2;
!Output2 = A3;
!Output2 = A4;
```

are written for an output, then the result (after the equations have been combined) will be

```
Output2 = A1 # A2 # !(A3 # A4);
```

Pin-to-Pin Registered Equations

ABEL's pin-to-pin registered assignment operator can be used when a design requires registered outputs. Registered outputs hold their values until a clock signal causes a new value to be loaded. The following equation describes a function that stores a shifted 2-bit data value in four output registers:

```
[Q3,Q2,Q1,Q0] := [D1,D0,0,0] & Shift
               # [0,0,D1,D0] & !Shift;
```

This equation describes the function as it will appear on output pins *Q0* through *Q3*. The := assignment operator describes pin-to-pin values, so the actual value stored in the associated flip-flop and the flip-flop type are left undefined. (Pin-to-pin registered descriptions such as this imply D-type flip-flops or latches, but logic synthesis routines found in some device fitters can convert the equations into alternativee flip-flop types.) When clocked functions (flip-flops) are described, the equation for the outputs must be accompanied by a clock equation such as the following:

```
[Q3,Q2,Q1,Q0].CLK = Clock;
```

This equation indicates that the flip-flops associated with the *Q3* through *Q0* outputs are to be clocked using the input signal *Clock*. In the absence of a clock equation, the ABEL logic simulator will be unable to load the desired function into the registers, so the design will not work as intended. The absence of a clock equation will also cause problems during the synthesis and device fitting process, since the register type (flip-flop or latch) is ambiguous.

Writing Flip-flop-oriented Equations

The .CLK dot extension used in the previous example is the most common of the flip-flop-related dot extensions. An equation for a .CLK is required for every output signal that has a flip-flop associated with it. There are many other dot extensions associated that allow you to have more complete control over the behavior of the flip-flop. These dot extensions were described in Section A.3 and include extensions for reset and preset and for the specific data inputs to various types of flip-flops.

When the data inputs to flip-flips (the .D, .T, .J, .K, .S, and .R dot extensions) are used in logic descriptions or when you are writing equations describing flip-flop reset and preset functions, the focus of your design description is moving to the various flip-flop inputs, rather than to the design outputs. This is particularly important to understand if your design will be implemented in a programmable logic device that has inverted outputs. When the outputs are inverted, the values that appear on the output pins will be inverted from the values that are actually stored in the associated flip-flop.

Figures A.8 and A.9, for example, show two forms of the simple shifter function previously described, using flip-flop stimulus (detailed) design equations. The first of these examples shows how the design would be written if the outputs were not inverted, while the second example shows how the design would be written if the outputs were inverted. The difference in these two representations is in the inversion of the L-value when istype 'invert' is specified. Without this additional inversion, the data appearing on the design outputs would be reversed from what had

```
MODULE Shift1


    Clock, Shift  PIN;         "Control inputs
    D1,D0         PIN;         "Data inputs
    Q3,Q2,Q1,Q0   PIN ISTYPE 'reg_D,buffer'; "Outputs

EQUATIONS

    [Q3,Q2,Q1,Q0].D = [D1,D0,0,0] & Shift
                    # [0,0,D1,D0] & !Shift;
    [Q3,Q2,Q1,Q0].CLK = Clock;

END
```

Figure A.8 Simple shifter described using a flip-flop oriented (.D) equation

```
MODULE Shift2

    Clock, Shift  PIN;        "Control inputs
    D1,D0         PIN;        "Data inputs
    Q3,Q2,Q1,Q0   PIN ISTYPE 'reg_D,invert'; "Outputs

EQUATIONS

   ![Q3,Q2,Q1,Q0].D   = [D1,D0,0,0] & Shift
                      # [0,0,D1,D0] & !Shift;
    [Q3,Q2,Q1,Q0].CLK = Clock;

END
```

Figure A.9 Shifter described with .D equation and 'invert' attribute

been applied on the two data inputs. Both examples have the same operation as the pin-to-pin version described earlier. Since flip-flop specific features such as preset and reset are not required for this simple design, the earlier pin-to-pin version is preferable.

On, Off and DC Assignments

ABEL supports the specification of logic that includes don't-care input conditions. To specify don't-cares for logic that has been described in equation form, you can use alternative assignment operators to completely specify the don't-care conditions for an output. Figure A.10 lists ABEL's assignment operators.

Operator	Description
=	Combinational assignment
?=	Combinational don't-care assignment
:=	Registered assignment
?:=	Don't-care registered assignment

Figure A.10 ABEL assignment operators

These assignment operators can be used in addition to the standard assignment operators to completely specify a logic function that includes don't-cares.

A.5 STATE DIAGRAMS

ABEL's STATE_DIAGRAM language allows the pin-to-pin description of complex sequential circuits. Each STATE_DIAGRAM section consists of one state diagram header followed by any number of state descriptions. Each state description must include a state value and at least one transition statement. A sample state diagram is shown in Figure A.11.

State Diagram Header

The state diagram header lists which signals (either pins or nodes) will be used to store the current state of the machine being described. This list of signals is called the state register, and each signal in the state register represents one state bit. The list is entered as a set of registered signals, as shown in the sample state diagram of figure A.11. Since each state of the machine must have a unique value stored in the state register, the maximum number of states that can be represented is 2^n, where n is the number of state bits in the state register. It is not necessary to define all possible states of the machine, however. Many (perhaps most) state machines have some number of states that are undefined, since the number of states required for a large sequential application is rarely an exact power of 2.

In addition to the common state diagram header seen in most state diagram examples, ABEL allows a modified form of the header to be used for state machines that will be implemented using combinational signals. This form is often used for state machines that are being implemented in PROM-type devices with external memory elements or

```
STATE_DIAGRAM [S1,S0]

    STATE [0,1]: IF Restart THEN [0,1]
                 ELSE [1,1];

    STATE [1,1]: IF Restart THEN [0,1]
                 ELSE [1,0];

    STATE [1,0]: IF Restart THEN [0,1]
                 ELSE [0,0];

    STATE [0,0]: IF Restart THEN [0,1]
                 ELSE [0,1];
```

Figure A.11 Sample state diagram

```
DECLARATIONS

    C3,C2,C1,C0  PIN;
    N3,N2,N1,N0  PIN ISTYPE 'com';

STATE_DIAGRAM [C3,C2,C1,C0]->[N3,N2,N1,N0]
          .
          .
          .
```

Figure A.12 Alternate form of ABEL state diagram header

when asynchronous state machines are being described. The state diagram header shown in Figure A.12 uses this alternative header representation.

In the state diagram associated with this header, the first set of signals (consisting of *C3* through *C0*) represents the current state of the machine, while the second set of signals (*N3* through *N0*) represents the calculated next state of the machine. The state descriptions that follow this type of state diagram header are in all respects identical to the state descriptions that follow standard state diagram headers.

State Values

State values are entered in state descriptions to describe the current state of the machine (as represented in the state register) and to specify the value of the next state for each transition statement. State values indicate the actual values that will appear on the state bit outputs (whether declared as PINs or NODEs) during each state of the machine. Because of the possibility of output inversions, state values do not necessarily correspond directly to the values stored in the state register flip-flops. This is why state diagrams are said to be pin-to-pin design descriptions.

State values can be entered as sets of binary values or as numbers. State values can also be assigned identifier names and declared as constants, as shown in Figure A.13. Using constant declarations for state values can help make the design easier to read and easier to modify if you later need to optimize the state values.

Transition Statements

In our first simple example, each state description includes two possible transitions that were chained using an IF-THEN-ELSE transition state-

```
DECLARATIONS
    Start  = [0,1];
    Delay1 = [1,1];
    Delay2 = [1,0];
    Delay3 = [0,0];

STATE_DIAGRAM [S1,S0]
    STATE Start:  IF Restart THEN Start
                  ELSE Delay1;

    STATE Delay1: IF Restart THEN Start
                  ELSE Delay2;

    STATE Delay2: IF Restart THEN Start
                  ELSE Delay3;

    STATE Delay3: IF Restart THEN Start
                  ELSE Start;
```

Figure A.13 Using constants for state values

ment. Three basic types of transition statements can be used individually or combined to create complex transition logic. The first and simplest type of transition is the GOTO statement. A GOTO statement specifies an unconditional transition to the indicated state. Since the last state of our sample state machine has only one possible next state, this state description could be rewritten using a simpler GOTO statement as follows:

```
STATE Delay3: GOTO Start;
```

The second type of transition statement is the IF-THEN or IF-THEN-ELSE statement. We have seen IF-THEN-ELSE statements used in our simple example. When multiple IF-THEN-ELSE statements are chained, all the transitions in the chain are guaranteed to be mutually exclusive. This is done in much the same way as in the WHEN-THEN statements described for equations. IF-THEN-ELSE statements also allow transitions to be nested to an arbitrary level of complexity. Figures A.14 and A.15 show how the same transition logic can be described using either chained or nested IF-THEN-ELSE statements.

```
State5: IF Restart THEN State0          "Restart
        ELSE IF Down & Skip THEN State3 "!Restart&Down&Skip
        ELSE IF Down THEN State4        "!Restart&Down&!Skip
        ELSE IF Skip THEN State7        "!Restart&!Down&Skip
        ELSE State6;                    "!Restart&!Down&!Skip
```

Figure A.14 Chained IF-THEN-ELSE statement

```
State5: IF Restart THEN State0
        ELSE
        {   IF Down THEN
            {   IF Skip THEN State3
                ELSE         State4
            }
            ELSE
            {
                IF Skip THEN State7
                ELSE         State6;
            }
        }
```

Figure A.15 Nested IF-THEN-ELSE statements

The third type of transition statement is the CASE-ENDCASE statement. CASE-ENDCASE statements can be used to describe transition logic that you know is mutually exclusive. The following shows how the transition for the *Start* state can be written using a CASE-ENDCASE statement to describe the same transition logic that was previously written using an IF-THEN-ELSE statement:

```
STATE Start: CASE Restart: Start;
                 !Restart: Delay1;
             ENDCASE;
```

The CASE-ENDCASE statement is rarely used, since any CASE-ENDCASE statement can be written more clearly as a series of IF-THEN statements. In addition, the use of CASE-ENDCASE often results in the accidental creation of nonmutually exclusive transition logic.

State Output Equations

Most state machines are designed to drive outputs of some sort, so ABEL allows equations for these outputs to be entered within the state descriptions. There are two basic ways to write equations within state descriptions. You can write them for the current state, or for the current transition.

To associate one or more equations with the current state of a machine, you insert the equations into the state description between the state value and first transition statement, as follows:

```
STATE_DIAGRAM [S1,S0]

STATE [1,0]:
          MyOutput1 = 1;
          MyOutput2 = A & B;
          IF ... THEN ... ;
```

In this example, the output *MyOutput1* will be true whenever the state registers (*S1* and *S0*) contain the values 1 and 0 (respectively) and the output *MyOutput2* will be true whenever the state registers contain the indicated values and the inputs *A* and *B* are also true.

To write equations for outputs associated with specific transitions or to write equations for registered outputs that must become active in a destination state, you can use the WITH statement. The following state description uses a WITH statement to include an output equation with each of the two possible transitions out of the indicated state:

```
STATE Compare:
    IF (Data == ^hAA06) THEN
        Write WITH {
                       status := STATUS_WRITE;
                    }
    ELSE
        Incr WITH {
                       status := STATUS_HOLD;
                  }
```

Timing Considerations for State Outputs

When writing equations using WITH statements, it's important to understand the timing of the resulting logic. Since the only information available to decode as inputs to any state equation is the current state and state machine inputs, there is no way to directly associate an equation with the actual next state of the machine. Instead, the equations that are written within WITH statements are always created by using the specified transition logic and the current state. As a result, combinational outputs described in a WITH section will be active one state early and will probably not be the values that you expect when the machine advances to the destination state. This is why equations written within WITH statements are usually registered.

Equations written within state descriptions that are not enclosed in WITH statements exhibit the opposite behavior. If a state output is declared and used as registered, and is defined by an equation in a state description, then that output will not become active until the next clock

```
DECLARATIONS

    State1, State2,
    State3,State4  = 1,2,3,0;    "State values

    ToState1  MACRO
                {State1 WITH IN_1 := 1; IN_3 := 0; ENDWITH};
    ToState2  MACRO
                {State2 WITH IN_1 := 0; IN_3 := 0; ENDWITH};
    ToState3  MACRO
                {State3 WITH IN_1 := 0; IN_3 := 1; ENDWITH};
    ToState4  MACRO
                {State4 WITH IN_1 := 0; IN_3 := 0; ENDWITH};

STATE_DIAGRAM [S1,S0]
        .
        .
        .
    STATE State3:
                 IF (Reset) THEN ToState1
                 ELSE IF (Hold) Then ToState3
                 ELSE ToState4;
```

Figure A.16 Using macros for WITH statements

cycle, by which time the state machine will probably have advanced to a different state.

If you require a registered output to be active every time you transition into a particular state, without regard to the previous state, you can use a macro to attach one or more WITH equations to the primary state as shown in Figure A.16.

A.6 TRUTH TABLES

Truth tables are used to describe direct tabular relationships between the values observed on a set of input signals and the corresponding values that are to appear on a set of output pins. Truth tables consist of a truth table header followed by one or more truth table entries.

Truth Table Headers

Like state diagrams, truth tables have a header that defines the input and output signals to be used in the subsequent description. A sample truth table header is

```
TRUTH_TABLE ([A,B,C,D]->[Out1,Out2])
```

This truth table header lists the signals *A*, *B*, *C* and *D* as inputs and signals *Out1* and *Out2* as outputs. The use of the -> operator indicates that the outputs are combinational and is analogous to the = assignment operator. If the outputs of the truth table are registered, the :> operator is used. The :> operator is analogous to the := pin-to-pin registered assignment operator. The following example is a truth table header for a set of two registered outputs:

```
TRUTH_TABLE ([A,B,C,D]:>[Reg1,Reg2])
```

Truth tables can include both combinational and registered outputs in the same header by using a three-part header such as the following:

```
TRUTH_TABLE ([A,B,C,D]:>[Reg1,Reg2]->[Out1,Out2])
```

This combined representation is often used as an alternative to state diagrams when describing state machines.

Nested sets can be used in truth table headers to simplify the entry of truth table values. Truth tables can be further simplified by declaring these sets using constant declarations as follows:

```
DECLARATIONS

    InSet  = [A,B,C,D];
    OutSet = [Out,Out2];

TRUTH_TABLE (InSet -> OutSet)
    .
    .
    .
```

Truth Table Entries

The actual entries in the truth table must consist of sets of numeric values or .X. no-connect special constants. The set widths used in the entries must match the set widths specified in the truth table headers. The following truth table contains four entries that define the function of the circuit (a simple XOR operation):

```
TRUTH_TABLE ([ A, B ]->[ Y])
             [ 1, 0 ]->[ 1];
             [ 0, 1 ]->[ 1];
             [ 0, 0 ]->[ 0];
             [ 1, 1 ]->[ 0];
```

As in equations, numbers can be substituted for sets of binary values, so it is not necessary to type in large sets of binary values. The truth table shown in Figure A.17, for example, uses nested sets to describe a complex decoder function

In this example, the *Address* and *StringData* sets are used to simplify the truth table. Nonbinary numeric values are substituted for sets of binary values in the truth table entries for *Address* and *StringData*, and these numeric values are expanded by the compiler to match the width of the header entries. When sets are nested in this way, it is important to keep in mind that the nesting level of the truth table entries must match the nesting of the truth table headers. If you ignore the nesting levels, the results can be incorrect and confusing. For example, the truth table shown in Figure A.18 will not compile correctly because the nestings of the truth table's header and entries do not match.

The input of this truth table is correctly described; the *Bits* and *A* sets are combined to form the input set, and the input values written in the truth tables are expressed as nonbinary values that are properly evaluated and mapped to the inputs. The output side, however, is not correctly written. Since set *Y* is declared as a set and then nested within the output set of the truth table header, the effective set width of the truth table header is 1 bit. This means that the values appearing on the output side of the truth table entries will be evaluated as a single bit and truncated before being applied to it. The solution is to either remove the unnecessary set of brackets from the output side of the truth table header, or add a pair of brackets around each of the values appearing on the output side of the truth table entries.

```
DECLARATIONS

     Address     =  [A15..A0];
     StringData  =  [D7..D0];

 TRUTH_TABLE ([AddrValid,Address]->StringData)
             [    1    , ^h0100]-> ^h46;
             [    1    , ^h0101]-> ^h52;
             [    1    , ^h0102]-> ^h45;
             [    1    , ^h0103]-> ^h44;
```

Figure A.17 Truth table with nested sets

```
DECLARATIONS

     Bits  =  [B1,B0];
     A     =  [A1,A0];
     Y     =  [Y3,Y2,Y1,Y0];

TRUTH_TABLE ([Bits,  A ]->[  Y  ])  "Nested too far
                [ 1  ,^b01]->^b0010;
                [ 1  ,^b10]->^b0100;
                [ 2  ,^b01]->^b0100;
                [ 2  ,^b10]->^b1000;
```

Figure A.18 Truth table with set nesting problem

A.7 TEST VECTORS

Test vectors are used to describe stimulus and expected outputs for logic simulation. The test vector format is similar to the truth table format; it has a header that describes the input and output signals, and corresponding entries describing input and output values.

Test Vector Headers

Like truth table headers, test vector headers consist of sets of signals that define the inputs and outputs to be used in the subsequent entries. A sample test vector header is

```
TEST_VECTORS ([Clock,Address,Reset]->[Sflag,Data])
```

This truth table header lists the identifiers *Clock*, *Address*, and *Reset* (which may be either signals or sets of signals) as inputs and identifiers *Sflag* and *Data* as outputs. The use of the -> operator is required and is used regardless of whether the outputs being tested are combinational or registered.

Test Vector Entries

The entries in a test vector section, like those of a truth table, must consist of sets of numeric values or special constants, and the set widths used in the entries must match the set widths specified in the header. All the special constants listed in Figure A.19 are valid.

The test vector section shown in Figure A.20 demonstrates how special constants are entered in the test vector entries. Note that the .X. special

Constant	Description
.C.	Clock (drive input low-high-low).
.D.	Down edge (drive input high-low).
.F.	Floating input or output.
.K.	Negative clock (high-low-high).
.P.	Output registers preload.
.U.	Up edge (drive input low-high).
.X.	Don't-care input or output.
.Z.	High impedence (three-state)

Figure A.19 ABEL special constants

constant used in test vectors has a different meaning than no-connect. When .X. is used in equations and other logic descriptions, it is used strictly as a place holder to create sets of matched widths. In truth tables, .X. can be used to indicate that an output is unspecified, which may or may not result in a don't-care being generated by the compiler (depending on the optimization-related signal attributes used, and whether @DCSET was specified). In test vectors, .X. indicates a don't-care in simulation. Depending on the simulator used, this may be an actual don't-care logic value or an input or output whose value is simply ignored.

A.8 MACROS

ABEL macros are used to define sequences of ABEL statements or other text that can be invoked repeatedly within an ABEL module. Macros can include parameters, making it possible to describe context-sensitive, parameterized functions. A macro declaration has the following form:

macro_name MACRO (optional_arguments) { text_body} ;

```
TEST_VECTORS ([Clock, Reset ,Address]->[Sflag, Data])
             [ .C. ,   0   ,  .X.   ]->[  0  , .X. ];
             [ .C. ,   1   ,^h0100  ]->[  1  , ^h46];
             [ .C. ,   1   ,^h0101  ]->[  1  , ^h52];
             [ .C. ,   1   ,^h0102  ]->[  1  , ^h45];
             [ .C. ,   1   ,^h0103  ]->[  1  , ^h44];
```

Figure A.20 Test vectors section

ABEL macros provide a basic text substitution mechanism. The body of the macro, which is enclosed in curly braces, is not processed by the compiler until the macro is actually referenced elsewhere in the design description. Wherever a macro name appears in the ABEL module in which it was declared, the text of the macro body is inserted into the ABEL design description. If the macro includes parameters, then the actual parameters specified when the macro is invoked are substituted for the dummy arguments appearing in the macro body. The following macro declaration includes such parameters:

```
Compare MACRO (Arg1, Arg2)  {?Arg1 == ?Arg2};
```

In this example, *Arg1* and *Arg2* are listed as macro parameters and appear in the text body of the macro as dummy arguments. Dummy arguments are identified by the ? character prefix. When this macro is invoked with actual arguments, the resulting text is inserted into the design description and processed by the language compiler. Since the text appearing in the macro body is not processed by the language compiler until the macro is invoked, syntax errors within the macro body are not detected until the macro is invoked.

Macros Versus Constant Declarations

The ability to pass arguments into macros is one reason to use macros rather than constant expressions. Another reason to use macros is for general text-substitution purposes. In many situations, either a macro or a constant declaration could be used to achieve the same purpose, and in these cases which to use is really a matter of style. There are important differences between macros and constant declarations, however, which must be understood. Since macros are a simple text-substitution mechanism, you must be careful not to make assumptions about operator priority when specifying logic expressions in them. Consider, for example, the following macro and constant declarations:

```
Select1 MACRO {Addr > ^hE000};

Select1    =   Addr > ^hE000;
```

Although these two declarations appear very similar, they are actually very different. If *Select1* is used in the context of a larger expression, such as in

```
Trigger = Select1 & !Reset;
```

then the two types of declarations will result in different equations. Since macros are simply a text substitution mechanism, the result of using the *Select1* macro will be an equation of the form

```
Trigger = Addr > ^hE000 & !Reset;
```

which is probably not the desired result. Why? Because the AND operator (&) has a higher priority than relational operators such as >. This means that the equation will evaluate incorrectly. To get around this problem, the macro must be modified with parentheses:

```
Select1  MACRO {(Addr > ^hE000)};
```

Constant expressions, since they are evaluated at the time they are declared, will always maintain the expected operator priority and order of operation, so it is not necessary to enclose them in parenthesis. To avoid evaluation errors when using macros, it is a good idea to place parentheses around all macros containing logic expressions. For consistency, it doesn't hurt to do the same for all nontrivial constant expressions as well.

A.9 DIRECTIVES

Directives instruct the language compiler to interrupt processing and perform such things as text insertion, looping, and conditional compilation. Directives can also be used to change certain compiler options such as the default number system (radix) or the processing of don't-cares. Directives can appear any place in an ABEL design description and do not have to be located within a module. Figure A.21 lists the directives supported in ABEL.

Like other ABEL keywords, directive names may be entered in uppercase, lowercase or mixed case.

ABEL's many directives can seem confusing at first, but they are actually extremely powerful and should be used whenever possible to simplify design descriptions. The most common use of directives is in the specification of tabular or repetitive data. Test vectors (and truth tables) tend to involve repeated sequences of values, so for these applications the looping and text substitution capabilities of ABEL directives are particularly useful.

@ALTERNATE	@IFDEF	@IRPC
@CONST	@IFIDEN	@MESSAGE
@DCSET	@IFNB	@ONSET
@EXIT	@IFNDEF	@PAGE
@EXPR	@IFNIDEN	@RADIX
@IF	@INCLUDE	@REPEAT
@IFB	@IRP	@STANDARD

Figure A.21 ABEL compiler directives

Looping and Text Processing Directives

@INCLUDE

The @INCLUDE directive inserts the contents of the specified file at the point where the directive appears. @INCLUDE directives can be nested, so multiple levels of include files can be used. Include files are useful for storing commonly used macros, constant declarations and other ABEL statements or for breaking a large design into multiple files. The following statement, for example, includes a file into the current design description:

```
@INCLUDE 'stateval.inc'    "Include the state values
```

@CONST

The @CONST directive provides a simple method for declaring or redefining numeric constants. @CONST is useful for creating or modifying values that are used within loops or macros. When a constant is defined using @CONST, any existing declaration for that constant is overridden. The following is an example of how a @CONST directive can be used to create a sequence of incrementing values that count from 0 to 99 (in this case within a test vector section):

```
test_vectors([SelectA,SelectB,   A   , B ]->[DataOut])
@CONST addr = 0;
@REPEAT 100 {
                 [    1  ,    0   ,  addr ,.x.]->[ addr  ];
      @CONST addr = addr + 1;
}
```

The terminating semicolon is required for all @CONST directives.

@IF

The @IF directive is used to conditionally include blocks of text into the design description. This directive tests a numeric expression and, if the expression is nonzero, causes the text within the block to be inserted in the design description. The following example causes a block of text to be included when the indicated relational expression evaluates to true:

```
@IF (bit > (msb - shiftby))
{@EXPR {ShftR} bit; = 1;}      "Shift in 1s from left
```

A terminating semicolon is not required after an @IF directive or in any other place where a terminating curly brace is provided. The expression specified after the @IF directive can include dummy arguments, so @IF directives can be included within macros.

@IFB

The @IFB directive is similar to the @IF directive, but operates on text sequences rather than on numeric values. If the text within the @IFB directive's argument is blank (has zero characters), the text located in the block will be inserted in the design description. The following example tests whether the indicated character sequence (which is a passed into the directive as a macro dummy argument) is zero length:

```
@IFB (?text)  {@MESSAGE 'The argument was blank.'}
```

Unlike other ABEL language statements, spaces are significant in @IFB directives. A common mistake is to add extra spaces around an actual or dummy argument, resulting in the directive evaluating false no matter what sequence of characters is passed into the directive:

```
@IFB ( ?text )  {@MESSAGE 'This will never print!'}
```

As with other @IF... directives, no terminating semicolon is required after the closing curly brace.

@IFNB

The @IFNB directive is the inverse of the @IFB directive and tests if the text within the directive's argument is nonblank (has one or more characters.) If the argument is nonblank, the text located in the subsequent block will be inserted in the design description. The following example tests whether the indicated character sequence (which is passed into the directive as a macro dummy argument) is nonblank:

```
@IFNB (?text)  {@MESSAGE 'The argument was not blank.'}
```

As with @IFB, no terminating semicolon is required after the closing curly brace.

@IFDEF

The @IFDEF directive tests whether an identifier name has been previously defined and, if so, includes the specified block of text in the design description. This is useful for error checking in complex macros and include files or for conditionally including blocks of ABEL statements based on arguments specified on the language compiler's command line. The following sample shows how an ABEL statement can be included in the design if a user-defined command line argument is specified:

```
MODULE mydesign (myoptimizeflag)

@IFDEF (?myoptimizeflag) {
    @DCSET                  "Turn on don't-care processing
}
```

Since the @IFDEF tests only whether the specified argument is defined, and not its value, then this example will detect only whether an argument was specified on the command line, not that argument's value. If the user typed in "ABELCOMP mydesign -arg 0", for example, the *myoptimizeflag* argument would be defined, and the @IFDEF test would still cause the text to be inserted.

@IFNDEF

@IFNDEF is the inverse of the @IFDEF directive and tests whether an identifier name is currently undefined. If the specified argument is undefined, the specified block of text is included in the design description. The following sample shows how an ABEL statement can be included in the design if a macro identifier is undefined:

```
@IFNDEF (MyFavoriteMacro) {
    @INCLUDE 'mymacros.inc'
}
```

@IFIDEN

The @IFIDEN directive allows two sequences of text characters to be compared and a subsequent block of text be included if the two sequences are identical. The comparison is case sensitive, so the following @IFIDEN test will fail:

```
@IFIDEN (A_String,a_string)
    {@MESSAGE 'They are identical'}
```

Notice that the specified sequences are not enclosed in single quotes, and therefore do not have to be ABEL strings. The text sequences can be dummy arguments, which allows @IFIDEN directives to be used within macros.

As with the @IFB and @IFNB directives, the @IFIDEN and @IFNIDEN directives are sensitive to extra spaces appearing in the argument list. The following @IFIDEN directive will not work:

```
@IFIDEN (Fred, Fred)
    {@MESSAGE 'This message will not print!'}
```

@IFNIDEN

The @IFNIDEN directive is the inverse of the @IFIDEN directive. @IFNIDEN compares two sequences of text characters and includes the subsequent block of text if the two sequences are different. The comparison is case sensitive, so the following @IFNIDEN test will pass:

```
@IFNIDEN (A_String,a_string)
    {@MESSAGE 'They are different'}
```

@REPEAT

The @REPEAT directive causes a block of text to be repeated the number of times specified. Any valid numeric expression can be used as the repeat count. When a block of text is specified using @REPEAT (or any other text-substitution directive), the text to be included will be exactly as specified between the curly brace delimiters. This means that no line terminators (line feed or equivalent) or spaces will be added if the block of text is entered as a single line. This is normally of no consequence, since multiple ABEL statements can be placed on a single line of a design description. The following example demonstrates how a @REPEAT directive can be used to create a repetitive block of test vectors:

```
test_vectors
    ([clock,Reset,Halt,select]->[Statereg,data])
     "Reset the machine and let the outputs stabilize...
@REPEAT 10  {
     [ .C. ,  1  ,  1 ,  .x. ]->[  .x.  , .x. ];
}
```

@IRP

The @IRP directive allows a block of text to be repeated *n* times, where *n* is the number of arguments appearing in the directive's argument list. The @IRP directive includes one dummy argument that is used to indicate where in the block of text the actual arguments are to be substituted each time the block of text is repeated. Each time the block of text is repeated, a new argument is substituted until all arguments have been used.

The following example of @IRP generates test vectors for six different input values:

```
test_vectors([addr15..addr0]->[databyte])
@IRP address(^hA030,^hA060,^hC030,^hC060,^hE030,^hE060)
{              ?address     ->    ^hFF; }
```

Since the @IRP directive is a text substitution mechanism, the actual arguments listed in the argument list can be of any type. If the arguments are sequences of characters that are being concatenated or used in some other way in which their precise format is important, then you must remember that any spaces found in the argument list will also be passed into the body of the directive.

@IRPC

The @IRPC directive is similar to the @IRP directive, but operates on individual characters rather than on distinct arguments. An @IRPC directives accepts one actual argument, and repeats *n* times, where *n* is the number of characters in that argument. The following example uses an @IRPC directive to generate a sequence of 20 state names and corresponding sequential values:

```
@CONST sval = 0;
@IRPC stateid (ABCDEFGHIJKLMNOPQRST) {
    State?stateid = sval; "StateA = 0, StateB = 1, etc.
    @CONST sval = sval + 1;
}
```

@EXPR

The @EXPR directive is used to insert a numeric value into the ABEL file. The numeric text can be created from any valid expression that evaluates to a number. This directive can also include an optional text string (enclosed in curly braces) that is to be appended to the beginning of the numeric value. The @EXPR directive is most often used to create signal names from complex expressions and can be used to perform set indexing operations or other functions that require compile-time name generation. For example, the following @REPEAT loop uses the @EXPR directive to repetitively index into the members of an input and output set, creating the equivalent of a shift function:

```
equations
@const index = 0;
@REPEAT 15
{@EXPR {A} index-1; = @EXPR {B} index; ;}     "An-1 = Bn
```

A terminating semicolon is required for all @EXPR directives. This terminating semicolon is not inserted into the design description, so for this example an additional semicolon is required to complete the equation produced in each iteration of the @REPEAT loop.

@SETSIZE

The @SETSIZE directive is used to determine the set width of any ABEL set expression. This directive is useful within macros or @REPEAT blocks when a set argument of unknown size is being operated on. The @SETSIZE directive can be combined with set range operations to perform complex manipulations on sets of arbitrary sizes. The following ABEL macro, for example, will cause a 4-bit value (represented by the first argument *A*) to be assigned to the highest four bits of a set of unknown size (represented by argument *B*):

```
SetHighNibble  macro(A,B)
    {  @if (@setsize ?A; != 4)
            {@message 'Set A is not 4 bits!'}
       @if (@setsize ?B; < 4)
            {@message 'Set B is too small!'}
       ?B[@setsize(?B);-1..@setsize(?B);-4] = ?A;
    };
```

Compile Options Directives

@RADIX

The @RADIX directive changes the default number system. If @RADIX is not specified, the default number system is base 10 (decimal). The @RADIX directive can be used to specify base 2 (binary), base 8 (octal), base 10 (decimal) or base 16 (hexadecimal) number systems. If you are changing the default number system repeatedly within a design description, you must remember to use the number system currently in effect when respecifying the @RADIX directive. The following example changes the radix from base 10 to base 16, and then back to base 10:

```
@RADIX 16        "Change to hexadecimal
     .
     .
     .
@RADIX A         "Change back to decimal
```

No matter what default number system is being used, the ^b, ^o, ^d or ^h radix modifiers can be used at any point to enter a number in a different number system. The statement

```
@RADIX ^d10
```

will change the default number system back to base 10 regardless of the default number system in effect.

@CARRY

The @CARRY directive allows you to specify how a large set expression (such as a counter, comparators, or adders) is converted into equations by the language compiler. Normally (when @CARRY is not specified) the compiler will flatten all equations into two-level (sum of products) equations using the rules defined in Section A.3. When @CARRY is specified, though, these expressions are translated into multilevel equations that preserve carry operations from one stage of the operation to another. The @CARRY directive accepts a numeric argument that specifies how many bits of a set expression are to be grouped between carries. This allows you to tailor the design to the specific needs of a target device or architecture.

The following example shows how @CARRY can be specified to allow a very large (56-bit) counter to be described. In this case, a value of 2 was specified for the design, meaning that the equations generated for the counter will have one carry signal (in the form of an automatically generated combinational node) for every two bits of the counter:

```
@carry 2;
[q55..q0]  :=  ([q55..q0] + 1) & !Reset;
```

Without the @CARRY directive, this design equation would expand to an impractical amount of logic (so much logic, in fact, that it would overwhelm the language compiler.)

To disable carry generation, a value of 0 can be specified in the @CARRY directive:

```
@carry 0;    "Turn off automatic node generation
```

@DCSET

The @DCSET directive enables don't-care generation and optimization for all subsequent state diagrams and truth tables. When @DCSET is specified, on-set and off-set equations (see Chapter 3 for a complete description of these terms) are generated for transitions to 1 and 0 states, respectively. Unspecified state diagram transitions or truth table input combinations become don't-cares when processed by the compiler. When @DCSET is specified, it is important to write logic descriptions that do not conflict. If the same input combination is covered by two or more transitions to different states, for example, an error will occur in the compiler or in the logic optimizer (ABELOPT).

The @ONSET directive disables don't-care generation and optimization for all subsequent logic descriptions. If @ONSET is specified (or @DCSET was not specified earlier in the design description) then the language compiler will generate only the on-set portion of the logic for state diagrams and truth tables. All input conditions not included in the on-set, including unspecified conditions and transitions, will become part of the off-set when processed by the compiler. An alternative to the @DCSET directive, if you need pin-by-pin control over don't-care processing, is to use the 'dc' signal attribute. (The @DCSET and @ONSET directives override the 'pos', 'neg', and 'dc' directives, if these signal attributes have been specified.)

@DCSTATE

The @DCSTATE directive enables don't-care generation and optimization for state diagrams outputs. When @DCSTATE is specified, all state diagram transitions (consisting of current state and transition condition combinations) are combined and logically complemented to form a set of don't-care conditions for the circuit. These don't-cares are then applied to all equations in the design. This can result in a substantial reduction of logic for state machine outputs. This processing can be extremely time consuming for large state machines, however.

The @DCSTATE directive must be used in combination with either the 'dc' directive (applied to each state machine output) or with the @DCSET directive.

@ALTERNATE

The @ALTERNATE directive instructs the compiler to use an alternative set of Boolean operator symbols. The alternate operators are those used in the PALASM language and are shown in the table of Figure A.22.

When the @ALTERNATE directive is in effect, the arithmetic operations normally represented by the +, * and /symbols are not available.

@STANDARD

The @STANDARD directive changes the Boolean operators back to the standard set.

Miscellaneous Directives

@EXIT

The @EXIT directive will cause the language compiler to stop processing the file and exit immediately without attempting to write an Open-ABEL output file. This directive is useful for error checking within complex

Boolean Operation	Standard Symbol	Alternate Symbol
NOT	!	/
AND	&	*
OR	#	+
XOR	$	:+:
XNOR	!$	:*:

Figure A.22 ABEL alternative operators

macros, or when a design description requires that arguments be specified on the command line. The following example uses this feature to check that the user defined command line arguments have been properly specified:

```
MODULE MyModule (myoption)

@IFNDEF (?myoption) {
    @MESSAGE 'You must specify an argument'
    @EXIT
}
```

@MESSAGE

The @MESSAGE directive (which we have used in examples of previous directives) allows you to specify a text string that is to be displayed when the language compiler is running. @MESSAGE directives are useful for debugging complex design descriptions and for reporting status during long compilations.

@PAGE

The @PAGE directive instructs the language compiler to send a form-feed character to the list file. If no list file is been specified on the command line, this directive is ignored.

A.10 REFERENCES

Data I/O Corporation, *ABEL User's Guide*, Data I/O, Redmond, WA, 1988.

B

The ABEL Software

ABEL (Advanced Boolean Expression Language) is a design language and set of supporting software programs that allow complex logic designs to be entered, compiled, simulated, and mapped into a programmable logic device. To create a design using ABEL, you use the text editor provided in the ABEL design environment or any text editor that produces standard ASCII files. When your design is ready for processing, you invoke the various ABEL programs to simulate the design, choose a target device, and produce a JEDEC format data file for device programming. The ABEL design environment is easy to use, and the on-line help available in the software contains detailed information about program options and operation.

The software provided with this book includes all the significant features of the ABEL version 5.0 software. These features include the following:

- Extended memory support
- Windows compatible user interface
- On-line help
- ABEL 5.0 language features
- Espresso logic reduction
- Automatic pin and node assignment (device fitting)
- Simulation (equation and JEDEC)

This software has been made available by Data I/O Corporation. Data I/O releases new versions of software on a regular basis, and new features are frequently being added to the language and to the product in general. For this reason, the commercial software available from Data I/O at any given time may be significantly different from the software supplied with this book.

Device Support

The software supplied with this book supports a limited number of devices. These devices are representative of the range of PLDs available, and most of them are available from several sources. The primary manufacturer of each device is listed along with the device name in the chart of Figure B.1.

The list of devices includes one complex PLD that requires additional information related to device fitting and optimization. This device, the Mach 215 complex PLD, is described in detail in Appendix C. We do not recommend that you try using the Mach 215 device until you have a complete understanding of simpler configurable devices, such as the 18CV8.

Device	Manufacturer	Pins	Notes
P16L8	AMD[1], others	20 (DIP)	Combinational PAL (neg. polarity)
P16H8	AMD, others	20 (DIP)	Combinational PAL (pos. polarity)
P16P8	AMD, others	20 (DIP)	Combinational PAL (prog. polarity)
P16R4	AMD, others	20 (DIP)	PAL, D-type flip-flops (4)
P16R6	AMD, others	20 (DIP)	Registered PAL, D-type flip-flops (6)
P16R8	AMD, others	20 (DIP)	Registered PAL, D-type flip-flops (8)
P20X8	AMD, others	24 (DIP)	Registered PAL, XOR outputs
E0320	Altera, others	20 (DIP)	Configurable macrocell PLD
P18CV8	ICT[2], others	20 (DIP)	Configurable macrocell PLD
Mach 215	AMD, others	44 (LCC)	Complex PLD with buried nodes

[1]Advanced Micro Devices, Incorporated
[2]International CMOS Technology, Incorporated

Figure B.1 Devices supported by the ABEL software included with this book

Installation and System Requirements

The ABEL software requires an IBM-compatible computer with an 80386 or higher processor and at least 4 MB of extended memory to operate. The software is fully compatible with Microsoft Windows, and can be operated from the DOS command line, or from within the Windows environment.

To install the ABEL software, insert disk number 1 into your disk drive and type

```
A:INSTALL
```

at the DOS prompt (assuming the disk drive is drive A). The installation software will prompt you for an installation directory and will copy the files accordingly. The installation software will also, if requested, modify your AUTOEXEC.BAT and CONFIG.SYS files to add the software to your system path. If you do not wish to have the installation software modify your AUTOEXEC.BAT and CONFIG.SYS files, you must add the installation directory to your path and add the ABEL5DEV environment variable to your environment. If the software is installed in the C:\ABEL directory, for example, you would add C:\ABEL to your path and add the statement

```
SET ABEL5DEV=C:\ABEL
```

to your AUTOEXEC.BAT file.

You should also check to make sure that the FILES variable in your CONFIG.SYS file is set to at least 30 before running the ABEL software.

Running the Software

To invoke the ABEL design environment from DOS, type the command

```
ABEL5
```

After you have invoked the ABEL5 design environment, you can load a design by selecting Open from the File menu. As an alternative, you can run the ABEL5 design environment from within Windows, either by creating a File Manager association (associate files that have a .ABL file name extension with the ABEL5.EXE program) or by creating a new program item. You can customize your Windows installation of ABEL5 (for example, have it come up in a window, rather than full screen) by creating a .PIF file. For detailed information on how to use Windows .PIF files with DOS-based applications, refer to your Windows documentation.

For detailed information about the operation of the ABEL software and the ABEL design environment, refer to the on-line help available in the Help menu. For help on any of the menus, dialogue boxes, or program options, press function key F1.

Note: The ABEL software is memory-extended using a DPMI-compatible DOS extender. If you are unable to compile ABEL designs due to DOS extender and memory manager incompatibilities (indicated by a DPMI-related error message appearing during the compile operation), you should run the ABEL software from a Windows DOS box, in either full-screen or windowed mode, rather than running the software directly from DOS.

C

Mach 215 Special Device Information

This appendix describes the Mach 215 device and the Mach 215 device fitting software that is included with this book. Several examples are presented to show how various fitting strategies can be used with the Mach 215.

C.1 MACH 215 DEVICE ARCHITECTURE

General Architecture

The Mach 215 device has four PAL blocks whose inputs are connected by a switch matrix (see Figure C.1). Configurable output macrocells are tied to the outputs of the PAL blocks, with each PAL block having eight macrocells. The device also has four dedicated inputs and two global clocks that go directly into the switch matrix. The global clocks can be used as both clocks and inputs. The function of the switch matrix is to take the outputs of the macrocells and the connections of the dedicated inputs and clocks and provide a routing path to the inputs of the PAL blocks.

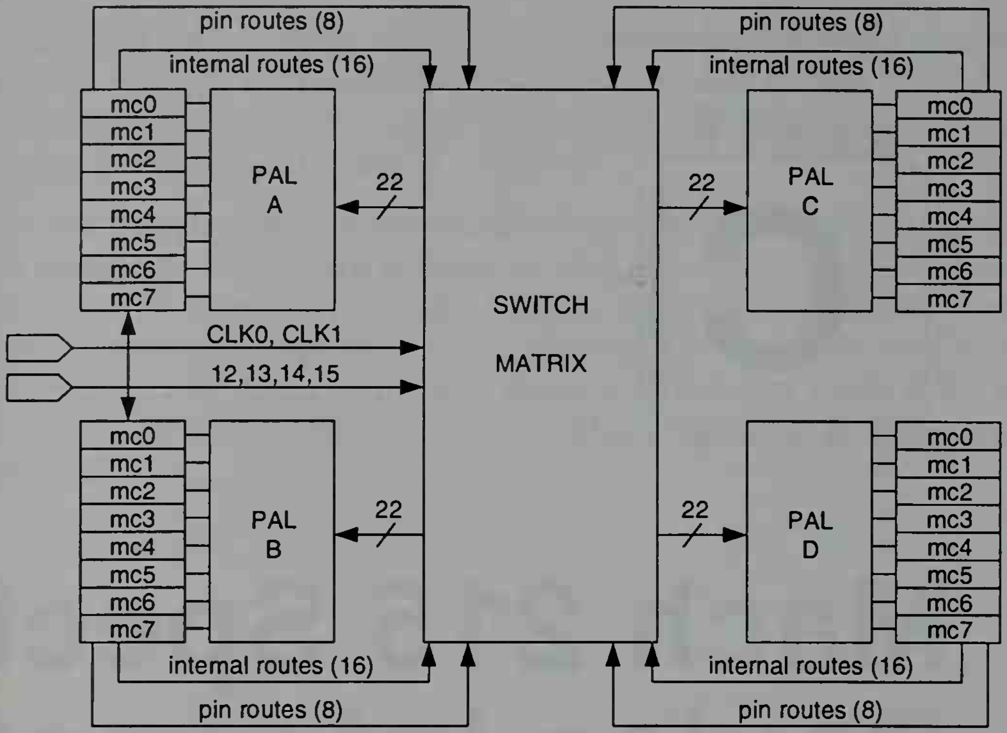


Figure C.1 Mach 215 device architecture

Device Resources

The table of Figure C.2 lists the resources of the Mach 215 device.

Resource	Count
Pins	44
I/O Pins	32
Clock Pins	2
Input Pins	4
Macrocells	32
PAL Blocks	4
Inputs to PAL Blocks	22

Figure C.2 Mach 215 device resources

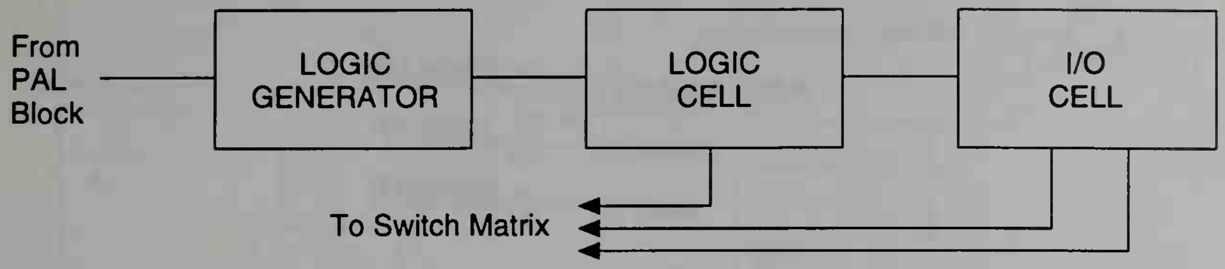


Figure C.3 Mach 215 output macrocell block diagram

Mach 215 Macrocell

The macrocell of the Mach 215 device is made up of three components: the logic generator, the logic cell, and the I/O cell. Figure C.3 shows how these components are connected together and to the switch matrix and PAL block.

Logic Generator

The function of the logic generator is to implement the combinational logic equations for the macrocell. The logic generator has four local product terms that can be connected to the local OR gate or steered away to the OR gate of the macrocell above or below. With product term steering, the logic generator can implement a function of up to 12 product terms (Figure C.4). The macrocells on the top and bottom of the PAL block can only generate eight product terms because they do not have a macrocell above or below them, respectively.

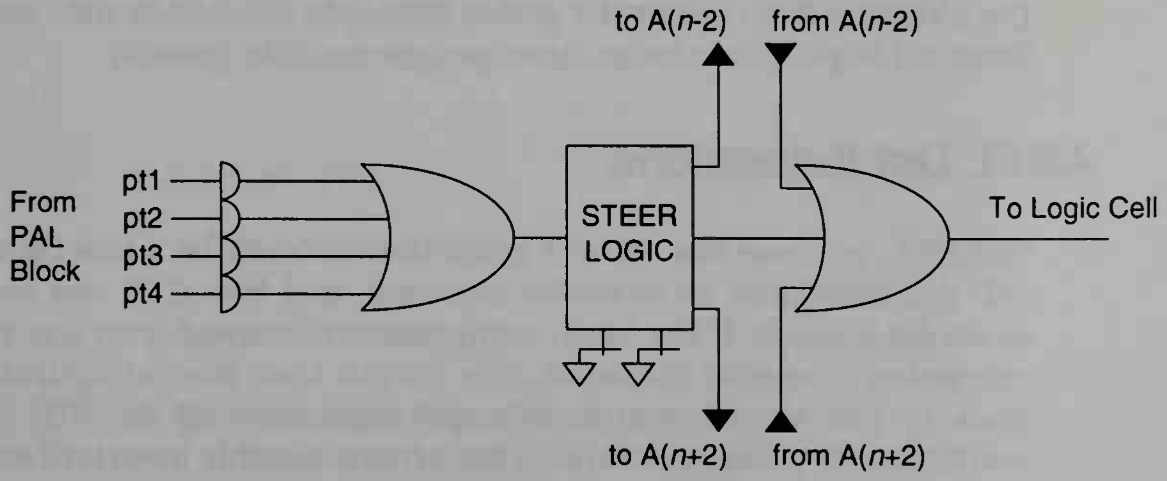


Figure C.4 Mach 215 logic generator

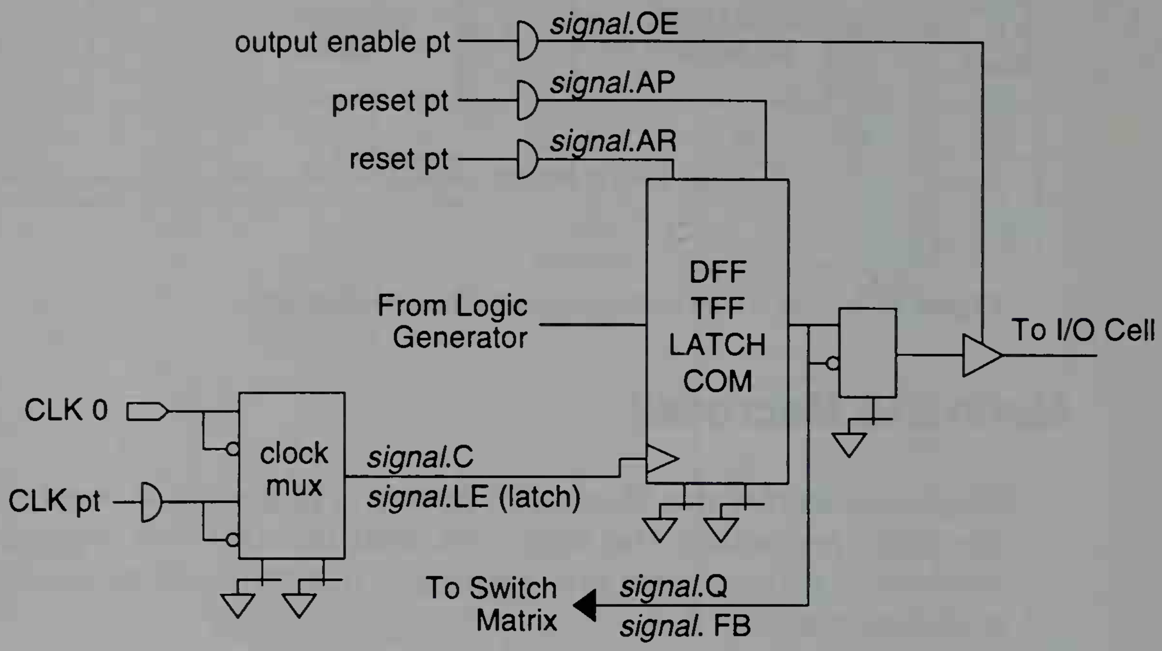


Figure C.5 Mach 215 Logic cell

Logic Cell

The logic cell can be used to register the data from the logic generator function and pass the information to the I/O cell (Figure C.5). The logic cell can be configured as a D flip-flop, T flip-flop, active-low latch, or combinational buffer. If configured as a storage element, the reset product term and the preset product term are used to reset the register or latch to 0 or preset it to 1, and the clock multiplexer (mux) generates the clocking from either the global clock pin (CLK 0) or the clock product term (CLK pt). The clocks have programmable polarity.

ABEL Dot Extensions

In ABEL, you use the .AR dot extension to describe a reset function, the .AP dot extension to describe a preset, and the .CLK dot extension to describe a clock. If the latch configuration is used, you use the .LE dot extension to enable the latch. The output from the configurable register goes to the switch matrix (via dot extension .Q or .FB) and to the configurable polarity control. This programmable inverter feeds into the tristate buffer, which is controlled by the output enable product term (dot extension .OE).

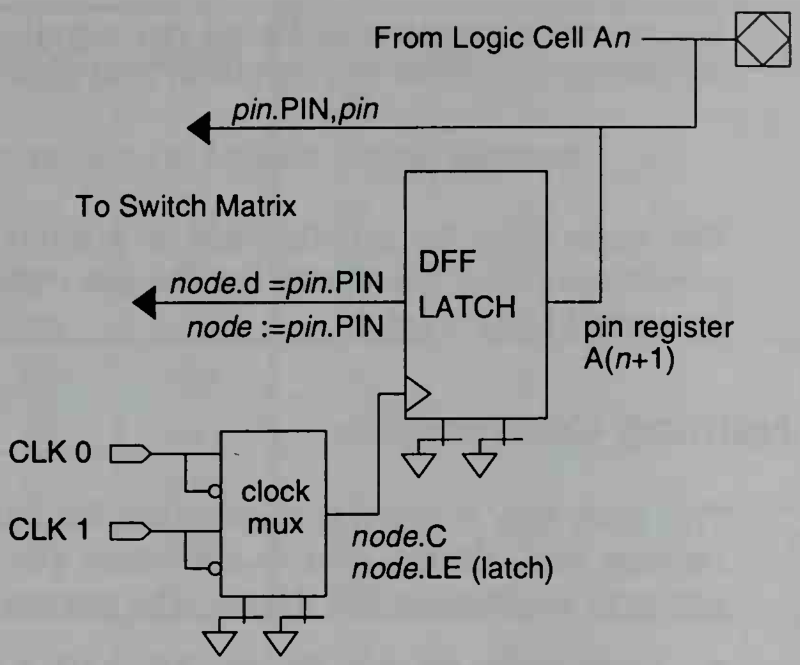


Figure C.6 Mach 215 I/O cell

I/O Cell

Figure C.6 shows the third component of the macrocell, the I/O cell. The I/O cell consists of an I/O pin and a pin register. The I/O pin connects to the switch matrix (.PIN or no dot extension) and to the pin register. The pin register can be configured as a D flip-flop or an active-low latch. The clock for the pin register comes from one of two global clock pins (CLK0 or CLK1) and can be used in either positive or negative polarity. The pin register is generated when the ABEL equation

```
node := pin.PIN
```

or

```
node := pin
```

or

```
node.d= pin.PIN
```

is used. The pin signal can also be inverted:

```
node:=!pin
```

For inverted (negative polarity) pin signals, the fitter inverts the signal on the input side of the equation and generates the message

```
        Warning 4445: Signal signal_name is an istype 'neg'.
```

The node must be a D flip-flop or a latch and cannot have a reset or preset specified. The clock for the pin register is specified with *node*.C (or *node*.LE for a latch).

Naming Convention

The fitter has a naming convention for both the logic cell and the pin register that allows you to associate pin and node numbers with a physical location in the device. The names for block A are as follows:

- Logic cells: A0, A2, A4, A6, A8, A10, A12, A14
- Pin registers: A1, A3, A5, A7, A9, A11, A13, A15

The logic cell A*n* is tied to the pin register A(*n* + 1). Block B, C, and D have logic cells B*n*, C*n*, and D*n* and pin registers B(*n* + 1), C(*n* + 1), and D(*n* + 1) respectively. In all cases, the logic cells have even numbers and the pin registers have odd numbers.

Figure C.7 lists the pin and node numbers of the MACH 215 device and their relationship with logic cells and pin registers. You can use this information to assign signals to physical locations on the device. The examples in the next section show how this is done.

C.2 DESIGN STRATEGIES

When a large design is placed into a Mach 215 by the device fitting software, many resource assignments must be made in addition to pin assignment and macrocell configuration. Because of the segmented architecture of the Mach 215, several factors affect the possibility of a given design fitting into the device. These factors include the following:

- Pin and node preassignments
- Signal groups
- Fitter options
- Design specification
- Routing and dot extensions

Pin	Name	Node	Name	Pin	Name	Node	Name
1	GND	--	--	23	GND	--	--
2	A0	45	A1	24	C0	61	C1
3	A2	46	A3	25	C2	62	C3
4	A4	47	A5	26	C4	63	C5
5	A6	48	A7	27	C6	64	C7
6	A8	49	A9	28	C8	65	C9
7	A10	50	A11	29	C10	66	C11
8	A12	51	A13	30	C12	67	C13
9	A14	52	A15	31	C14	68	C15
10	I0	--	--	32	I3	--	--
11	I1	--	--	33	I4	--	--
12	GND	--	--	34	GND	--	--
13	CLK0	--	--	35	CLK1	--	--
14	B14	60	B15	36	D14	76	D15
15	B12	59	B13	37	D12	75	D13
16	B10	58	B11	38	D10	74	D11
17	B8	57	B9	39	D8	73	D9
18	B6	56	B7	40	D6	72	D7
19	B4	55	B5	41	D4	71	D5
20	B2	54	B3	42	D2	70	D3
21	B0	53	B1	43	D0	69	D1
22	VCC	--	--	44	VCC	--	--

'--' indicates no register for this pin
CLK0, CLK1: Global clock signals. May be used as inputs
I0, I1, I3, I4: Dedicated input signals.

Figure C.7 Mach 215 pin and node numbers

You can control the fitting process by preassigning pins and nodes and specifying Keep Pin Assignments in the Fit Options dialog box of the ABEL design environment. The preassignment of a single pin can have a large impact on the rest of the logic in the design. For example, placing an output on a macrocell within a PAL block restricts fitting in the following ways:

- The equation for the output must be placed in the macrocell(s) within that block.
- The output's preset, reset, and output enable product terms must be placed within that block.
- All inputs used by the output equation and its preset, reset, and output enable product terms must be routed to that block.
- If the output's feedback is used, it must be routed to all equations that require it.

A preassignment can affect the fitting of other signals and may cause fitting to fail. Make preassignments only when necessary and with consideration for the Mach 215 architecture.

Reserving Macrocells

Preassignments can be used to reserve a logic cell within a device by assigning a signal to a pin, but not using the pin in the logic description. The fitter treats the unused signal as an output and reserves the logic cell and the four local product terms of the logic generator. Unused signals without preassignments are removed by the fitter and do not appear in the fitted design.

Using Clock Pins as Inputs

The clocks in the Mach 215 device can be used as inputs. The design can use up to six dedicated inputs (four coming from dedicated input pins and two from the clock pins).

Switch Matrix Restrictions

The switch matrix routes the feedback signals from the macrocells to the PAL block inputs. The sparseness of the switch matrix increases the speed of the Mach 215 device, but it also limits the number of ways a feedback path can be routed. A PAL block can have more inputs than available paths through the switch matrix. In this case, the extra inputs cannot be used.

Clock Restrictions

The fitter takes all the following into account and will find a clock assignment that will minimize routing and maximize the number of pin registers used.

- The logic cell is tied to the global clock CLK0 or to the clock product term.
- When CLK0 is used, that signal has a direct connection to the logic cell and does not need to be routed through the switch matrix.
- All inputs to the clock product term must be routed.
- The pin register is tied to global clocks CLK0 or CLK1. These clocks have direct connections to the pin register and are not routed. There is no clock product term on the pin register.

When making preassignments, you should also take these clock restrictions into account to make the most efficient use of the device.

Signal Groups

The GROUP property statement is another way to control the fitting process. Property statements are an extension to the ABEL language that allow device-specific information to be passed to ABEL fitters. The syntax for the GROUP property statement is

```
AMDMACH property 'GROUP block signal-list';
```

where *block* is the name of the PAL block (A, B, C, or D), and *signal-list* is a list of signals (separated by spaces) that are to be placed in that block.

The fitter attempts to place all the signals of the group (the *signal-list*) within the specified block. The fitter must place all the specified signals in the block or fitting will be unsuccessful. If no signal groups are specified, the fitter tries to partition the logic across the device so that the amount of logic in the blocks is balanced and the routing between the blocks is minimized.

Like preassignments, specifying signal groups can affect how the design is fit in the device. Closely related signals make good grouping candidates. An output signal is closely related to another output signal if it uses the same reset, preset, and output enable resources and has similar inputs.

Grouping output signals is usually more effective in controlling the partitioning process than grouping inputs. Forcing an input to a block causes the fitter to find a route from that input to all blocks that use it, but has little effect on the partitioning for other signals. On the other hand, grouping a single output could give the fitter a different starting point and dramatically different results.

Design Specification

The way a logic description is specified affects the fitter's solution. You can improve the chance of a fit by verifying that the device has enough inputs, outputs, product terms, and clocks, and by the way you specify the logic. The "Design Examples" section gives several examples on how to specify logic to improve the chances of a fit in the Mach 215.

Routing and Dot Extensions

Logic cells have two feedback paths: one from the I/O pin and one from the logic cell internal location (see "Internal Feedback" for more information). The router in the Mach fitter routes from the internal feedback path if you specify .Q or .FB and from the pin feedback path for .PIN. If a signal is declared as a node, the router uses only the internal path. If a route is not available internally, the router attempts to route the signal from the pin and displays the message

```
Warning 4443: No fit for internal signal -- changing to
PIN to re-try.
```

If the following conditions are met, the router routes from the pin or internally, normalizing the pin polarity, if necessary, to maintain the same functionality:

- A signal is declared as a pin
- An explicit route is not specified (with .Q, .FB, or .PIN)
- The tristate control is not specified (that is, it is always an output)

For example, the router may route from the pin route to blocks A and B and from the internal route to blocks C and D.

C.3 DESIGN EXAMPLES

Example files for the Mach 215 device can be found in the \MACH215 directory located on the distribution disk and installation directory.

Automatic Equation Splitting

The fitter automatically splits equations that have more than 12 product terms for the Mach 215 device. For example, the design shown in Figure C.8 has a signal named *Parity* that has 36 product terms. When fitting to the Mach 215 device, the fitter generates and places the following message into the fitter report file:

```
Warning 4421: Signal Parity has 36 terms (max is 12).

Number of nodes created = 3

  Output Parity is now a function of:

    Parity__0 with 12 terms.

    Parity__1 with 12 terms.

    Parity__2 with 12 terms.
```

The fitter rewrites the equation for *Parity* so that it is a function of the created subequations, *Parity_ _0*, *Parity_ _1*, and *Parity_ _2*. These subequations have 12 product terms each and implement the 36 product terms. *Parity* itself becomes a function of three product terms. To see the actual equations that are generated by the fitter, process the split215.abl design example and select the View/Fitted Equations menu item in the ABEL design environment

After an equation is split, it has an extra level of delay. *Parity* above will have two levels of delay. The fitter will always generate a message when an equation is split, and will create new nodes by appending *_ _number* to the signal being split. (For this reason, you should avoid using two consecutive underscore characters when naming signals in your ABEL source file.)

```
MODULE split215
TITLE 'Illustrates splitting in Mach 215 fitter'

DECLARATIONS
        split215 DEVICE  'mach215a';
"Inputs
        Clk,M1,M0                          PIN;
        Reset1,Reset2,Preset1,Preset2      PIN;
"Outputs
        Carry0,Cnt7Reg                     PIN  ISTYPE 'REG,BUFFER';
        Cnt7,Cnt15up,Cnt15dn,Ld0           PIN  ISTYPE 'COM';
        Parity, Pulse16,Div32              PIN  ISTYPE 'COM';
        Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0            PIN  ISTYPE 'REG_T,BUFFER';
"Sets and Constants
        Z,X,C = .Z.,.X.,.C.;
        Count   = [Q7..Q0];
        Mode    = [M1,M0];
        Up      = [ 0, 0];
        Down    = [ 0, 1];
        Load    = [ 1, X];
        Reset   = [Reset2 ,Reset1 ];
        Preset  = [Preset2,Preset1];

EQUATIONS
      Count.t   = ((Count.q + 1) & (Mode == Up)
                #  (Count.q - 1) & (Mode == Down)
                #   Count        & (Mode == Load)) $ Count.q;
      Count.oe  = (Mode == Up) # (Mode == Down);
      Count.clk = Clk;
      Count.ar  = Reset1  & Reset2;
      Count.ap  = Preset1 & Preset2;
      Carry0   := (Count.q == 254) & (Mode == Up)
                # (Count.q == 1)   & (Mode == Down);
      Cnt7Reg  := (Count.q == 7);
      [Carry0,Cnt7Reg].clk = Clk;
      [Carry0,Cnt7Reg].AR = 0;
      [Carry0,Cnt7Reg].AP = 0;
      Cnt7      = (Count.q == 7);
      Cnt15up   = (Count.q == 15) & (Mode == Up);
      Cnt15dn   = (Count.q == 15) & (Mode == Down);
      Ld0       = (Count.q == 0)  & (Mode == Load);
      Pulse16   = (Count.q == [X,X,X,X,1,1,1,1]);
      Div32     = (Count.q == [X,X,X,1,X,X,X,X]);
      Parity    = (Q6 $ (Q5 $ Q4)) $ ((Q3 & Q2) $ (Q1 & Q0)) ;

DECLARATIONS
      XOR,TMP   PIN  ISTYPE 'COM';

EQUATIONS
      XOR       = (TMP $ Q6.q) !$ Q7.q;
      TMP       = Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & q5.q;

END
```

Figure C.8 Design that requires equation splitting

Intermediate Expressions

The Mach 215 device limits the number of product terms allowed on any one macrocell, as well as the number of inputs to the PAL block. Thus, the number of product terms in an equation and the number of inputs that the product terms use affect the chances for a fit. Equations that have more product terms than allowed by the device are automatically split into multiple levels of logic by the fitter. While this does generate a viable solution, your knowledge of the logic may help you to create a more efficient solution.

Whenever possible, specify intermediate expressions: subequations used within two or more other equations. This specification introduces an extra level of delay, but reduces the number of macrocells used. Intermediate expressions can be used to decrease the number of product terms required and to decrease the number of inputs required for one or more product terms.

Probably the most useful multilevel synthesis technique is rewriting two or more outputs so that they are functions of an intermediate variable. (This technique was described in Chapter 3.)

Decreasing the Number of Product Terms

In some cases, there are no subexpressions common to two or more outputs. In this case, splitting off a piece of logic to reduce the number of product terms can increase the chance of a fit. The fitter automatically splits equations that have too many product terms in an attempt to find a split that minimizes the amount of routing required and the number of macrocells needed to implement the new equations.

You can also do manual equation splitting, if required. For an equation with too many product terms, creating an intermediate variable is done as follows.

Original equation:

```
x = a # b # c # d # e # f # g # h # i;
```

Split equation:

```
tmp = a # b # c # d ;
x = tmp # e # f # g # h # i;
```

Both of the above implementations require three macrocells. However, the second uses an additional level of logic, and the fitter now has two smaller pieces of logic to fit. The signal *tmp* can be placed in one block, and the signal x can be placed in another, so all the inputs do not need to be routed to one block.

Decreasing the Inputs to a Product Term

If the number of inputs to a product term exceeds the number of inputs to a PAL block, the fitter cannot place the signal that uses that product term. Even if the number of inputs in the product term is less than is allowed in a PAL block, a product term may not fit because all its inputs must be routed to that block, and no route may be available in the switch matrix.

A technique similar to that used in the previous section can be used to split large AND terms. In the Mach 215 device, each PAL block can have up to 22 inputs. Any one AND term can have at most 22 inputs or it will not fit into a block. You may want to split AND terms that are smaller than the PAL input limit but large enough to cause routing problems. Split large AND terms as follows:

Original AND term:

```
x = a & b & c & d & e & f & g & h & i;
```

Split AND term:

```
tmp = a & b & c & d & e;
x = tmp & f & g & h & i;
```

The fitter will route the second solution more easily.

The GROUP Property Statement

The source file listed in Figure C.9 shows a sample use of the GROUP property statememt.

Internal Feedback

The MACH 215 architecture allows two feedback paths from the logic cell register: the pin and an internal path. Pin feedback is specified by using only the signal name of the pin (or the signal name with a .PIN

```
MODULE bus215
TITLE 'Bus-loadable 8-bit up/down counter   Data I/O Corp '
        buscntr  DEVICE  'mach215a';
"Inputs
        Clk,M1,M0                           PIN;
        Reset1,Reset2,Preset1,Preset2       PIN;
"Outputs
        Carry0,Cnt7Reg                      PIN  ISTYPE 'REG,BUFFER';
        Cnt7,Cnt15up,Cnt15dn,Ld0            PIN  ISTYPE 'COM';
        Pulse16,Div32                       PIN  ISTYPE 'COM';
        Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0             PIN  ISTYPE 'REG_T,BUFFER';
"Sets and Constants
        Z,X,C = .Z.,.X.,.C.;
        Count    = [Q7..Q0];
        Mode     = [M1,M0];
        Up       = [ 0, 0];
        Down     = [ 0, 1];
        Load     = [ 1, X];
        Reset    = [Reset2 ,Reset1 ];
        Preset   = [Preset2,Preset1];

AMDMACH property 'GROUP A Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0';
AMDMACH property 'GROUP B Carry0 Cnt7 Cnt15up';
AMDMACH property 'GROUP B Cnt15dn Cnt7Reg Pulse16 Div32';

EQUATIONS
      Count.t    = ((Count.q + 1) & (Mode == Up)
                 #  (Count.q - 1) & (Mode == Down)
                 #   Count        & (Mode == Load)) $ Count.q;
      Count.oe   = (Mode == Up) # (Mode == Down);
      Count.clk = Clk;
      Count.ar   = Reset1  & Reset2;
      Count.ap   = Preset1 & Preset2;
      Carry0    := (Count.q == 254) & (Mode == Up)
                 # (Count.q == 1)   & (Mode == Down);
      Cnt7Reg   := (Count.q == 7);
      [Carry0,Cnt7Reg].clk = Clk;
      [Carry0,Cnt7Reg].AR = 0;
      [Carry0,Cnt7Reg].AP = 0;
      Cnt7       = (Count.q == 7);
      Cnt15up    = (Count.q == 15) & (Mode == Up);
      Cnt15dn    = (Count.q == 15) & (Mode == Down);
      Ld0        = (Count.q == 0)  & (Mode == Load);
      Pulse16    = (Count.q == [X,X,X,X,1,1,1,1]);
      Div32      = (Count.q == [X,X,X,1,X,X,X,X]);
DECLARATIONS
      XOR,TMP    PIN  ISTYPE 'COM';
EQUATIONS
      XOR        = (TMP $ Q6.q) !$ Q7.q;
      TMP        = Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q;
END
```

Figure C.9 Design with GROUP property statements

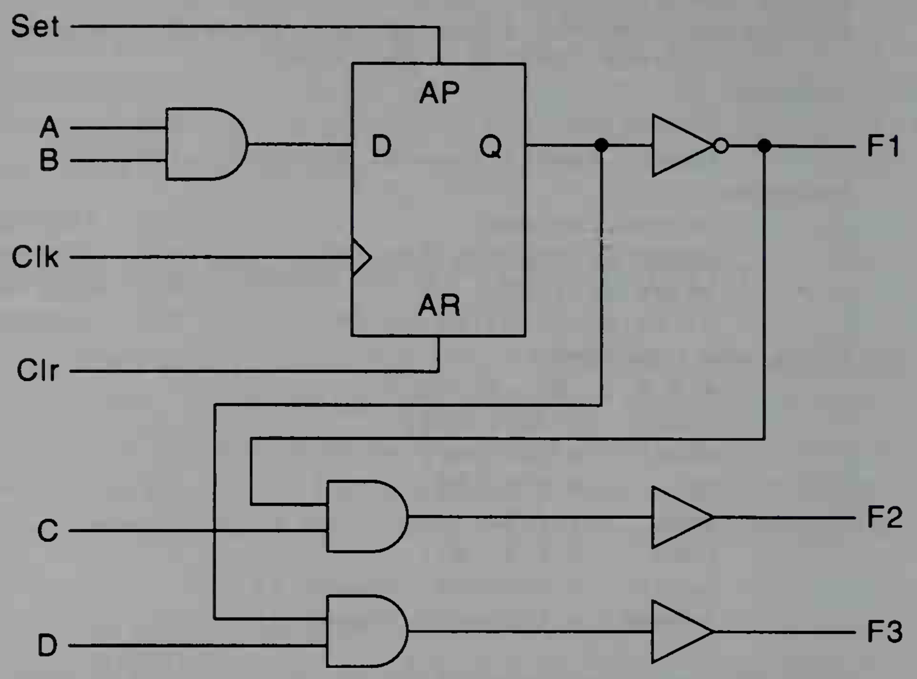


Figure C.10 Dual register feedback circuit

extension). Internal register feedback is specified with the signal name and a .FB or .Q extension. The internal combinational path is specified with the signal name and .D. No nodes are declared to specify these internal feedback paths.

The schematic in Figure C.10 shows a simple registered circuit with both feedback paths used. Figure C.11 shows how to specify both feedback paths with ABEL.

F1.Q is the output from the register before the output inverter, and *F1.PIN* is the output path after the inverter. The fitted equations are shown below:

```
F1.D = (A & B);
F1.C = (Clk);
F1.AR = (Clr);
F1.AP = (Set);
F2 = (C & F1.PIN);
F3 = (D & F1.Q);
```

The circuit shown in the schematic of Figure C.10 can also be described with the ABEL description shown in Figure C.12. Note that since *F1.FB* is normalized to the pin polarity, *!F1.FB* must be used to get the same functionality as *F1.Q* in the previous example.

```
module ex215a
title 'Dual feedback in AMD MACH'

        ex215a   device  'MACH215a';
        A,B,C,D          pin;
        Clk,Clr,Set      pin;
        F1               pin istype 'reg_D,invert';
        F2,F3            pin;

equations
        F1.D     = A & B;
        F1.Clk   = Clk;
        F1.AR    = Clr;
        F1.AP    = Set;
        F2 = C & F1.PIN;     "Pin Feedback
        F3 = D & F1.Q;       "Internal Feedback

test_vectors
       ([A,B,C,D,Clk,Clr,Set] -> [F1 ,F2 ,F3 ])
        [0,0,0,0, 0 , 0 , 1 ] -> [ 0 , 0 , 0 ];
        [0,0,1,1, 0 , 0 , 0 ] -> [ 0 , 0 , 1 ];
        [0,0,0,0, 0 , 1 , 0 ] -> [ 1 , 0 , 0 ];
        [0,0,1,1, 0 , 0 , 0 ] -> [ 1 , 1 , 0 ];
        [1,1,0,0,.C., 0 , 0 ] -> [ 0 , 0 , 0 ];
        [0,0,1,1, 0 , 0 , 0 ] -> [ 0 , 0 , 1 ];
        [0,0,1,1,.C., 0 , 0 ] -> [ 1 , 1 , 0 ];
        [1,1,1,1,.C., 0 , 0 ] -> [ 0 , 0 , 1 ];
end
```

Figure C.11 Specifying register feedback paths

```
module ex215b
title 'Dual feedback in AMD MACH'
        ex215b   device  'MACH215a';

        A,B,C,D          pin;
        Clk,Clr,Set      pin;
        F1               pin istype 'reg,invert';
        F2,F3            pin;

equations
       !F1      := A & B;
        F1.Clk   = Clk;
        F1.AR    = Clr;
        F1.AP    = Set;

        F2 = C & F1.PIN;  "Pin Feedback
        F3 = D & !F1.FB;  "Internal Feedback
end
```

Figure C.12 Specifying register feedback paths with pin-to-pin dot extensions

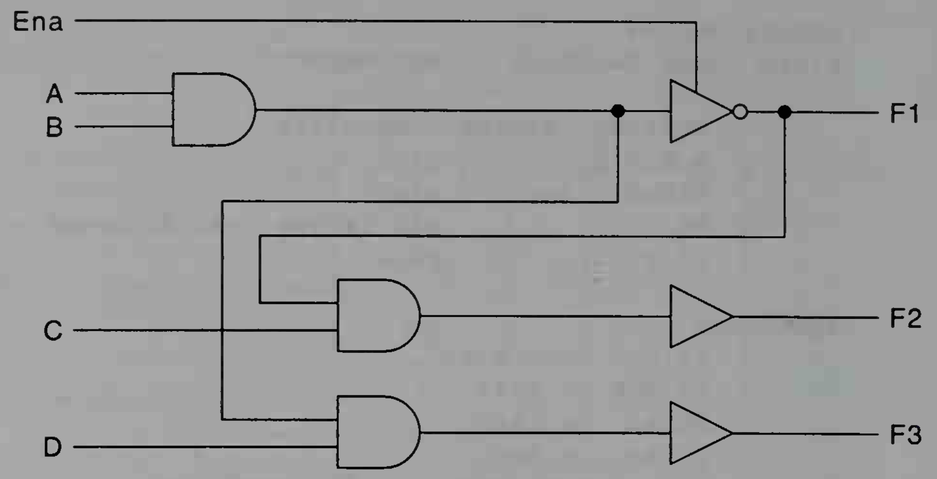


Figure C.13 Dual feedback combinational circuit

Combinational Feedback

Combinational designs may also use dual feedback. The combinational internal feedback is specified by using the .D extension on the right side of an equation. The circuit in Figure C.13 is described in ABEL in Figure C.14.

Buried Registers

A buried register circuit is shown in Figure C.15 and its ABEL equivalent description is shown in Figure C.16. All registers in the Mach 215 are tied to pins. To bury the register, the fitter assigns the ABEL node to a pin number. This prevents the pin from being used as an output, but it still can be used as an input. The example shown in Figure C.16 demonstrates how you can preassign both a node and input pin to the same location. (The ABEL compiler generates a warning that two signals are tied to the same location, but this is a valid operation for the Mach 215 device.)

Input Registers

The Mach 215 fitter program automatically detects and assigns input pin registers when you write an ABEL node that is a function of a single pin. The pin can be either an input or output. You can also force a pin register with preassignments by assigning the node to the pin register directly below the pin. An input register circuit is shown in Figure C.17 and its ABEL description is shown in Figure C.18.

```
module ex215c
title 'Dual feedback on combinational output'

        ex215c    device  'MACH215a';

        A,B,C,D,Ena       pin;
        F1                pin istype 'com,invert';
        F2,F3             pin;

equations
        F1 = A & B;
        F1.OE = Ena;

        F2 = C & F1.PIN;     "Pin Feedback
        F3 = D & F1.D;       "Internal Feedback

test_vectors ([A,B,C,D,Ena] -> [F1 ,F2 ,F3 ]) "Array feedback
              [0,0,0,0, 0 ] -> [.Z., 0 , 0 ];
              [0,0,0,0, 1 ] -> [ 0 , 0 , 0 ];
              [1,1,0,0, 1 ] -> [ 1 , 0 , 0 ];
              [1,1,1,0, 1 ] -> [ 1 , 1 , 0 ];
              [1,1,1,1, 1 ] -> [ 1 , 1 , 1 ];
              [1,1,1,1, 1 ] -> [ 1 , 1 , 1 ];
              [1,1,0,1, 1 ] -> [ 1 , 0 , 1 ];

test_vectors ([A,B,C,D,Ena,F1 ] -> [F1 ,F2 ,F3 ]) "Pin feedback
              [0,0,0,0, 0 ,.X.] -> [.Z., 0 , 0 ];
              [1,1,1,1, 0 , 0 ] -> [.X., 0 , 1 ];
              [1,1,1,1, 0 , 1 ] -> [.X., 1 , 1 ];

end
```

Figure C.14 Dual combinational feedback described with ABEL

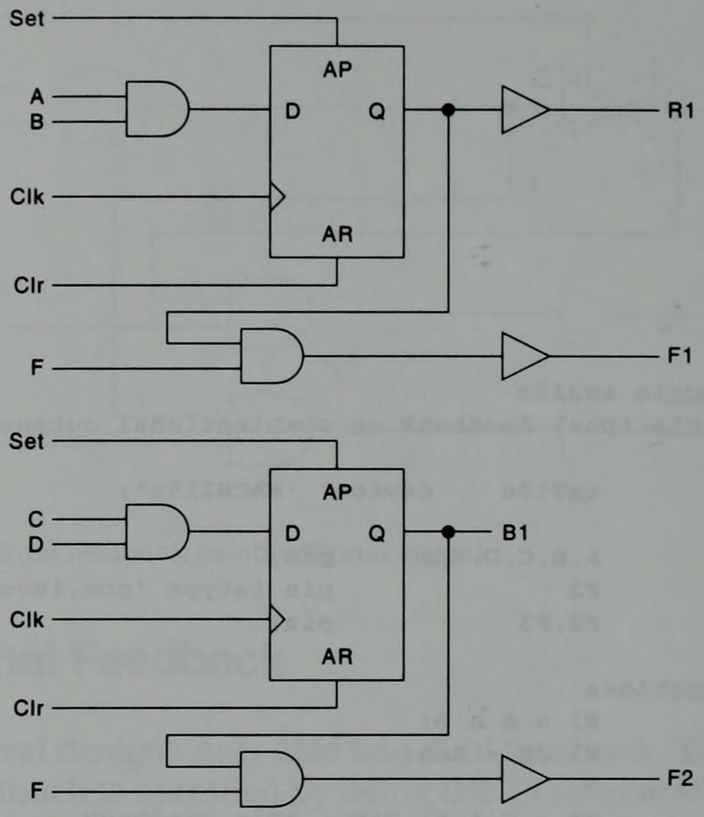


Figure C.15 Buried register circuit

```
module ex215d
title 'Buried register example  Data I/O Corp.'

        ex215d   device   'mach215a';
        A                 pin 2;
        B,C,D,E,F         pin;
        Clk,Set,Clr       pin;
        R1                pin     istype 'reg_D,buffer';
        B1                node 2  istype 'reg_D,buffer';
        F1,F2             pin     istype 'com';

equations
        R1.d      = A & B;
        R1.clk    = Clk;
        R1.ar     = Clr;
        R1.ap     = Set;
        F1        = R1.Q & E;
        B1.d      = C & D;
        B1.clk    = Clk;
        B1.ar     = Clr;
        B1.ap     = Set;
        F2        = B1.Q & F1;
end
```

Figure C.16 Buried register described with ABEL

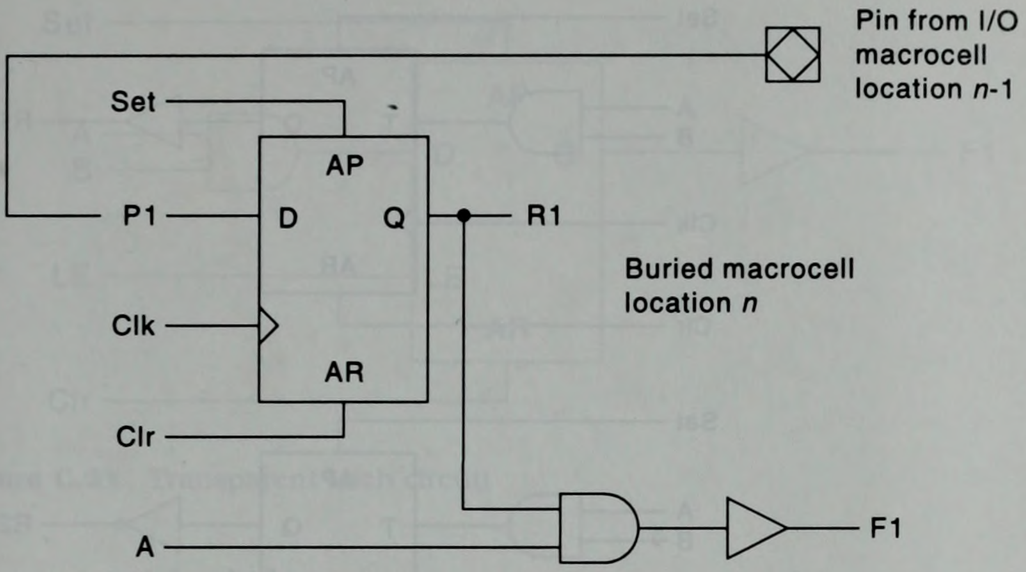


Figure C.17 Input register circuit

```
module ex215e
title 'Input register example  Data I/O Corp.'

        ex215e              device  'Mach215a';
        A,B                 pin;
        Clk                 pin;
        R1,R2               node    istype 'reg';
        F1                  pin     istype 'com';

equations
        R1.clk  = Clk;
        R2.clk  = Clk;
        R1 := A;        "positive input pin register
        R2 := !B;       "negative input pin register
        F1 = R1 & R2;

test_vectors
        ([Clk,A,B] -> [F1])
        [.c.,0,0] -> [0 ];
        [.c.,1,0] -> [1 ];
        [ 0 ,1,1] -> [1 ];
        [.c.,1,1] -> [0 ];
        [ 0 ,1,0] -> [0 ];
        [.c.,1,0] -> [1 ];
end
```

Figure C.18 Input register described with ABEL

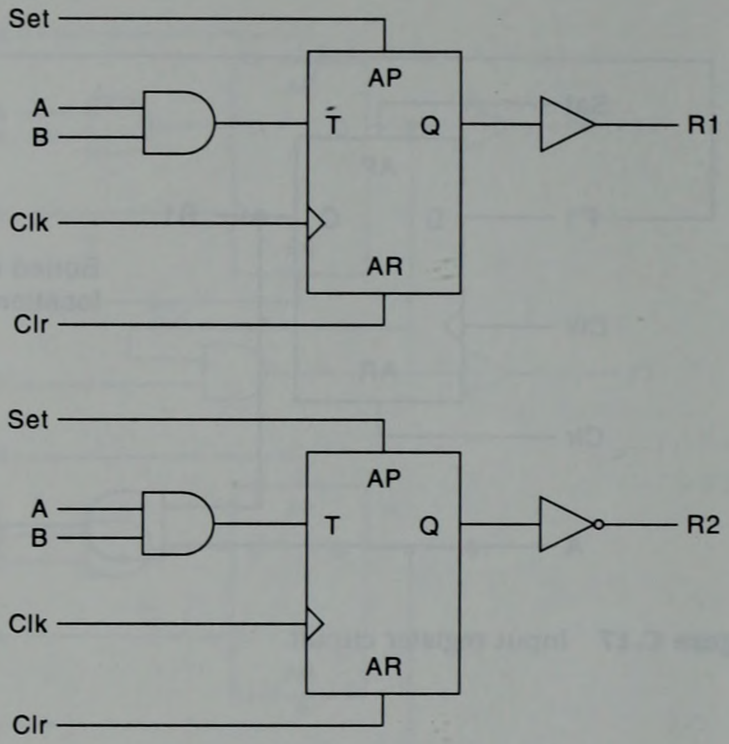


Figure C.19 T flip-flop circuit

T Flip-flops

Figures C.19 and C.20 demonstrate how to describe T flip-flops with ABEL.

```
module ex215f
title 'T FFs in AMD MACH'

        ex215f    device  'MACH215a';
        A,B               pin;
        Clk,Clr,Set       pin;
        R1                pin istype 'reg_T,buffer';
        R2                pin istype 'reg_T,invert';

equations
        R1.T    = A & B;
        R1.Clk  = Clk;
        R1.AR   = Clr;
        R1.AP   = Set;
        R2.T    = A & B;
        R2.Clk  = Clk;
        R2.AR   = Clr;
        R2.AP   = Set;
end
```

Figure C.20 T flip-flop design file

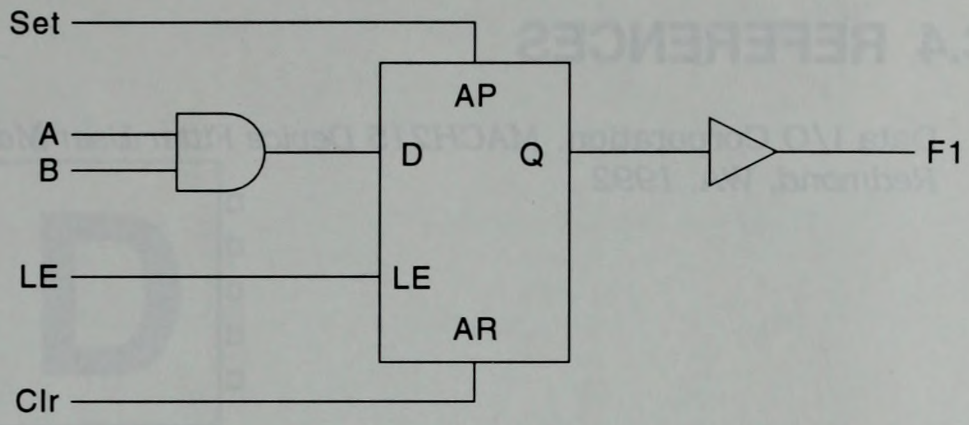


Figure C.21 Transparent latch circuit

Transparent Latches

Figures C.21 and C.22 show an ABEL-HDL description using a transparent latch.

```
module ex215g
title 'Transparent Latch in AMD MACH 215'

        ex215g   device   'MACH215a';
        A,B               pin;
        LE,Clr,Set        pin;
        F1                pin istype 'reg_D,buffer';

equations
        F1.D     = A & B;
        F1.LE    = LE;
        F1.AR    = Clr;
        F1.AP    = Set;

test_vectors
        ([A,B,LE ,Clr,Set] -> F1 )
         [0,0, 1 , 0 , 1 ] ->  1 ;
         [0,0, 1 , 1 , 0 ] ->  0 ;
         [1,1, 1 , 0 , 0 ] ->  0 ; "Latched
         [1,1, 0 , 0 , 0 ] ->  1 ; "Flow Through
         [0,0, 0 , 0 , 0 ] ->  0 ;
         [1,1, 0 , 0 , 0 ] ->  1 ;
         [1,1, 1 , 0 , 0 ] ->  1 ; "Latched
         [0,0, 1 , 0 , 0 ] ->  1 ;
end
```

Figure C.22 Transparent latch design file

C.4 REFERENCES

Data I/O Corporation, *MACH215 Device Fitter User Manual, Data I/O, Redmond, WA, 1992*

D Supplementary Listings for Chapter 12

vfg.c

```
/* Video Frame Grabber Control Program                    */
/* Copyright 1993   Michael Holley and David Pellerin */

/* Written for MicroSoft C version 5.1 */
#include <conio.h>
#include <stdio.h>

#define FILESIZE 131071L

char name_buf[45];

main()
{
    int ch;

    print_menu();
    while(1)
    {
        ch = getch();
        switch(toupper(ch))
        {
            case 'F':   outp(0x316,0);  /* Freeze Picture */
                        break;

            case 'L':   outp(0x314,0);  /* Live Picture */
                        break;

            case 'W':   save_RAM();     /* Write Image */
                        break;

            case 'R':   load_RAM();     /* Read Image */
                        break;

            case 'Q':   exit(0);        /* Quit Program */
                        break;

            default:    print_menu();
                        break;
        }
    }
}

print_menu()
{
    printf("\nFreeze video frame grabber\n");
    printf("Press 'L' for live image\n");
    printf("Press 'F' to freeze image\n");
    printf("Press 'W' to write image RAM to disk file\n");
    printf("Press 'R' to read disk file into image RAM\n");
    printf("Press 'Q' to return to DOS\n");
}

/* Copy image in VFG RAM to disk file */
save_RAM()
{
    unsigned char data;
    long i;
    FILE *fp;

    name_buf[0] = '\0';
    printf("\nEnter file name: ");
    scanf("%40s",name_buf);
    if( (fp = fopen(name_buf,"wb") ) == NULL)
    {
        fprintf(stderr,"can't open output file '%s'\n",name_buf);
        return;
    }

    outp(0x313,0);     /* Clear address counter */

    for (i=1L; i < FILESIZE; i++)
    {
        outp(0x313,0);      /* Next Address */
        outp(0x312,0);      /* Hold Address */
        data = inp(0x300);
        putc(data,fp);
    }

    outp(0x316,0);           /* Freeze Picture */
    close(fp);
}

/* Copy a saved image from disk into VFG RAM for playback */
load_RAM()
{
    unsigned char data;
    long i;
    FILE *fp;

    name_buf[0] = '\0';
    printf("\nEnter file name: ");
    scanf("%40s",name_buf);
    if( (fp = fopen(name_buf,"rb") ) == NULL)
    {
        fprintf(stderr,"can't open input file '%s'\n",name_buf);
        return;
    }

    outp(0x313,0);           /* Clear address counter */

    for (i=1L; i < FILESIZE; i++)
    {
        outp(0x313,0);       /* Next Address */
        outp(0x312,0);       /* Hold Address */
        data = getc(fp);
        outp(0x300,data);
    }

    outp(0x316,0);           /* Freeze Picture */
    close(fp);
}
```

vfg2bmp.c

```
/*********************************************************************
* Program 2 - Format Video Image and Translate to a Windows BMP File
* Copyright 1993  Michael Holley and David Pellerin
*/

/* Written for MicroSoft C Version 5.1 */

#include <stdio.h>
#include <conio.h>

#define XMAX       360  /* Number of pixel in X direction (width) */
#define YMAX       230  /* Number of pixel in Y direction (lines) */
#define TOP_SKIP    1   /* Scan lines skipped at top of frame */
```

```
#define SIDE_SKIP  0   /* Samples skipped at start of scan line */

FILE *outfile;
FILE *infile;


/***********************************************
* Open the file specified on the command line
* and convert to a .BMP file. Output file name
* is PRTFILE.BMP.
*/
main(argc,argv)
int  argc;
char *argv[];
{
    int x, y;
    int sample;

    printf("Process video frame grabber disk file\n");

    /* Open data file */

    if(argc  1)
    {
        printf("Opening input file '%s'\n",argv[1]);
        if( (infile = fopen(argv[1],"rb") ) == NULL)
        {
            fprintf(stderr,"can't open input file '%s'",argv[1]);
            exit(1);
        }
    }
    else
    {
        printf("Opening input file 'temp'\n");
        if( (infile = fopen("temp","rb") ) == NULL)
        {
            fprintf(stderr,"can't open input file '%s'","temp");
            exit(1);
        }
    }

    /* Open output file */

    printf("Opening output file 'prtfile.bmp'\n");
    if( (outfile = fopen("prtfile.bmp","wb") ) == NULL)
    {
        fprintf(stderr,"can't open output file '%s'","prtfile");
        exit(1);
    }

    init_plot();

    for(y=0; y < TOP_SKIP; y++)
    {
        find_hsync();
        }

        /* Read scan lines */

        for(y = 0; y < YMAX; y++)
        {
            for(x = 0; x < XMAX; x++)
            {
                sample = getc(infile);
                fprintf(outfile,"%c",sample);
            }
            find_hsync();   /* Find start of next scan line */
        }

        close(infile);
        close(outfile);
        exit(0);
    }

    /*************************************************
    * Find Horizontal sync pulse
    */
    find_hsync()
    {
        int sync_cnt;
        unsigned char sample;

        sync_cnt = 0;

        /* Look for 20 samples of zero (synclevel) */

        while(sync_cnt < 20)
        {
            sync_cnt++;
            sample = getc(infile);
            if(sample != 0)
            {
                sync_cnt = 0;
            }
        }

        /* Skip color burst samples */

        while(sync_cnt < SIDE_SKIP)
        {
            sample = getc(infile);
            sync_cnt++;
        }
    }

    /************************************************
    * Initialize plot
    */
    init_plot()
    {
```

```
    long total_size,width,height;
    long offset;
    int  i,blue,green,red;

    /* write header */

    width = XMAX;
    height = YMAX;
    offset = 12L + 40L + (256L * 4L);
    total_size = width * height + offset;

    /* BitMapFileHeader                  */

    fprintf(outfile,"BM");
    write_long(outfile,total_size);     /* Total size of file   */
    write_short(outfile,0);             /* Set to 0             */
    write_short(outfile,0);             /* Set to 0             */
    write_long(outfile,offset);         /* Offset to bitmap     */

    /* BitMapInfoHeader  */

    write_long(outfile,40L);            /* Size of structure    */
    write_long(outfile,width);          /* Width of bitmap      */
    write_long(outfile,heigth);         /* Heigth of bitmap     */
    write_short(outfile,1);             /* Set to 1             */
    write_short(outfile,8);             /* Color bits per pixel */
    write_long(outfile,0L);             /* Compression scheme   */
    write_long(outfile,0L);             /* Size of bitmap       */
    write_long(outfile,0L);             /* Horizontal res       */
    write_long(outfile,0L);             /* Vertical res         */
    write_long(outfile,0L);             /* Number of colors     */
    write_long(outfile,0L);             /* Important colors     */

    /* RGBQUAD */

    for(i = 0; i < 256; i++)
    {
        blue  = i & 0xFE;
        green = i & 0xFE;
        red   = i & 0xFE;
        fprintf(outfile,"%c%c%c%c",blue,green,red,0);
    }
}

/**********************************************
*/
write_long(fp,lnum)
FILE  *fp;
long  lnum;
{
    int b0,b1,b2,b3;

    b0 = (lnum & 0x00ff);
    lnum = lnum >> 8;
    b1 = (lnum & 0x00ff);
    lnum = lnum >> 8;
    b2 = (lnum & 0x00ff);
    lnum = lnum >> 8;
    b3 = (lnum & 0x00ff);
    lnum = lnum >> 8;

    fprintf(fp,"%c%c%c%c",b0,b1,b2,b3);
}

/*********************************************
*/
write_short(fp,num)
FILE  *fp;
int   num;
{
    int b0,b1;

    b0 = (num & 0x00ff);
    num = num >> 8;
    b1 = (num & 0x00ff);

    fprintf(fp,"%c%c",b0,b1);
}
```

Index

'buffer' attribute 233
'com' attribute 233
'dc' attribute 234
'invert' attribute 233
'neg' attribute 234
'pos' attribute 233
'reg' attribute 231
'reg_D' attribute 231
'reg_G' attribute 232
'reg_JK' attribute 232
'reg_SR' attribute 232
'reg_T' attribute 232
'retain' attribute 74
'xor' attribute 234
7-segment display 2
@ALTERNATE directive 277
@CARRY directive 275
@CONST directive 269
@DCSET directive 276
@EXIT directive 277
@EXPR directive 270, 274
@IF directive 270
@IFB directive 270
@IFDEF directive 271
@IFIDEN directive 272
@IFNB directive 270
@IFNDEF directive 271
@IFNIDEN directive 272
@IRP directive 273
@IRPC directive 273
@MESSAGE directive 278
@ONSET directive 276
@PAGE directive 278
@RADIX directive 275
@REPEAT directive 272
@SETSIZE directive 274
@STANDARD directive 277

A

ABEL 1, 77
    basic syntax 228
    declarations 228
    declarations section 226
    equations 250
    equations section 226
    expressions 236
    identifiers 228
    keywords 228
    language 225
    macros 266
    module 226
    sets 79, 243
    signals 237
    software 279
    source file 226
    state diagram 85, 256
    test vectors section 226
    truth tables 81
Absorption 18
Active Level 234
Active-low 106, 133
Addend 152
Adders 150
    chained 150, 153
    described with high-level equations 152
    fast carry look-ahead 153
    full adder 150
    half adder 152
    ripple 153
    using @CARRY 153
Address counter 219
Address decoder 133
Advanced Boolean Expression Language 1, 77
Analog-to-digital converter 204
AND 11
Arithmetic functions 147

Arithmetic operators 246
ASCII 236
Assignment
  combinational 250
  registered 254
Assignment operator 15
Associative laws 17
Asynchronous 70
  state machines 70
Asynchronous state machines 198
Attributes 230
  'buffer' 233
  'com' 233
  'dc' 82, 167, 234
  'invert' 123, 233
  'neg' 82, 234
  'pos' 233
  'reg' 231
  'reg_D' 231
  'reg_G' 232
  'reg_JK' 232
  'reg_SR' 232
  'reg_T' 232
  'retain' 74
  'xor' 148, 234
Augend 152

B

Barrel shifter 99
Binary operators 16
Bistable 68
BMP file 207
Boolean
  algebra 14
  equation 14
  operators 14, 245
Boolean equation 15
Boolean logic
  standard forms 25
Borrow flag 155
Bubble diagram 60
Bubble logic 11
Buffers 91
Buried register 129, 300
Bus transceiver 93

C

C language 77, 225
Canonical product 24
Carry logic 127
Carry look-ahead 153
CASE-ENDCASE statement 259
Check bits 167
Clock equations 254
Clock inhibit 102
Clocked JK flip-flop 58
Clocked SR flip-flop 57
Clocked T flip-flop 57
Code converters 133, 165
  gray code 167
Color burst 205
Combinational logic 11
Commutative laws 17
Comparators 135
  equality 137
  identity 135
  inequality 137
  magnitude 138
  polarity of 140
  ripple 141
  using @CARRY 138
Complement 13, 15
Complement array 184
Complement identities 17
Composite expressions 244
Condition product terms 63
Constants
  numeric 235
  special 244
Constraints 5-6
Counter/decoder 185
Counters 109
  @CARRY directive 127
  arbitrary length 122
  chained 127
  D flip-flop 113
  described using high-level equations 113
  for waveform generation 127
  loadable 114
  look-ahead 111
  reset and preset 117
  reset schemes 115
  ripple 109
  short counter 123
  SR flip-flop 119, 129
  T flip-flop 109
  up and down counters 119
Cover, minimal 36
CRC 167
CRC-CCITT 170
CRC generation
  parallel 173
  serial 169
Cube 30
Current state 56, 257
Cyclic redundancy check (CRC) 167

D

Data I/O Corporation 77
Dc-set 40, 256
Declaration
  active-low 234
  constant 79
  pin 79
Declarations 228
Decoder 133, 135
  address 133
  BCD to 7-segment 165
Default state 183
Delay states 75
DeMorgan's theorem 18
  for XOR optimization 49
Demultiplexers
  TTL 104
Design abstraction 2
Design entry 1
Device fitting 226
Device independent 1
Directives 268
  @ALTERNATE 277
  @CARRY 127, 275
  @CONST 269
  @DCSET 276
  @EXIT 277
  @EXPR 270, 274
  @IF 270
  @IFB 270
  @IFDEF 271
  @IFIDEN 272
  @IFNB 270
  @IFNDEF 271
  @IFNIDEN 272
  @IRP 273
  @IRPC 273
  @MESSAGE 278
  @ONSET 276
  @PAGE 278
  @RADIX 275
  @REPEAT 272
  @SETSIZE 274
  @STANDARD 277
  looping 269
  text processing 269
Disjunctive normal form 24
Distributive laws 17
Division
  remainder 247
  result of 247
Don't-care
  assignment operators 42
  in tabular form 34
Don't-cares 234
  during JK flip-flop toggles 64
  in state assignments 68
  minimization using 42
Dot extensions 237, 286
  .ACLR 237
  .AP 237
  .AR 238
  .ASET 238
  .CE 238
  .CLK 238
  .CLR 238
  .D 238
  .FB 239
  .FC 239
  .J 239
  .K 239
  .LD 240
  .LE 240
  .LH 240
  .OE 240
  .PIN 240
  .PR 241
  .Q 241
  .R 241
  .RE 241
  .S 242
  .SET 242
  .SP 242
  .SR 242
  .T 242
Dynamic hazard 72

E

Edge-triggered D flip-flop 57
ELSE keyword 252
Equation splitting 293
Equation tree 245
Equations 1, 250
  ABEL 78
  Boolean 78
  don't-cares in 256
  high-level 78, 113
  in state diagrams 260
  multiple equations for a signal 253
  pin-to-pin 254
  using blocks in 253
  WHEN-THEN 252
Error correcting code 147
Espresso 6
Essential prime implicant 36
Exclusive-OR 12
  optimization 47, 49, 51

Expressions 236
    numeric 251

F

Factor 45
Factoring 45
    constraints 47
    with XORs 50
Feedback 55
    asynchronous 201
    combinational 300
    controlling 292
    internal 296
    pin 296
Fibonacci sequence 155
    generator circuit 155
Fitting strategies 283
Flip-flop
    D-type 57, 98
    emulation 59, 113
    G-type 232
    JK-type 58
    set-reset 70
    setup and hold time 57
    SR-type 57
    T-type 57
    transition tables 63
Flip-flop oriented equations 255
Flip-flops
    configurable 59
    in state machines 57
    metastability 68
FPGA 77
Frame grabber 204
Frame memory 207
Full adder 150
Full subtracter 155
Function hazard 72

G

GAL
    20V8 123
Generalized Reed-Muller form 48
Generator function 169
Generator polynomial 169
Genlocking 206
Glitches 71
GOTO statement 258
Gray code 28, 167
GRM form 48
Group property 291

H

Half adder 152
Hardware description languages 226
Hazards 71
    avoidance 73
    dynamic 72
    function 72
    logic 72-73
    static 72
HDLs 78, 226
Hypercube 31

I

IBM prototyping card 219
Identifiers 228
Identity comparators 135
IF-THEN-ELSE statement 181, 259
Illegal states 181
Implicant 35
    prime 35
Indempotence 17
Input forming logic 190
Input timing 76
Inputs 230
    asynchronous 76
Intermediate expressions 295
Internal feedback 296
Inverted output 233
ISTYPE
    modifier 230
    statement 230

K

Karnaugh map 26

L

L-value 250
Latch 94
    addressable 96
    octal 94
LFSR 169
Linear feedback shift register 169
Literals 15
Loadable counters 114
Logic
    array 12
    bubble 11
    circuits 9
    combinational 11

gates 10
minimization 26
multilevel 21
values 10
Logic functions 9
Logic generator 285
Logic hazard 72-73
Logic synthesis 5-6
Logical sum 20
Look-ahead counters 111

M

Mach 215 device 283
Macros 266
versus constant declarations 267
Magnitude comparators 138
Maxproduct 25
Maxterms 24
Mealy model 66
Memory elements 57
Metastability 68, 76
Metastable
hardened 69
protection 69
state 68
Minimal cover
defined 36
determining 39
Minimization
2-level 23
logic minimization methods 26
multilevel 44-45
multiple-output 43
of state machine logic 67
using don't-cares 42
Minsum 24
determining for a logic function 25
Minterm 24
in tabular form 34
Minuend 155
Mixed GRM form 48
Module 226
Moore model 66
Multilevel logic 21, 45, 295
for comparators 137
Multilevel Optimization 44-45
Multiple-output minimization 43
Multiplexer
12-input, 4-output 78
TTL 104
Multiplicand 159
Multiplication
binary 158
Multiplier
sequential binary 158

N

n-ary operator 16
NAND 11
Next state 56
No-connect 135, 244
for set padding 153
Node 229
automatically generated 127
declarations 229
NOR 11
NOT 11
NTSC standard 203
Numbers
accuracy 235
as expression elements 244
in constants 235
in set expressions 251
radix 235
using strings for 236
Nyquist frequency 205

O

Off-set 40, 63, 256
On-set 40, 63, 256
One-hot encoding 68, 188
Operator
AND 14
arithmetic 246
assignment 15
binary 16
Boolean 245
formal rules 16
in expression 245
NOT 14
OR 14
Order of evaluation 15
Precedence 15
Priority 247
Relational 79, 246
Unary 15
XOR 14
Optimization
multilevel 44-45
OR 11
Order of evaluation 15
Output enable 91
Output inversion 234
Output variable 15
Outputs 230

- active-low 133
- inverted 255
- of asynchronous state machines 75
- programmable inversion 117

Overlap condition 42

P

PAL
- 16H8 134
- 16L8 134
- 16R4 183
- 16R8 113, 116
- 18CV8 208
- 20R8 124
- 20X8 49, 113
- 22V10 117, 123

PAL block 283
Parallel-out serial shift register 100
Parity
- detection 147
- even 147
- odd 147

Partial product 159
Pascal 77, 225
Pattern detector 198
PEEL device 208
Pin 229
- declarations 229
- feedback 296
- numbers 230

PIN keyword 79
Pin-to-pin 113-114, 231, 254
- dot extensions 235
- feedback 235

PLA 12
PLD 77
PLS105 119, 184
Polarity 117
- equation 251
- negative 234
- positive 233

Power-up 181
Prime implicant
- multiple-output 43
- single-output 44

Prime Implicants 35
- determining 37

Product 19
Product term 19
- redundant 75

Programmable logic array 12
Programmable output inversion 117
Programming languages 78
PROM
- for state machines 257

Property statement 291
Propagation delay 71, 111

Q

Quine-McCluskey procedure 37

R

R-value 250
Radix modifier 235
Range operator
- in declarations 230
- in sets 243

Redundant product term 75
Reed-Muller form 48
Reflected code 28, 167
Register
- buried 129, 300
- bypass 117
- input 300

Registers 98
Relational operator 79, 246
Reset and preset swapping 118
Ripple counter 109
Rotate 99

S

Scan line 203
SDLC 170
Sequential
- logic circuits 55

Serial transmitter 190
Set indexing 163, 243
Set notation 82
Set-reset flip-flop 70
Sets 243
- padding with no-connects 244
- restricted operations 246
- rules of evaluation 248

Setup and hold 76
Setup and hold time 57
Shannon's theorems 19
Shift register 98
- linear feedback 169

Short count comparator 123
Signals 229, 237
- active-low 234
- bidirectional 230, 234
- dot extensions 237
- inputs 230
- outputs 230

Simulation 5
Special constants 244
Specialized variable 33
Splitting 293
State assignment 67
  asynchronous 75
  one-hot encoding 68
  random 67
State bit 56, 257
State diagram 4, 85, 180, 256
  header 257
  symbolic 188
State diagrams
  transition statements 258
State encoding 67
  one-hot 188
State graph 60
State machines 56, 177
  asynchronous 56, 70, 73, 198
  determining logic for 60
  illegal states 181
  logic minimization 67
  Mealy model 66
  Moore model 66
  partitioning 188
  state encoding methods 67
  timing considerations 261
  transitions 60
  using state diagrams for 85
State register 257
State table 60, 177
State value 60, 258
Static hazard 72
Strings 236
Subcubes
  finding 32
Subtracter
  Boolean equations 155
  chained 155
  full subtracter 155
Subtraction 155
Subtrahend 155
Sum 20
Sum-of-products 20
  tabular representation 34
Switch matrix 283
Symbolic state diagram 188
Synchronous sequential circuit 55
Synthesis 5

T

Terminology 9
Test vectors 5, 265
  entries 265
  header 265
Timing chains 99
Toggle flip-flop 57
Transition 60
Transition logic
  determining 63
Transition tables 63
Truth tables 2, 81, 262
  entries 263
  for state machine design 60, 82
  for state machines 177
  header 262
  incompletely specified 41, 167
  numbers in 264
  overlapping logic in 42
TTL
  74153 multiplexer 104
  74154 decoder/demultiplexer 106
  74164 serial-to-parallel converter 99
  74165 shift register 100
  74166 shift register 101
  74241 octal buffer 91
  74245 bus transeiver 93
  74259 addressable latch 96
  74373 octal latch 94
  74374 octal register 98
  Data Book 91
  demultiplexers 104
  functions 91
  gates 14
  multiplexers 104
Two-level logic 20
Twos complement 246

U

UART 191
Unary operator 15
Unity 18
University of California at Berkeley 6
Unspecialized variable 33

V

Variable 15
Video frame grabber 203

W

Waveform generator 127
WHEN-THEN equations 252
Windows BMP file 207

X

XOR 12
- 20X8 PAL device 49
- constrained 49
- factoring 50
- for flip-flop emulation 59, 113
- for parity detection 147
- function hazard in 72
- of sum-of-products 47
- optimization 47

XOR complement 18

XOR identities 18

ABEL™

DESIGN SOFTWARE

Version 5.03 MS-DOS 1.44 MB

B:INSTALL

© 1994 Data I/O Corporation ISBN 0-13-605874-4
DataI/O Corporation All rights reserved

ABEL-EDU Disk 1 of 2

ABEL™

DESIGN SOFTWARE

Version 5.03 MS-DOS 1.44 MB

B:INSTALL

© 1994 Data I/O Corporation ISBN 0-13-605874-4
DataI/O Corporation All rights reserved

ABEL-EDU Disk 2 of 2

ELECTRICAL ENGINEERING
Electronic and Digital Design

DISKETTES INCLUDED.
DO NOT DEMAGNETIZE

Includes ABEL Software

DIGITAL DESIGN USING ABEL™

DAVID PELLERIN / MICHAEL HOLLEY

Digital Design Using ABEL gives you all the information you need to get started designing digital circuits using ABEL—Advanced Boolean Expression Language. This book is intended to introduce beginning users to Hardware Description Languages (HDLs) and to increase high-level design skills for more advanced users.

The authors lead you step-by-step through the creation of actual circuits using automated tools. Topics include:

- Introduction to the ABEL Hardware Description Language
- Logic Design Concepts for HDL Users
- Logic Optimization Methods—Manual and Automated
- Examples of Common Circuits Described with ABEL
- Example of a Large, Multi-Chip Design
- Comprehensive ABEL Language Reference

The sample circuits introduced here can be used as building blocks for much larger designs. The book concludes with an example showing how a complex design described at a high level is implemented in actual programmable devices. Appendices include a comprehensive ABEL language reference, and detailed information relating to the device-fitting software supplied with the book.

Digital Design Using ABEL includes software for a complete ABEL design entry system. The software supports a limited number of readily available Programmable Logic Devices (PLDs). Two 3.5" diskettes hold the ABEL compiler, logic optimizer, simulator, and device-fitting software, and include an MS-DOS®-and Microsoft® Windows™-compatible user interface with on-line help. All of the sample circuits referenced in the book are provided.

ISBN 0-13-605874-4
90000
9 780136 058748

PRENTICE HALL
Englewood Cliffs, NJ 07632