

9261

ABEL-PLD™

PLD Design Software

User Manual

DATA I/O
Corporation

096-0095-001

March 1990

096-0095-001

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented by manual distribution.

Data I/O Corporation provides this manual "as is," without warranty of any kind, either expressed or implied. Data I/O reserves the right to make improvements and/or changes in this document or in the product and/or program(s) described herein at any time. Information on these changes will be incorporated in new editions of this publication.

Data I/O Corporation
10525 Willows Road N.E., P.O. Box 97046
Redmond, Washington 98073-9746 USA
(206) 881-6444

Acknowledgments:

Data I/O and PLDtest are registered trademarks of Data I/O Corporation. ABEL, ABEL-PLD, PLDlinx, PLDtest Plus, Personal Silicon Foundry, PSF, DASH-PLD, DASH, FutureDesigner, GATES, UniSite LogicPak, UniPak 2B, GangPak and PROMlink are trademarks of Data I/O Corporation.

© 1990 Data I/O Corporation
All rights reserved

Table of Contents

Preface

Operation

Chapter 1. Introduction

System Requirements	1-2
-------------------------------	-----

Chapter 2. Understanding ABEL-PLD

Source File	2-2
Elements of the Source File	2-2
ABEL-PLD Output Files	2-4
Language Processor	2-5
Parse	2-6
Transform	2-7
Reduce	2-7
Fusemap	2-7
Simulate	2-8
Document	2-11
Functional Device Testing	2-11

Chapter 3. Using ABEL-PLD

Creating a Source File	3-1
Using ABEL-PLD	3-1
Using the PSF Menus	3-7
Command Line Processing	3-10
ABEL-PLD Programs	3-12
PARSE	3-12
TRANSFOR	3-18

REDUCE	3-20
FUSEMAP	3-25
SIMULATE	3-30
DOCUMENT	3-36
Utilities	3-43
TOABEL — PALASM to ABEL Converter	3-43
ABELLIB — Library Manager	3-44
Cleanup.bat	3-46
Downloading the Programmer Load File	3-46

Chapter 4. Using Advanced Features

Controlling Device Nodes	4-1
Using Node Numbers	4-1
Using Dot Extension Notation	4-2
Solving Timing Problems with REDUCE	4-4
Passing Arguments from the Command Line	4-7
Simulation	4-9
Test Vectors and Simulation	4-10
Debugging State Machines	4-10
Multiple Test Vector Tables	4-11
Using Macros and Directives to Create Test Vectors	4-12
Don't Cares in Simulation	4-16
Preset and Preload Registers	4-18
Asynchronous Circuits	4-27
Devices with Clock Inputs	4-29
Trace Levels and Breakpoints	4-30
Simulation and Designs With Buffered Outputs	4-38
Simulation and Unspecified Inputs	4-39
Simulation for Designs With Feedback	4-39

Language Reference

Chapter 5. Source File

Elements of the Source File	5-2
Examining a Source File	5-4
Purpose of the Address Decoder	5-5
The MODULE Statement	5-6
The FLAG Statement	5-6
The TITLE Statement	5-6
The DEVICE Declaration	5-6
PIN and NODE Declarations	5-6
CONSTANT Declarations	5-7
EQUATIONS Statements	5-7
Test Vectors	5-8
Processing a Source File	5-9

Chapter 6. Language Elements

Basic Syntax	6-1
Valid ASCII Characters	6-2
Identifiers	6-2
Reserved Identifiers (Keywords)	6-3
Choosing Identifiers	6-4
Strings	6-4
Comments	6-5
Numbers	6-5
Special Constants	6-7
Operators, Expressions, and Equations	6-8
Logical Operators	6-8
Arithmetic Operators	6-8
Relational Operators	6-9
Assignment Operators	6-11
Expressions	6-11
Equations	6-13
Sets	6-15
Set Operations	6-16
Set Assignment and Comparison	6-17
Set Evaluation	6-18
Limitations/Restrictions on Sets	6-20
Blocks	6-21
Arguments and Argument Substitution	6-21

Chapter 7. Language Structure

Basic Structure	7-1
MODULE Statement	7-3
FLAG Statement	7-4
TITLE Statement	7-5
Declarations	7-5
Declaration Keyword	7-6
Device Declaration Statement	7-6
Pin Declaration Statement	7-7
Node Declaration Statement	7-8
Constant Declaration Statement	7-9
Macro Declaration Statement and Macro Expansion	7-10
ISTYPE Declaration Statement	7-12
Attribute Definitions	7-13
Library Statement	7-15
Equations Statement	7-16
Truth Table Statement	7-17
Truth Table Format	7-18
Programmable Polarity Registers	7-19
State Descriptions	7-20
STATE_DIAGRAM Statement	7-20
IF-THEN-ELSE Statement	7-22

Chained IF-THEN-ELSE Statements	7-23
CASE Statement	7-23
GOTO Statement	7-24
WITH-ENDWITH Statement	7-24
XOR_Factor	7-25
Fuses Section	7-28
Test Vectors	7-29

Chapter 8. Directives

@ALTERNATE Directive	8-2
@CONST (Constant) Directive	8-3
@EXIT Directive	8-3
@EXPR (Expression) Directive	8-4
@IF Directive	8-4
@IFB (If Blank) Directive	8-5
@IFDEF (If Defined) Directive	8-5
@IFIDEN (If Identical) Directive	8-6
@IFNB (If Not Blank) Directive	8-6
@IFNDEF (If Not Defined) Directive	8-7
@IFNIDEN (If Not Identical) Directive	8-7
@INCLUDE Directive	8-8
@IRP (Indefinite Repeat) Directive	8-9
@IRPC (Indefinite Repeat, Character) Directive	8-10
@MESSAGE Directive	8-10
@PAGE Directive	8-11
@RADIX Directive	8-11
@REPEAT Directive	8-12
@STANDARD Directive	8-12

Applications Guide

Chapter 9. Design Considerations

Exclusive OR Equations	9-1
Polarity Control	9-2
Polarity Control in ABEL-PLD	9-3
Polarity Control in State Machines	9-3
Register Preloads in the Simulator	9-6
Assigning Pins	9-6
Using State Machines	9-6
Mealy and Moore	9-7
Debugging State Machines with Simulate	9-13
Express State Registers as a Set	9-14
Use Identifiers Rather Than Numbers for States	9-14
Powerup Register States	9-17
Designing with Programmable Polarity Outputs	9-17

Unsatisfied Transition Conditions, D-Type Flip-Flops	9-17
Unsatisfied Transition Conditions, Other Flip-Flops	9-18
Number Adjacent States for One-Bit Changes	9-18
Use State Register Outputs to Identify States	9-19

Chapter 10. Design Examples

6809 Memory Address Decoder	
Equations, P14L4	10-3
Design Specification	10-3
Design Method	10-4
Test Vectors	10-5
12 to 4 Multiplexer	
Equations, P14H4	10-6
Design Specification	10-6
Design Method	10-7
Test Vectors	10-8
1 to 8 Demultiplexer	
Equations, P16L8	10-9
Design Specification	10-9
Design Method	10-10
Test Vectors	10-10
4-Bit Counter/Multiplexer	
Equations, P16R4	10-12
Design Specification	10-12
Design Method	10-13
Test Vectors	10-14
Multiple Equations to the Same Signal	10-16
8-Bit Barrel Shifter	
Equations, P20R8	10-19
Design Specification	10-19
Design Method	10-19
Test Vectors	10-21
7-Segment Display Decoder	
Truth Table, RA5P8	10-22
Design Specification	10-22
Design Method	10-23
Test Vectors	10-25
4-Bit Comparator	
Macros, Equations, F153	10-26
Design Specification	10-26
Design Method	10-27
Test Vectors	10-29
Blackjack Machine	10-30
Design Specification — MUXADD	10-32
Design Method — MUXADD	10-33
Test Vectors — MUXADD	10-34
Design Specification — BINBCD	10-35
Design Method — BINBCD	10-35
Test Vectors — BINBCD	10-35

Design Specification — BJACK	10-38
Design Method — BJACK	10-39
Test Vectors — BJACK	10-42

Chapter 11. Advanced Devices

Output Enables	11-1
Pin Controlled Output Enable	11-1
Term Controlled Output Enable	11-1
Configurable Output Enable	11-2
Output Macro Cell Control with ISTYPE	11-4
Controlling Macro Cell Polarity	11-4
Controlling Macro Cell Feedback Point	11-6
Selecting or Bypassing Device Registers	11-8
Controlling Register Type	11-9
Selectable Product Term Sharing	11-9
More On Feedback	11-9
Selectable Feedback Type	11-11
The .Q Dot Extension	11-12
Using Complement Arrays	11-12
Equations for XOR PALs	11-15

Chapter 12. Downloading to Programmers

Downloading to a Model 29 with LogicPak	12-1
Downloading to a UniSite™ Universal Programmer	12-3
PROM Download (Model 29/UniPak2)	12-3

Appendixes

Appendix A. Error Messages

Appendix B. JEDEC Standard Number 3A

Appendix C. Programmable Logic Device Information

Appendix D. ASCII Table

Appendix E. Set Operations

Appendix F. The Terminal Emulator Utility

Index

Reference Card

List of Figures

Figure 1-1. Logic Design Steps with ABEL-PLD	1-3
Figure 2-1. Role of ABEL-PLD in the Logic Design Process	2-2
Figure 2-2. Processing Flow of the Language Processor	2-6
Figure 2-3. Simulate Processing Flow Diagram	2-10
Figure 3-1. PSF Run Series Menu	3-9
Figure 3-2. Parse Menu	3-12
Figure 3-3. PARSE Listing File with Errors from m6809err.abl	3-16
Figure 3-4. Corrected Source File, m6809a.abl	3-17
Figure 3-5. Transfor Menu	3-18
Figure 3-6. Reduce Menu	3-20
Figure 3-7. Fusemap Menu	3-25
Figure 3-8. Simulate Menu	3-30
Figure 3-9. Document Menu	3-36
Figure 3-10. Document Output for m6809a.abl	3-40
Figure 3-11. TOABEL Menu	3-43
Figure 4-1. Using Node Numbers for Reset/Preset Functions	4-2
Figure 4-2. Using Dot Extensions for Reset/Preset Functions	4-4
Figure 4-3. Circuit Using an Input and Its Complement	4-5
Figure 4-5. Device Type Passed From Command Line, Memory Address Decoder	4-8
Figure 4-6. Source File With Multiple Test Vectors Sections	4-11
Figure 4-7. Test Vectors Described With a Macro and @IF and @IRP Directives	4-13
Figure 4-8. PARSE Output (partial) Showing Expanded Vectors for a Macro and @IF/@IRP Directives	4-14
Figure 4-9. Test Vectors Described With a Macro, @CONST and @REPEAT Directives	4-15
Figure 4-10. PARSE Output (partial) with Expanded Output for a Macro and @CONST/@REPEAT Directives	4-16
Figure 4-11. Assignment of Don't Care Value (.x.) to Design Outputs	4-17
Figure 4-12. SIMULATE Results with Outputs Specified as Don't Care	4-18
Figure 4-13. Test Vectors for Special Preset Conditions	4-20
Figure 4-14. Timing Diagram Showing Test Vector Action	4-21
Figure 4-15. The Illegal States Defined	4-23
Figure 4-16. Test Vectors for Illegal States	4-24

Figure 4-17. Using Test Vectors to Preload A State Machine	4-25
Figure 4-18. Controlling Reset/Preset by Product Term	4-26
Figure 4-19. A Cross-Coupled Flip-Flop	4-28
Figure 4-20. Using the Input Side of the Test Vectors to Define Outputs	4-28
Figure 4-21. Clock Inputs Shown in Trace Level 2 Output	4-29
Figure 4-22. Trace Level 0 Output for m6809er.abl	4-31
Figure 4-23. Test Vectors Used to Create Simulation Error	4-31
Figure 4-24. Level 1 Simulation Output, All Vectors	4-32
Figure 4-25. Level 2 Simulation Output, Single Vector	4-33
Figure 4-26. Trace Level 3 Simulation Output	4-34
Figure 4-27. Trace Level 4 Simulation Output	4-36
Figure 4-28. Trace Level 5 Simulation Output	4-37
Figure 4-29. Trace Levels 4 and 5 in the Same Output File	4-38
Figure 4-30. Synchronous Feedback Circuit	4-39
Figure 4-31. Source File: Synchronous Feedback Circuit	4-40
Figure 4-32. Simulation Output, Trace Level 1: Synchronous Feedback Circuit	4-41
Figure 4-33. Simulation Output (partial), Trace Level 2: Synchronous Feedback Circuit	4-42
Figure 4-34. Asynchronous Feedback Circuit	4-42
Figure 4-35. Source File: Asynchronous Feedback Circuit	4-43
Figure 4-36. Simulation Output, Trace Level 1: Asynchronous Feedback Circuit	4-43
Figure 4-37. Simulation Output, Trace Level 2: Asynchronous Feedback Circuit	4-44
Figure 5-1. Source File Template	5-2
Figure 5-2. Block Diagram; 6809 Memory Address Decoder	5-4
Figure 5-3. Source File Describing an Address Decoder	5-5
Figure 7-1. Structure of an ABEL-PLD Source File	7-2
Figure 7-2. Feedback Paths for an E0310	7-15
Figure 7-3. Pictorial State Diagram	7-22
Figure 9-1. Flow Diagram for a Mealy State Machine	9-7
Figure 9-2. Flow Diagram for a Moore State Machine	9-8
Figure 9-3. Mealy State Diagram of a Sequence Detector	9-9
Figure 9-4. Moore State Diagram of a Sequence Detector	9-10
Figure 9-5. Source File for a Mealy State Machine	9-11
Figure 9-6. Source File for a Moore State Machine	9-12
Figure 9-7. Source File With Multiple Test Vectors Sections	9-14

Figure 9-8. Source File: State Machine	9-16
Figure 9-9. D-Type Register with False Inputs	9-17
Figure 10-1. Block Diagram: 6809 Memory Address Decoder	10-3
Figure 10-2. 6809 Memory Address Decoder Source File	10-4
Figure 10-3. Simplified Block Diagram: 6809 Memory Address Decoder	10-5
Figure 10-4. Block Diagram: 12 to 4 Multiplexer	10-6
Figure 10-5. Simplified Block Diagram: 12 to 4 Multiplexer	10-7
Figure 10-6. Source File: 12 to 4 Multiplexer	10-8
Figure 10-7. Block Diagram: 1 to 8 Demultiplexer	10-9
Figure 10-8. Simplified Block Diagram: Demultiplexer	10-9
Figure 10-9. 1 to 8 Demultiplexer	10-11
Figure 10-10. Block Diagram: 4-Bit Counter With 2 Input Multiplexer	10-12
Figure 10-11. Simplified Block Diagram: 4-Bit Counter With 2 Input Multiplexer	10-13
Figure 10-12. Source file: 4-bit Counter with 2 Input Mux	10-15
Figure 10-13. Multiple Equations Sections, 4-Bit Counter	10-17
Figure 10-14. Block Diagram: 8-Bit Barrel Shifter	10-19
Figure 10-15. Simplified Block Diagram: 8-Bit Barrel Shifter	10-20
Figure 10-16. Source File: 8-Bit Barrel Shifter	10-21
Figure 10-17. Block Diagram: Seven-Segment Display Decoder	10-22
Figure 10-18. Simplified Block Diagram: Seven-Segment Display Decoder	10-23
Figure 10-19. Source file: 4-bit Counter with 2 Input Mux	10-24
Figure 10-20. Block Diagram: 4-Bit Comparator	10-26
Figure 10-21. Simplified Block Diagram: 4-bit Comparator	10-26
Figure 10-22. Source File: 4-Bit Comparator	10-28
Figure 10-23. Schematic of a Blackjack Machine Implemented in Three PLDs	10-31
Figure 10-24. Source File: Multiplexer/Adder/Comparator	10-34
Figure 10-25. Source file: 4-bit Counter with 2 Input Mux	10-37
Figure 10-26. Pictorial State Diagram: Blackjack Machine	10-40
Figure 10-27. Source File: State Machine (Controller)	10-43
Figure 11-1. Output Enable Controlled by Device Pin	11-2
Figure 11-2. Output Enable Controlled by Product Term	11-2

Figure 11-3. Typical Multiplexer for Output Enable Modes	11-4
Figure 11-4. Controlling Macro Cells with ISTYPE	11-5
Figure 11-5. Controlling Macro Cells with ISTYPE — Source File	11-7
Figure 11-6. Macro Cell, Configurable to Combinatorial or Registered Output	11-9
Figure 11-7. Registered and Combinatorial Feedback (P16R4)	11-10
Figure 11-8. Selectable Feedback Paths (E0310)	11-11
Figure 11-9. Location of the Q Signal in Registered Devices	11-12
Figure 11-10. Transition Equations for a Decade Counter	11-14
Figure 11-11. Abbreviated F105 Schematic	11-16
Figure 12-1. Cable Configuration for Transfer Between an IBM-XT and a Data I/O Programmer	12-2
Figure 12-2. Cable Configuration for Transfer Between an IBM-AT and a Data I/O Programmer	12-3
Figure B-1. Example of a PLD Data File	B-2
Figure B-2. Computing the Transmission Checksum	B-6
Figure B-3. 8-bit Words Formed from Fuse States for Checksum	B-11
Figure B-4. Computing the Fuse Checksum	B-11

List of Tables

Table 2-1. Transfor Logic Reduction Rules	2-7
Table 3-1. ABEL-PLD Program Options	3-2
Table 3-2. Batch Processing Options	3-9
Table 3-3. Parse Options	3-13
Table 3-4. Transfor Options	3-19
Table 3-5. Reduce Options	3-21
Table 3-6. Basic Rules of Boolean Algebra	3-22
Table 3-7. Fusemap Options	3-26
Table 3-8. Data Translation Format Codes and File Extensions	3-28
Table 3-9. Simulate Options	3-31
Table 3-10. Document Options	3-37
Table 3-11. TOABEL Options	3-44
Table 4-1. Dot Extensions	4-3
Table 4-2. Notation Used in Simulation Output Files	4-35
Table 6-1. Number Representation in Different Bases	6-6
Table 6-2. Special Constants	6-7
Table 6-3. Logical Operators	6-8

Table 6-4. Arithmetic Operators	6-9
Table 6-5. Relational Operators	6-10
Table 6-6. Assignment Operators	6-11
Table 6-7. Summary of Operators and Priorities	6-12
Table 6-8. Valid Set Operations	6-18
Table 7-1. Attributes for Pin, Node and ISTYPE	7-13
Table 8-1. Directives	8-1
Table 8-2. Alternate Operator Set	8-2
Table 10-1. Design Examples Supplied with ABEL-PLD	10-2
Table B-1. Field Identifiers and Descriptions	B-3
Table B-2. Field Identifiers	B-7
Table B-3. Test Conditions	B-13
Table C-1. TOABEL Supported Devices	C-2
Table C-2. Signetics Supported Devices	C-2
Table C-3. Nodes Used by ABEL-PLD	C-5
Table F-1. Term-100 General Mode	F-5
Table F-2. Term-100 UniSite Mode	F-6

Table of Contents

Preface

Technical Questions?

If you need technical assistance with your Data I/O product, our Customer Resource Center (CRC) Support Engineers are available between 6:00 a.m. and 5:00 p.m. Pacific Standard Time. To help us provide quick and accurate assistance, please be at your programmer or computer when you call, and have the following ready:

- Product version number
- Detailed description of the problem you are experiencing
- Error messages (if any)
- Device manufacturer and part number (if device related)
- Product manual

USA

Anywhere in USA
Fax No.

1-800-247-5700
206-882-1043

International

Data I/O Japan
Data I/O Europe
Data I/O Canada
Data I/O Intercontinental

(03) 432-6991
+31 (0)20 6622866
416-678-0761
206-881-6444

Bulletin Board

Phone No. 206-882-3211
1200/2400 Baud, 8 data bits, no parity, 1 stop bit.

Written Inquiries

Data I/O Corporation
Customer Resource Center
10525 Willows Road N.E.
P.O. Box 97046
Redmond, WA 98073-9746 USA

Typographic Conventions

The table below lists typographic conventions used throughout this manual to describe interactions with the computer.

Convention	Description
Examples	Items in this typeface are system output (prompts, reports and displays) and user input.
<Function Key>	Function keys such as <Enter> and <F1> are enclosed in angle marks.
OR OR ...	Items separated by a vertical bar are mutually exclusive; that is, only one of the options listed can be specified.
[Optional Default Entry]	The program's default choice is shown in brackets following a question. To choose the default, press <Enter>. Optional elements of a command are also shown in brackets.
Variables	Italicized items in a command are variables and should be replaced with the requested information. For example, <i>input filename</i> should be replaced with the name of the desired input file. If an italicized item is enclosed in brackets, it is also an optional item as above.

Data I/O Device Support Policy/Liability

1. Data I/O strives to achieve more device support approvals from semiconductor manufacturers than any other programmer manufacturer or software developer.
2. Every effort is made to program an adequate number of samples according to the manufacturer-supplied specification, and verify waveforms as per that specification prior to release of support. Manufacturers' approvals are to be sought in parallel with this process.
3. Data I/O's objective is to seek and obtain approvals on all devices.
4. Data I/O has made every attempt to ensure that the device information (as provided by the device manufacturer) contained in our programmers, software and documentation is accurate and complete. However, Data I/O assumes no liability for errors, or for any damages, whether direct, indirect, consequential or incidental, that result from use of documents provided with equipment, or from the equipment or software which it accompanies, regardless of whether or not Data I/O has been advised of the possibility of such loss or damage.

Data I/O Companion Products

ABEL

ABEL™ PLD Design Software is the industry-standard PLD design software for single PLD designs. It allows you to use any combination of schematics (with PLDlinx), equations, truth tables or state diagrams to describe a design. ABEL software automatically converts these high-level behavioral design descriptions into the proper structural representation for single PLD implementation. Then, logic synthesis, including reduction, can be performed. This ensures optimization of the design to the selected target devices.

GATES

GATES™ Logic Synthesizer is a multiple PLD design software tool that allows you to use any combination of schematics, equations, truth tables or state diagrams to enter a multiple PLD design. The GATES software includes an interactive user interface, and a high-level functional verifier that ensures that design errors are caught up front. Then, logic synthesis can be performed, including reduction, factoring, partitioning and device selection. This ensures optimization of the design to the selected target device technology.

FutureDesigner

FutureDesigner™ Design Entry System is an integrated design entry system combining industry-standard DASH schematic capture, GATES behavioral design entry, and logic synthesis capabilities. This flexibility allows the engineer to use any combination of schematics, equations, truth tables or state diagrams to enter a circuit design. Logic synthesis is automatically performed, optimizing the circuit for the target vendor and technology. As output, the FutureDesigner system supports multiple technologies including TTL, PLD, Logic Cell™ Array (LCA), gate array and other ASIC parts. Because the FutureDesigner system is technology independent, migration between these technologies is easy.

Data I/O Device Programmers

Standard PLD programmer load files can be used by any of Data I/O's logic programmers, including the UniSite™ 40 Universal Programmer, the 29B Logic System, or the 60A/H Logic Programmers.

DASH

The FutureNet® DASH™ Schematic Designer can be used with the PLDlinx program and any of Data I/O's PLD development tools to describe PLD designs using schematics. DASH software is extremely easy to use and features an extensive symbol library, powerful postprocessors, and a host of interfaces to other systems and services.

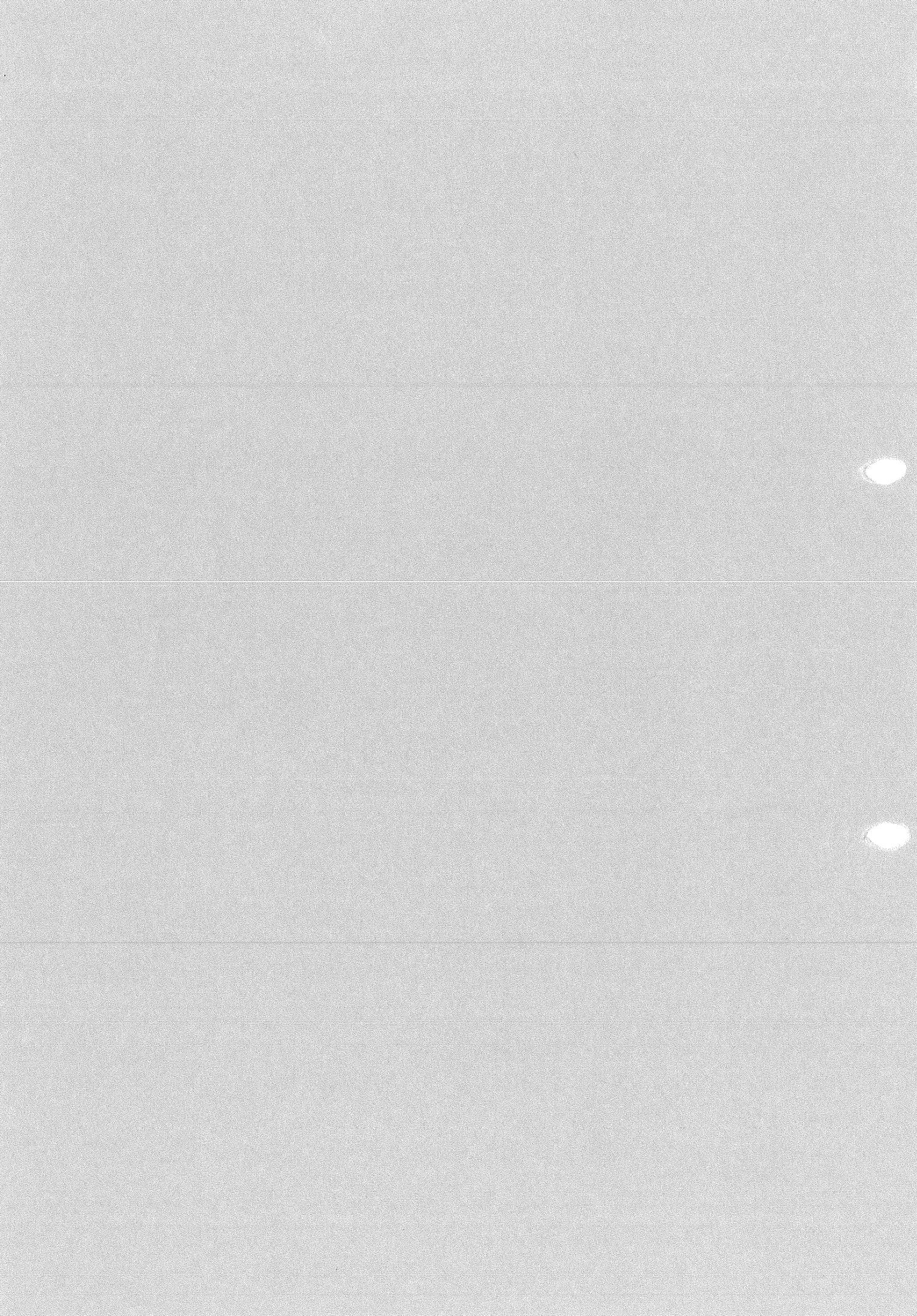
PLDlinx

PLDlinx™ Schematic Translator automatically transfers designs entered in the DASH program to ABEL or GATES format. This allows PLD designs to be described using schematics.

PROMlink

PROMlink™ PC Interface permits control of and communication with Data I/O programmers by means of a personal computer.

Operations



Chapter 1

Introduction

ABEL-PLD™ PLD Design Software is a complete logic design tool that lets you easily describe and implement programmable logic designs in PLDs and PROMs. The ABEL-PLD software consists of a special-purpose, high-level language that is used to describe logic designs, and a language processor that converts logic descriptions to programmer load files. Programmer load files contain the information necessary to program and test programmable logic devices.

Features of the ABEL-PLD design language are

- Universal syntax for all programmable logic types
- High-level, structured design language
- Flexible forms for describing logic:

High-level and/or Boolean equations
Truth tables
State descriptions

- Test vectors for simulation and testing
- Time-saving macros and directives

The ABEL-PLD language processor also has many powerful features:

- Syntax checking
- Verification that a design can be implemented with a chosen part
- Logic reduction
- Design simulation
- Automatic design documentation
- Creation of programmer load files in JEDEC and PROM format

Together, the ABEL-PLD design language and language processor make it easy to design and test logic functions that are to be implemented with programmable logic devices. For example, you can design a three-input AND function with the inputs A, B, and C and the output Y using a truth table like this:

```
truth_table "3-input AND gate"  
([ A, B, C ] -> Y)  
[ 0,.X.,.X. ] -> 0 ;  
[ .X., 0,.X. ] -> 0 ;  
[ .X.,.X., 0 ] -> 0 ;  
[ 1, 1 , 1 ] -> 1 ;
```

The ".X."s in the table indicate "don't care" conditions, and the output Y is set to 1 only when all three inputs equal 1. You also could have specified the output Y in terms of simple Boolean operators and have achieved the same result. This is done here, where "&" is the logical AND operator:

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

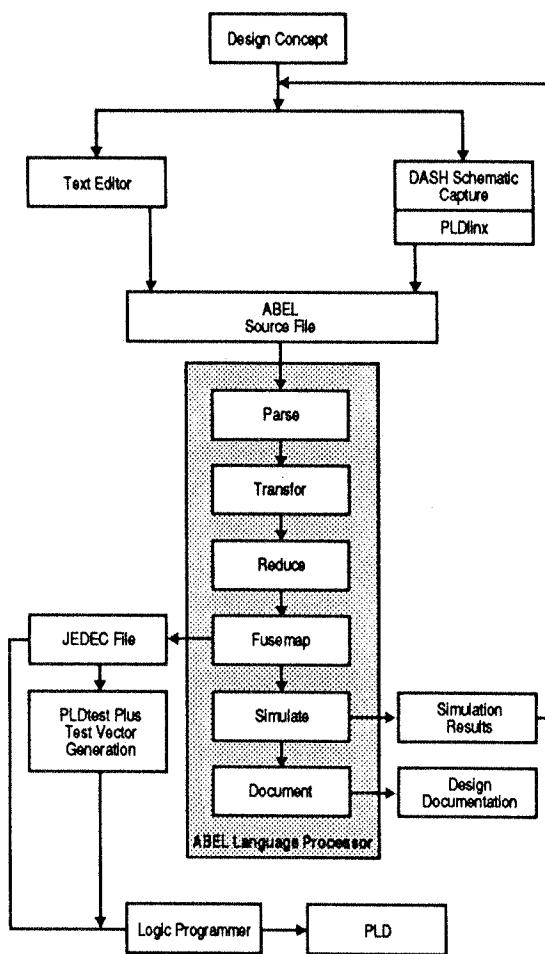
ABEL-PLD software lets you choose the type of description that is best suited to the logic being described, or the type of description you feel most comfortable with. In most cases, the same description can be used for many different devices by changing the device specification. The way the ABEL-PLD program enters the design process reduces errors and saves time. You can think about designs in a logical, functional way, describe them in that fashion, and then test your design to see that it operates as expected, all without worrying about which fuses should be blown or left intact.

System Requirements

ABEL-PLD presently runs on IBM/AT/XT/PS2 and MS-DOS compatibles. To install the ABEL-PLD software, refer to the *Installation Guide* supplied with your ABEL-PLD package. To create ABEL-PLD source files, you will also need an editor or word processor. This may be any editor of your choice as long as it produces a standard ASCII file.

For downloading programmer load files to a logic programmer, you will need an RS232 port and a cable to connect to the programmer. Cable and programmer configurations are discussed in the "Applications Guide."

Figure 1-1
*Logic Design Steps with
ABEL-PLD*



Introduction

Chapter 2

Understanding ABEL-PLD

This chapter explains the process used by ABEL-PLD software to create a programmer load file from your design.

First, you must describe your design in a source file, then the ABEL-PLD software processes the source file to produce a programmer load file, and simulate and document your design.

The source file is described briefly below to familiarize you with its basic elements. When you want to begin creating your own source files, refer to the "Language Reference" for the complete requirements.

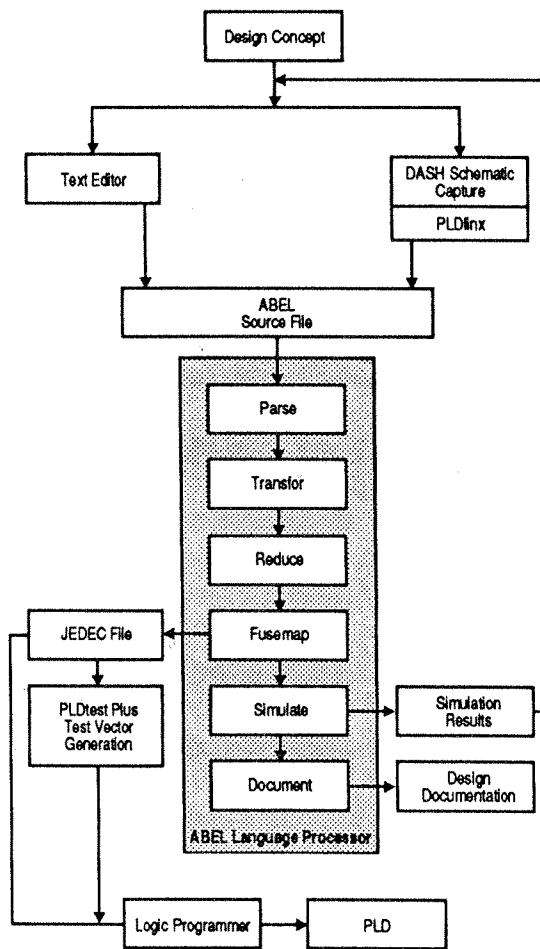
The functions that each element of the ABEL-PLD package performs are discussed in this chapter. For information on running the ABEL-PLD software, refer to the chapter "Using ABEL-PLD."

The output files produced are introduced below, with complete information in "Using ABEL-PLD."

The ABEL-PLD package contains certain utilities in addition to the language processor to improve the interface between ABEL-PLD input and output files and other programs. These utilities are discussed under "Utilities" in the chapter "Using ABEL-PLD."

Figure 2-1 shows the logic design process and the role the ABEL-PLD software takes in it.

Figure 2-1
Role of ABEL-PLD in the Logic Design Process



Source File

Beginning with the design concept, you create the ABEL-PLD source file required by the language processor to generate the programmer load file. The source file contains a complete description of your logic design. You can create the source file manually by means of a text editor or word processor that generates ASCII files, or if you have PLDlinx, you can use it to convert a DASH schematic of the design to an ABEL-PLD source file.

Elements of the Source File

The basic element of the source file is the module, which contains a logic description. A source file may contain one or any number of modules, each containing a different logic description.

The source file module is made up of several elements. These elements are listed below. See the Language Reference for complete requirements and syntax for source files.

- **MODULE STATEMENT** — This names the module and also indicates whether dummy arguments are used.
- **FLAG** — This can be used to control processing of the source file by the language processor with command line options, and is optional.
- **TITLE** — The title is optional and is used to describe the module. The text entered will be used as a header for the output files.
- **DEVICE DECLARATIONS** — The device declaration associates one or more device identifiers with specific programmable logic devices.
- **PIN DECLARATIONS** — This defines the pins of the programmable logic device with identifiers used in the source file.
- **NODE DECLARATIONS** — Nodes in the programmable logic device, if present, are defined here.
- **CONSTANT DECLARATIONS** — Constant values are defined.
- **MACRO DEFINITIONS** — Macros are defined.
- **EQUATIONS** — Contains the Boolean or high level equations required to describe the logic design. (You can also use truth tables or state descriptions.)
- **TRUTH TABLE** — Contains the truth tables required to describe the logic design. (You can also use equations or state descriptions.)
- **STATE DIAGRAM** — Contains the state descriptions required to describe the logic design. (You can also use equations or truth tables.)
- **TEST VECTORS** — Contains the optional test vectors that you write to verify the functionality of the logic design. Design test vectors you write can be used in conjunction with device test vectors generated by the PLDtest or PLDtest Plus programs.
- **END STATEMENT** — Ends the module.

ABEL-PLD Output Files

Five types of output files are created during execution of ABEL-PLD software:

<i>filename.lst</i>	parser listing file
<i>device.jed</i>	programmer load files (JEDEC format)
<i>filename.sim</i>	simulation output
<i>filename.doc</i>	design documentation
<i>filename.out</i>	intermediate file used by SIMULATE and DOCUMENT

Note: During processing, ABEL-PLD creates several temporary files. Files with the same file name as the input source file and extensions .tm1, .tm2, and .tm3 are generated; and files named fusein.tmp and fuseout.tmp may be generated. Any files with a file name and extension identical to those of a temporary file will be overwritten.

Parser Listing File

Contains the source code, error messages, and the effect of @INCLUDE directives. (See the Language Reference for information on macros and directives.) A complete description on the information that can be included in the parser listing file is given in the chapter "Using ABEL-PLD."

Programmer Load File (*.jed)

When you process an ABEL-PLD source file, one or more files named *filename.jed* are created in your working directory. These files are the programmer load files, and contain the data necessary for a logic programmer to program a logic device with your design. The programmer load file contains the fuse states, test vectors and other design information as defined by the JEDEC standard. This file is loaded into a logic programmer to program and test a programmable logic device. (The programmer load file may also be applied to PLDtest/PLDtest Plus programs.)

One programmer load file is created for each device specified in the source file. To transfer a programmer load file to the logic programmer, you must download it from your system over an RS232-compatible communications link. Data I/O's PROMlink™ PC Interface, version 2.0 or greater, can be used to transfer programmer load files from DOS systems. If the PROMlink interface is used, refer to the PROMlink manual for details. For information on downloading ABEL-PLD files to Data I/O programmers, see the "Applications Guide." If you have a non-Data I/O programmer, see your programmer manual.

Complete specification of the JEDEC standard format is given in an appendix.

Intermediate File (*.out)	The language processor creates a file named filename.OUT that contains intermediate output data created by FUSEMAP. This intermediate file can be used as input to SIMULATE and DOCUMENT. See the chapter "Using ABEL-PLD" for information on FUSEMAP options and the results of each option on the *.out file.
Simulation Output File (*.sim)	Output from the simulation step indicating whether simulation was completed successfully, and if not, where predicted and actual results differed. See the chapter "Using ABEL-PLD" for information on SIMULATE options and the results of each option on the *.sim file.
Design Documentation File (*.doc)	Contains a chip diagram, reduced logic equations and other design information. This file is created by the DOCUMENT step of the language processor.

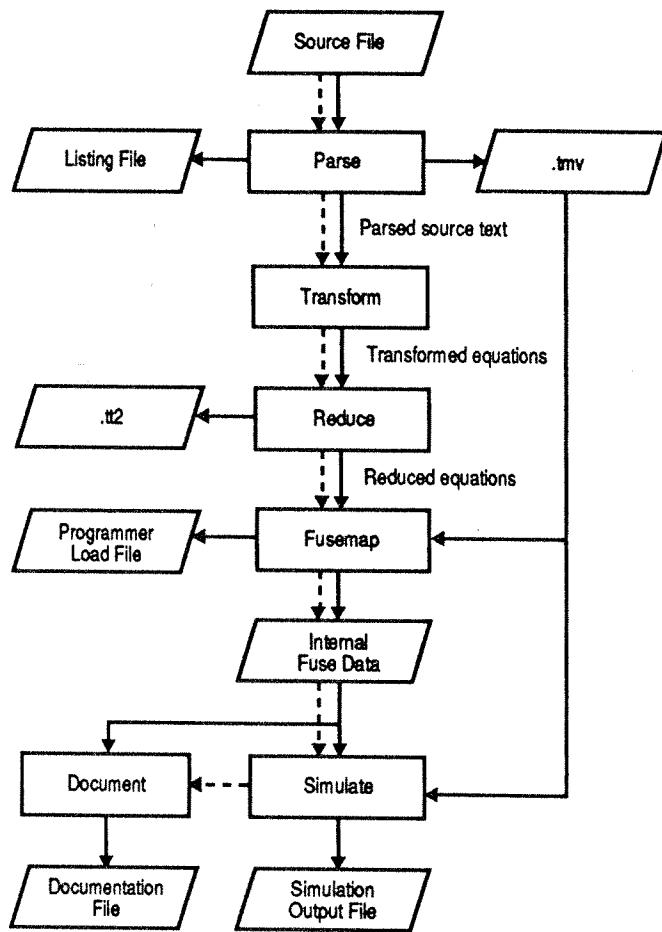
Language Processor

The ABEL-PLD language processor converts logic descriptions in the source file to programmer load files that can be downloaded to a logic programmer. Processing an ABEL-PLD source file is a six-step process:

1. PARSE — Reads the source file, checks for correct syntax, flags errors, expands macros, and acts on directives.
2. TRANSFOR — Converts the description to an intermediate form.
3. REDUCE — Performs logic reduction.
4. FUSEMAP — Creates the programmer load file.
5. SIMULATE — Simulates the function of a design with test vectors contained in the source file and reports any functional failure of the design.
6. DOCUMENT — Creates design documentation, including a listing of the source file, a drawing of the logic device pin assignments, and a listing of the programmer load file.

The ABEL-PLD command is a batch file or command script that invokes all six steps of the language processor. This chapter discusses the functions of the batch file and each of the six steps. Figure 2-2 shows the processing flow of the language processor. You can either run the batch file to process a design, or you can run each step of the language processor separately. The chapter "Using ABEL-PLD" discusses how to run the ABEL-PLD software using both of these options.

Figure 2-2
Processing Flow of the Language Processor



- - - - - ➤ indicates batch file processing

→ indicates data flow

Parse

PARSE reads the source file, converts state descriptions and truth tables to equations, translates test vectors, expands macros, and checks for correct syntax. If any syntax errors are found, the approximate place at which the error occurs and the type of error are displayed on your monitor. The language processor also checks your design to see if it can be implemented on the chosen device. For example, if a device input pin is used as an output in an equation, the language processor detects and reports the error.

Transform

TRANSFOR reads the equations generated by PARSE and manipulates them to

- Replace sets with equivalent equations without sets.
- Replace all operators with equivalent operations using only NOTs, ANDs, ORs and XORs.
- OR together equations that cause multiple assignments to the same identifier.
- Perform logic reduction based on the following rules:

Table 2-1
Transfor Logic Reduction Rules

Rule	Description
A & 1 = A	A AND 1 = A
A & 0 = 0	A AND 0 = 0
A # 1 = 1	A OR 1 = 1
A # 0 = A	A OR 0 = A
A \$ 1 = !A	A XOR 1 = NOT A
A \$ 0 = A	A XOR 0 = A
A !\$ 1 = A	A XNOR 1 = A
A !\$ 0 = !A	A XNOR 0 = NOT A

The transformed equations can then be reduced by the REDUCE program.

Reduce

REDUCE reduces your logic description so that fewer product terms are used in the programmable logic device, so that you do not have to perform the tedious task of logic reduction by traditional methods such as Karnaugh maps. You may choose different levels of reduction based on the design and the device. REDUCE reduces the equations in the input file and writes the reduced equations to the output file.

Fusemap

FUSEMAP processes the output of REDUCE and creates

- an intermediate output file (*.out) for input to SIMULATE and DOCUMENT.

- programmer load files (*.jed) that are loaded into a logic programmer to program and test devices.

**Standard JEDEC
Format
Programmer Load
File**

The standard programmer load file created by the language processor conforms with the JEDEC standard No. 3A for data transfer to logic programmers. JEDEC format files are used to transfer designs to the logic programmer. Other formats for PROM programmers are supported.

Simulate

SIMULATE uses design and device information to simulate the operation of a programmable device. The simulation facility uses device characteristics, a fuse map, and test vectors to simulate the actual operation of the device. The fuse map and test vectors used for simulation are the same as those that will be used to program and test the real device. SIMULATE can use either the design information created by FUSEMAP or any programmer load file conforming with the JEDEC standard to simulate the operation of PALs, CPLAs, and FPLSs. FUSEMAP intermediate file output must be used to simulate PROMs.

SIMULATE does not execute equations or apply inputs to ABEL-PLD truth tables or state descriptions; it simulates the operation of a device as though it were already programmed with the information contained in the input file.

The simulation facility is a powerful feature of ABEL-PLD that allows you to find and correct design errors before programmable logic devices are programmed. This can mean great savings in time and expense during the logic design cycle. By simulating the operation of a part, you can make sure that a design is correct before you program the part, thus eliminating trial-and-error programming and testing of devices. Simulation is performed by the SIMULATE step of the language processor.

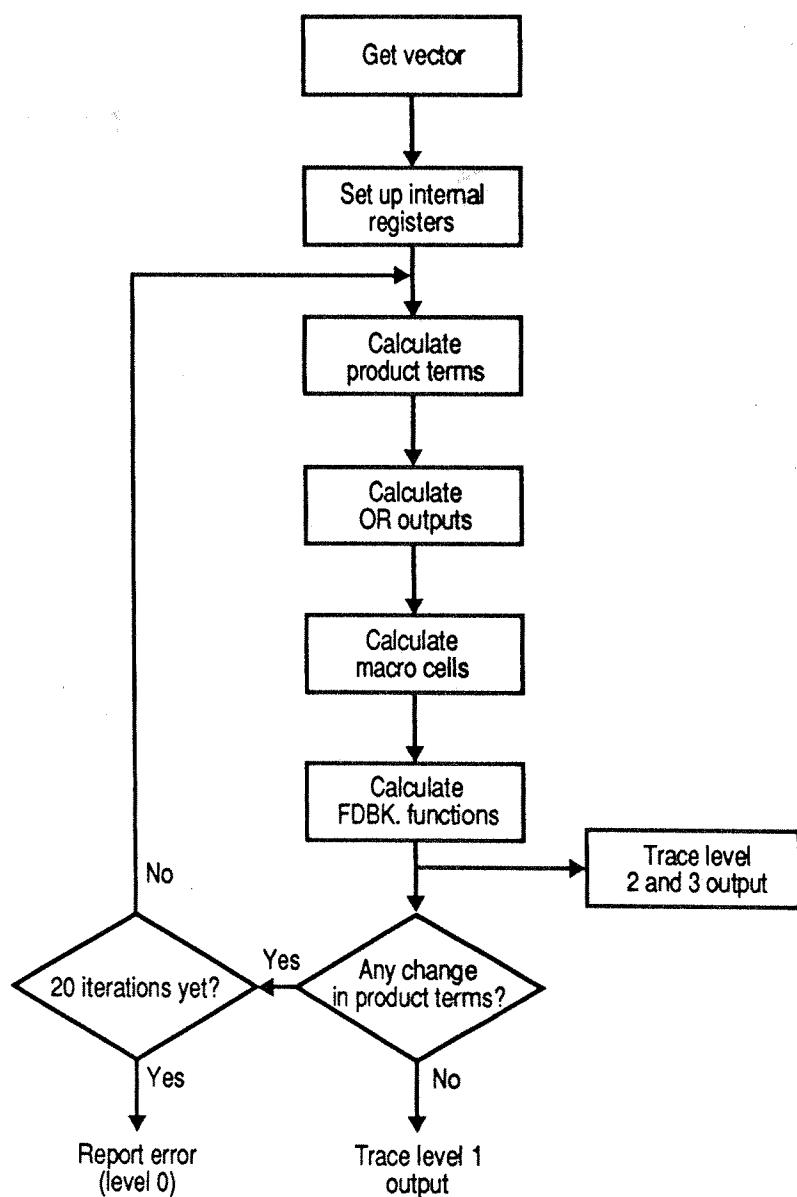
SIMULATE simulates the actual operation of a programmed device, as opposed to simply simulating the logic description provided in the source file. This is an important distinction. Simple simulation of the logic description, as contained in the source file, would not take the device that is to be programmed into account, and, clearly, such simulation would not truly indicate whether the programmed device would function as desired. To simulate the real operation of the device, device specific information must be used in conjunction with the programming information. This is how the ABEL-PLD software performs simulation.

SIMULATE Program Operation

Figure 2-3 shows a flow diagram that depicts SIMULATE operation when evaluating the inputs to the output. SIMULATE gets the first test vector and performs any setup of internal registers (within the PLD) that results from the vector applied to the inputs. SIMULATE then calculates the product terms that result from the test vector, the OR outputs that result from the product terms, any macrocell outputs that result from the OR outputs, and then any feedback functions. As indicated in Figure 2-3, the results of the SIMULATE calculations are written to the *.SIM file when trace level 2 or trace level 3 are specified (see the chapter "Using ABEL-PLD" for more information on trace levels).

The outputs of devices with feedback cannot always be determined by one evaluation of the input-to-output function, but may require several successive evaluations until the outputs stabilize. SIMULATE uses an iterative method to compute the outputs. After the feedback paths have been calculated, SIMULATE checks to see if any changes have occurred with the device since the product terms were last calculated. If changes have occurred due to feedback functions, the calculations are again repeated. This iterative process continues until no changes are detected, or until 20 iterations have taken place. If 20 iterations take place, the design is determined to be unstable and an error is reported. More detailed information on simulating devices with feedback, and other advanced uses of the simulation program are presented in "Simulation and Designs with Buffered Outputs" in the chapter "Advanced Features."

Figure 2-3
Simulate Processing Flow Diagram



Document

DOCUMENT creates design documentation from information provided by previous steps of the language processor.

Depending on the options chosen, the design documentation may contain the following information for each device in the source file:

- Symbol table
- Reduced equations
- Transformed equations
- Equations
- Test vectors
- Fuse map
- Chip Diagram

Functional Device Testing

If test vectors are specified in a source file, the programmer load file created by the language processor contains these vectors in a form that can be used to test a programmed device with a logic programmer.

Chapter 3

Using ABEL-PLD

Creating a Source File

Before ABEL-PLD can work with your design, you need to create a *source file* that contains a description of your design. The ABEL-PLD software allows you to describe your design in any combination of high-level equations, Boolean equations, truth tables and state descriptions. In order for the software to understand your description, the source file needs to be in a standard format. This format is described in detail in the Language Reference.

If you want to try running the software before creating a source file, several source files are provided with the ABEL-PLD package for your use. These files all have .abl extensions. The chapter "Design Examples" in the Applications Guide contains descriptions of many of these files to assist you in creating your own source files.

Using ABEL-PLD

The ABEL-PLD software can be run as a batch process, or each step can be individually executed. The ABEL-PLD software can be run from the command line, or from a menu system called the Personal Silicon Foundry (PSF). Both methods are discussed for each ABEL-PLD program. Headers in the left margin give command line options in normal text, and PSF menu options in boxed text.

The following table is a summary of option usage in ABEL-PLD.

Table 3-1
ABEL-PLD Program Options

Option	Parse	Transform	Reduce	Fusemap	Simulate	Document	Batch*	Flag*	PSF Menu
-i <i>infile</i> — Input File Name	♦	♦	♦	♦	♦	♦			●
-o <i>outfile</i> — Output File Name	♦	♦	♦	♦	♦	♦			●
-l <i>listfile</i> — List File	♦								●
-e — Expanded Output	♦						●		●
-p — Expanded Output & Directives	♦						●		●
-a <i>argument...</i> — Pass Arguments To Source File	♦						●		
-h <i>path</i> — Include File Path	♦						●		
-y <i>path</i> — Device File Path	♦						●		
-r(0 1 2 3 4 5) — Reduction Level			●				●	●	●
-j <i>path</i> — JEDEC File Path				♦			●	●	
-c(0 1 2) — Checksum				♦		●	●		●
-d(0 82 83 87 91 92 93) — Programmer Load File Output Format				♦		●	●	●	●
-k(y n) — Connect/disconnect Unused OR Fuses				♦		●	●	●	●
-t(0 1 2 3 4 5) — Trace Level					♦	●	●	●	●
-bp1,p2 [,trace_level] — Break Points					♦				●
-n <i>device</i> — Device Type					♦	●	●		●
-x(0 1 h l) — Value for Don't Cares (.X.)					♦	●	●	●	●
-z(0 1 h l) — Value for High Z (.Z.)					♦	●	●	●	●

Option	Parse	Transfer	Reduce	Fusemap	Simulate	Document	Batch*	Flag*	PSF Menu
-w $pin\#,pin\#,\dots$ — Watch Points				◆		●	●	●	
-u(0 1 h l) — Power Up State				◆		●	●	●	
-f(0 1 2) — Fusemap and/or Terms					◆	●	●	●	
-g[0 1] — Chip Diagram					◆	●	●	●	
-s — List Symbol Table					◆	●	●	●	
-q[0,1,2,3]... — List Equations					◆	●	●	●	

*Batch denotes commands valid for the abel.bat batch file. Flag denotes commands valid for the Flag statement in the source file. PSF Menu denotes commands available through the PSF menus.

● denotes the option is valid; ◆ denotes the option is valid and is documented with that program.

Remarks	<p>The input file that you specify must be a valid source file, and devices used in the source file must be supported by the ABEL-PLD device library. See the <i>ABEL-PLD User Notes</i> for the complete device listing.</p> <p>As the table shows, any of the options associated with the six steps of the language processor, except for input file name, output file names and breakpoints, can be entered on the command line to control how a source file is processed. The input file name is already specified, output file names are determined by the defaults, and breakpoints can only be specified when running SIMULATE separately.</p> <p>If you do not specify any options, and no Flag statements are given in the source file (see the "Language Reference" for information on Flag statements), default values are used to perform the following actions:</p> <ul style="list-style-type: none"> • A parsed listing file is created. Text included with the @INCLUDE directive is shown. • Level 1 (simple) logic reduction is performed. • Programmer load files are created according to the JEDEC standard. • Level 0 simulation is performed (shows only simulation errors). • Design documentation is created. • Processing status, statistics and error messages are displayed on the screen. <p>When any of the ABEL-PLD programs are run individually (without being invoked by the abel.bat batch file), using -i (input file) or -o (output file) without a file name will cause the ABEL-PLD passes to suppress prompting for input and output file names. This is useful for redirecting program inputs and outputs. The SIMULATE program will assume a default file extension of .OUT for the input file name.</p> <p>If the JEDEC file contains the name of the device, the -n flag need not be used with SIMULATE. The device name will be included as the default at the device prompt. To accept the default device, press <Enter> and processing will continue.</p>
ABEL-PLD Output Files	<p>The following output files are produced. These output files are described in detail in the chapter "Understanding ABEL-PLD": <i>filename.lst</i>, <i>device.jed</i> (one for each device in the source file), <i>filename.sim</i>, <i>filename.doc</i>, and <i>filename.out</i>.</p>
Example Batch Run	<p>The following example uses the address decoder source file, <i>m6809a.abl</i>, provided on the Examples disk in the ABEL-PLD package. Copy this file to the current directory for processing by ABEL-PLD.</p> <p>Enter the following command line to process the address decoder example (.abl file extension not required):</p> <pre>abel m6809a</pre>

The processing statistics for each step of the language processor's operation are printed to the screen, as shown below. Processing times vary between systems.

PARSE ABEL-PLD(tm) 3.21
Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

PARSE complete. Time: 3 seconds

TRANSFOR ABEL-PLD(tm) 3.20

Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

TRANSFOR complete. Time: 2 seconds

REDUCE ABEL-PLD(tm) 3.20

Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

device U09a

Simple Reductions (Sum Of Products)

REDUCE complete. Time: 2 seconds

FUSEMAP ABEL-PLD(tm) 3.21

Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

device U09a 'P14L4'

6 of 16 terms used, 8 vectors included

FUSEMAP complete. Time: 2 seconds

SIMULATE ABEL-PLD(tm) 3.20

Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

device U09a 'P14L4'

.....

8 out of 8 vectors passed.

SIMULATE complete. Time: 2 seconds

DOCUMENT ABEL-PLD(tm) 3.20

Copyright 1983-1990 Data I/O Corp. All Rights Reserved.

module m6809a

device U09a

DOCUMENT complete. Time: 2 seconds

When the processing is complete, several new files are created. These files are

filename.ext	Description
m6809a.lst	Listing from PARSE, used to check syntax
u09.jed	Programmer load file for design transfer and input to PLDtest/PLDtest Plus
m6809a.sim	Simulation output file for error checking
m6809a.doc	Documentation file for design
m6809a.out	Intermediate file for SIMULATE, DOCUMENT, and PLDtest/PLDtest Plus

The name, u09, of the programmer load file is taken from the device identifier in the source file. You may want to look at the listing, plus the simulation (.sim) and documentation (.doc) files to see what the language processor creates.

ABEL-PLD Libraries

Device Libraries

Device specification files are provided in a single library file (abel3lib.dev) that contains specifications on all supported devices. When the ABEL-PLD software is invoked, it searches for a *.dev device file corresponding to each device referenced in the source file. It searches the current (local) directory first, then the abel3dev directory. If you have included the device library, abel3lib.dev, in the abel3dev directory, ABEL-PLD will look for the device specifications in the abel3lib.dev library if no appropriate *.dev file is found.

You can obtain individual device (*.dev) files from the library file as described in the section "ABELLIB, Library Manager," later in this chapter. By extracting individual device files from the library, you can load only those files you need instead of reading the entire library file each time you use ABEL-PLD.

The device library file should be in the abel3dev directory on your system. Refer to the Installation Guide provided with your ABEL-PLD package for information on installing the device library file.

Include Libraries

ABEL-PLD include files may also be contained in a library. When ABEL-PLD requires an include file (a file that has been referenced in an @INCLUDE or LIBRARY statement), it tries to find the required file in the local directory. If the file is not found, ABEL-PLD looks for the library, abel3lib.inc, and tries to find the specified Include file within it. A source file can have one or more Include files that will be concatenated into the source file during processing.

Refer to the section "ABELLIB, Library Manager" later in this chapter for information on libraries and on the library manager utility (used to add and delete files from abel3lib.dev and abel3lib.inc).

Using the PSF Menus

The ABEL-PLD Utilities disk contains a batch file named psf.bat used to invoke the PSF menus. After these programs are copied to your system, you can access and use the Personal Silicon Foundry menus to execute ABEL-PLD and related functions. The basic operation of the PSF menus is given below. You can also use the command line to execute ABEL-PLD. Use the PSF menus that refer to ABEL, since these menus are used for both ABEL and ABEL-PLD.

Note: Personal Silicon Foundry is for the IBM-PC/XT/AT and compatibles. To run PSF you must have 512K of RAM plus an EGA, VGA, or monochrome graphics board installed in your computer. A mouse is also recommended, and any mouse that is compatible with the Microsoft mouse driver will perform satisfactorily. If you do not have a mouse, you can use the + and - keys (on the right side of the keyboard) for the left and right mouse buttons respectively.

Accessing the Main Menu

When ABEL-PLD and PSF are installed as described in the *Installation Guide*, entering

`psf`

on the command line will access the main PSF menu. Move the mouse pointer to highlight the Logic Synthesis selection and press the right mouse button. This displays the list of menus available for running ABEL-PLD. You can select any of these menus by moving the pointer to highlight the desired function and pressing the right mouse button.

Use In-File Setting

This menu selection is a choice for many options. Use In-File Setting will use the option value selected in the source file, or the default value if no options are specified in the source file.

Command Buttons

Each PSF menu contains one or more of the following command buttons, which can be selected by moving the pointer to the button and pressing the left mouse button:

RUN
ESC
EDIT
LIST

RUN

The RUN button initiates processing by invoking the necessary program(s) and passing the filenames and other parameters entered on the menu.

ESC

The ESC button cancels the entries made to the menu and returns you to the opening PSF menu.

EDIT

The EDIT button invokes the text editor in menus where it is meaningful to edit the contents of a file. When editing is complete, the system returns to the current menu.

LIST

The LIST button directs the output file generated by running the SIMULATE and DOCUMENT operations to the display.

The PSF Run Series Menu

The following table describes each of the menu selections available on the Run Series menu.

Figure 3-1
PSF Run Series Menu

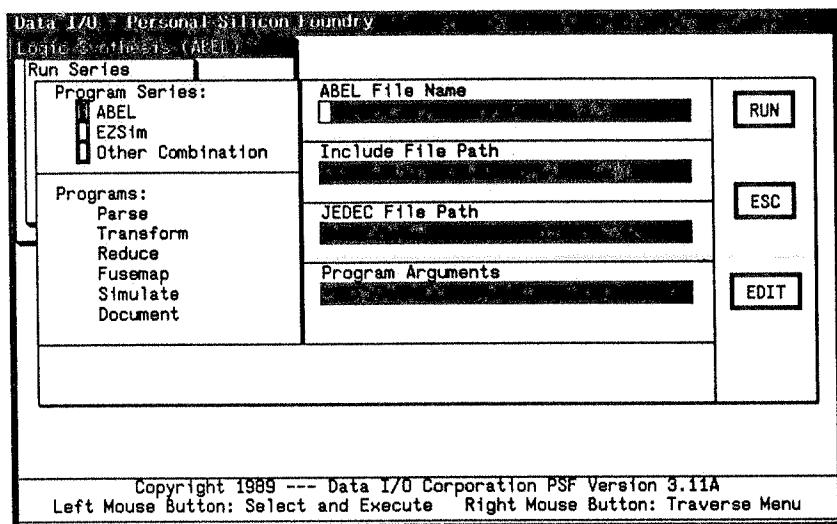


Table 3-2
Batch Processing Options

Run Series Menu Option	Command Line Equivalent
Program Series: ABEL	<code>abel</code>
Program Series: EZsim	Not implemented
Program Series: Other Combination	(See below)
ABEL File Name	<code>-filename</code>
Include File Path	<code>-hpath</code>
JEDEC File Path	<code>-jpath</code>
Program Arguments	any valid batch options

**Program Series:
ABEL**

This selection runs the ABEL-PLD source file through all modules of the ABEL-PLD language processor. Refer to "Processing an ABEL-PLD Source File."

**Program Series:
Other Combination**

This selection allows you to pick which modules of the ABEL-PLD language processor to run the ABEL-PLD source file through. Selection and de-selection of the different modules is made by clicking the left mouse button on the list of programs that appears below the Program Series window. Initially, all programs are selected. See also "Creating Your Own Batch or Command Files."

ABEL File Name

Enter the name of the ABEL-PLD source file in this entry field.

Include File Path

Not implemented.

JEDEC File Path

Not implemented.

Program Arguments

Enter any valid command line options in this entry field. See Table 3-1.

Note: Each of the ABEL-PLD menus allows you to enter filenames. If you do not specify a file extension, the default file extension used by ABEL-PLD will be inserted when the <Enter> key is pressed.

Command Line Processing

Abel.bat is a batch file that contains the commands needed to run all six steps of the language processor automatically. The basic operation of the command line is given below. You can also use the PSF menus to execute ABEL-PLD.

**Command Line
Syntax**

abel *filename* [*options*]

filename

The name of an ABEL-PLD source file. The file extension must be .abl, but is not entered.

options

Options must be separated by spaces. Refer to Table 3-1 for short descriptions of available options, and to the individual ABEL-PLD programs for more detailed information.

**Calling the
ABEL-PLD Batch
File from Another
Batch File**

If you want to run the ABEL-PLD batch file from another batch file you may need to restore ECHO at the end of the batch file. Just add the line "ECHO ON" to the end of the abel.bat file.

**Creating Your Own
Batch or Command
Files**

The ABEL-PLD batch or command file executes all steps of the language processor. You may find that you want to create your own batch file that invokes only some of the steps in the language processor. Or, you may want to run the language processor with special option settings at each step. For example, you could write a batch file that runs PARSE, TRANSFOR, REDUCE and FUSEMAP to create a programmer load file without generating design documentation, and without running a simulation.

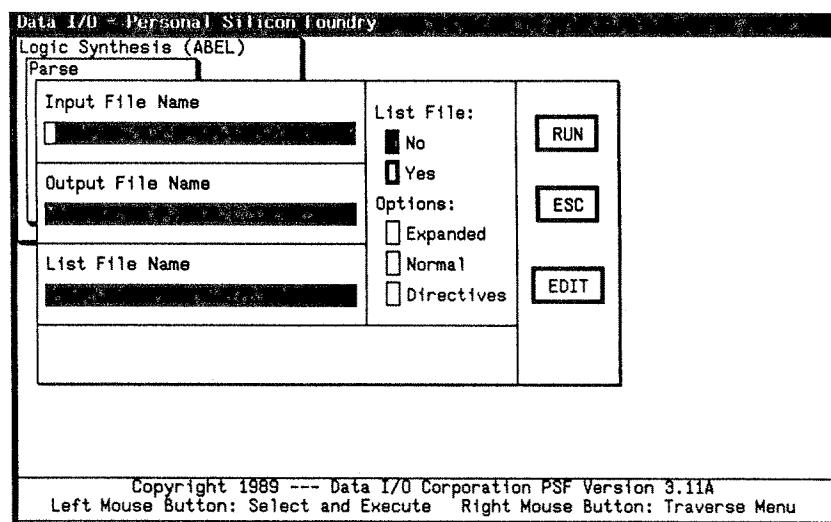
You can write your own custom batch file by following the rules governing batch files as described in your DOS manual. Combine the individual language processor steps in the way that best suits your application.

ABEL-PLD Programs

PARSE

PARSE reads the source file, converts state diagrams and truth tables to equations, translates test vectors, expands macros, and checks for correct syntax. If any syntax errors are found, the approximate place at which the error occurs and the type of error are displayed on your monitor. Error messages are also written to a listing file if one is being created (-l). An intermediate file is written to *outfile* if -o is specified. This intermediate file provides input to TRANSFOR.

Figure 3-2
Parse Menu



PARSE can be executed from either the PSF menus or the command line.

Command Line Syntax

```
parse [-iinfile] [-ooutfile] [-llistfile] [-e] [-p] [-aargument]... [-hpath] [-yopath]
```

Table 3-3
Parse Options

Command Line	PARSE Menu
<code>-filename</code>	Input Filename
<code>-ofilename</code>	Output Filename
<code>-lfilename</code>	List File
	Options:
<code>-e</code> [default]	Expanded
<code>-p</code>	Normal
<code>-aarg</code>	Directives
<code>-hpath</code>	N/A (Path to Include Files)
<code>-yঃpath</code>	N/A (Path to Device Files)

Remarks

All options are optional. If the `-i` and/or `-o` options are omitted (no file specified), you will be prompted for the input and/or output filenames.

Consider the following example command line (with default file extensions named):

```
parse -aP14L4 -aGND -im6809a.abl -om6809a.tm1 -lm6809a.lst -e
```

This command (entered on a single line to the operating system) invokes PARSE to process the source file m6809a.abl which contains the logic description for the address decoder described in section 3. The output is written to the file, m6809a.tm1, and a listing file named m6809a.lst is created. Expanded output is shown in the listing because the `-e` option is included. Two arguments, P14L4, and GND, are passed to the processor for argument substitution.

The command options can be entered in lowercase characters and in any order. Each option is discussed in detail below. The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-ifilename

Input File Name

Use **-ifilename** to specify the file containing your source file. If no input file is specified, the program will prompt you to enter the input file name from your keyboard. The input is assumed to come from your standard input device (usually the keyboard). This feature is useful for piping file output.

-ofilename

Output File Name

Use **-ofilename** to specify the name of the output file. If no output file is specified, output will go to your standard output device (usually the monitor). The output file contains the parsed source code and is used as input to the TRANSFOR processor.

-lfilename

List File

-lfilename creates a listing file. The filename is specified directly after the **-l** option. A listing file contains the parsed source code with error messages (if any) and with macro expansions and inclusion of code due to directives if the **-e** option was specified. If the **-p** option is specified, the listing also shows the directive that caused the inclusion of code. If **-l** is not used, no listing file will be created.

-e

Options:
Expanded

The **-e** option causes the parsed and expanded source code to be written to the listing file. Text included by macros and directives is shown. If **-e** is not specified, the listing contains the source file as it was before processing plus error messages: expanded text is not shown.

-p

Options:
Directives

In addition to the listing information provided by **-e**, **-p** lists the directives that caused code to be added to the source.

-aarg
(Pass Arguments to Source)

The **-aarg** option lets you pass arguments to a source file. These arguments are substituted for dummy arguments in the source. As many arguments as are needed can be specified, but each argument must be preceded by the **-a** option. Argument substitution is discussed further in the chapter "Language Elements" in the Language Reference. This option is not available via the PSF menus.

-hpath
(Path to Include Files)

The @INCLUDE directive, described in the chapter "Directives," lets you include source text from one file in another source file. By default, the included file is assumed to be in the default directory; **-hpath** allows you to override that default value and explicitly specify where included files are to be found. Specify a drive and path immediately after **-h**. For example, the following specifies that include files are to be found in the directory named examples:

-h\examples

See your DOS manual for more information about path and drive specifications. This option is available via the PSF Run Series menu, not the PARSE menu.

-ypath
**(Path to Device
 Files)**

The language processor uses device specification files in conjunction with your source file to process a design. The **-ypath** option lets you specify which drive and directory contains the device files. Specify a path immediately after **-y**.

For example,

-y\devices

indicates that device specification files are in a system subdirectory, **\devices**. If **-y** is not used, device files are assumed to be in the default directory. See your DOS manual for more information about path and drive specifications.

PARSE Listing File

Figure 3-3 shows a listing file created by the PARSE step of the language processor. This listing was created by running PARSE on an altered version of the **m6809a.abl** source file described in section 3. Syntax errors were introduced into the **m6809a.abl** source file to create the file **m6809err.abl**, also provided with the design examples. **M6809err.abl** contains two syntax errors.

The first error is a missing semicolon in the equation:

IDRAM = (Address < - ^hDFFF)

The second error is a missing left parenthesis in the truth table header:

test_vectors Address -> [ROM1,ROM2,IO,DRAM])

These two errors are displayed on your monitor and shown in the listing file. The approximate places at which the errors occur are indicated by circumflexes (^). The type of error is also indicated.

Figure 3-3
**PARSE Listing File with
 Errors from m6809err.abl**

```

0001 |module m6809err
0002 |title '6809 memory decode
0003 |Jean Designer      Data I/O Corp Redmond WA'
0004 |
0005 |          U09      device 'P14L4';
0006 |  A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007 |  ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008 |
0009 |  H,L,X   = 1,0,.X.;
0010 |  Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];
0011 |
0012 |equations
0013 |      !DRAM   = (Address <= ^hDFFF)
0014 |
0015 |      !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);
0016 |
0017 |      !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
0018 |
0019 |      !ROM1   = (Address >= ^hF800);
0020 |
0021 |test_vectors Address -> [ROM1,ROM2,IO,DRAM])
0022 |          ^
0023 |          ? Syntax error: ';' expected
0024 |          ^
0025 |          ^
0026 |          ^
0027 |          ^
0028 |          ^
0029 |          ^
0030 |end m6809err

```

Figure 3-4 shows the corrected source file with the proper semicolon and left parenthesis added.

Figure 3-4
Corrected Source File,
m6809a abl

```

0001 |module m6809a
0002 |title '6809 memory decode
0003 |Jean Designer      Data I/O Corp Redmond WA'
0004 |
0005 |          U09a    device 'P14L4';
0006 |          A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007 |          ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008 |
0009 | H,L,X   = 1,0,.X. ;
0010 | Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011 |
0012 |equations
0013 | !DRAM   = (Address <= ^hDFFF);
0014 |
0015 | !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);
0016 |
0017 | !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
0018 |
0019 | !ROM1   = (Address >= ^hF800);
0020 |
0021 |test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
0022 |          ^h0000 -> [ H , H , H, L ];
0023 |          ^h4000 -> [ H , H , H, L ];
0024 |          ^h8000 -> [ H , H , H, L ];
0025 |          ^hC000 -> [ H , H , H, L ];
0026 |          ^hE000 -> [ H , H , L, H ];
0027 |          ^hE800 -> [ H , H , H, H ];
0028 |          ^hF000 -> [ H , L , H, H ];
0029 |          ^hF800 -> [ L , H , H, H ];
0030 |end m6809a

```

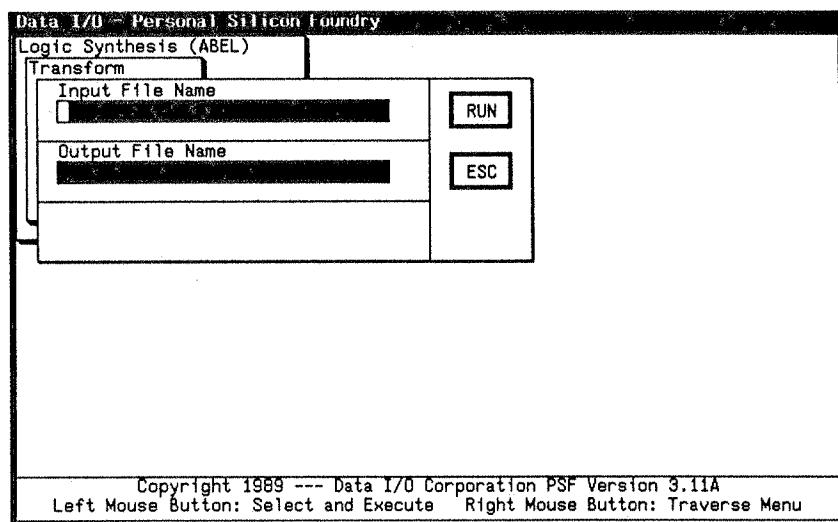
TRANSFOR

TRANSFOR manipulates the equations generated by PARSE to

- Replace sets with equivalent equations without sets.
- Replace all operators with equivalent operations using only NOTs, ANDs, ORs and XORs.
- OR together equations that cause multiple assignments to the same identifier.
- Perform logic reduction (see "Understanding ABEL-PLD for logic reduction rules used by TRANSFOR").

TRANSFOR can be executed from the PSF menus or the command line.

*Figure 3-5
Transfor Menu*



Command Line
Syntax

transfor [-iinfile] [-ooutfile]

Table 3-4
Transfor Options

Command Line	TRANSFOR Menu
-ifilename	Input File Name
-ofilename	Output File Name

Remarks

The transformed equations can then be reduced by the REDUCE program. The file specifications given for input and output filenames must be valid file specifications for DOS. (Note that you can assign any legal file extension to the input and output filenames.)

For example,

transfor -im6809a -om6809a.tm2

This command transforms the parsed source code contained in m6809a.tm1 (created by PARSE) and creates an output file named m6809a.tm2.

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-ifilename

Input File Name

Enter the name of the intermediate file created by the parser.

-ofilename

Output File Name

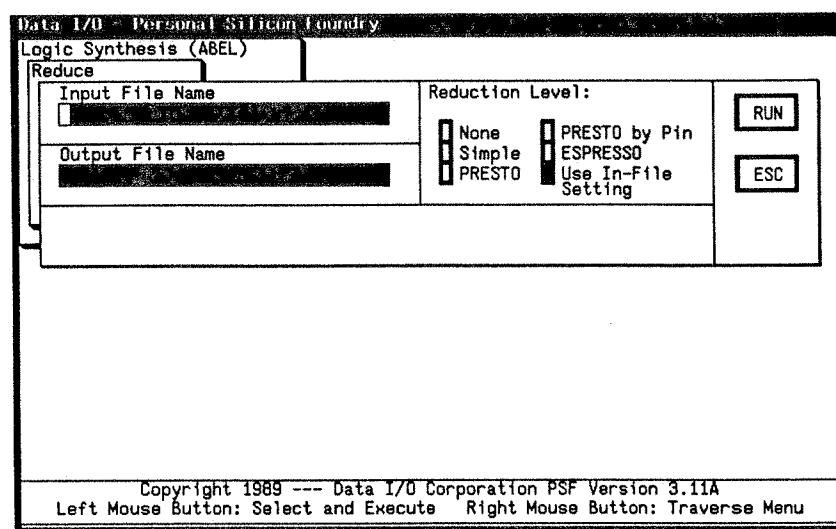
Enter the name of the file for the transformed output to be written to. If no filename is entered here, the input filename will be used and the transform program will assign a .tm2 extension.

REDUCE

REDUCE performs reduction on your logic description so that fewer product terms are used in the programmable logic device. The equations in *infile* are reduced and then written to the file specified by *outfile*. The equations in *infile* must be in the form produced by TRANSFOR. If *infile* is not specified, the input is assumed to come from your standard input device (usually your keyboard). If *outfile* is not specified the output is written to your standard output device (usually your monitor).

REDUCE can be executed from the PSF menus or the command line.

Figure 3-6
Reduce Menu



Command Line
Syntax

reduce [-i infile] [-o outfile] [-rn]

Table 3-5
Reduce Options

Command Line	REDUCE Menu
<code>-ifilename</code>	Input File Name
<code>-ofilename</code>	Output File Name
<code>-rn</code>	Reduction Level:
<code>-r0</code>	None
<code>-r1</code>	Simple
<code>-r2</code>	Presto
<code>-r3</code>	Presto by Pin
<code>-r4</code>	Espresso
<code>-r5</code>	N/A (Espresso by Pin)
<code>N/A</code>	Use In-File Setting

Remarks

You may choose one of six levels of reduction: 0, 1, 2, 3, 4 or 5. If you do not explicitly choose a reduction level with `-r`, level one reduction is performed.

When using exclusive-OR parts, such as P20X8 and P20X4, and exclusive-OR conditions are encountered in the input file to REDUCE, REDUCE will leave the exclusive-OR operator intact for that output. If more than one exclusive-OR condition is encountered for one output, only the first will use the exclusive-OR operator; others will be converted to multiple ANDs and ORs.

If a T-type flip-flop is being emulated in a D-type registered device, all exclusive-OR operators for that output will be expanded to multiple ANDs and ORs.

The "group" reduction options (`-r2` and `-r4`) will minimize equations for an FPLA architecture where a single product term can be used by multiple outputs. The "by the pin" options (`-r3` and `-r5`) are significantly faster and produce the best fit for PAL architecture devices and a satisfactory fit for FPLAs.

The Espresso options produce two temporary files with the device ID and a `.tt1` and `.tt2` extension. The device ID is used for the file names, not the device type, since more than one of a device type can be used in a single source file.

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-filename

Input File Name

Enter the name of the intermediate file created by the Transform program.

-ofilename

Output File Name

Enter the name of the file for the REDUCE output to be written to. If no filename is entered here, the output will be directed to the standard output device (usually the monitor).

-r0

**Reduction Level:
None**

No reduction is performed for level 0 reduction. The transformed equations are checked for valid outputs for the specified device and written to the output file.

-r1

**Reduction Level:
Simple**

If you choose level 1 reduction (-r1), the logic equations are expanded to a sum of products form and then reduced according to the basic rules of Boolean algebra:

Table 3-6
Basic Rules of Boolean Algebra

Rule	Description	
$\text{!}0 = 1$	NOT 0	= 1
$\text{!}1 = 0$	NOT 1	= 0
$A \& 0 = 0$	A AND 0	= 0
$A \& 1 = A$	A AND 1	= A
$A \# 0 = A$	A OR 0	= A
$A \# 1 = 1$	A OR 1	= 1
$A \# A = A$	A OR A	= A
$A \& A = A$	A AND A	= A
$A \& \text{!}A = 0$	A AND (NOT A)	= 0
$A \# \text{!}A = 1$	A OR (NOT A)	= 1

Level 1 reduction is the default if no level is chosen with the -r option.

-r2

Reduction Level:
Presto

Level 2 reduction should be used when the number of product terms used to implement a function is near to or more than the number of product terms available in the device. If you choose level 2 reduction (-r2), both level 1 and Presto logic reduction are performed.

The following command line performs simple and Presto reduction on the transformed equations contained in m6809a.tm2 (created by TRANSFOR) and writes the reduced equations to the file named m6809a.tm3. See below for more details on choosing between Espresso and Presto reduction.

```
reduce -r2 -im6809a.tm2 -om6809a.tm3
```

-r3

Reduction Level:
Presto by Pin

Level 3 reduction reduces logic associated with each pin in a device on a pin-by-pin basis. There are two major advantages to pin reduction. First, reduction by pin is faster than normal Presto reduction. Second, this type of reduction is well-suited to PALs which do not share terms among the outputs. See below for more details on choosing between Espresso and Presto reduction.

Presto by the pin (-r3) is the fastest reduction option for small to medium designs. If either Presto option (-r2 or -r3) takes more than 30 seconds, interrupt the program and use Espresso (-r4 or -r5).

-r4

Reduction Level:
Espresso

Level 4 reduction uses the Espresso reduction method. Espresso's main advantage over Presto is its speed, which makes it the best choice for large designs. In some cases, it will also produce better reduction than Presto. See below for more details on choosing between Espresso and Presto reduction.

-r5

(Espresso by Pin)

Espresso by the pin (-r5) performs logic reduction for each output, independent of the other outputs. This is similar to the -r3 option, which uses the Presto algorithm. This option is not available through the PSF menus.

**Espresso and
Presto Reduction**

Both Espresso and Presto are heuristic logic reduction algorithms that attempt to reduce logic to its minimal form within a reasonable runtime. The difference between these algorithms is the tradeoff ratios for degree of optimization versus program performance (runtime).

Espresso's strength is in unusual reduction cases. With simple reduction cases, Espresso will usually take longer to run than Presto. However, with more complex reduction cases, Espresso can achieve better results in a shorter period of time.

Espresso, as implemented in the ABEL-PLD REDUCE software, performs global minimization (similar to ABEL-PLD's level 2 reduction). It attempts to share product terms where possible. Global minimization is useful in devices which share product terms, such as FPLAs available from Signetics.

Presto, on the other hand, reduces either globally (level 2) or pin-by-pin (level 3). For devices which don't share product terms (PAL type devices), use reduction level 3 (Presto, pin-by-pin). For devices that share product terms, which level to use depends on how extensive the reduction will be. Typically, Espresso is the better choice; the exception being when the design is very simple.

One added advantage of Espresso is that it produces a file containing the reduced equations in Berkeley format. ABEL-PLD software does not use this file, but tools which have standardized on the Berkeley format, such as those from LSI Logic, can import the reduced equations produced by Espresso.

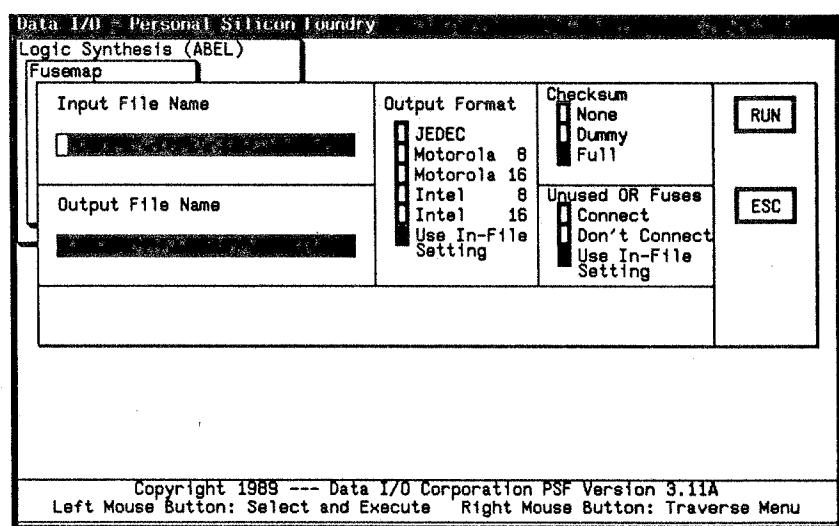
FUSEMAP

FUSEMAP processes the output of REDUCE and creates

- an intermediate output file (*.out) for input to SIMULATE and DOCUMENT.
- programmer load files that are loaded into a logic programmer to program and test devices.

FUSEMAP can be executed from the PSF menus or the command line.

Figure 3-7
Fusemap Menu



Command Line
Syntax

`fusemap [-iinfile] [-ooutfile] [-dn] [-cn] [-k(y|n)] [-jpath]`

Table 3-7
Fusemap Options

Command Line	FUSEMAP Menu
<code>-filename</code>	Input Filename
<code>-ofilename</code>	Output Filename
Output Format:	
<code>-d82</code>	Motorola 8
<code>-d87</code>	Motorola 16
<code>-d83</code>	Intel 8
<code>-d88</code>	Intel 16
<code>-d0, -d91</code>	JEDEC (Brief fuse list) [default]
<code>-d92</code>	N/A (JEDEC — Normal fuse map)
<code>-d93</code>	N/A (JEDEC — Hex fuse list)
<code>N/A</code>	Use In-File Setting
Checksum:	
<code>-c0</code>	None
<code>-c1</code>	Dummy
<code>-c2</code>	Full
Unused OR Fuses:	
<code>-ky</code>	Connect
<code>-kn</code>	Don't Connect
<code>N/A</code>	Use In-File Setting

Remarks

One programmer load file is created for each device specified in the original source file. A load file contains fuse states for programming the device and test vectors to test it once it has been programmed. FUSEMAP creates a JEDEC format load file unless you specify a different format with the `-d` option.

The file name of each load file is the name of the device for which the file contains a fuse map. The file extension is ".jed" for JEDEC format programmer load files and ".pxx" for all other load file formats, where xx corresponds to the number specified with the `-d` option. The load file is written to the path specified by `-jpath:`, or to the default path if no drive is specified.

For example,

```
fusemap -im6809a -om6809a.out
```

This command invokes FUSEMAP. The file m6809a.tm3 (output of REDUCE) is read, one programmer load file (u09.jed) is created for device U09 in the source file (see Figure 3-4 for naming of the programmable logic device), and the output file m6809a.out is created. If multiple devices are named in a source file, then multiple programmer load files will be created by FUSEMAP. For example, if a source file has three devices named device1, device2, and device3, FUSEMAP would create the programmer load files device1.jed, device2.jed, and device3.jed.

Each file is a programmer load file (conforming with the JEDEC standard) that describes the logic and test functions to be programmed into the device. These files can be processed by PLDtest/PLDtest Plus to generate a full set of device test vectors.

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-filename

Input File Name

Enter the name of the intermediate file created by the REDUCE program.

-ofilename

Output File Name

Enter the name of the file for the FUSEMAP output to be written to.

-dn

Output Format

The **-dn** option specifies the format of the programmer load file. **-d0** is the default and indicates that the programmer load file is to be in the JEDEC brief fuse list format. Other formats, including compacted JEDEC formats, can be specified by following **-d** with the appropriate microprocessor format code (**-d83**, for example). Supported microprocessor formats, their codes, and the file extension given to the programmer load files are given in Table 3-8.

Note: Non-JEDEC programmer load file formats do not contain test vectors, even if test vectors were specified in the source file.

Table 3-8
*Data Translation Format Codes
 and File Extensions*

Format	Format Code	File Name Extension
JEDEC - Brief fuse list (default)	91	.jed
JEDEC - Normal fuse map	92	.jed
JEDEC - Hex fuse list*	93	.jed
Motorola 8 (Exorciser)	82	.p82
Motorola 16 (Exormax)	87	.p87
Intel 8 (Intellec 8/MDS)	83	.p83
Intel 16 (MCS-86 Hexadecimal Object)	88	.p88

* -d93 works with UniSite Version 2.7 exclusively.

-c(0|1|2)**Checksum**

Normally, STX is placed at the beginning of the programmer load file, and ETX and a checksum are placed at the end of the file in accordance with the JEDEC standard. The -c option allows you to control whether and how STX, ETX, and the checksum are written.

-c0**Checksum: None**

Omits the STX, ETX, and checksum from the programmer load file.

-c1**Checksum: Dummy**

Causes STX and ETX to be written to the programmer load file as usual, but writes a dummy checksum, 0000, thereby disabling checksum checking.

-c2**Checksum: Full**

The default. Causes the full STX, ETX, and valid checksum to be written.

-k(y|n)**Unused OR Fuses**

The -k option lets you specify what to do with unused fuses.

-ky Leaves the fuses connected.

-kn Disconnects unused fuses. If -k is not specified, the unused fuses are left connected.

Unused fuses in an IFL or FPLA OR array can either be left connected or disconnected. Leaving unused fuses connected allows the addition of logic functions to the device, alteration of an existing design in the device, and may improve programming yield. However, some speed and power improvements are achieved by disconnecting all unused fuses. Disconnecting the fuses prevents any future modifications to the device.

-jpath
(Path to JEDEC
Files)

Specify drive and path for programmer load file output

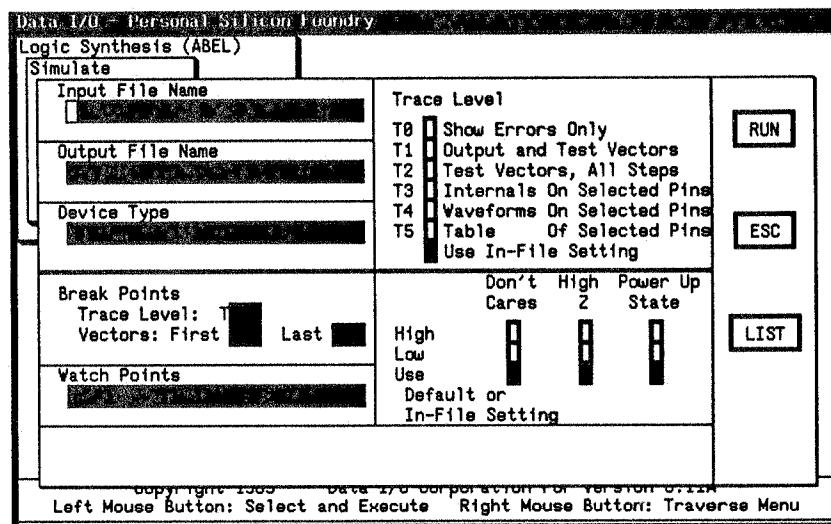
The **-jpath** option allows you to indicate where the programmer load file is written. Specify the desired drive and/or path after **-j**.

SIMULATE

SIMULATE uses design and device information to simulate the operation of a programmable device. SIMULATE can use either the design information created by FUSEMAP or any programmer load file conforming with the JEDEC standard to simulate the operation of PALs, FPLAs, and FPLSs. FUSEMAP intermediate file output must be used to simulate PROMs.

SIMULATE can be executed from the PSF menus or the command line.

Figure 3-8
Simulate Menu



Command Line Syntax

```
simulate [-iinfile] [-ooutfile] [-ndevice] [-bp1,p2[,trace_level]] [-wpin#,pin#,...] [-ttrace_level] [-x(0|1|h|l)] [-z(0|1|h|l)] [-u(0|1)]
```

Table 3-9
Simulate Options

Command Line	SIMULATE Menu
-filename	Input File Name
-ofilename	Output File Name
-device	Device Type
-bp1,p2 [,trace_level]	Break Points
-w	Watch Points
-ttrace_level	Trace Level: -t0 T0, Show Errors Only -t1 T1, Output and Test Vectors -t2 T2, Test Vectors, All Steps -t3 T3, Internals on Selected Pins -t4 T4, Waveforms on Selected Pins -t5 T5, Table of Selected Pins N/A Use In-File Setting
-x(0 1 h l)	Don't Cares: -x1, -xh High -x0, -xl Low N/A Use In-File Setting
-z(0 1 h l)	High Z: -z1, -zh High -z0, -zl Low N/A Use In-File Setting
-u(1 0 h l)	Power Up State: -u1, -uh High -u0, -ul Low N/A Use In-File Setting

Remarks

If the **-i** and/or **-o** options are omitted (no file specified), *filename.out* is the default input file and you will be prompted for the output file names. Also, if the device part number (**-n**) is not specified, you will be prompted. Refer also to "Input Files" in this chapter.

SIMULATE does not execute equations or apply inputs to ABEL-PLD truth tables or state diagrams; it simulates the operation of a device as though it were already programmed with the information contained in the input file. If a programmer load file is used as the input to SIMULATE, part number information must be provided by using the **-n** option. If the output file from FUSEMAP (specified with **-o** at invocation of FUSEMAP) is used for simulation, the part number information is already available and **-n** is not needed.

Furthermore, if a programmer load file (*.jed) is used as input to SIMULATE, the operation of the one device associated with that load file will be simulated. If the intermediate file output (*.out) from FUSEMAP is used as input, the operation of all devices initially specified in the source file will be simulated.

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-filename

Input File Name

Enter the intermediate filename created by REDUCE. The default file extension is .out.

-filename

Output File Name

The output created by SIMULATE is written to the file specified with the **-o** option, or to the standard output device if no output file is given. The contents of this file will vary depending on the trace level set with the **-t** option. See the examples on the following pages for more information on the simulation output.

-device

Device Type

If a programmer load file is used for simulation input rather than FUSEMAP output, no device-specific information is available. (If the programmer load file was generated by ABEL-PLD or GATES, the SIMULATE program can extract the device type from a string in the file header.) **-device** specifies the device associated with the load file, and the device information is read from the appropriate device specification file on the distribution disk. The device must be supported by ABEL-PLD, and is specified by an industry part number following **-n**.

Example:

`simulate -iU09.jed -om6809a.sim -nP14L4`

The above example invokes the simulator with the file named U09.jed as input, m6809a.sim as the output file, and p14l4.dev as the device specification file. If p14l4.dev cannot be found, an error message is issued.

Note: JEDEC files and -n cannot be used to simulate PROMs. The input to SIMULATE must come through FUSEMAP in ABEL-PLD batch processing.

-bp1,p2 [,trace_lvl]

Break Points

It may be useful, particularly in large designs, to view simulation output for only some of the test vectors. **-bp1,p2 [,trace_lvl]** allows this type of selective tracing. A beginning and an ending breakpoint must be specified. A breakpoint is specified with the number of the test vector at which the break is to occur. The new trace level is set at the beginning break point. The trace level is returned to its original value after the ending breakpoint. Only one -b option is allowed.

Note: On some versions of MS-DOS, the use of commas to separate breakpoint values will cause an error. If this occurs, periods may be substituted.

The new trace level can be specified explicitly with *trace_level*. If the new trace level is not specified, the trace level between breakpoints will be one level higher (in number and detail) than before the break occurred.

For example:

```
simulate -im6809a.out -om6809a.sim -B5,8,3
```

This invokes the simulator with the default trace level 0. At the fifth test vector, the trace level is set to 3 and remains there until after vector 8, when the trace level is reset to level 0.

For example:

```
simulate -t1 -im6809a.out -om6809a.sim -b5,8
```

In this case, the breakpoints are again vectors 5 and 8. But the initial trace level is set to level 1 by the -T option and no new trace level is specified with the breakpoints. Thus, the new trace level between vectors 5 and 8 (inclusive) is level 2 (one level higher than 1).

-wpin#,pin#,...

Watch Points

When trace level 3, 4, or 5 is specified, the -w option may be used to specify the device pins to be watched during the simulation process. Each specified pin number is separated by a comma when using the -w option; or a range of pins can be specified by separating the pin numbers with an ellipsis (...).

For example,

```
simulate -t4 -im6809a.out -om6809a.sim -w1..6,14..17
```

The above example starts the simulator with the file named m6809a.out as the input file and m6809a.sim as the output file, with device pins 1 through 6 and 14 through 17 specified for inclusion in the output file. The order the pin numbers are entered on the command line determines the order of the data (by pin number) in the output file. For example, to list pin 1 through 6 in the reverse order, enter

-w6..1,14..17

on the command line. Also, although the device pin numbers are entered on the command line, the listing from the output file indicates the pins by their identifiers named in the pin declarations of the ABEL-PLD source file.

You can also insert a blank column in the trace level 4 and 5 printouts by entering any number greater than the number of pins and nodes in the particular device (such as 999) as a -w option. For example, to insert a blank column between pins 1 and 14, enter

-w6..1,999,14..17

-ttrace_level

Trace Level

-ttrace_level determines the level of information that is provided by SIMULATE. The trace level can be 0, 1, 2, 3, 4, or 5. Errors are listed regardless of the trace level. By choosing the appropriate trace level, you can see only the final outputs for registered devices, or the outputs before and after the clock pulse.

Trace level 0 shows the final output (after the outputs have stabilized) and test vectors for errors only.

Trace level 1 shows final output (not the output for each SIMULATE iteration) and test vectors for the device.

Trace level 2 shows the outputs after each iteration of the simulator and the test vectors. For registered parts, the outputs are shown before and after each clock pulse. For designs with feedback, the outputs are shown for each iteration until the outputs stabilize.

Trace level 3 shows the internal nodes and the device outputs for each iteration plus the test vectors. Use level 3 for the most help with determining where and why simulation errors occur.

Trace level 4 shows the output level that appears on each specified device pin during the simulation process. The output pin voltages are shown as a waveform in the output file that contains a trace for each pin. Each trace represents the logic high and logic low output levels for each test vector. If no input or output pins are specified with the -w option, the first 14 outputs will appear in the output file by default.

Trace level 5 is similar to trace level 4 except that the waveform is replaced by H, L, and Z for logic high, logic low, and high-impedance state.

A detailed discussion of trace levels with examples is given in the chapter "Advanced Features."

**-x(0|1|h|l) and
-z(0|1|h|l)**

**Don't Cares and
High Z**

Don't care and high impedance values encountered in test vectors must be given some value during simulation. The -x(0|1|h|l) and -z(0|1|h|l) options allow you to override the default values. As a default, anytime a ".X." is encountered in a test vector the logical value 0 is substituted for it. As a default, 1 is substituted for a ".Z." value. You can specify default values of 0, 1, L, or H for ".X." or ".Z.". The default values are substituted only when ".X." or ".Z." are inputs to a design or outputs that are fed back as inputs. Outputs that are not fed back are shown in simulation output as they exist in the source file, with ".X." and ".Z." intact.

The simulator checks the design with a single voltage level for the don't care inputs, while the target circuit may place other levels on the input during actual operations. For complete simulation, it is recommended that you run the SIMULATE operation with the don't-cares set to 0 (option -x0), and then again with them set to 1 (option -x1). Refer also to "Don't Cares in Simulation" in the chapter "Using Advanced Features."

-u(0|1|h|l)

Power Up State

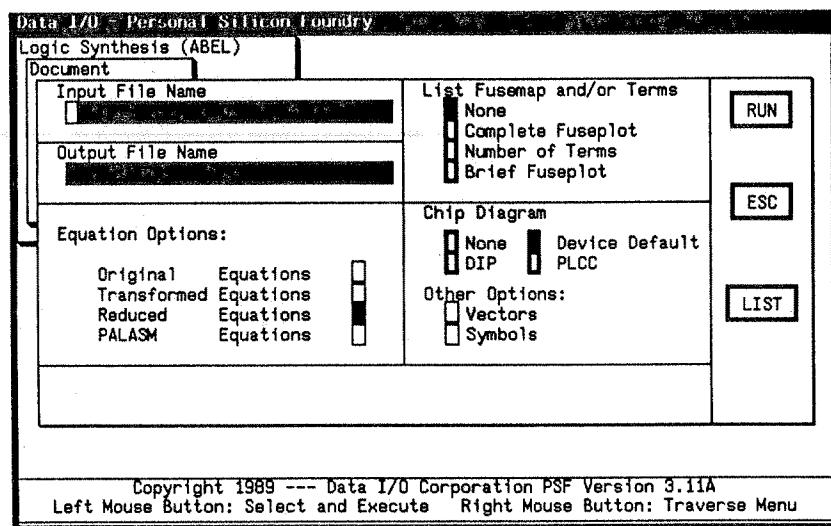
The powerup state of all registers in a device can be set with the -u flag. -u1 or -uh sets all registers to 1, while -u0 or -ul sets all registers to 0. If no -u option is specified, registers are set to the default state specified in the device file.

DOCUMENT

DOCUMENT creates design documentation from information provided by previous steps of the language processor.

DOCUMENT can be executed from the PSF menus or the command line.

Figure 3-9
Document Dialog Box



Command Line Syntax

```
document [-iinfile] [-ooutfile] [-q(0,1,2,3)] [-fn] [-g[n]] [-s]
```

Table 3-10
Document Options

Command Line	DOCUMENT Menu
-ifilename	Input File Nname
-ofilename	Output File Name
-q(0,1,2,3)	Equation Options:
-q0	Original Equations
-q1	Transformed Equations
-q2	Reduced Equations
-q3	PALASM Equations
-f(0 1 2)	Fusemap and/or Terms:
-f0	Complete Fuseplot
-f1	Number of Terms
-f2	Brief Fuseplot
-g[0 1]	Chip Diagram:
N/A	None
-g	Device Default
-g0	DIP
-g1	PLCC
	Other Options:
N/A	Vectors (Not implemented)
-s	Symbols

Remarks

The documentation is written to the file specified by the -o option or to the standard output device if no output file is specified. If the -i and/or -o options are omitted (no file specified), you will be prompted for the input and/or output file names.

Depending on the options chosen, the design documentation may contain the following information for each device in the source file:

Symbol table	constant, pin, node, module and macro identifiers listed alphabetically.
Reduced equations	equations produced by REDUCE.
Transformed equations	equations produced by TRANSFOR.
Equations	original equations from the source file and equations generated by PARSE from truth tables and state diagrams.

Test vectors	test vectors described as inputs and outputs, taken from the PARSE intermediate output file.
Fuse map	graphical representation of the fuse states from the programmer load file.
Chip Diagram	a diagram showing the device pin outs and the identifiers assigned to each pin.

If no options are given when starting DOCUMENT, no documentation listing will be generated.

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-filename

Input File Name

filename is an intermediate output file from FUSEMAP.

-ofilename

Output File Name

Gives the name for the design documentation file.

-q(0,1,2,3)

Equations Option

Select equations to be listed, where any combination of the four numbers can be specified.

0 : list original equations

1 : list transformed equations

2 : list reduced equations (default)

3 : create the output file in PALASM format

The -q option controls which equations, if any, appear in the documentation output file. Up to four numbers (0, 1, 2, and 3) can be specified following -q, or the -q option can be repeated with a different number for each type of equations listing desired. For example, both "-q02" and "-q0 -q2" cause the original and reduced equations to be included as part of the documentation output file.

-f(0|1|2)

**List Fuse Map
and/or Terms**

The -f(0|1|2) option controls the listing of fuse maps and terms:

0 : list the fuse map and the device utilization information

1 : list only the device utilization information

2 : brief fuse map

-f0 lists a complete fuse map, the number of terms used, and the utilization of each term with respect to the outputs (utilization report). -f1 lists only the number of terms used and the utilization report. -f2 or -f lists an abbreviated fuse map, where fuse rows that have all their fuses intact are not shown. -f2 eliminates showing many rows of intact fuses in devices where only a small portion of the product terms are used in the design.

-g(0|1)

Chip Diagram

The -g(0|1) option lists the chip diagram (default), or the diagram listed below:

0 : DIP diagram

1 : PLCC package diagram

-s

Symbols

The symbol table lists constant, pin, node, module and macro identifiers alphabetically.

Sample Output

Figure 3-10 shows a sample output from DOCUMENT. This output was created by processing the source file, m6809a.abl (Figure 3-3) using the options -f, -g, -s, and -q2. The listing contains a symbol table, the reduced equations, a chip diagram, and a fuse map. In the fuse map, intact connections are shown as "X"s, and blown fuses (no connection) are shown as dashes.

Figure 3-10
*Document Output for
m6809a.abl*

Page 1

ABEL-PLD(tm) 3.20 - Document Generator 5-Feb-90 02:54 PM
6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987
Symbol list for Module m6809a

m6809a	Module Name
U09a	device P14L4
A15	Pin 1 pos, com
A14	Pin 2 pos, com
A13	Pin 3 pos, com
A12	Pin 4 pos, com
A11	Pin 5 pos, com
A10	Pin 6 pos, com
ROM1	Pin 14 neg, com
IO	Pin 15 neg, com
ROM2	Pin 16 neg, com
DRAM	Pin 17 neg, com

Page 2

ABEL-PLD(tm) 3.20 - Document Generator 5-Feb-90 02:54 PM
6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987
Equations for Module m6809a

Device U09a

- Reduced Equations for device U09a:

```
!DRAM = (!A13 # !A14 # !A15);  
!IO = (A15 & A14 & A13 & !A12 & !A11);  
!ROM2 = (A15 & A14 & A13 & A12 & !A11);  
!ROM1 = (A15 & A14 & A13 & A12 & A11);
```

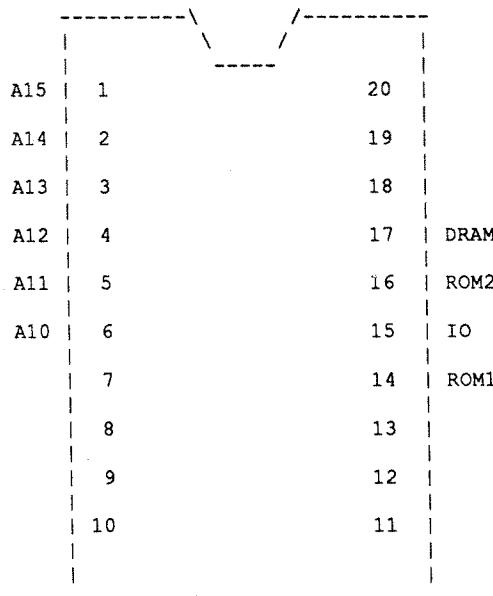
Page 3

ABEL-PLD(tm) 3.20 - Document Generator
 6809 memory decode
 Jean Designer Data I/O Corp Redmond WA 24 Feb 1987
 Chip diagram for Module m6809a

5-Feb-90 02:54 PM

Device U09a

P14L4



Page 4

ABEL-PLD(tm) 3.20 - Document Generator
 6809 memory decode
 Jean Designer Data I/O Corp Redmond WA 24 Feb 1987
 Fuse Map for Module m6809a

5-Feb-90 02:54 PM

Device U09a

	0	10	20
0:	-----X-----	-----	-----
28:	-X-----	-----	-----
56:	---X-----	-----	-----
112:	X-X-X---X-	--X-----	-----
224:	X-X-X---X	--X-----	-----
336:	X-X-X---X-	--X-----	-----

ABEL-PLD(tm) 3.20 - Document Generator
6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987
for Module m6809a

Device U09a

Device Type: P14L4

Terms Used: 6 out of 16

Pin #	Name	Terms				Pin Type
		Used	Max	Term Type		
1	A15	--	--	---	---	Input
2	A14	--	--	---	---	Input
3	A13	--	--	---	---	Input
4	A12	--	--	---	---	Input
5	A11	--	--	---	---	Input
6	A10	--	--	---	---	Input
7		--	--	---	---	Input
8		--	--	---	---	Input
9		--	--	---	---	Input
10		--	--	---	---	GND
11		--	--	---	---	Input
12		--	--	---	---	Input
13		--	--	---	---	Input
14	ROM1	1	4	Normal	---	Output
15	IO	1	4	Normal	---	Output
16	ROM2	1	4	Normal	---	Output
17	DRAM	3	4	Normal	---	Output
18		--	--	---	---	Input
19		--	--	---	---	Input
20		--	--	---	---	VCC

end of module m6809a

Utilities

This section describes the utilities provided in addition to the ABEL-PLD software. These utilities are

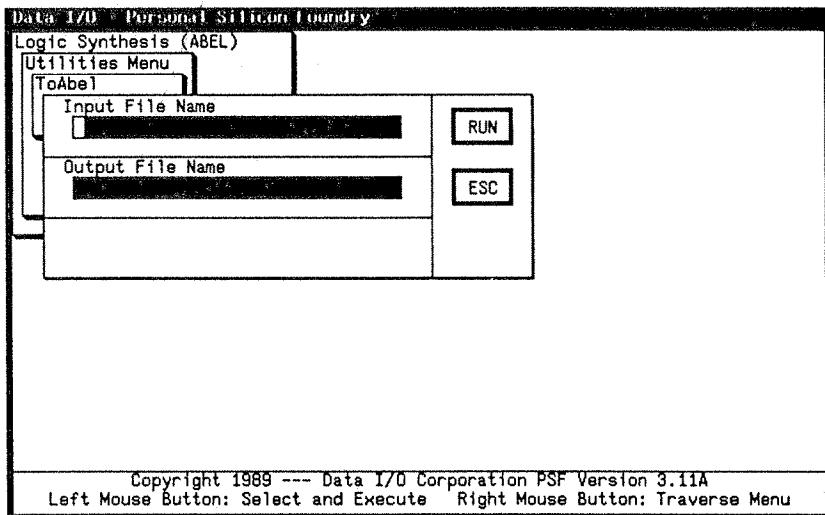
- TOABEL — PALASM to ABEL-PLD converter
- Abellib — Library manager
- Cleanup.bat — Cleans up temporary files generated by ABEL-PLD
- Finddev.bat — Finds a specific device. For more information, see Device Support in the *User Notes*.

TOABEL — PALASM to ABEL Converter

TOABEL converts PALASM (version 1.0 only) logic descriptions to ABEL-PLD logic descriptions. TOABEL creates an ABEL-PLD source file that can be processed normally with the ABEL-PLD language processor. Note that test vectors for devices with programmable inputs and outputs (i.e., P16L8) may require editing because TOABEL cannot determine whether a pin is an input or an output.

TOABEL can be executed from the PSF menus or the command line.

Figure 3-11
TOABEL Menu



Command Line
Syntax

`toabel -iinfile -ooutfile`

Table 3-11
TOABEL Options

Command Line	TOABEL Menu
-filename	Input File Name
-filename	Output File Name

The first line is the command line option, and the second line in boxed text is the PSF menu option. If a menu option is not given, the option can only be accessed from the command line.

-filename

Input File Name

Enter the name of the PALASM input file to be converted.

-filename

Output File Name

Enter the name of the TOABEL output file.

ABELLIB — Library Manager

Abellib is used to maintain the ABEL-PLD library files. The device library (abel3lib.dev) is a single file that contains all the device files for devices currently supported by the ABEL-PLD software package. The "include" library (abel3lib.inc) is a single file that contains text files that may be included in an ABEL-PLD source file via the LIBRARY statement. Abellib is not accessible via the PSF menus.

Syntax

abellib [*library*] [*command*] { *files* }

Remarks

The abellib library manager is useful in cases where disk space is at a premium. This program allows you to extract individual device files and/or edit the libraries so that unused device files or "include" files are eliminated from the libraries.

With abellib, you can add, delete, replace, and extract files from a library as well as list its contents. The command options available are

- a** add files to the library
- d** delete files from the library
- e** extract files from the library
- l** list the contents of the library

When using abellib, you must specify the full path and filename of the library file to be examined and modified, unless the file is in your current directory.

The default library filename is `abel3lib.dev`. If you do not specify a library name on the command line, this file is referenced by `abellib`. If no commands are specified on the command line, you will be prompted for the function desired.

The following is the suggested procedure to extract device specifications from a library into one or more individual device (*.dev) files:

1. Change your current directory to the one that contains the library. For DOS systems with standard installations, use the command

```
cd c:\dataio\lib3
```

2. Create a device file, such as one for the P16R8, with the following command:

```
abellib abel3lib.dev -e p16r8.dev
```

where `abel3lib.dev` is the name of the library and `p16r8.dev` is the name of the new device file.

3. Repeat step 2 as necessary to extract the desired devices from the library. Newly created *.dev files will be contained in the current directory.

To create a device library that contains only those devices you will use,

1. Change your current directory to the one that contains the device files.
2. Create a device library, such as one named `newlib`, with the following command:

```
abellib newlib.dev -a p16r8.dev
```

where `p16r8` is the name of the device file to be placed in the library.

3. Repeat step 2 as necessary to add the desired device files to the library. The newly created library will be contained in the current directory.
4. Since ABEL-PLD looks for "abel3lib.dev" for device specifications, you must rename any existing "abel3lib.dev" library file, then rename your new library "abel3lib.dev." For example, MS-DOS users can use the commands:

```
ren abel3lib.dev oldlib.dev  
ren newlib.dev abel3lib.dev
```

to ensure that ABEL-PLD accesses the newly created library.

Library File Usage

When an external file is required to process your ABEL-PLD design, such as a device or "include" file, ABEL-PLD will first search for the required file, and then if it is not found, will attempt to find the file in a library.

This search will be repeated for each of the following directories, and in the listed order:

1. The current directory
2. The directory indicated in the ABEL3DEV environment variable
3. Directories indicated in the PATH environment variable

Cleanup.bat

The ABEL-PLD programs produce several temporary files in addition to the *.jed, *.sim, *.doc and *.pxx files. To clean up the internally used ABEL-PLD temporary files, enter

```
cleanup
```

This command will delete files with the extensions .tmv, .tm1, .tm2, .tm3, .ttl and .tt2.

To clean up all files produced when running ABEL-PLD software, enter

```
cleanup all
```

Cleanup all will delete all the temporary files, and all the *.doc, *.jed and *.pxx. The *.abl files will not be affected.

*Note: Running an ABEL-PLD source file that contains test vectors creates a temporary file, *.tmv. If you subsequently delete the test vectors from the source file, and rerun the ABEL-PLD software without running cleanup, the *.tmv file from the original run will be read by FUSEMAP, and the test vectors will be added to the JEDEC file.*

Cleanup.bat is not available via the PSF menus.

Downloading the Programmer Load File

Information on downloading programmer load files for your particular system is provided in the chapter "Downloading to Programmers."

Chapter 4

Using Advanced Features

This chapter covers advanced features of ABEL-PLD that assist in designing for more complex devices, debugging designs, creating test vectors, device-independent design, and other tasks. The following is a brief overview of the topics covered:

- Controlling Device Nodes with Node Numbers and Dot Extensions
- Solving Timing Problems with REDUCE
- Passing Arguments from the Command Line
- Advanced Use of SIMULATE

Controlling Device Nodes

When designing for devices with complex macro cells and features, it is often necessary to refer to internal nodes within the device. All such internal nodes have been given numbers specific to each device, and these numbers may be found on the logic diagram for the device.

Using Node Numbers

In the P22V10 device, reset and preset functions appear on the logic diagram as nodes 25 and 26. These nodes may be declared and used in the same way you would declare and use an actual device output.

In Figure 4-1, the declaration statements assign the identifiers "reset" and "preset" to nodes 25 and 26. Once declared, the identifiers can be used in equations to specify the logic for each signal.

The test vectors for this design verify that the device registers will be asynchronously reset when pins 4 and 6 are pulled high and low, respectively, and that the device will be synchronously preset on the next clock pulse when pins 5 and 6 are pulled high and low, respectively.

Figure 4-1
Using Node Numbers for
Reset/Preset Functions

```

module _reset22
  title 'Demonstrates Asynchronous Reset and Synchronous Preset'
  Dave Pellerin    Data I/O Corp'

  reset22 device   'P22V10';

  Clk,I1,I2,R,S,T      Pin 1,2,3,4,5,6;
  Q1,Q2                  Pin 14,15;
  reset,preset           Node 25,26;

  Q2,Q1 istype 'pos';
  Ck,Z,H,L    = .C., .Z., 1, 0;
  Input        = [I2,I1];
  Output       = [Q2,Q1];

  equations
    Output  := Input;      "Registered buffer

    reset   = R & !T;

    preset   = S & !T;

  test_vectors
    ([Clk,Input,R,S,T] -> Output)
    [ Ck, 0 ,0,0,0] -> 0;
    [ Ck, 1 ,0,0,0] -> 1;
    [ Ck, 2 ,0,0,1] -> 2;
    [ 0 , 3 ,0,0,1] -> 2;  "Hold
    [ Ck, 3 ,0,0,1] -> 3;
    [ 0 , 3 ,1,0,1] -> 3;  "Reset = R & !T
    [ 0 , 3 ,1,0,0] -> 0;  "Async Reset
    [ 0 , 0 ,0,1,0] -> 0;  "Preset requires clock
    [ Ck, 0 ,0,1,0] -> 3;  "Sync Preset
  end

```

Using Dot Extension Notation

For most device nodes it is easier to refer to the required node by using dot extension notation. For example, the reset term (node 25) of the P22V10 could be accessed by writing an equation of the form:

$$\text{Q2.RE} = \text{R} \& \text{!T};$$

Similarly, the preset term (node 26) could be referred to with the equation:

$$\text{Q2.PR} = \text{S} \& \text{!T};$$

When dot extension notation is used, nodes do not need to be declared in the declaration section of your ABEL-PLD source file. Figure 4-2 shows the P22V10 design example using dot extension notation for the reset and preset functions. Node numbers are not used in this example to declare the reset and preset nodes. The dot extensions supported in the ABEL-PLD language are listed below, and detailed information about which devices can use these dot extensions is contained in the appendix "Programmable Device Information."

Table 4-1
Dot Extensions

.AP	Asynchronous Preset
.AR	Asynchronous Reset
.C	Clock
.J	J input to JK Flip Flop
.K	K input to JK Flip Flop
.L	Load input for registers
.OE	Output Enable
.PR	Preset
.Q	Q feedback
.R	R input to SR Flip Flop
.RE	Reset
.S	S input to SR Flip Flop

Figure 4-2
*Using Dot Extensions for
 Reset/Preset Functions*

```

module _reset22a
title 'Demonstrates Asynchronous Reset and Synchronous Preset'
Dave Pellerin Data I/O Corp'

reset22a device 'P22V10';

Clk,I1,I2,R,S,T Pin 1,2,3,4,5,6;
Q1,Q2 Pin 14,15;

Q2,Q1 istype 'pos';
Ck,Z,H,L = .C., .Z., 1, 0;
Input      = [I2,I1];
Output     = [Q2,Q1];

equations

    Output := Input;      "Registered buffer

    Q2.RE = R & !T;
    Q2.PR = S & !T;

test vectors
([Clk,Input,R,S,T] -> Output)
[ Ck, 0 ,0,0,0] -> 0;
[ Ck, 1 ,0,0,0] -> 1;
[ Ck, 2 ,0,0,1] -> 2;
[ 0 , 3 ,0,0,1] -> 2; "Hold
[ Ck, 3 ,0,0,1] -> 3;
[ 0 , 3 ,1,0,1] -> 3; "Reset = R & !T
[ 0 , 3 ,1,0,0] -> 0; "Async Reset
[ 0 , 0 ,0,1,0] -> 0; "Preset requires clock
[ Ck, 0 ,0,1,0] -> 3; "Sync Preset

end

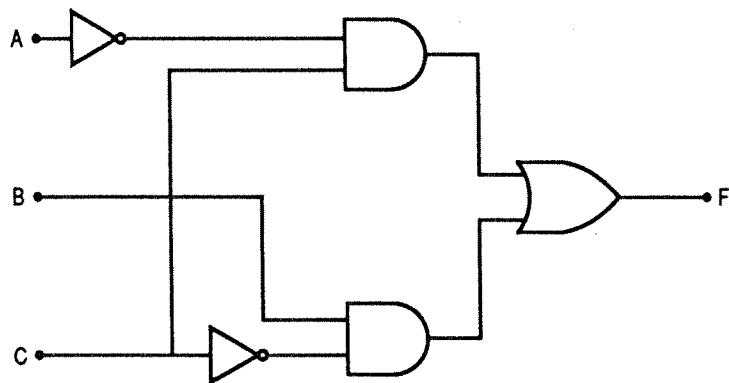
```

Solving Timing Problems with REDUCE

Timing problems sometimes occur in logic circuits where propagation delays vary throughout the design or where one path to an output is longer or shorter than others. Many methods exist to eliminate such hazards, one of them being the introduction of redundancy into the circuit. If you use this technique, you must be aware that the reduction pass of the language processor, when run at reduction level 2 or 3, is designed to eliminate redundancy, whether it was intentional or not. Thus, if you run REDUCE (the reduction pass) at -r2 (reduction level 2) or at -r3 (reduction level 3), you will counteract any attempts at solving timing problems by the introduction of redundancy. Reduction levels 0 and 1 do not eliminate redundant terms and may be used safely with intentionally redundant logic.

Figure 4-3 shows a circuit that has a timing problem because both the complement of the signal C and the uncomplemented signal are used. Notice that the part of the circuit using !C has one additional unit of propagation delay.

Figure 4-3
Circuit Using an Input and Its Complement



Thus the output F defined by the equation:

$$F = B \& \text{!}C \# \text{!}A \& C$$

which uses both C and its complement experiences a glitch as C undergoes a transition from 1 to 0, as shown in the timing diagram in Figure 4-4.

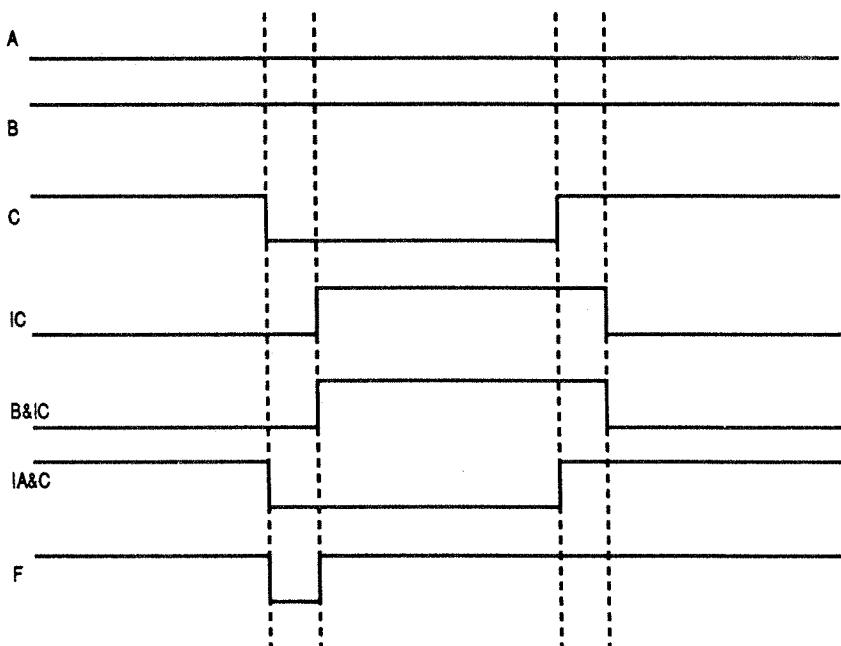
The hazard can be eliminated by the introduction of a redundant term, $\text{!}A \& B$, so that the full equation becomes:

$$F = B \& \text{!}C \# \text{!}A \& C \# \text{!}A \& B$$

This new equation does, in fact, remove the timing problem. But, if the language processor is run with level 2 or level 3 reduction in effect, the redundant term will be eliminated, resulting in the original equation with the hazard present.

Information on the language processor, the reduction pass, and the different reduction levels is presented in "Using ABEL-PLD."

Figure 4-4
Timing Diagram for $F = B \& !C$
 $\# !A \& C$



Passing Arguments from the Command Line

Dummy arguments can be used within a source file so that variable values can be passed from the command line when the ABEL-PLD language processor is invoked. This section describes the use of dummy arguments to provide a variable device type to the memory address decoder presented in the design examples.

Device types must always be chosen explicitly before an ABEL-PLD logic description can be converted to a programmer load file. This requirement arises because the language processor uses device-specific information to convert logic descriptions to programmer load files and to check designs for compatibility with the device. The processor detects design errors such as too many product terms for the chosen device, too many inputs or outputs and other device-dependent restrictions. The device type is specified in the declarations section of the source file with a device declaration.

The situation could exist in which many devices are suited to a particular logic design problem but in which the availability of any certain part is uncertain, or where power requirements dictate the use of one part in one system design and another in a different system, both to provide the same function. Given such a situation, it would be convenient to specify the device type when the source file is processed rather than having to edit the file before each run.

This idea is used in the example logic design, M6809D, shown in Figure 4-5. M6809D allows the user to specify the device type from the language processor command line rather than in the source file itself. Thus, the same source file can be used to create many different programmer load files for different devices. M6809D also provides a default device type so that if none is specified, a P14L4 is used by default.

Dummy arguments and directives are used to create the variable device type, as follows.

The variable device type is implemented in three of the first eleven lines of the source file. These three lines are discussed in detail here; the remainder of the source file is described in the design examples.

Line 1:

```
module M6809D (dev);
```

The first line of the source file is a MODULE statement that names the module M6809D and indicates that a dummy argument, dev, is to be used within the module. When the language processor is invoked, an argument can be passed to the module, as shown here:

```
abel m6809d -aP16L8
```

Figure 4-5
Device Type Passed From
Command Line, Memory
Address Decoder

```

module m6809d (dev)
title '6809 memory decode
Jean Designer Data I/O Corp Redmond WA'

" The device type may be specified on the command line with
" ABEL m6809d -aP16L8

@ifnb (?dev) { U09d device '?dev'; @message 'Using "?dev".'};

@ifb (?dev) { U09d device 'P14L4'; @message 'Using "P14L4".'};

A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
ROM1,IO,ROM2,DRAM      pin 14,15,16,17;

H,L,X = 1,0,.X.;
Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
  !DRAM = (Address <= ^hDFFF);
  !IO   = (Address >= ^hE000) & (Address <= ^hE7FF);
  !ROM2 = (Address >= ^hF000) & (Address <= ^hF7FF);
  !ROM1 = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
  ^h0000 -> [ H , H , H, L ];
  ^h4000 -> [ H , H , H, L ];
  ^h8000 -> [ H , H , H, L ];
  ^hC000 -> [ H , H , H, L ];
  ^hE000 -> [ H , H , L, H ];
  ^hE800 -> [ H , H , H, H ];
  ^hF000 -> [ H , L , H, H ];
  ^hF800 -> [ L , H , H, H ];

end m6809d

```

This command line invokes the language processor to process the file named m6809d.abl. The argument P16L8 will be substituted for dev wherever dev is preceded by a question mark in the source file. (A P16H8 or F153 can also be used.) The question mark is a required marker that must precede all dummy arguments where values are to be substituted in their place.

Line 8:

```

@ifnb (?dev) { U09 device '?dev';
@message 'Using "?dev".'};

```

Line 8 uses the "IF NOT BLANK" directive, @IFNB, to insert the appropriate device declaration into the source file. If the value passed for dev is not blank, the text enclosed by the left and right braces is inserted into the source with the value of dev substituted. The text within the braces is a standard device declaration followed by the @MESSAGE directive. @MESSAGE causes the text within single quotes to be written to the terminal. In this case, the message indicates what device is being used.

For example, if the argument P16L8 were passed from the command line, as shown above, line 8 would cause,

U09 device 'P16L8';

to be inserted into the source for processing.

Line 10 functions similarly to line 8 but covers the opposite situation: If no device type is passed from the command line, the device type of P14L4 is used. This is because ?dev would be blank, and the text enclosed by braces would be inserted into the source.

Line 10:

```
@ifb (?dev) { U09 device 'P14L4';
@message "Using \"P14L4\"."};
```

Lines 1, 8, and 10 allow the user to pass a device type to the source file when it is processed. If no device type is specified, a default value is used. Arguments can be passed to any source file in a similar manner. The dummy argument must be specified in the MODULE statement, the argument is preceded by a question mark in the module wherever substitution is desired, and the argument value is passed to the source when the language processor is invoked.

Simulation

This section explains some of the finer points of using the simulation program and gives some suggestions concerning its use. Each subsection covers a separate topic. The following topics are discussed:

- Test vectors and simulation
- Debugging state machines
- Multiple test vector tables
- Using macros and directives to create test vectors
- Don't cares in simulation
- Preset, reset and preload registers
- Asynchronous circuits
- Trace levels and breakpoints

Test Vectors and Simulation

SIMULATE simulates the operation of a programmed device, but the simulation output is only as good as the input you provide with your test vectors. With test vectors, you specify the required outputs (at declared device pins) for given input combinations (at declared device pins). SIMULATE applies the inputs specified in the test vectors to your design and compares the required outputs with the actual outputs; if there is a difference between the required and actual outputs, an error is indicated. However, if you do not specify outputs for some combination of inputs, that combination of inputs is not applied to the design, and therefore is not tested. It is to your advantage to create complete sets of test vectors that test all functions of your logic design, and to use SIMULATE regularly as design changes are made.

Note: The nodes shown on ABEL-PLD logic diagrams are outputs for writing equations; for example, the OR terms for the RS flip/flops in an F167. These nodes cannot be used as direct flip/flop inputs for simulation test vectors.

Debugging State Machines

State machines can be difficult to debug once an error occurs because each state is dependent on previous states and points to next state. A simple error in the description of one state transition can cause an incorrect output that cascades through the state machine causing so many errors during simulation that the original error is difficult to isolate. Therefore, when you debug state machines of any size, you should periodically force (with your test vectors) the machine to a known state and then let the state transitions take place. In this manner, you limit the cascading of errors to a smaller number of states and make it much easier to find initial errors.

It is also helpful with large state machines to start with a small set of test vectors that tests only part of the state machine's function. When operation of that function has been verified, add a set of test vectors that test another function, then add another, and so on, until you have tested the full function of the state machine. Combining this technique of gradual simulation with the forcing vectors discussed above greatly simplifies the testing of large state machines.

In summary, to test and debug larger state machines:

- Write small sets of test vectors that test individual functions of the state machine, and gradually add them to the simulation.
- Add test vectors that periodically force the machine to known states to eliminate cascading of errors.

Multiple Test Vector Tables

More than one set of test vectors can be used to simulate the function of a device. This may be useful in the representation of the test in the source file. Take for example, the source file presented in Figure 4-6 that describes AND and NAND gates implemented on the same device as well as the test vectors used to simulate the operation of that device. The test vectors are described in two separate sections. The first test vectors section lists the test vectors that simulate the operation of the AND portion of the design, and the second section tests the NAND function. With the test vectors written as they are, in two separate sections, the correspondence between test vectors and the function being tested is readily apparent in the source file.

In a similar manner, any time you have two or more distinct functions being performed by the same device, you may want to describe the vectors in separate sections for each function. You are not, however, required to do so.

Figure 4-6
Source File With Multiple
Test Vectors Sections

```

module simple
  title 'Simple ABEL-PLD example'
  Dan Poole  Data I/O Corp'

  U7      device  'P14H4';
  A1,A2,A3      pin 1,2,3;
  N1,N2,N3      pin 4,5,6;
  AND,NAND      pin 14,15;

  equations
    AND      = A1 & A2 & A3;
    !NAND    = N1 & N2 & N3;

  test_vectors  'Test And Gate'
  ( [A1,A2,A3] - AND )
    [ 0, 0, 0] - 0;
    [ 1, 0, 0] - 0;
    [ 0, 1, 0] - 0;
    [ 0, 0, 1] - 0;
    [ 1, 1, 1] - 1;

  test_vectors  'Test Nand Gate'
  ( [N1,N2,N3] - NAND )
    [ 0, 0, 0] - 1;
    [ 1, 0, 0] - 1;
    [ 0, 1, 0] - 1;
    [ 0, 0, 1] - 1;
    [ 1, 1, 1] - 0;
end simple

```

Using Macros and Directives to Create Test Vectors

Macros and directives can be used to write test vectors in a concise form that is often helpful in large designs that require many test vectors to fully test the device operation. In this section, two examples of using macros and directives to create test vectors are shown for a simple design. The advantages gained through writing test vectors in this way are even greater for more complicated designs.

Figure 4-7 shows a modified version of the memory address decoder presented in the design examples. In this version, a macro and the @IRP directive are used to create the test vectors. The macro Between is defined after the constant and set declarations and before the equations section. The Between macro uses the @IF directive to produce the character "L" if an address is in a given range and the character "H" if the address is not in that range.

The macro has three dummy arguments, a, b, and c, whose values are supplied when the macro is invoked. Argument a represents the address that either falls or does not fall in the range between arguments b and c. The body of the macro is specified in the block defined by the outermost left and right braces. Within the block, the @IF directive is used to check the value supplied for a against the range values supplied for b and c. The @IF directive itself contains blocks that contain the "L" or "H" character. If the condition in parentheses following the @IF directive is true, the block following the condition is inserted into the text. Thus, there are two @IF directives: the first produces an "L" if the address is in the range; the second produces an "H" if the address is out of the range.

Figure 4-7
Test Vectors Described With a Macro and @IF and @IRP Directives

```

module m6809b
  title '6809 memory decode
  Jean Designer Data I/O Corp Redmond WA'

  U09b device 'P14L4';
    A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
    ROM1,IO,ROM2,DRAM      pin 14,15,16,17;

    H,L,X = 1,0,.X.;
    Address = [A15,A14,A13,A12,A11,A10,X,X,X,X,X,X,X,X,X,X];

    " This macro will return an 'L' if the address (a) is between the
    " two limits (b,c), otherwise it returns an 'H'.
    between macro (a,b,c)
      (@if ((?a>=?b)&(?a<=?c)){L}@if!((?a>=?b)&(?a<=?c)){H});

    equations
      !DRAM = (Address <= ^hDFFF);
      !IO    = (Address >= ^hE000) & (Address <= ^hE7FF);
      !ROM2  = (Address >= ^hF000) & (Address <= ^hF7FF);
      !ROM1  = (Address >= ^hF800);

    test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
      @IRP addrs
        (^h0000,^h8000,^hE100,^hE800,^hF100,^hFC00) {
          ?addrs -> [ between (?addrs,^h0000,^hDFFF),      " DRAM
                        between (?addrs,^hE000,^hE7FF),      " IO
                        between (?addrs,^hF000,^hF7FF),      " ROM2
                        between (?addrs,^hF800,^hFFFF)];      " ROM1
        }
      end m6809b

```

Once the Between macro has been defined, it can be used in the test vectors section to help build the test vectors. The @IRP directive invokes Between to insert the "L" or "H" needed to properly define the vectors. @IRP causes the block following it (that text enclosed by braces), to be repeated once for each value contained in the parentheses in the @IRP statement. Each time the block is repeated, one of those values is successively substituted for the dummy argument addrs declared in the @IRP statement. This means that the block will be repeated six times, and addrs will take on six different values each time between is invoked in the block. addrs is substituted for a in the macro and the values for b and c are given explicitly. The resulting source is shown in the expanded listing produced by the PARSE step of the language processor. Figure 4-8, shows how the directive and macro were expanded to produce the test vectors. The listing was produced by PARSE with the -e parameter.

In the second source file, shown in Figure 4-9, the @CONST and @REPEAT directives are used in conjunction with the between macro to create the test vectors for the same design. The definition of between is identical to that used in the previous example. @REPEAT 6 causes the block containing the vectors to be repeated 6 times. Within that block, @CONST is used to increment the value of addrs before the next repetition of the block. Figure 4-10 shows the expanded output produced by the PARSE step of the language processor invoked with the -e parameter. Notice that in the test vectors shown in the listing, the constant addrs appears instead of the actual numeric value that the constant represents. The correct value is retained internally, and these test vectors are functionally the same as those shown in Figure 4-8

Figure 4-8
PARSE Output (partial)
Showing Expanded Vectors
for a Macro and @IF/@IRP
Directives

```

0001e|module m6809b
0002e|title '6809 memory decode
0003e|Jean Designer Data I/O Corp Redmond WA'
0004e|
0005e|U09b device 'P14L4';
0006e| A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007e| ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008e|
0009e| H,L,X = 1,0,.X.;;
0010e| Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e|
0012e|"This macro will return an 'L' if the address (a) is between
0013e|"the two limits (b,c), otherwise it returns an 'H'.
0014e|between macro (a,b,c)
0015e| {@if ((?a>=?b) & (?a<=?c)) {L}@if!((?a>=?b) & (?a<=?c)) {H}};
0016e|
0017e|equations
0018e| !DRAM = (Address <= ^hDFFF);
0019e|
0020e| !IO    = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e|
0022e| !ROM2  = (Address >= ^hF000) & (Address <= ^hF7FF);
0023e|
0024e| !ROM1  = (Address => ^hF800);
0025e|
0026e|test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
0027e|
0028e|
0029e| ^h0000 -> [ L,      "DRAM
0030e|           H,      "IO
0031e|           H,      "ROM2
0032e|           H];     "ROM1
0033e|
0034e| ^h8000 -> [ L,      "DRAM
0035e|           H,      "IO
0036e|           H,      "ROM2
0037e|           H];     "ROM1
0038e|
0039e| ^hE100 -> [ H,      "DRAM
0040e|           L,      "IO
0041e|           H,      "ROM2
0042e|           H];     "ROM1
0043e|
0044e| ^hE800 -> [ H,      "DRAM
:
:
:
```

Figure 4-9

Test Vectors Described With a Macro, @CONST and @REPEAT Directives

```

module m6809c
  title '6809 memory decode
  Jean Designer    Data I/O Corp Redmond WA'
  U09c    device 'P14L4';
  A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
  ROM1,IO,ROM2,DRAM      pin 14,15,16,17;

  H,L,X    = 1,0,.X.;
  Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];

  " This macro will return an 'L' if the address (a) is between the
  " two limits (b,c), otherwise it returns an 'H'.
  between macro (a,b,c)
    {@if ((?a>=?b)&(?a<=?c)){L}@if!((?a>=?b)&(?a<=?c)){H}};

  equations
    !DRAM   = (Address <= ^hDFFF);
    !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);
    !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
    !ROM1   = (Address >= ^hF800);

  test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
    @CONST addrs = ^hD000;
    @REPEAT 6 {
      addrs -> [ between (addrs,^h0000,^hDFFF),      " DRAM
                  between (addrs,^hE000,^hE7FF),      " IO
                  between (addrs,^hF000,^hF7FF),      " ROM2
                  between (addrs,^hF800,^hFFFF)];      " ROM1
      @CONST addrs = addrs + ^h800;
    }
  end m6809c

```

Figure 4-10
**PARSE Output (partial) with
 Expanded Output for a Macro
 and @CONST/@REPEAT**

```

0001e|module m6809c
0002e|title '6809 memory decode
0003e|Jean Designer      Data I/O Corp Redmond WA'
0004e|
0005e|U09c device 'P14L4';
0006e|      A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007e|      ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008e|
0009e|      H,L,X = 1,0,.X.;;
0010e|      Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e|
0012e|" This macro will return an 'L' if the address (a) is between the
0013e|" two limits (b,c), otherwise it returns an 'H'.
0014e|between macro (a,b,c)
0015e|      (@if ((?a>=?b) & (?a<=?c)) {L}@if!((?a>=?b) & (?a<=?c)) {H});
0016e|
0017e|equations
0018e|      !DRAM = (Address <= ^hDFFF);
0019e|
0020e|      !IO    = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e|
0022e|      !ROM2 = (Address >= ^hF000) & (Address <= ^hF7FF);
0023e|
0024e|      !ROM1 = (Address >= ^hF800);
0025e|
0026e|test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
0027e|
0028e|
0029e|      addrs -> [ L,      "DRAM
0030e|                  H,      "IO
0031e|                  H,      "ROM2
0032e|                  H];     "ROM1
0033e|
0034e|      addrs -> [ L,      "DRAM
0035e|                  H,      "IO
0036e|                  H,      "ROM2
0037e|                  H];     "ROM1
0038e|
0039e|      addrs -> [ H,      "DRAM
0040e|                  L,      "IO
:
:
:
```

Don't Cares in Simulation

In ABEL-PLD you can use the special constant .X. in a test vector to denote a don't care input or output. The .X. tells the ABEL-PLD simulator to choose a value for the input designated by the .X. in the test vector(s). The default value used in the simulator for the don't care inputs is zero; however, the don't care flag (-xn) can be used in the SIMULATE command line to specify zero or one (0 or 1) for the don't care value. (Refer also to the SIMULATE section of "Using ABEL-PLD."

Input pins that are not specified in the test vectors are given the default don't care value zero (0) by the simulator unless the don't care flag is set to -x1. In this case, all unspecified pins will be assigned a value of 1 in the test vectors.

If you experience trouble with devices not working in a circuit or programmer/tester, it may be helpful to recheck the don't care assumptions. There may be a combination of 1's and 0's in a test vector that needs to be checked by the ABEL-PLD simulator.

Note: *The Data I/O LogicPak, Model 60, and Unisite programmers do not test the device in the same manner as the ABEL-PLD simulator. If the LogicPak displays ERROR 75 you may determine the failed vector and pins by using the programmer/LogicPak in the terminal mode as described in the LogicPak manual.*

Also, the simulator checks the design with a single level for the don't care inputs, while the target circuit may place other levels on the input during actual operation of the device. For complete simulation, you must run the SIMULATE operation with the don't cares set to 0 (flag -x0), and then again with them set to 1 (flag -x1).

Output pins that are not specified in the test vectors are disregarded by the simulator and no error will be indicated due to conflict between a specified value and the value determined by the simulator. The .X. constant at an output pin tells the simulator not to compare the outputs (the output produced by the design and the output specified in the test vector) but still allow them to be displayed. Figure 4-11 shows how the .X. value can be assigned to the outputs prior to the SIMULATE step of the language processor.

Figure 4-11
Assignment of Don't Care Value (.x.) to Design Outputs

```
module findout flag '-t1'
title 'The ABEL simulator will find the output levels
Ngoc Nicholas Data I/O'

F1      device 'P16L8';

A,B,Y1,Y2      pin 1,2,14,15;

X = .X.;

equations
  !Y1 = A # B;
  !Y2 = A $ B;

test_vectors
  ([A,B] -> [Y1,Y2])
  [0,0] -> [ X, X];
  [0,1] -> [ X, X];
  [1,0] -> [ X, X];
  [1,1] -> [ X, X];
end
```

Using trace level 1, 2, or 3, you can observe the actual output values determined by the simulator. In Figure 4-12, the X entries (in the test vectors) for pins 14 and 15 allow the simulator to display an H or L to indicate the output value for the specified inputs. (All unused inputs are set to the default condition, 0.) The N entries indicate outputs that are not to be tested.

Figure 4-12
SIMULATE Results with
Outputs Specified as Don't
Care

```
Simulate ABEL-PLD(tm) 3.20
The ABEL simulator will find the output levels
Ngoc Nicholas Data I/O

File:'findout.out' Module:'findout'
Device:'F1' Part:'P16L8'

Vector 1
Vector In [00.....]
Device Out [.....ZZHHZZZZ.]
Vector Out [.....XX....]

Vector 2
Vector In [01.....]
Device Out [.....ZZLLZZZZ.]
Vector Out [.....XX....]

Vector 3
Vector In [10.....]
Device Out [.....ZZLLZZZZ.]
Vector Out [.....XX....]

Vector 4
Vector In [11.....]
Device Out [.....ZZLHZZZZ.]
Vector Out [.....XX....]

4 out of 4 vectors passed.
```

Preset and Preload Registers

Preset, reset, and preload are terms used to define a specific action and resultant output of one or more registers contained in a programmable logic device. Preset forces all register outputs to one, reset forces all register outputs to zero, and preload forces all registers to specified states. Synchronous preset, reset, and preload functions require a clock input. Asynchronous functions require no clock input.

To verify the operation of these devices, appropriate test vectors must be written and placed in the source file. These test vectors allow the SIMULATE step of the language processor to verify operation of the design by performing the required operations of these registers.

Note: ABEL-PLD assumes that devices have inversion between the register outputs and the device outputs. When preloading devices that have noninverting outputs or that have outputs programmable to noninverting, the data to be preloaded must be complemented to obtain the desired preload condition.

Also note that it is not possible to preset and preload at the same time. Preset and preload must not contend during preload with other inputs, preset, or register functions.

Special Preset Considerations

Certain programmable logic devices, such as F105 and F167, do not respond to the first clock pulse following a preset condition (invoked by power-on or the preset input). These devices allow normal clocking only after a high-to-low transition of the clock input following the preset condition. This means that simulation for these devices requires an additional test vector following the preset condition just to provide the high-to-low transition of the clock input, thereby allowing normal clocking to take place without the loss of a clock pulse.

To illustrate the preset considerations for these devices, a four-state counter with clock and preset inputs is presented in Figure 4-13, along with the test vectors required to properly verify the design. The equation for the preset condition is written using the dot extension for the two registers (see "Controlling Device Nodes" earlier in this chapter). This counter is targeted for a circuit that provides a power-on preset condition; so the test vectors must verify operation of the counter after power-on preset as well as after the preset input has been active.

Figure 4-13
Test Vectors for Special Preset Conditions

```

module _preset
flag '-KY' "leave unused OR terms connected
title '2-bit counter to demonstrate power on preset
Michael Holley Data I/O Corp'

    preset device 'F167';

    Clk,Hold      pin 1,2;
    PR            pin 16; "Preset/Enable
    P1,P0         pin 15,14;

    Ck      = .C.;

equations

    [P1.PR,P0.PR]      = PR "preset

    [P1.R,P0 ]       := !P1 & !P0 & !Hold;   " state 0
    [P1 ,P0.R]       := !P1 &  P0 & !Hold;   " state 1
    [P1 ,P0 ]       :=  P1 & !P0 & !Hold;   " state 2
    [P1.R,P0.R]      :=  P1 &  P0 & !Hold;   " state 3

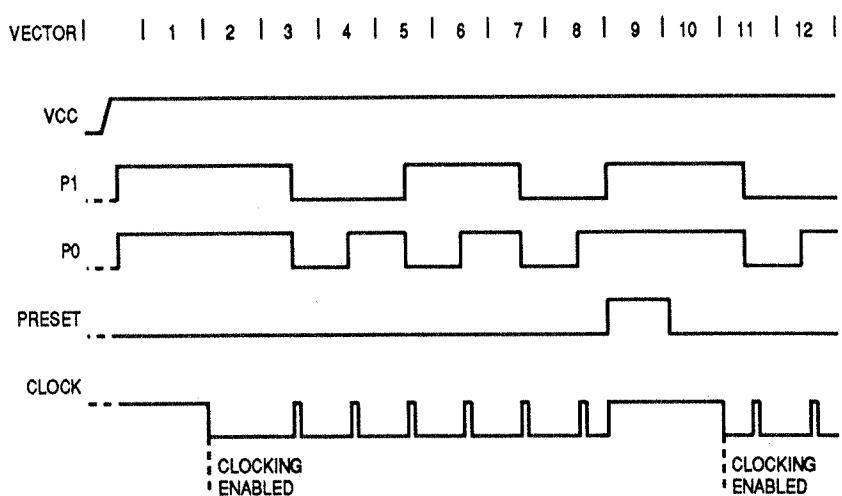
test_vectors
    ([Clk,PR,Hold] -> [P1,P0])
        [ 1 , 0, 0 ] -> 3; " Provides a High-to-Low on clock
        [ 0 , 0, 0 ] -> 3; " to enable clocking
        [ Ck, 0, 0 ] -> 0;
        [ Ck, 0, 0 ] -> 1;
        [ Ck, 0, 0 ] -> 2; " Hold count
        [ Ck, 0, 1 ] -> 2;
        [ Ck, 0, 0 ] -> 3;
        [ Ck, 0, 0 ] -> 0; " Roll over
        [ Ck, 0, 0 ] -> 1;
        [ 1 , 1, 0 ] -> 3; " Preset high
        [ 1 , 0, 0 ] -> 3; " Preset low
        [ Ck, 0, 0 ] -> 0;
        [ Ck, 0, 0 ] -> 1;

    " Notes on preset from the Signetics Data Sheet
    "
    "The PR input provides an asynchronous preset to logic '1' of all
    "State and Output Register bits. Preset overrides the Clock, and
    "when held High, clocking is inhibited and all outputs are High.
    "Normal clocking resumes with the first clock pulse following
    "a High-to-Low clock transition after PR goes Low.
    "
    "The power on preset also inhibits clocking until a High-to-Low
    "clock transition. This is provided by the first 2 test vectors.
end

```

Figure 4-14 is a timing diagram that shows the action of the test vectors in Figure 4-13. As indicated in the timing diagram, the preset input overrides the clock input and when held high, inhibits clocking of the counter. Assuming that the device is powered up in the preset condition, the first test vector pulls the clock input high while the second vector pulls it low to provide the high-to-low transition on the clock line, required for normal clocking.

Figure 4-14
Timing Diagram Showing Test Vector Action



Preset overrides clock, and when held high, clocking is inhibited and the registers are high.
Normal clocking resumes with the first clock pulse following a high-to-low clock transition after preset goes low.

The next six test vectors provide clock inputs to increment the counter through all states and back to state one. As shown in the timing diagram, the 9th test vector invokes the preset function while the 10th test vector pulls the preset input low and maintains the clock input high. The 10th test vector allows the preset line to go low before the high-to-low transition of the clock. The preset line must go low before the clock so that the high-to-low clock transition can enable the clock pulse of the 11th test vector. The high-to-low transition that follows the 10th test vector resumes normal clocking of the device.

If the 2nd and 10th test vectors are not included in the source file, the clock pulse of the 3rd and 11th vectors would be lost. That is, the high-to-low transition of the clock pulses in these vectors would cause the resumption of normal clocking, but do not increment the counter as required by the design.

Supervoltage Preload

Supervoltage preload allows the setting of registers within certain devices, such as the P16R4, to the logic levels placed on their registered outputs. Supervoltage preload is accomplished by means of the P test condition (.P. special constant) that is used to "jam load" registers within the logic device to the desired state. When the P test condition is applied to the clock pin, the logic level applied to the register output is loaded into the register. Devices with separate banks of registers require that the P test condition be applied to each clock pin. Also during preload, certain device pins, such as the output enable pin, may have to be in a defined state.

To verify the preload operation, use a separate test vector to test the outputs. This vector must follow the vectors that perform the preload operation.

Supervoltage preload can be used to test state machine designs that could assume one or more illegal states, or designs that contain branch conditions. An illegal state for a state machine is a state that the design does not allow, but the device is capable of assuming under certain conditions (such as powerup or noise). A typical decade counter, having states 0 through 9 and made up of four registers, could possibly assume any of six additional (and illegal) states (10 through 15). The decade counter should be designed so that when an illegal state is achieved, the next clock pulse returns the counter to the 0 state. During simulation it is necessary to not only test the counter for normal up/down/clear operation (performed by the test vectors) but also to insure that it will clock to state S0 from any illegal state.

To test that a decade counter will clock to state 0 from any illegal state, it is necessary to do two things:

1. Define all illegal states to be tested.
2. Create test vectors that verify the return to state 0 from any illegal state.

To test the illegal states for a decade counter, it is necessary to define the illegal states (i.e., 10 through 15). The example in Figure 4-15 shows that the following additional entries are included to define the six illegal states.

```
S10= ^b0101;  
S11= ^b0100;  
S12= ^b0011;  
S13= ^b0010;  
S14= ^b0001;  
S15= ^b0000;
```

Figure 4-15
The Illegal States Defined

```
module cnt10p
flag '-r3'
title 'decimal counter
Note: preload the data on pins into the registers
Denny Siu    Data I/O Corp'

U10p      device 'P16R4';

Clk,Clr,OE pin 1,2,11;
Q3,Q2,Q1,Q0 pin 14,15,16,17;

Ck,X,Z,P = .C. , .X., .Z., .P.;

" Counter States
S0 = ^b1111;   S4 = ^b1011;   S8 = ^b0111;   S12= ^b0011;
S1 = ^b1110;   S5 = ^b1010;   S9 = ^b0110;   S13= ^b0010;
S2 = ^b1101;   S6 = ^b1001;   S10= ^b0101;   S14= ^b0001;
S3 = ^b1100;   S7 = ^b1000;   S11= ^b0100;   S15= ^b0000;
```

To verify that the design will recover from each of the illegal states, appropriate test vectors are included as shown in Figure 4-16. This group of test vectors preloads the device to each possible illegal state and then verifies that the device clock to state S0. The test vectors preload the counter by means of the .P. special constant applied to the clock pin and a logic high applied to the output enable (OE) pin. The test vector that follows the preload test vector verifies the result of the preload operation, while the following test vector verifies the clocking of the counter from the illegal state to state S0.

Figure 4-16
Test Vectors for Illegal States

```

module cnt10p
flag '-r3'
title 'decimal counter
Note: preload the data on pins into the registers
Denny Siu FutureNet Division, Data I/O Corp'

U10p      device 'P16R4';

Clk,Clr,OE pin 1,2,11;
Q3,Q2,Q1,Q0 pin 14,15,16,17;

Ck,X,Z,P = .C. , .X., .Z., .P.;

" Counter States
S0 = ^b1111;   S4 = ^b1011;   S8 = ^b0111;   S12= ^b0011;
S1 = ^b1110;   S5 = ^b1010;   S9 = ^b0110;   S13= ^b0010;
S2 = ^b1101;   S6 = ^b1001;   S10= ^b0101;   S14= ^b0001;
S3 = ^b1100;   S7 = ^b1000;   S11= ^b0100;   S15= ^b0000;

state_diagram [Q3,Q2,Q1,Q0]
State S0: IF !Clr THEN S1 ELSE S0;

State S1: IF !Clr THEN S2 ELSE S0;

State S2: IF !Clr THEN S3 ELSE S0;

State S3: IF !Clr THEN S4 ELSE S0;

State S4: IF !Clr THEN S5 ELSE S0;

State S5: IF !Clr THEN S6 ELSE S0;

State S6: IF !Clr THEN S7 ELSE S0;

State S7: IF !Clr THEN S8 ELSE S0;

State S8: IF !Clr THEN S9 ELSE S0;

State S9: GOTO S0;

test_vectors
([Clk ,OE, Clr ] -> [Q3,Q2,Q1,Q0])
[ Ck , 0, 1 ] -> S0;
[ Ck , 0, 0 ] -> S1;
[ Ck , 0, 0 ] -> S2;
[ Ck , 0, 0 ] -> S3;
[ Ck , 0, 0 ] -> S4;
[ Ck , 0, 0 ] -> S5;
[ Ck , 1, 0 ] -> Z ;
[ Ck , 0, 0 ] -> S7;
[ 0 , 0, 0 ] -> S7;
[ Ck , 0, 0 ] -> S8;
[ Ck , 0, 0 ] -> S9;
[ Ck , 0, 0 ] -> S0;
[ Ck , 0, 0 ] -> S1;
[ Ck , 0, 0 ] -> S2;
[ Ck , 0, 1 ] -> S0;

```

```

test_vectors 'preload to illegal states'
( [Clk ,OE, Clr,[Q3,Q2,Q1,Q0]] -> [Q3,Q2,Q1,Q0])
[ P , 1, 0 , S10 ] -> X;
[ 0 , 0, 0 , X ] -> S10;
[ Ck , 0, 0 , X ] -> S0;
[ P , 1, 0 , S11 ] -> X;
[ 0 , 0, 0 , X ] -> S11;
[ Ck , 0, 0 , X ] -> S0;
[ P , 1, 0 , S12 ] -> X;
[ 0 , 0, 0 , X ] -> S12;
[ Ck , 0, 0 , X ] -> S0;
[ P , 1, 0 , S13 ] -> X;
[ 0 , 0, 0 , X ] -> S13;
[ Ck , 0, 0 , X ] -> S0;
[ P , 1, 0 , S14 ] -> X;
[ 0 , 0, 0 , X ] -> S14;
[ Ck , 0, 0 , X ] -> S0;
[ P , 1, 0 , S15 ] -> X;
[ 0 , 0, 0 , X ] -> S15;
[ Ck , 0, 0 , X ] -> S0;
end cnt10p

```

An example of a state machine that contains branch conditions is given in the blackjack machine in the chapter "Design Examples." If it was not possible to preload the state machine to each branch condition, it would be necessary to repeat the test vectors down to the Test22 state for each branch of the Test22 state. The test vectors in Figure 4-17 show how this state machine can be preloaded to test these three branches of the design.

Figure 4-17
Using Test Vectors to Preload
A State Machine

```

test_vectors ' Test 3 way branch at Test22'
([Ena,Clk,LT22,Ace,Qstate] -> [Ace,Qstate ])
[ 1 ,..P., X 1 ,Test22] -> [ X,      X      ];
[ 0 , 0 , 1 X ,   X  ] -> [ H, Test22 ]; "Verify preload
[ 0 , C , 1 X ,   X  ] -> [ H, ShowStand];

[ 1 ,..P., X 0 ,Test22] -> [ X,      X      ];
[ 0 , 0 , 0 X ,   X  ] -> [ L, Test22 ];
[ 0 , C , 0 X ,   X  ] -> [ L, ShowStand];

[ 1 ,..P., X 1 ,Test22] -> [ X,      X      ];
[ 0 , C , 0 X ,   X  ] -> [ H, sUB10 ];

```

**Preset/Reset
Controlled by
Product Term**

In programmable logic devices such as the P22V10, preset and reset functions can be controlled by product terms placed in the "equations" portion of the ABEL-PLD source file. (Note that in devices that have programmable polarity at the output of the registers, preset and reset functions may complement the output pins.) An example of controlling the preset and reset functions is given in Figure 4-18. In this listing, two nodes within the P22V10 are declared as the reset and preset functions. (Nodes are internal signal lines defined by Data I/O to aid in the programming of parts with internal signals. Refer to "Controlling Device Nodes" earlier in this chapter for additional details on nodes.) The declared nodes are nodes 25 and 26. Appendix C lists nodes 25 and 26 of the P22V10 as usable for the reset and preset functions.

Figure 4-18
*Controlling Reset/Preset by
Product Term*

```
module _reset22
  title 'Demonstrates Asynchronous Reset and Synchronous Preset'
  Dave Pellerin  Data I/O Corp'

  reset22 device  'P22V10';

  Clk,I1,I2,R,S,T Pin 1,2,3,4,5,6;
  Q1,Q2             Pin 14,15;
  reset,preset      Node 25,26;

  Q2,Q1           istype 'pos';
  Ck,Z,H,L        = .C., .Z., 1, 0;
  Input           = [I2,I1];
  Output          = [Q2,Q1];

  equations
    Output := Input;           "Registered buffer

    reset     = R & !T;
    preset   = S & !T;

  test_vectors
    ([Clk,Input,R,S,T] -> Output)
    [ Ck, 0 ,0,0,0] -> 0;
    [ Ck, 1 ,0,0,0] -> 1;
    [ Ck, 2 ,0,0,1] -> 2;
    [ 0 , 3 ,0,0,1] -> 2;           "Hold
    [ Ck, 3 ,0,0,1] -> 3;
    [ 0 , 3 ,1,0,1] -> 3;           "Reset = R & !T
    [ 0 , 3 ,1,0,0] -> 0;           "Async Reset
    [ 0 , 0 ,0,1,0] -> 0;           "Preset requires clock
    [ Ck, 0 ,0,1,0] -> 3;           "Sync Preset
  end
```

In the P22V10, the reset is asynchronous while preset is synchronous (to the clock input). The test vectors in Figure 4-18 verify the reset and preset functions. The first three vectors verify the loading of input data. (Note that the third vector pulls the T input high; this is in preparation for T to be pulled low later in the simulation.) The fourth vector verifies operation of the clock input by changing the input data without providing a clock input. The sixth vector verifies that the R input without a low T input will not provide the asynchronous reset.

The seventh vector verifies operation of the asynchronous reset by pulling the T input low. The next vector verifies that a high S input and low T input do not preset the device without the clock input. The final vector verifies the synchronous preset by providing the clock pulse and testing the output for both output pins at logic high (decimal 3).

Preset/Reset Controlled by Pin

Some devices have a direct Preset coming from a pin. Be sure to include this in the test vectors or simulation errors may occur.

Powerup States

Some devices power up with registers set to 1, some set to 0 and some set to an unknown value. For example, some TI devices power up with registers set and outputs low while some AMD devices power up with registers clear and outputs high. The first test vector should place the device in a known state. The ABEL-PLD simulator assumes D, JK, and T registers power up to 0 and RS registers power up to 1.

Asynchronous Circuits

In some asynchronous circuits, the ABEL-PLD simulator may not maintain the output state and will report an error. In these cases, the device outputs should be specified on both sides of the test vector. Figure 4-19 shows a common cross-coupled flip-flop, an asynchronous circuit that can cause erroneous error reporting by the simulator.

The problem with this type of asynchronous circuit is that it can powerup in an undetermined state. That is, the output can be either high or low, depending on the electrical characteristics of the device. In order for the simulator to verify the functionality of the circuit, it must be made to assume some beginning state for the outputs. Test vectors are used to define the output states so that the simulator can test the design.

Figure 4-20 shows two sets of test vectors that could be used to test the design of a cross-coupled flip-flop such as that shown in Figure 4-19. The first set of test vectors defines the inputs only, and tests the outputs for the desired results. Because of the circuit's asynchronous nature, the simulator may not be able to maintain the output state and may report an error.

The second set of test vectors specifies the outputs on both sides of each vector. If the outputs are not defined in this manner, the simulator may report an error due to the conflict between outputs tied back to the inputs. The second set of test vectors defines the outputs for the simulator and prevents any error indications due to the asynchronous nature of the circuit.

Figure 4-19
A Cross-Coupled Flip-Flop

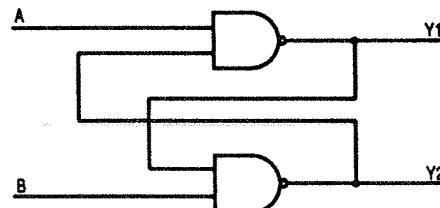


Figure 4-20
Using the Input Side of the
Test Vectors to Define
Outputs

```

module _flipflop
  title 'Simulation of an asynchronous flip flop'
  Lloyd Hyden      Data I/O Corp'

  flipflop          device 'P16L8';
  A,B,Y1,Y2        pin 1,2,13,14;

  equations
    Y1      = !(A & Y2);      "Cross coupled flip flop
    Y2      = !(B & Y1);

  test_vectors
    ([A,B] -> [Y1,Y2])
    [1,0] -> [ 0, 1];
    [1,1] -> [ 0, 1];
    [0,1] -> [ 1, 0];
    [1,1] -> [ 1, 0];

  test_vectors
    ([A,B,Y1,Y2] -> [Y1,Y2])
    [1,0, 0, 1] -> [ 0, 1];
    [1,1, 0, 1] -> [ 0, 1];
    [0,1, 1, 0] -> [ 1, 0];
    [1,1, 1, 0] -> [ 1, 0];
end

```

Devices with Clock Inputs

Since devices with registered outputs must be clocked before the outputs reflect any change in inputs, a clock pulse must be specified as one of the inputs in the test vectors for such devices. A clock input is indicated by a C in the test vector for a low-high-low pulse, and a K for a high-low-high pulse. The clock input in the test vector causes SIMULATE to evaluate the inputs to the outputs prior to the first clock pulse transition (low-to-high or high-to-low depending on the polarity of the clock signal). The evaluation consists of the iterative steps described in "Simulation Program Operation." The inputs to outputs are then evaluated with the clock input at its active state, and then again with the clock input at its inactive state.

When running SIMULATE with trace levels 2 or 3, simulation data will be written to the *.sim file for all three evaluations. That is, internal test vectors are generated to evaluate the design before the first clock transition, after the first clock transition, and after the second clock transition, thus effectively expanding the number of test vectors. An example of SIMULATE output for a device with a clock input, and using a single test vector, is shown in Figure 4-21. This output was generated by the command

```
simulate regfb -b1,1,2 -w14
```

which invokes SIMULATE with test vector 1, trace level 2, and output pin 14 of the source file regfb abl.

Figure 4-21
Clock Inputs Shown in Trace Level 2 Output

```
simulate ABEL-PLD(tm) 3.20
File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'
Operation of the simulator on devices with feedback
DATA I/O Corp. 24 Feb 1990

Vector 1
Vector In [C0111.....0.....]
Device In [001110000000111110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [001110000000111110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [101110000000111110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [001110000000111110001111]
Device Out [.....ZHHHHHZZ.....]
Vector Out [.....HH.....]

4 out of 4 vectors passed.
```

In Figure 4-21, the clock input is represented by the C on the Vector In line. On the four subsequent Device In lines, the C input goes from 0 to 1 and back to 0 to provide one complete clock pulse.

Note: Note that the first two Device In/Device Out lines of the listing are identical. This is a result of SIMULATE evaluating the outputs against the inputs twice to insure that the first evaluation did not alter functions internal to the device, and thus affect the outputs. See "Simulate Program Operation."

Trace Levels and Breakpoints

Through careful use of trace levels and breakpoints, you can easily control the amount of information that SIMULATE provides and make the analysis of a faulty design much easier. SIMULATE provides everything from simple error messages indicating that the actual outputs differ from the outputs you predicted in your test vectors to detailed information about the states of internal registers and gates of a device during simulation. It is suggested that you first simulate a design at the lowest trace level (level 0) to determine whether any errors exist. Once you find errors, use the higher trace levels (2, 3, 4, or 5) successively to increase the amount of information provided until you have enough information to solve the problem. Examples of the output produced by the different trace levels are given below.

With small designs, you can rerun Simulate with different trace levels to obtain the level of information you desire or need. With a larger or more complex design with many test vectors, however, this may result in so much information that you have difficulty finding that which is pertinent to the error. This is when breakpoints become helpful. Assume that you have run a simulation and detected an error in the twentieth test vector out of fifty vectors. You want to see more information to determine the cause of the error. If you simply rerun Simulate with a higher trace level, more detailed information will be collected for every one of the fifty test vectors, when in fact you only need detailed information for vector 20. If you run Simulate with breakpoints set before and after test vector 20, the higher level of information will be provided only for that vector, thus reducing significantly the amount of information collected.

Detailed information on the different trace levels and on using breakpoints is provided below. In summary:

- Simulate designs at trace level 0 to determine the existence of errors.
- Once an error is found, increase the trace level until you have enough information to correct the error.
- Use breakpoints to limit simulation data collection to the vectors of concern.

Simulate Output File, Trace Level 0

Figure 4-22 shows the output of Simulate created during processing of m6809a.abl (Figure 4-3), with pointers to the various parts of the output. For the purposes of this description, one of the test vectors was changed to produce an error, and simulation was run at trace level 0 (only errors are shown). Figure 4-23 shows the test vectors used to create the simulation error, in which the fourth vector has been changed. If the original test vectors shown in Figure 4-3 were used, no simulation errors would occur.

Figure 4-22
Trace Level 0 Output for
m6809er.abl

```
simulate ABEL-PLD(tm) 3.20
File:'m6809er.out' Module:'m6809er' Device:'U09er' Part:'P14L4'
6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1984

Vector 4
ROM1 14, 'H' found 'L' expected

7 out of 8 vectors passed.
```

The simulation output for trace level 0 lists the number of the vector that failed, the name and number of the failed output, and the nature of the failure. In this example, vector 4 failed at pin 14 (ROM1) which produced an active-high output instead of an active-low as expected by the test vector.

Figure 4-23
Test Vectors Used to Create
Simulation Error

```
test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
^h0000 -> [ H , H , H, L ];
^h4000 -> [ H , H , H, L ];
^h8000 -> [ H , H , H, L ];
^hC000 -> [ L , H , H, L ];
^hE000 -> [ H , H , L, H ];
^hE800 -> [ H , H , H, H ];
^hF000 -> [ H , L , H, H ];
^hF800 -> [ L , H , H, H ];
```

SIMULATE Output File, Trace Level 1

Figure 4-24
Level 1 Simulation Output,
All Vectors

Figure 4-24 shows a trace level 1 simulation output for the same source file (including the incorrect vector) that produced Figure 4-22.

```

Simulate ABEL-PLD(tm) 3.20
File:'m6809er.out' Module:'m6809er' Device:'U09er' Part:'P14L4'

6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1984

Vector 1
Vector In [000000.....]
Device Out [.....HHHL...]
Vector Out [.....HHHL...]

Vector 2
Vector In [010000.....]
Device Out [.....HHHL...]
Vector Out [.....HHHL...]

Vector 3
Vector In [100000.....]
Device Out [.....HHHL...]
Vector Out [.....HHHL...]

Vector 4
Vector In [110000.....]
Device Out [.....HHHL...]
Vector Out [.....LHHL...]

Vector 4
ROM1 14, 'H' found 'L' expected

Vector 5
Vector In [111000.....]
Device Out [.....HLHH...]
Vector Out [.....HLHH...]

Vector 6
Vector In [111010.....]
Device Out [.....HHHH...]
Vector Out [.....HHHH...]

Vector 7
Vector In [111100.....]
Device Out [.....HHLH...]
Vector Out [.....HHLH...]

Vector 8
Vector In [111110.....]
Device Out [.....LHHH...]
Vector Out [.....LHHH...]

7 out of 8 vectors passed.

```

The trace level 1 output shows the actual signal outputs and the test vectors used to perform the simulation. The actual output associated with each test vector is shown on one line followed by the input portion of the test vector, Vector In, on the next line. The output portion of the test vector; i.e., the expected output of the device appears on the Vector Out line below the actual output.

SIMULATE Output File, Trace Level 2

Trace level 2 provides information similar to that described for level 1, except that level 2 output shows the device inputs and outputs for each iteration of the simulator. Figure 4-25 shows the level 2 output for the same source file used to generate Figure 4-24. The command used to produce Figure 4-25 is simulate add5 -b5,5,2 -w19. Note that the output is expanded over that shown in Figure 4-24 to show each iteration of the simulate operation that takes place to stabilize the device output at pin 19.

Figure 4-25
*Level 2 Simulation Output,
Single Vector*

```
Simulate ABEL-PLD(tm) 3.20
File:'decade.out' Module:'decade' Device:'TE10' Part:'F105'
Decade Counter - Uses Complement Array and Default Node Names
Michael Holley Data I/O Corp 12 Oct 1989
Vector 5
Vector In [1.....0.....1.....]
Device In [1000000001111011110000000000000001111100000010000000]
Device Out [. .... HHHH.HHHH..... LLLLLLHHHHHHHLLL.....]
Device In [100000000111101111000000000000001111100000000000000]
Device Out [. .... HHHH.HHHH..... LLLLLLHHHHHHHLLL.....]
Device In [10000000011110111100000000000000111111111000000000000]
Device Out [. .... HHHH.HHHH..... LLLLLLHHHHHHHLLL.....]
Device In [10000000011110111100000000000000111111111000000000000]
Device Out [. .... HHHH.HHHH..... LLLLLLHHHHHHHLLL.....]
Device In [10000000011110111100000000000000111111111000000000000]
Device Out [. .... HHHH.HHHH..... LLLLLLHHHHHHHLLL.....]
Vector Out [. .... H..... HHHH.....]
```

20 out of 20 vectors passed.

SIMULATE Output File, Trace Level 3

Trace level 3 simulation provides all trace level 2 information, plus internal device information such as OR-gate outputs, register outputs, and the final outputs. Figure 4-26 shows one portion of a trace level 3 listing with pointers to its various parts and a portion of the corresponding logic diagram. Only a portion of the trace level 3 simulation output is shown since level 3 output files can be quite large.

Figure 4-26
Trace Level 3 Simulation
Output

```

PT input[0110 1010 1010 1010 0110 0110 0110 0110 0101 0101 0101]
PT 5368 [T]-----|-----|
                               |-----|
                               Q1 Pin 14 | \-----|
                               | | >---- L Vec=L
                               | | /-----|
                               |-----|
                               | Q = L | --
                               ]---| OR = H |
                               ]---| CK = L |
                               ]---| AR = H |
                               ]---| SP = L |
                               -----|-----|
PT 5412 [TFFFFFFF
Pin 1 [0
Node 25 [1
Node 26 [0
                               ]-----|
                               Q2 Pin 15 | \-----|
                               | | >---- L Vec=L
                               | | /-----|
                               |-----|
                               | Q = L | --
                               ]---| OR = H |
                               ]---| CK = L |
                               ]---| AR = H |
                               ]---| SP = L |
                               -----|-----|
PT 4884 [T-----|
PT 4928 [TFFFFFFFFF
Pin 1 [0
Node 25 [1
Node 26 [0

```

Fuse and node numbers shown on the table are numbers assigned by Data I/O to the fuses in the device and are shown in the Logic Diagrams provided with the ABEL-PLD package. The OR-gate and register outputs shown in the simulation output are internal signals not available as pin outputs that can be very useful for debugging designs.

The large size of trace level 3 simulation files is due not only to the complete listing of all device internals, but also due to expanded test vectors caused by clock inputs (see "Devices with Clock Inputs" earlier in this chapter), and the numerous iterations of the simulator required to stabilize the outputs of some designs (see "Simulation and Designs with Buffered Outputs"). Trace level 3 may be used with a breakpoint to specify a limited number of test vectors, and the -w option to limit the number of outputs included in the simulation output file.

In Figure 4-26, the product term line shows the state of 45 product terms. This is because an F105 is used in this example, and the input array has 45 possible terms. (Refer to the F105 logic diagram in the Logic Diagram Package provided with ABEL-PLD for details.) The product terms line is followed by the states of various nodes, input pins, and product terms that exist for indicated states on the product terms line.

Table 4-2 defines the notation used in the simulation output files to identify product terms and nodes.

Table 4-2
*Notation Used in Simulation
Output Files*

Notation	Description
Current Nodes	
OE	Output enable
AR	Asynchronous Reset
SR	Synchronous Reset
AP	Asynchronous Preset
SP	Synchronous Preset
LD	Register Load
CK	Register Clock
OR	Normal output OR gate
IN1	First input to a Flip/Flop ("J")
IN2	Second input to a Flip/Flop ("K")
OR Node Types	
PTnnnn	One or more product term
LOW	Always logic level 0
HIGH	Always logic level 1
Pin nn	Input from pin
Node nn	Input or feedback from a internal node
Pin nn & nn	The AND of two pins
Pin nn # nn	The OR of two pins
PROM nn	Bit nn of a prom output
PRODUCT Term Display	
Pin nn [1]	Logic level 1 from a pin or node
Pin nn [0]	Logic level 0 from a pin or node
nn & nn [0 & 1]	Logic level 0 from a pin or node
PTnnnnn [TFFFFIT]	Multiple product terms
PTnnnnn [TT \$ FT]	XOR of two groups of product terms
PTnnnnn [TT # FT]	OR of two groups of product terms
PTnnnnn [T-FF-T]	Shared product terms ('-' term not connected)
PTnnnnn [TTTTTFTTTTFFFFFTTF] [TFFFFFFFTTF]	Multiple line display of large OR

Trace level 3 will produce very large files if all pins and nodes are traced for all vectors. To get a reasonably sized file, use the **-w** and **-b** options to specify the pins or nodes for only the desired vectors. If a **-w** option is not specified, the first I/O pin in the device will be traced.

SIMULATE Output File, Trace Level 4

Trace level 4 provides a waveform representation of the inputs and/or outputs of the device for each of the specified test vectors. The **-b** flag (breakpoints) can be used to specify vectors to be used and the **-w** flag can be used to specify which inputs and outputs are to appear in the output file. Up to 14 pins can be specified; any blank columns inserted with 999 in the **-w** option count as a pin in the output. If no **-w** is used, SIMULATE automatically generates the signals appearing at the first 14 output pins.

Figure 4-27 shows the waveform generated by running the same source file used in Figures 4-22 and 4-24 (address decoder, m6809a.abl). The command

```
simulate m6809a -t4 -w1..6,999,17
```

was used to generate a trace level 4 waveform of pins 1 through 6, and 17 of the address decoder. Each of the specified pins is shown with the logic high and low levels for each of the eight test vectors.

Figure 4-27
Trace Level 4 Simulation Output

```
Simulate ABEL-PLD(tm) 3.20
File:'m6809a.out' Module:'m6809a' Device:'U09a' Part:'P14L4'

6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987

          A   A   A   A   A   A   D
          1   1   1   1   1   1   R
          5   4   3   2   1   0   A
                                M

V0001 |   |   |   |   |   |   |
V0002 |   ~|   |   |   |   |   |
V0003 ~|   | ~|   |   |   |   |
V0004 |   ~|   |   |   |   |   |
V0005 |   |   | ~|   |   |   ~|
V0006 |   |   |   |   | ~|   |
V0007 |   |   |   | ~|   | ~|   |
V0008 |   |   |   |   | ~|   |   |

8 out of 8 vectors passed.
```

SIMULATE Output File, Trace Level 5

Trace level 5 is similar to level 4 except that instead of a pictorial waveform representation of the specified pins, the signal levels are represented by H, L, and Z for logic high, logic low, and high-impedance state. Since the output produced by trace level 5 is more compact than that of trace level 4, it allows specification of more device pins than trace level 4.

Figure 4-28 shows the level 5 output for the same m6809a.abl shown in Figure 4-27. The command used to generate this report is

```
simulate m6809a -t5 -w1..6,999,17
```

Note that more pins can be shown with level 5 than can with level 4. Also, pins that do not have declared signal names are named by their pin numbers and a "P" prefix, such as for pins 6 through 9 and 16 through 19.

Figure 4-28
Trace Level 5 Simulation Output

```
Simulate ABEL-PLD(tm) 3.20
File:'m6809a.out' Module:'m6809a' Device:'U09a' Part:'P14L4'
6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987

          D
A A A A A A   R
1 1 1 1 1 1   A
5 4 3 2 1 0   M

V0001  0 0 0 0 0 0   L
V0002  0 1 0 0 0 0   L
V0003  1 0 0 0 0 0   L
V0004  1 1 0 0 0 0   L
V0005  1 1 1 0 0 0   H
V0006  1 1 1 0 1 0   H
V0007  1 1 1 1 0 0   H
V0008  1 1 1 1 1 0   H
8 out of 8 vectors passed.
```

SIMULATE Output File, Two Trace Levels

As shown in the preceding examples, you can use **-b** (breakpoints) to change trace levels at specific test vectors. In many of the preceding examples, the default trace level 0 was used to a certain test vector, such as in trace level 4, where trace level 5 was invoked at test vector 14. Figure 4-29 shows the result of changing from level 4 to level 5 at vector 5. The command used to generate this listing is

```
simulate m6809a -t4 -b5,8,5 -w1..6,999,14..17.
```

Figure 4-29
Trace Levels 4 and 5 in the
Same Output File

```

Simulate ABEL-PLD(tm) 3.20
File:'m6809a.out' Module:'m6809a' Device:'U09a' Part:'P14L4'

6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987

          A   A   A   A   A   A   R   R   D
          1   1   1   1   1   1   O   O   R
          5   4   3   2   1   0   M   I   M   A
                                1   O   2   M   M

V0001 |   |   |   |   |   |   ~|   ~|   ~|   |
V0002 |   ~|   |   |   |   |   |   |   |   |   |
V0003 ~|   |~|   |   |   |   |   |   |   |   |
V0004 |   ~|   |   |   |   |   |   |   |   |   |

          R   R   D
          A   A   A   A   A   O   O   R
          1   1   1   1   1   1   M   I   M   A
          5   4   3   2   1   0   1   O   2   M

V0005 1 1 1 0 0 0 H L H H
V0006 1 1 1 0 1 0 H H H H
V0007 1 1 1 1 0 0 H H L H
V0008 1 1 1 1 1 0 L H H H
8 out of 8 vectors passed.

```

Simulation and Designs With Buffered Outputs

When a design with 3-state buffered outputs is simulated with trace levels 4 and/or 5, the states of the outputs are reported as H, L, 1, 0, Z, or X, depending on the test vectors used, whether or not the pin is bidirectional, and whether the output buffer is enabled or not.

With Simulation trace level 5, device pins that are output-only, or are bidirectional and configured as outputs, the output will be reported as follows (in order of significance):

- if the buffer is enabled, the active state (H or L) of the output (that results from the levels applied at the input pins by the input test vector) is reported.
or
- if the buffer is not enabled, the same value (1, 0, Z, or X) applied to that output by the input test vector is reported.
or
- if the output is not enabled and no 1 or 0 is applied to that output by the input test vector, z is reported.

Simulation and Unspecified Inputs

When the input test vector does not specify a logic level to be applied to a particular input, or set of inputs, simulation uses the default value assigned by the `-x` parameter. Using don't cares (X's) in the input test vector can cause the input(s) to be unspecified.

Simulation for Designs With Feedback

Logic designs containing feedback present a unique simulation problem because the current output on one or more gates in the design depends on the outputs of other gates. Thus, determining the outputs of a design with feedback is not a simple input-to-output determination. Propagation delays, the number of gates in the feedback path, and, in synchronous feedback circuits, clock inputs must be taken into account. When an input to the design changes, the outputs may not assume their new state (stabilize) immediately. Synchronous circuits must be clocked before the outputs reflect changes in the inputs.

SIMULATE determines the final outputs of feedback circuits through iteration, calculating and monitoring the outputs until they stabilize or are clocked out to give the final outputs. (If outputs do not stabilize after 20 iterations, an error message is given.) The iterations, final outputs and the states of the internal register are provided in the simulation output file depending on the trace level you choose to simulate under. Figure 4-30 shows a simple synchronous circuit with feedback. One clock pulse is required after the inputs change to cause a corresponding change in the outputs. The source file describing this circuit and the simulation output for trace levels 1 and 2 are shown in Figures 4-31 through 4-33.

*Figure 4-30
Synchronous Feedback Circuit*

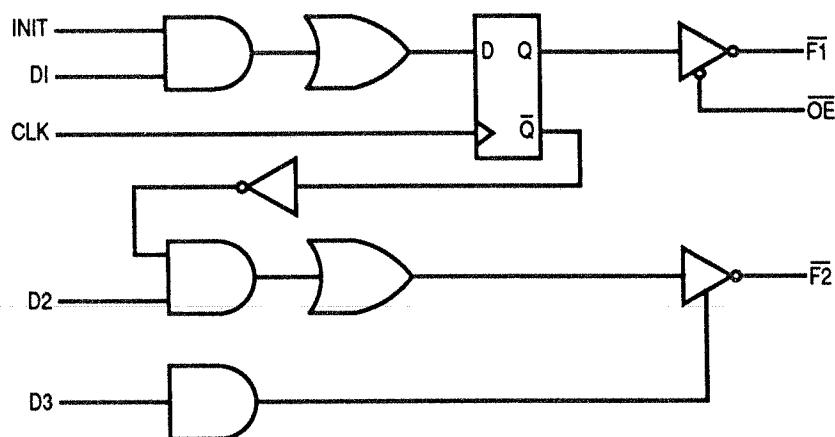


Figure 4-31
Source File: Synchronous
Feedback Circuit

```
module regfb
  title 'Operation of the simulator on devices with feedback
          DATA I/O Corp.'
  FB2 device 'P16R4';

  Clk,OE      pin 1,11;
  INIT,D1,D2,D3 pin 2,3,4,5,;
  F1,F2      pin 14,13;

  equations
    !F1 := D1 & INIT;
    !F2 = D2 & !F1;
    F2.OE = D3;

  test_vectors ([Clk,OE,INIT,D1,D2,D3] - [ F1, F2])
    [.C., 0, 0 , 1, 1, 1] - [ 1, 1 ];
    [.C., 1, 0 , 0, 0, 0] - [.Z.,.Z.];
    [.C., 1, 1 , 1, 1, 1] - [.Z., 0 ];
    [ 0 , 0 , 0, 0, 0] - [ 0 ,.Z.];
end regfb
```

Trace level 1 output shows the test vectors and the final outputs after the clock pulse. Trace level 2 shows the test vectors and the value of the outputs before and after the clock. Trace level 3 results in a large simulation output file. If you wish to examine the trace level output for this circuit, you can run ABEL-PLD on "regfb.abl" with trace level 3 and examine the "regfb.sim" file.

Figure 4-32
Simulation Output, Trace Level
1: Synchronous Feedback Circuit

```

Simulate ABEL-PLD(tm) 3.20
File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'
Operation of the simulator on devices with feedback
DATA I/O Corp.

Vector 1
Vector In [C0111.....0.....]
Device Out [.....ZHHHHHZZ.....]
Vector Out [.....HH.....]

Vector 2
Vector In [C0000.....1.....]
Device Out [.....ZZZZZZZZ.....]
Vector Out [.....ZZ.....]

Vector 3
Vector In [C1111.....1.....]
Device Out [.....ZLZZZZZZ.....]
Vector Out [.....LZ.....]

Vector 4
Vector In [00000.....0.....]
Device Out [.....ZZLHHHZZ.....]
Vector Out [.....ZL.....]

4 out of 4 vectors passed.

```

As a second feedback example, Figure 4-34 shows an asynchronous circuit that requires more than one simulation iteration before the outputs stabilize. Figure 4-35 shows the source file describing the circuit and Figures 4-36 and 4-37 show the simulation output for trace levels 1 through 2. Trace level 0 output is not shown since there are no simulation errors in this design and level zero only reports errors.

Trace level 1 shows the final outputs after they have stabilized, and also the test vectors. Trace level 2 shows the output values at the different iterations as the outputs stabilize, as well as the final outputs and the test vectors. Notice that for the inputs provided in vector 2, three iterations are needed before the outputs stabilize. Vector 1 requires only one iteration to provide stable outputs. Trace level 3 output is not shown but can be generated by running ABEL-PLD with "feedback.abl" to generate the "feedback.sim" file shown in Figure 4-35.

Figure 4-33
*Simulation Output (partial),
 Trace Level 2: Synchronous
 Feedback Circuit*

```

Simulate ABEL-PLD(tm) 3.20
File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'
Operation of the simulator on devices with feedback
DATA I/O Corp.

Vector 1
Vector In [C0111.....0.....]
Device In [00111000000011110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [00111000000011110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [101110000000111110001111]
Device Out [.....ZHHHHHZZ.....]

Device In [001110000000111110001111]
Device Out [.....ZHHHHHZZ.....]
Vector Out [.....HH.....]
.
.
.
Vector 4
Vector In [00000.....0.....]
Device In [000000000000001110000111]
Device Out [.....ZZLHHHZZ.....]

Device In [000000000000001110000111]
Device Out [.....ZZLHHHZZ.....]
Vector Out [.....ZL.....]

4 out of 4 vectors passed.

```

Figure 4-34
Asynchronous Feedback Circuit

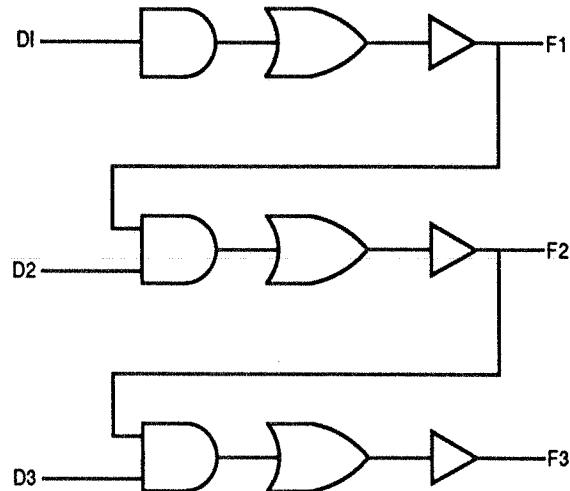


Figure 4-35
Source File: Asynchronous Feedback Circuit

```

module feedback
title
'Operation of the simulator on devices with feedback
FB1    device 'P16HD8';

D1,D2,D3 pin 1,2,3;
F1,F2,F3 pin 13,14,15;

equations

F1 = D1;
F2 = D2 & F1;
F3 = D3 & F2;

test vectors
([D1,D2,D3] -> [F1,F2,F3])
[ 0, 0, 0 ] -> [ 0, 0, 0 ];
[ 1, 1, 1 ] -> [ 1, 1, 1 ];
end feedback

```

Figure 4-36
Simulation Output, Trace Level 1: Asynchronous Feedback Circuit

```

Simulate ABEL-PLD(tm) 3.20
File:'feedback.out' Module:'feedback' Device:'FB1' Part:'P16HD8'
Operation of the simulator on devices with feedback
DATA I/O Corp. 24 Feb 1983

Vector 1
Vector In [000.....]
Device Out [.....LLLLLLL.]
Vector Out [.....LLL....]

Vector 2
Vector In [111.....]
Device Out [.....LHHHLLL.]
Vector Out [.....HHH....]

2 out of 2 vectors passed.

```

Figure 4-37
*Simulation Output, Trace Level
2: Asynchronous Feedback
Circuit*

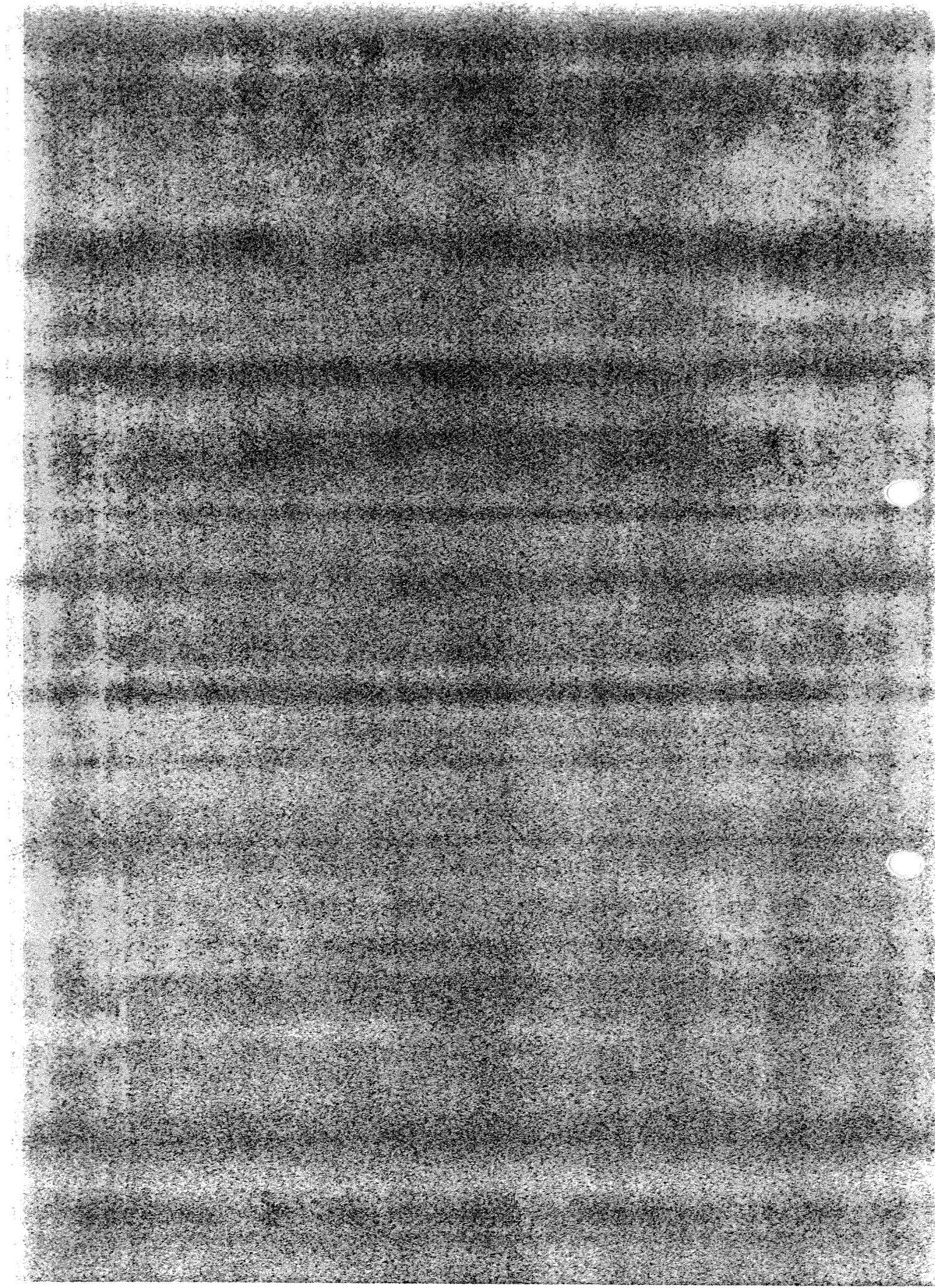
```
Simulate ABEL-PLD(tm) 3.20
File:'feedback.out' Module:'feedback' Device:'FB1' Part:'P16HD8'
Operation of the simulator on devices with feedback
DATA I/O Corp. 24 Feb 1983

Vector 1
Vector In [000.....]
Device In [000000000000000000000000]
Device Out [.....LLLLLLL.]
Device In [000000000000000000000000]
Device Out [.....LLLLLLL.]
Vector Out [.....LLL....]

Vector 2
Vector In [111.....]
Device In [1110000000010000000]
Device Out [.....LHLLLLLL.]
Device In [1110000000011000000]
Device Out [.....LHHLLLLL.]
Device In [1110000000011100000]
Device Out [.....LHHHLLLL.]
Device In [1110000000011100000]
Device Out [.....LHHHLLLL.]
Vector Out [.....HHH....]

2 out of 2 vectors passed.
```

Language Reference



Chapter 5

Source File

The language processor uses logic descriptions contained in a source file to convert logic descriptions to programmer load files. Before you can create a load file that contains the fusemap of a particular logic design, you must create a source file that reflects that logic design. The source file is an ASCII text file written according to the requirements of the ABEL-PLD language processor. These requirements are described in detail in the following chapters.

Elements of the Source File

You can write a source file using any word processor or text editor that produces ASCII files. The elements of the source file are shown in the template in Figure 5-1. Figure 5-1 shows the basic element of the source file, the module. A source may contain any number of modules, each of which contain a different logic description.

Figure 5-1 shows that the source file module (or complete source file in this case) is made up of several elements. These elements are illustrated in Figure 5-2, a source file for an address decoder that is to be placed in a P14L4.

Figure 5-1
Source File Template

```

module ....;
flag '.....';
title
'.....'

"device declaration
"      name   DEVICE  'device
type';
"      ...   device  '....';

"pin and node declarations
"      names          PIN pin #s;
"      names          NODE node #s;
      .....          pin ....;
      .....          pin ....,...,...;
      .....          node ....;

"constant declarations
H,L,X = 1,0,.X. ;
.... = ....;
..   = ..;

equations
"      name = expression ;
      ... = ... ;
enable ... = ....;
... := ....;

test_vectors ([.....] -> ...)
      "inputs           outputs
[...] -> [..., ..., ..., ..., ..., ..., ...];
[...] -> [..., ..., ..., ..., ..., ..., ...];
[...] -> [..., ..., ..., ..., ..., ..., ...];

end ....;

```

The elements of the source file are listed here. Syntax and command line options are detailed in the chapter "Language Structure."

- MODULE STATEMENT — This names the module and also indicates whether any dummy arguments are used.
- FLAG — This can be used to control processing of the source file by the language processor, and is optional.
- TITLE — The title is optional and is used to describe the module. The text entered will be used as a header for the output files.
- DEVICE DECLARATIONS — The device declaration associates one or more device identifiers with specific programmable logic devices.
- PIN DECLARATIONS — This defines the pins of the programmable logic device with identifiers used in the source file.
- NODE DECLARATIONS — If nodes exist on the programmable logic device, they are defined here.
- CONSTANT DECLARATIONS details.
- MACRO DEFINITIONS

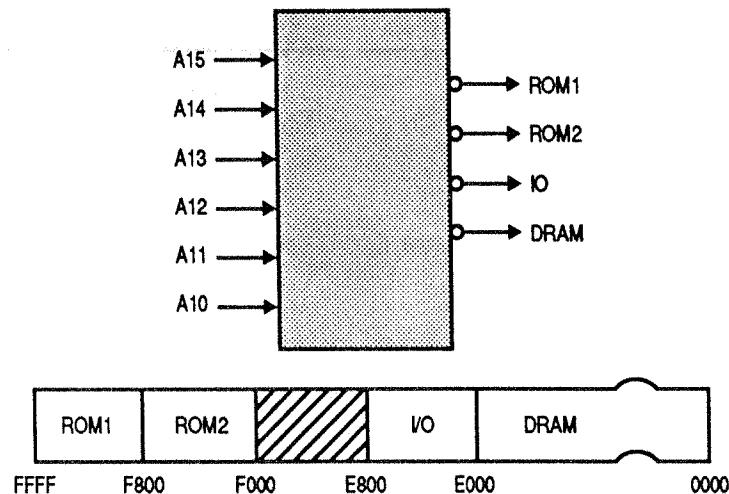
Note: The source file elements below are completed as necessary and can exist in any order, provided no symbol (identifier) is referenced before it has been defined.

- EQUATIONS — Contains the high-level and/or Boolean equations required to describe the logic design. (You can also express the design by means of truth tables or state diagram.)
- TRUTH TABLE — Contains the truth tables required to describe the logic design. (You can also express the design by means of equations or state diagrams.)
- STATE DIAGRAM — Contains the state diagrams required to describe the logic design. (You can also express the design by means of equations or truth tables.)
- TEST VECTORS — Contains the optional test vectors that you write to verify the functionality of the logic design. Test vectors can be used in conjunction with test vectors generated by PLDtest or PLDtest Plus, which functionally test the programmed device.
- END STATEMENT — Ends the module.

Examining a Source File

As previously stated, you must write a source file that reflects the design of the circuit to be programmed into the programmable logic device before ABEL-PLD can generate the required programmer load file. In order to write a meaningful source file, you must be familiar with its make-up and content.

Figure 5-2
Block Diagram; 6809 Memory
Address Decoder



The following paragraphs examine the source file shown in Figure 5-3, which describes the address decoder shown in the block diagram, Figure 5-2. This source file is typical of that used with the language processor to produce a programmer load file for downloading to a logic programmer.

This source file does not illustrate all the features of ABEL-PLD, nor does it show examples of source file syntax rules. Complete source file details are given later in this chapter.

Figure 5-2
Source File Describing an
Address Decoder

```

module m6809a
title '6809 memory decode
Jean Designer Data I/O Corp. Redmond WA'

U09a    device 'P14L4';
A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
ROM1,IO,ROM2,DRAM      pin 14,15,16,17;

H,L,X = 1,0,.X.;
Address = [A15,A14,A13,A12,A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
  !DRAM = (Address <= ^hDFFF);
  !IO   = (Address >= ^hE000) & (Address <= ^hE7FF);
  !ROM2 = (Address >= ^hF000) & (Address <= ^hF7FF);
  !ROM1 = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
  ^h0000 -> [ H , H , H, L ];
  ^h4000 -> [ H , H , H, L ];
  ^h8000 -> [ H , H , H, L ];
  ^hC000 -> [ H , H , H, L ];
  ^hE000 -> [ H , H , L, H ];
  ^hE800 -> [ H , H , H, H ];
  ^hF000 -> [ H , L , H, H ];
  ^hF800 -> [ L , H , H, H ];

end m6809a

```

Purpose of the Address Decoder

An address decoder is a logic design problem that is easily solved with programmable logic. In the address decoder of Figure 5-1, the high-order six lines of an address bus are decoded into one of four active-low outputs.

Output	Low for addresses	Enables
ROM1	F800 to FFFF	ROM
ROM2	F000 to F7FF	ROM
I/O	E000 to EFFF	I/O ports
DRAM	0000 to DFFF	dynamic RAM

The MODULE Statement

The MODULE statement is a required element of the source file. It defines the beginning of the module (since a source file may contain several modules) and must be paired with an END statement. In Figure 5-2, the MODULE statement is module m6809a. The keyword is module. The identifier m6809a distinguishes the module from any others in the source file.

The FLAG Statement

The address decoder example in Figure 5-2 does not use the FLAG statement. The source file template in Figure 5-1 shows the location of the FLAG statement when used.

The TITLE Statement

The TITLE statement may be inserted in the source file to give a title to a module. Although the title is not acted on by the language processor, it will appear as a header in both the programmer load file and documentation file created by the language processor.

The TITLE statement consists of the keyword title followed by a string, which is the desired title for the module. The string is opened and closed by an apostrophe. In Figure 5-2, the title is:

```
'6809 memory decode  
Jean Designer Data I/O Redmond WA'
```

The DEVICE Declaration

The DEVICE declaration is used to associate the name of the device with the type of programmable logic device into which the logic design will be programmed. The name of the device in Figure 5-2 is U09, which is the schematic reference in this case. The type of programmable logic device is specified as 'P14L4'. The device name and device type are located on the same line of the source file but separated by device, the keyword of the DEVICE declaration.

PIN and NODE Declarations

In Figure 5-2, the two lines immediately following the DEVICE declaration are the PIN declarations. Two lines are used since there is insufficient horizontal space on an 80-column display to arrange the declaration on a single line. You can use as many lines as necessary to make the pin declarations.

The PIN declaration associates pin identifiers used in the module to actual pin numbers on the target programmable logic device. The first PIN declaration statement associates address line A15 with pin 1 of the P14L4 device, A14 with pin 2, and so on. The second PIN declaration identifies pin 14 of the P14L4 as the ROM1 output, pin 15 as the I/O output, etc.

Attributes can also be assigned to pins, although none are required in this memory decoder example.

There are no NODES to be declared in this example since the P14L4 logic device contains no internal signals that are not accessible at the external pins.

CONSTANT Declarations

A constant, described fully in the chapter "Language Structure," is an identifier that retains a constant value through a module. CONSTANT declarations are placed with the PIN and NODE declarations in the source file and use the = sign for their keyword. In Figure 5-2, the declaration H,L,X = 1,0,X.; tells the language processor to substitute a logic 1 whenever an uppercase H is encountered, a logic 0 whenever an upper-case L is encountered, and a "don't care" value whenever an upper-case X is encountered. The H, L, and X identifiers are used in subsequent test vectors. (.X. is a special constant used to denote a don't care condition. Refer to "Special Constants" in the chapter "Language Elements.")

The second CONSTANT declaration equates the identifier, Address, with a set consisting of A15, A14, A13, A12, A11, A10, and ten don't care (X) values. The X's are used for the low-order address lines, since the "equations" and "test vectors" sections of the source file relate the constant address to all 16 address lines.

EQUATIONS Statements

The EQUATIONS statement defines the beginning of a group of equations that specify the logic functions of a device. The actual equations follow the EQUATIONS statement. In Figure 5-2, there are four equations, one for each of the address decoder outputs. These four equations describe the logic function of the address decoder shown in Figure 5-2. Identifiers, defined in the declaration statements, are used in place of P14L4 pin numbers.

The first equation, !DRAM = (Address = hDFFF); equates the active-low DRAM output (pin 17 of the P14L4) with the address lines set to any address equal to or above DFFF (hex). For the sake of brevity, hexadecimal notation is used to specify the address input conditions. ABEL-PLD cross-references the identifier Address back to the P14L4 pin numbers by means of the preceding CONSTANT and PIN declaration statements. The second equation equates the active-low I/O output (pin 15) with the address lines set to any address in the range of E000 to E7FFF. Again, identifiers defined in the declaration statements are used to simplify the equation. The third and fourth equations are written in a similar fashion to define the remaining two address decoder outputs.

Test Vectors

Test vectors specify the expected functional operation of a logic device by defining its outputs as a function of its inputs. Figure 5-2 shows a typical set of test vectors and how they are arranged in a table. The form of the test vector table is determined by the header. In Figure 5-2, the test vector header is

```
test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
```

The left side of the test vector table specifies the state of the device inputs. Hexadecimal notation is used as a means of simplifying the writing of the vector; all 16 address lines could be indicated individually by 1's and 0's or H's and L's. The right side of the test vector table specifies the state of the device outputs. The H and L identifiers are used to indicate the expected state of each output for the corresponding inputs.

Sufficient test vectors should be included to enable the simulator within ABEL-PLD to test the intended function of the design. The test vectors provided in Figure 5-2 enable testing of the design at several, but not all possible, addresses. The test vectors are sufficient to verify the operation of each output at its lowest address, and pin 17 (DRAM) output is tested at four different addresses.

Some devices power up with registers set to 1, some set to 0 and some set to random value. The first test vector should place the device in a known state. The ABEL-PLD simulator assumes D, JK, and T registers power up to 0 and RS registers power up to 1. Some devices have functions (enables or presets) directly connected to pins. Be sure to include this function in the test vectors or simulation errors may occur.

Processing a Source File

Once ABEL-PLD is installed on your system as described in the *Installation Guide*, you can process a source file to create a programmer load file and, if you wish, download the load file to your logic programmer. The chapter "Using ABEL-PLD" describes how to process a source file and explains the different processes ABEL-PLD executes in order to achieve the load file and accompanying documentation.



Chapter 6

Language Elements

This section describes the various elements of the ABEL-PLD design language. These language elements are combined according to the structure described in the chapter "Source Files" in the Language Reference to create ABEL-PLD logic descriptions. Each element is presented on the following pages. The organization is from the most basic to the most complex topics.

Basic Syntax

Each line in an ABEL-PLD source file must conform with the following syntax rules and restrictions:

1. A line may be up to 131 characters long.
2. Lines are ended by a line feed character (hex 0A), by a vertical tab (hex 0B), or by a form feed (hex 0C). Carriage returns in a line will be ignored, thus accommodating common end-of-line sequences, such as carriage return/line feed. On most computers, an input line is ended simply by pressing the <Return> or <Enter> key.
3. Keywords, identifiers, and numbers must be separated from each other by at least one space. Exceptions to this rule are in lists of identifiers separated by commas, in expressions where identifiers or numbers are separated by operators, or in places where parentheses provide the separation.

4. Neither spaces nor periods can be imbedded in the middle of keywords, numbers, operators or identifiers. Spaces can appear in strings, comments, blocks and actual arguments. For example, if the keyword MODULE is entered as "MOD ULE", it will be interpreted as two identifiers, MOD and ULE. Similarly, if you enter "102 05" (instead of 10205), it will be interpreted as two numbers, 102 and 5.
5. Keywords (words defined as part of the language and that have specific uses) can be typed in either uppercase or lowercase.
6. Identifiers (user-supplied names and labels) can be typed in either uppercase, lowercase, or mixed-case, but are case sensitive: the identifier output, typed in all lowercase letters, is not the same as the identifier Output with an uppercase "O".

Valid ASCII Characters

All uppercase and lowercase alphabetic characters and most other characters on common keyboards are valid. Valid characters are listed or shown below.

```
a - z (lowercase alphabet)  
A - Z (uppercase alphabet)  
0 - 9 (digits)  
<space>  
<tab>  
! @ # $ ? + & * ( ) -  
_ = + [ { } ; : ' "  
' ~ \ | , < > . / ^ %
```

Identifiers

Identifiers are names that identify devices, device pins or nodes, sets, input or output signals, constants, macros, and dummy arguments. All of these items are defined later in section 8. The rules and restrictions for identifiers are the same regardless of what the identifier describes.

The rules governing identifiers are

1. Identifiers may be up to 31 characters long. Anything longer than 31 characters is considered an error and is flagged by the language processor.
2. Identifiers must begin with an alphabetic character or with an underscore.

3. Other than the first character (see rule 1), identifiers may contain uppercase and lowercase alphabetic characters, digits and underscores.
4. Spaces cannot be used in an identifier. Use underscores to provide separation between words.
5. Identifiers are case sensitive: uppercase letters and lowercase letters are not the same.
6. Periods cannot be used in an identifier, except when using a valid dot extension.

Some valid identifiers are

```
HELLO
hello
_K5input
_P_h
This_is_a_long_identifier
```

Note the use of underscores to separate words. Use different cases to make your source file easy to read.

Some invalid identifiers are

7	Does not begin with a letter or underscore
\$4	Does not begin with a letter or underscore
HEL.LO	Contains a period (.LO is not a valid dot extension)
b6 kj	Contains a space

The last of these invalid identifiers will be seen by the language processor as two identifiers, b6 and kj.

Reserved Identifiers (Keywords)

Keywords, listed below, are reserved identifiers that are part of the ABEL-PLD Design Language, and cannot be used to name devices, pins, nodes, constants, sets, macros or signals. When a keyword is used, it refers only to the function of that keyword. If a keyword is used in the wrong context, an error is flagged by the language processor. Note that WHEN is new to this version of software, and existing ABEL-PLD source files should be checked for inadvertent use of this keyword.

case	fuses	pin
device	goto	state
else	if	state_diagram
in	test_vectors	signal
end	istype	then
endcase	library	title
endwith	macro	truth_table
equations	module	when
flag	node	with
declarations	XOR_Factor	

Choosing Identifiers

The right choice in identifiers can make a source file easy to read and understand. The following suggestions are given to help make logic descriptions self-explanatory, eliminating the need for extensive documentation.

- Choose identifiers that match the function of what they describe. For example, the pin to be used as the carry-in on an adder could be named Carry_In. For a simple OR gate, the two input pins might be given the identifiers IN1 and IN2, and the output might be named OR.
- Avoid large numbers of similar identifiers. For example, do not name the outputs of a 16 bit adder: ADDER_OUTPUT_BIT_1 ADDER_OUTPUT_BIT_2 and so on. Such grouping of names makes the source file difficult to read.
- Use underscores to separate words in your identifier.

THIS_IS_AN_IDENTIFIER

is much easier to read than

THISISANIDENTIFIER

- Mixed-case identifiers can help make your source file readable; for example, CarryIn

Strings

Strings are series of ASCII characters enclosed by single quotes (apostrophes). Strings are used in the TITLE, MODULE and FLAG statements, and in pin, node, and attribute declarations. Spaces are allowed in strings.

Valid strings:

'Hello'
' Text with a space in front'
' '
'The preceding line is an empty string'
'Punctuation? is even allowed !!'

A single quote can be included in a string by preceding it with a backslash, "\".

'It\'s easy to use ABEL-PLD'
is the string,

It's easy to use ABEL-PLD

Backslashes can be put in a string by using two of them in succession.

'He\lshe can use backslashes in a string'

becomes the string:

He\lshe can use backslashes in a string

Note: Back-quotes ('') are also accepted as string delimiters and can be used interchangeably with the forward quote ('').

Comments

Comments are another way to make a source file easy to understand. Comments explain what is not readily apparent from the source code itself. Comments do not affect the meaning of the code.

A comment begins with a double quotation mark, "", and ends with either another double quotation mark or the end of line, whichever comes first. The text of the comment follows the opening quotation mark.

Comments cannot be imbedded within keywords.

Valid comments (shown in boldface):

```
MODULE Basic_Logic; "gives the module a name
TITLE 'ABEL design example: simple gates'; "title
"declaration section"
IC4 device 'P10L8'; "declare IC4 to be a P10L8
IC5 "decoder PAL" device 'P10H8';
```

The information inside single quotation marks (apostrophes) are identifiers, not comments, and are part of the statement.

Numbers

All operations in ABEL-PLD involving numeric values are done with 32-bit accuracy. Thus, valid numeric values fall in the range 0 to 2^{32} minus1. Numbers are represented in any of five forms. The four most common forms represent numbers in different bases. The fifth form uses alphabetic characters to represent a numeric value.

When one of the four bases other than the default base is chosen to represent a number, the base used is indicated by a symbol preceding the number. Table 6-1 gives the four bases supported by ABEL-PLD and their accompanying symbols. The base symbols can be typed in uppercase or lowercase.

Table 6-1
*Number Representation in
 Different Bases*

Base Name	Base	Symbol
binary	2	b
octal	8	o
decimal	10	d (default)
hexadecimal	16	h

When a number is specified and is not preceded by a base symbol, it is assumed to be in the default base numbering system. The normal default base is base 10, so numbers are represented in decimal form unless preceded by a symbol indicating that another base is to be used.

For special applications, the default base can be changed. See "@RADIX," in chapter 9 for more information.

Here are some examples of valid number specifications. The default base is base ten.

Specification	Decimal Value
75	75
h75	117
b101	5
o17	15
h0F	15

The circumflex, " $^$ ", must be entered as a character from the keyboard. It does not represent a control key sequence as in some other popular software.

Numbers may also be specified by alphabetic characters. In this case, the numeric ASCII code of the letter is used as the numeric value. For example, the character "a" is decimal 97, and hexadecimal 61 in ASCII coding. The decimal value 97 would be used if "a" were specified as a number.

Sequences of alphabetic characters are first converted to their binary ASCII values, and then are concatenated to form numbers (usually large). Some examples of numbers specified with characters are given below:

Specification	Hex Value	Decimal Value
a	h61	97
b	h62	98
abc	h616263	6382203

Special Constants

Constant, non-changing values can be used in ABEL-PLD logic descriptions. Constant values are used in assignment statements, truth tables and test vectors and are sometimes assigned to an identifier that then denotes that value throughout a module (see "Declarations" and "Module Statement" in the chapter "Language Structure"). Constant values may be either numeric or one of the non-numeric special constant values. The special values are listed in Table 6-2.

When one of the special constants is used, it must be entered as shown in Table 6-2 with surrounding periods. The periods indicate that a special constant is being used; without the periods, .C. would appear to be an identifier named C. Special constants can be entered in either uppercase or lowercase.

Table 6-2
Special Constants

Constant	Description
.C.	clocked input (low-high-low transition)
.F.	floating input or output signal
.K.	clocked input (high-low-high transition)
.P.	register preload
.SVn.	n = 2 through 9. Drive the input to super voltage 2 through 9.
.X.	don't care condition
.Z.	test input or output for high impedance

A new include file, constants.inc, supplied with ABEL 3.2 software contains definitions for the most used ABEL constants. This file may be included in your design with the command:

```
@LIBRARY CONSTANTS
```

Operators, Expressions, and Equations

Items such as constants and signal names can be brought together in expressions. Expressions combine, compare or perform operations on the items they include to produce a single result. The operations to be performed (addition and logical AND are just two examples) are indicated by operators within the expression.

ABEL-PLD operators are divided into four basic types: logical, arithmetic, relational, and assignment. Each of these types is discussed separately below, followed by a description of how they are combined into expressions. Following the descriptions is a summary of all the operators and the rules governing them, and finally an explanation of how equations utilize expressions.

Logical Operators

Logical operators are used in expressions. ABEL-PLD incorporates the standard logical operators used in most logic designs; these operators are listed in Table 6-3. Logical operations involving operands of more than one bit are performed bit by bit. Thus, 2 minus 4 equals 6. For alternate operators, refer to the @ALTERNATE Directive section in this Language Reference manual.

Table 6-3
Logical Operators

Operator	Example	Description
!	!A	NOT: ones complement
&	A & B	AND
#	A # B	OR
\$	A \$ B	XOR: exclusive OR
!\$	A !\$ B	XNOR: exclusive NOR

Arithmetic Operators

Arithmetic operators define arithmetic relationships between items in an expression. The shift operators are included in this class because each left shift of one bit is equivalent to multiplication by 2 and a right shift of one bit is the same as division by 2. Table 6-4 lists the arithmetic operators.

A minus sign has a different significance depending on its usage. When used with one operand, it indicates that the twos complement of the operand is to be formed. When the minus sign is found between two operands, the twos complements of the second operand is added to the first.

Table 6-4
Arithmetic Operators

Operator	Example	Description
-	-A	twos complement
-	A - B	subtraction
	A + B	addition
*	A * B	multiplication
/	A / B	unsigned integer division
%	A % B	modulus: remainder from /
<	A < B	shift A left by B bits
>	A > B	shift A right by B bits

Division is unsigned integer division: the result of division is a positive integer. The remainder of a division can be obtained by using the modulus operator, "%". The shift operators perform logical unsigned shifts. Zeros are shifted in from the left during right shifts and in from the right during left shifts.

Relational Operators

Relational operators are used to compare two items in an expression. Expressions formed with relational operators produce a Boolean true or false value. Table 6-5 lists the relational operator.

All relational operations are unsigned. For example, the expression $-1 > 4$ is true since the twos complement of 1 is 1111, which is 15 in unsigned binary, and 15 is greater than 4. For the purpose of this example, a four-bit representation was assumed; in actual use, -1, the twos complement of 1, is 32 bits all set to 1.

Table 6-5
Relational Operators

Operator	Example	Description
<code>==</code>	<code>A == B</code>	equal
<code>!=</code>	<code>A != B</code>	not equal
<code><</code>	<code>A < B</code>	less than
<code><=</code>	<code>A <= B</code>	less than or equal
<code>></code>	<code>A > B</code>	greater than
<code>>=</code>	<code>A >= B</code>	greater than or equal

Some examples of relational operators in expressions follow:

Expression	Value
<code>2 == 3</code>	false
<code>2 != 3</code>	true
<code>3 < 5</code>	true
<code>-1 > 2</code>	true

The logical values, true and false, are represented internally by numbers. Logical true is -1 in twos complement: all 32 bits are set to 1. Logical false is 0 in twos complement so all 32 bits are set to 0. This implies that an expression producing a true or false value (a relational expression) can be used anywhere a number or numeric expression could be used and -1 or 0 will be substituted in the expression depending on the logical result.

For example,

`A = D \$ (B == C);`

means that

A will equal the complement of D if B equals C
A will equal D if B does not equal C.

When using relational operators, always use parentheses to ensure the expression is evaluated in the order you expect. The logical operators & and # have a higher priority than the relational operators (see the section "Operators, Expressions, and Equations" in the chapter "Language Elements" of the ABEL manual). The following equation

```
Select = [A15..A0] == ^hD000 # [A15..A0] == ^h1000;
```

has a default parentheses grouping as follows:

```
Select = [A15..A0] == (^hD000 # [A15..A0]) == ^h1000;
```

which is not the intended equation. To get the desired results, write the equation as follows:

```
Select = ([A15..A0] == ^hD000) # ([A15..A0] == ^h1000);
```

Assignment Operators

Assignment operators are a special class of operators used in equations rather than in expressions. Equations assign the value of an expression to output signals. See "Equations" below for a complete discussion of equations. There are two assignment operators, unclocked and clocked. Unclocked or immediate assignment occurs without any delay as soon as the equation is evaluated. Clocked assignment occurs at the next clock pulse from the clock associated with the output. Table 6-6 shows the assignment operators.

Table 6-6
Assignment Operators

Operator	Description
=	Unclocked assignment (combinatorial outputs)
:=	Clocked assignment (registered outputs)

Expressions

Expressions are combinations of identifiers and operators that produce one result when evaluated. Any logical, arithmetic or relational operators may be used in expressions.

Expressions are evaluated according to the particular operators involved. Some operators take precedence over others, and their operation will be performed first. Each operator has been assigned a priority that determines the order of evaluation. Priority 1 is the highest priority, and priority 4 is the lowest. Table 6-7 summarizes the logical, arithmetic and relational operators, presented in groups according to their priority.

Table 6-7
Summary of Operators and Priorities

Priority	Operator	Description
1	-	negate
1	!	NOT
2	&	AND
2	<	shift left
2	>	shift right
2	*	multiply
2	/	unsigned division
2	%	modulus
3	+	add
3	-	subtract
3	#	OR
3	\$	XOR: exclusive OR
3	!\$	XNOR: exclusive NOR
4	==	equal
4	!=	not equal
4	<	less than
4	<=	less than or equal
4	>	greater than
4	>=	greater than or equal

When operations of the same priority exist in the same expression, they are performed in the order found from left to right in that expression. Parentheses may be used as in normal mathematics to change the order of evaluation, with the operation in the innermost set of parentheses performed first.

Some examples of valid expressions are given below. Note how the order of operations and the use of parentheses affect the evaluated result.

Expression	Result	Comments
$2 * 3 / 2$	3	operators with same priority
$2 * 3 / 2$	3	spaces are OK
$2 * (3/2)$	2	fraction is truncated
$2 + 3 * 4$	14	
$2\#4\$2$	4	
$2\#(4\$2)$	6	
$2 == ^HA$	0	false
$14 == ^HE$	-1	true

Equations

Equations assign the value of an expression to a signal or set of signals in a logic description. The identifier and expression must follow the rules already established for those elements.

Syntax	[WHEN condition THEN][!]... [ENABLE] element = expression; [ELSE equation] or [WHEN condition THEN][!]... [ENABLE] element := expression; [ELSE equation]
	condition any valid expression
	element an identifier naming a signal or set of signals, or an actual set, to which the value of expression will be assigned
	expression any valid expression
	= and := unclocked and clocked assignment operators
Remarks	Equations use the two assignment operators = (unclocked) and := (clocked) described above. The enable dot extension, .OE, is used to enable tristate output buffers. The identifier must be a tristate type output, and the value assignment applies only to the buffer enable rather than to the signal itself.

The complement operator, "!", can be used to express negative logic. The complement operator precedes the signal name and implies that the expression on the right of the equation is to be complemented before it is assigned to the signal. Use of the complement operator on the left side of equations is provided as an option; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

Examples of equations:

```
X = A & B;      " unclocked assignment to X
Y.OE = C # D;    " Y is enabled if C or D is true
Y: = A & B;      " clocked assignment to Y
!A = B & C # D;  " same as A = !(B & C # D);
WHEN B THEN A=B; ELSE A=C;
```

Multiple Assignments to the Same Identifier

When an identifier appears on the left side of more than one equation, the expressions being assigned to the identifier are first ORed together and then the assignment is made. If the identifier on the left side of the equation is complemented, the complement is performed after all the expressions have been ORed.

Examples:

<u>Equations Found</u>	<u>Equivalent Equation</u>
A = B; A = C;	A = B # C;
A = B; A = C & D;	A = B # (C & D);
A = !B; A = !C;	A = !B # !C;
!A = B; !A = C;	A = !(B # C);
!A = B; A = !C;	A = !C # !B;
!A = B; !A = C; A = !D; A = !E;	A = !D # !E # !(B # C);

Note that when the complement operator appears on the left side of multiple assignment equations, the right-hand sides are ORed first and then the complement is applied.

Sets

A set is a collection of signals and constants that is operated on as one unit. Any operation applied to a set is applied to each element in the set. Sets simplify ABEL-PLD logic descriptions and test vectors by allowing groups of signals to be referenced with one name.

For example, the outputs B0-B7 of an eight-bit multiplexer could be collected into the set named MULTOUT. The three selection lines might be collected in the set, SELECT. The multiplexer could then be defined in terms of MULTOUT and SELECT rather than being defined by all the input and output bits individually specified.

A set is represented by a list of constants and signals separated by commas, or the range operator (...), and surrounded by brackets. For example,

Sample Set	Description
[B0,B1,B2,B3,B4,B5,B6,B7]	outputs (MULTOUT)
[S0,S1,S2]	select lines (SELECT)

The above sets could also be expressed by using the range operator; for example,

[B0..B7]
[S0..S2]

Identifiers used to delimit a range must have compatible names: they must begin with the same alphabetical prefix and have a numerical suffix. Range identifiers can also delimit a decrementing range or a range which appears as one element of a larger set; for example,

[A7..A0] "decrementing range
[X..X..X..X..X..A10..A7] "range within a larger set

Note that for set specifications the brackets do not denote optional items. The brackets are required to delimit the set. Note also that ABEL-PLD source file sets are not mathematical sets.

The source equation

WHEN 1 THEN [x,y] := 1; ELSE [x,y] := 0;

produces the original equations

[x,y] := (1) & [1,1]; [x,y] := (!1) & [0,0];

and the transformed equations

x := 0; y := 1;

which are not the expected results. The reason for the above equations is the constant (1) in [x,y] := (1) & [1,1];

One solution to this is

WHEN [1,1] THEN [x,y] := 1; ELSE [x,y] := 0;

which makes the sets the same size. Another solution is

WHEN !0 THEN [x,y] := 1; ELSE [x,y] := 0;

which expands !0 to the width of the set.

Set Operations

Most operators can be applied to sets. In general, this means that the operation is performed on each element of the set, sometimes individually and sometimes according to the rules of Boolean algebra. Table 6-8 lists the operators that may be used with sets. The appendix "Set Operations" describes how these operators are applied to sets.

For operations involving two or more sets, the sets must have the same number of elements. The expression, "[a,b]+[c,d,e]", is invalid because the sets have different numbers of elements.

Some examples of set usage are given here.

The equation,

Chip_Sel = A15 & !A14 & A13;

represents an address decoder where A15, A14 and A13 are the three high-order bits of a 16-bit address. The decoder can easily be implemented with set operations. First, a constant set that holds the address lines is defined so that the set can be referenced by name. This definition is done in the constant declaration section of a module (described in the chapter "Language Structure").

The declaration is

Addr = [A15,A14,A13];

which declares the constant set Addr. The equation,

Chip_Sel = Addr == [1,0,1];

is functionally equivalent to

Chip_Sel = A15 & !A14 & A13;

If Addr is equal to [1,0,1], meaning that A15 = 1, A14 = 0 and A13 = 1, then Chip_Sel is set to true. Note that the set equation could also have been written as

Chip_Sel = Addr == 5;

because 101 binary equals 5 decimal.

In the example above, a special set with the high-order bits of the 16-bit address was declared and used in the set operation. The full address could have been used and the same function arrived at in other ways, as shown below.

Example 1

```
" declare some constants in declaration section
Addr = [a15..a0]; X = .X.; "simplify notation for don't care constant
Chip_Sel = Addr == [1,0,1,X,X,X,X,X,X,X,X,X,X,X,X];
```

Example 2

```
" declare some constants in declaration section
Addr = [a15..a0]; X = .X.;
Chip_Sel = (Addr >= ^HA000) & (Addr <= ^HBFFF);
```

Both of the solutions presented in these final two examples are functionally equivalent to the original equation and to the first solution in which only the high order bits are specified as elements of the set (Addr = [a15, a14, a13]).

Set Assignment and Comparison

Values and sets of values can be assigned and compared to a set. For example,

```
sigset = [1,1,0] & [0,1,1];
```

results in sigset being assigned the value, [0,1,0].

Numbers in any representation can be assigned or compared to a set. The preceding set equation could have been written as

```
sigset = 6 & 3;
```

When numbers are used for set assignment or comparison, the number is converted to its binary representation and the following two rules apply:

1. If the number of significant bits in the binary representation of a number is greater than the number of elements in a set, the bits are truncated on the left.
2. If the number of significant bits in the binary representation of a number is less than the number of elements in a set, the number is padded on the left with leading zeroes.

Thus, the following two assignments are equivalent:

```
[a,b] = ^B101011; "bits truncated to the left
[a,b] = ^B11;
```

And so are these two:

```
[d,c] = ^B01;
[d,c] = ^B1; "leading zero added
```

The set assignment,

$[a,b] = c \& d;$

is the same as the two assignments:

$a = c \& d;$
 $b = c \& d;$

Table 6-8
Valid Set Operations

Operator	Example	Description
=	$A = 5$	assignment
:=	$A := [1,0,1]$	clocked assignment
!	$!A$	NOT: ones complement
&	$A \& B$	AND
#	$A \# B$	OR
\$	$A \$ B$	XOR: exclusive OR
!\$	$A !\$ B$	XNOR: exclusive NOR
-	$-A$	negate
-	$A - B$	subtraction
	$A + B$	addition
==	$A == B$	equal
!=	$A != B$	not equal
<	$A < B$	less than
<=	$A <= B$	less than or equal
>	$A > B$	greater than
>=	$A >= B$	greater than or equal

Set Evaluation

How each operator will act when used with sets depends upon the types of its arguments.

When a set is written

$[a, b, c, d]$

"a" is the MOST significant bit and "d" is the LEAST significant bit.

The result, when most operators are applied to a set, will be another set. Note that the result of the relational operators ($==$, $!=$, $>$, \geq , $<$, \leq) is a value: TRUE (all ones) or FALSE (all zeros), which will be truncated to as many bits as are needed. The width of the result is determined by the context of the relational operator, not by the width of the arguments. See examples below.

The different contexts of the AND ($\&$) operator and the semantics of each usage are described below.

signal & signal
example: $a \& b$

This is the most straightforward usage. The expression is TRUE if both signals are TRUE.

signal & number
example: $a \& 4$

The number will be converted to binary and the least significant bit will be used, so this becomes $a \& 0$, which will be reduced to simply 0, or FALSE.

signal & set
example: $a \& [x, y, z]$

The signal will be distributed over the elements of the set to become $[a \& x, a \& y, a \& z]$

set & set
example: $[a, b] \& [x, y]$

The sets will be bit-wise ANDed resulting in: $[a \& x, b \& y]$. An error will be displayed if the set widths do not match.

set & number
example: $[a, b, c] \& 5$

The number will be converted to binary and truncated or padded with zeros as needed to match the width of the set. The sequence of transformations will be

$$\begin{aligned} & [a, b, c] \& [1, 0, 1] \\ & = [a \& 1, b \& 0, c \& 1] \\ & = [a, 0, c] \end{aligned}$$

number & number
example: $9 \& 5$

The number will be converted to binary and the least significant bit will be used, so this becomes 1 & 1, which will be reduced to simply 1, or TRUE.

Some example equations:

`select = [a15..a0] == ^H80FF`

`select` (signal) will be TRUE when the 16-bit address bus has the hex value 80FF.

`[sel1, sel0] = [a3..a0] > 2`

Both `sel1` and `sel2` will be true when the value of the four "a" lines (taken as a binary number) are greater than 2. Note that the width of the "sel" set and the "a" set are different. The "2" will be expanded to four bits (of binary) to match the size of the "a" set. The result of the comparison (which will be all ones or all zeros) will then be truncated to two bits to match the size of the "sel" set.

```
[out3..out0] = [in3..in0] & enable
```

If enable is TRUE, then the values on "in0" through "in3" will be seen on the "out0" through "out3" outputs. If enable is FALSE, then the outputs will all be FALSE.

Limitations/Restrictions on Sets

Since the widths of expression arguments are determined from context, there are some cases where the results are not as you might expect. For example, define count as a 3-bit set:

```
count = [a,b,c]
```

Then the following expression has a width of one:

```
9 & (count == 0)
```

It will be expanded as follows:

```
9 & ( la & !b & !c ) 1 & !a & !b & !c
```

There are also cases where an operator may not be commutative and associative because the results of its evaluation depend upon the context. Consider the following two equations. In the first, the constant "1" will be converted to a set; in the second, the "1" will be treated as a single bit.

$$\begin{aligned} [x1, y1] &= [a, b] \& 1 \& d & [x2, y2] &= 1 \& d \& [a, b] \\ &= ([a, b] \& 1) \& d & = (1 \& d) \& [a, b] \\ &= ([a, b] \& [0, 1]) \& d & = d \& [a, b] \\ &= ([a \& 0, b \& 1]) \& d & = [d, a] \& [d, b] \\ &= [0, b] \& [b, d] & = [d \& a, d \& b] \\ &= [0 \& d, b \& d] \\ &= [0, b \& d] \end{aligned}$$
$$\begin{aligned} x1 &= 0 & x2 &= a \& d \\ y1 &= b \& d & y2 &= b \& d \end{aligned}$$

If you are unsure about the interpretation of an equation, try the following hints:

1. Fully parenthesize your equation. Most errors are simply caused by ignoring the precedence rules in Table 6-7.
2. Write out numbers as sets of 1s and 0s instead of as decimal numbers. If the width is not what you expected, you will get an error message.

Because set assignment is applied to all elements of a set, sets with mixed combinational and registered outputs cannot be used on the left side of an equation. Assignments are either clocked (combinational) or unclocked (registered) according to the assignment operator used (= or :=). A clocked assignment to a combinational output or an unclocked assignment to a registered output constitutes an error and will be flagged by the language processor.

Blocks

Blocks are sections of ASCII text enclosed in braces, "(" and ")". Blocks are used in macros and directives. The text contained in a block can be all on one line or can span many lines. Some examples of blocks follow.

```
{ this is a block }
{
    this is also a block, and it
    spans more than one line.
}
{ A = B # C;
  D = [0, 1] + [1, 0];
}
```

Blocks can be nested within other blocks, as shown below, where the block { D = A } is nested within a larger block:

```
{ A = B $ C;
  {
    D = A;
  }
  E = C;
}
```

Blocks and nesting of blocks can be useful in macros and when used with directives. (See "Macro Declarations" in the chapter "Language Structure," and the chapter "Directives.")

If either a right or left brace is needed as a character in a block but does not denote the start or end of a nested block, it is preceded by a backslash. Thus,

```
\{\}
```

is the block containing the characters " { } ", with the spaces included.

Arguments and Argument Substitution

Variable values can be used in macros, modules and directives. These values are called the arguments of the construct that uses them. In ABEL-PLD, a distinction must be made between two types of arguments: actual and dummy. Their definitions are given here.

dummy argument

an identifier that is used to indicate where an actual argument is to be substituted in the macro, module, or directive.

actual argument

the argument (value) used in the macro, directive or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL-PLD.

Dummy arguments are specified in macro declarations and in the bodies of macros, modules and directives. The dummy argument is preceded by a question mark in the places where an actual argument is to be substituted. The question mark distinguishes the dummy arguments from other ABEL-PLD identifiers occurring in the source file.

Take for example, the following macro declaration arguments (macros are discussed fully in the chapter "Language Structure" and an example of usage is presented in the design example file MACRO.ABL):

```
OR_EM MACRO (a,b,c) { ?a # ?b # ?c };
```

This defines a macro named OR_EM that is the logical OR of three arguments. These arguments are represented in the definition of the macro by the dummy arguments, a, b, and c. In the body of the macro, which is surrounded by braces, the dummy arguments are preceded by question marks to indicate that an actual argument will be substituted.

The equation

```
D = OR_EM (x,y,z&1);
```

invokes the OR_EM macro with the actual arguments, x, y, and z&1. This results in the equation:

```
D = x # y # z&1;
```

Spaces (blanks) are significant in actual arguments. Actual arguments are substituted exactly as they appear. Note that in the example above, the actual argument z&1 contains no spaces in the equation referring to OR_EM, and that in the expanded equation the argument appears again without spaces. Had the argument been specified as "z & 1" (note the spaces), the resulting equation would have contained those spaces. For example, the equation,

```
D = OR_EM (x,y,z & 1)
```

results in the equation:

```
D = x # y # z & 1;
```

Argument substitution occurs before the source file is checked for syntactic or logical correctness. This means that the code is checked for correctness with the actual arguments in place. Thus, if an actual argument violates a syntactic or logical rule, the parser will detect and report the error.

In review:

- Dummy arguments are place holders for actual arguments.
- A question mark preceding the dummy argument indicates that an actual argument is to be substituted.
- Actual arguments replace dummy arguments before the source file is checked for correctness.
- Spaces are significant in actual arguments.

Further discussion and examples of argument usage are given in the Applications Guide, and in the Language Reference in the chapters "Language Structure" and "Directives."

Chapter 7

Language Structure

This section describes the structure of a complete ABEL-PLD design description. The language elements described in Chapter 6 are combined within the proper structures to define equations, state machines and truth tables. The device or devices being described must be indicated and initial declarations must be made. Definitions of expected output from simulation of the device may also be made.

Basic Structure

ABEL-PLD source files can be broken into independent parts called modules. Each module contains one or more complete logic descriptions. At the simplest level, an input file to the ABEL-PLD language processor consists of only one module; at the most complex level, an indefinite number of modules may be combined into one source file and processed at the same time.

Every module consists of several different sections, each with its own unique function. Possible sections are

- Declarations of devices, pins, nodes, constants, attributes, and macros
- Logic equations
- Truth tables
- State diagrams
- Explicit fuse declarations
- Test vectors

Some or all of these parts of a module may exist for any given application. Declarations must always be made. Figure 7-1 shows the structure of an ABEL-PLD source file. Because a source file is a collection of one or more modules, each with its own beginning and end, different source files can be combined to form complete system designs in one source file.

Figure 7-1
*Structure of an ABEL-PLD
Source File*

```
1st Module Start
Flags
Title
Declarations
    Constant Declarations
    Macro Definitions
    Device Declarations
    Pin and Node Assignments
    Attribute Declarations
    Library References
Boolean Equations
Truth Tables
State Diagrams
Fuse Declarations
Test Vectors
1st Module End

2nd Module Start
:
:
:
2nd Module End
:
:
```

MODULE Statement

Syntax	<pre>MODULE <i>modname</i> [(<i>dummy_arg</i> [,<i>dummy_arg</i>] ...)] [FLAG statement]... [TITLE statement] declarations [EQUATIONS]... [TRUTH_TABLE] ... [STATE_DIAGRAM]... [FUSES]... [TEST_VECTORS]... END [<i>modname</i>] [;]</pre>
	<p>modname a valid identifier naming the module</p> <p>dummy_arg a dummy argument</p> <p>FLAG processing options for the language processor</p> <p>TITLE the title of the module</p> <p>declarations declarations section of the module in which pin, node, device, attribute, and constant declarations are made</p> <p>EQUATIONS Logic equations section</p> <p>TRUTH_TABLE truth table specification</p> <p>STATE_DIAGRAM state machine specification</p> <p>FUSES explicit fuse declarations</p> <p>TEST_VECTORS specification of simulation test vectors</p>
Remarks	<p>The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.</p> <p>Each module in a source file should have a unique name. The module contains the different declarations, equations, truth tables, state diagrams and simulation tables necessary to complete the logic description(s). The following three rules apply to module structure:</p> <ol style="list-style-type: none"> 1. If the FLAG statement is used, it must be the first keyword used after the MODULE statement. 2. If the TITLE statement is used, it must be the first keyword used after the FLAG statement (if one exists). If FLAG is not used, the TITLE statement must be the first keyword used after the MODULE statement. 3. Only one declarations section can be used in a module. All other sections can be used as many times as needed and in any order (except see #1 and 2 above).

The optional dummy arguments in the module statement allow the actual arguments to be passed to the module when it is processed by the language processor. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value will be substituted by the parser.

For example,

```
MODULE MY_EXAMPLE (A,B)
:
:
C = ?B + ?A
:
:
END
```

In the module named MY_EXAMPLE, C will take on the value of "A + B" where A and B contain actual arguments passed to the module when the language processor is invoked. For further information about dummy arguments, see the chapter "Language Elements."

FLAG Statement

Syntax	<code>FLAG 'option [, option]...' [:]</code>
	<code>option</code> a string containing valid command line options, excepting -i, -o, -n, and all PARSE options
Remarks	The FLAG statement provides an alternate method of defining processing options that affect the way in which the source file is processed by the language processor. These options are normally passed from the command line or from a batch file when the language processor is invoked. FLAG allows the explicit statement of processing options directly in the source file. Options entered from the command line override options specified with the FLAG statement. PARSE options, and the -i, -o, and -n options are not allowed. Complete information on command line options is in the chapter "Using ABEL-PLD."
	FLAG is useful when a source file requires a specific type or level of processing for that processing to be successful. For example, a design may be large or complex enough that it will generate too many product terms for the specified device unless the PRESTO reduction algorithm is used in the reduction step of the processing. You might also want to simulate the design at trace level 3. This is done with the following command:

```
flag '-r2,-t3'
```

TITLE Statement

Syntax	<code>TITLE 'string'</code>
Remarks	<p>The title statement is used to give a module a title that will appear as a header in both the programmer load file and documentation file created by the language processor. The title is specified in the string following the keyword, TITLE.</p> <p>Use of the title statement is optional.</p> <p>If asterisks are found in the title string, they will not appear in the programmer load file header in order to conform with the JEDEC standard.</p> <p>An example of a title statement that spans three lines and describes the logic design follows:</p> <pre>module m6809a title '6809 memory decode Jean Designer Data I/O Corp Redmond WA'</pre>

Declarations

The declarations section of a module specifies the device being described and the correspondence between the names used in a module, and the pins and nodes of the device. Constants, attributes, and macros are also defined in the declarations section. Declarations stay in effect only in the module in which they are defined. Each module must have its own declarations section. There are several types of declaration statements:

- Attribute
- Constant
- Device
- Library
- Macro
- Node
- Pin

The syntax and use of each of these types is presented in the subsections that follow.

Declaration Keyword

A new Declarations keyword and associated section has been added. Using this new keyword, declarations (such as sets or other constants) can be declared in any part of the ABEL source file.

Syntax

DECLARATIONS
declarations

declarations can be replaced by any of the declaration statements described below.

Remarks

An example of declared equations, declare.abl, is shown below:

```
module DECLARE
    U1      device 'p22v10';
    A,B    pin 2,3 ;
    comout pin 14 ;

EQUATIONS
"combinatorial outputs
    comout = A # B ;

DECLARATIONS
    clk     pin 1;
    C,D    pin 4,5 ;
    regout pin 15 ;

EQUATIONS
"registered outputs
    regout := C # D ;
END
```

Device Declaration Statement

Syntax

device_id [, *device_id*]... DEVICE *real_device* ;

device_id an identifier used in a module for references to a device

real_device the ABEL-PLD device filename for the device

Remarks

The device declaration statement associates device names used in a module with the actual PLDs designs are implemented on. The device name used in the logic description is specified with *device_id*. Device identifiers used in device declarations should be valid filenames since JEDEC files are created by appending the extension .jed to the identifier. The ABEL-PLD device filename of the programmable logic device is indicated by the string, *real_device*. The ending semicolon is required.

The following example specifies two device names, u14 and u15, that represent F105 IFLs:

```
u14, u15 device 'F105' ;
```

Pin Declaration Statement

Syntax

```
[!]pin_id [, [!]pin_id [... PIN [ IN device_id ] pin# [=attr[,attr]...]
[ ,pin# [=attr[,attr]...]]]...
```

pin_id	an identifier used to refer to a pin throughout a module
device_id	a declared device name identifier indicating the device associated with these pins
pin #	the pin number on the real device
attr	a string that specifies pin attributes for devices with programmable pins. Valid strings are listed in Table 7-1 later in this chapter.

Remarks

The pin declaration statement indicates the correspondence between identifiers used in a module and the pins on a real device. The declaration can also specify pin attributes for devices with programmable pin characteristics.

When lists of pin_ids and pin #s are used in one pin declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one pin number associated with each identifier listed.

When more than one device is declared in one module, the optional IN portion of the pin declaration must be used so that the pins are associated with the proper device. A device declaration must be made before the pin declarations. The ending semicolon is required. An example of a simple pin declaration follows:

```
!Clock, Reset, S1 PIN IN U12 12,15,3;
```

This pin declaration assigns the pin name, Clock, to pin 12 of device U12, Reset to pin 15, and S1 to pin 3.

The use of the "!" operator in pin declarations indicates that the pin is active-low, and will be automatically negated when processed by the language processor.

The pin attribute, attr, specifies pin attributes. Attributes can be defined in this way or with the ISTYPE statement. The ISTYPE statement and attributes are discussed later in this chapter.

The pin declaration,

```
F0 pin 13 = 'neg, reg';
```

specifies that equations for F0, which corresponds to pin 13, should be optimized for a negative polarity registered output.

Node Declaration Statement

Syntax

```
[!]node_id [,!]node_id ]... NODE [ IN device_id ] node# [= 'attr  
,attr ]... ][,node# [= 'attr[,attr]...'] ]...
```

node_id an identifier used for reference to a node in a logic design

device_id a declared device name identifier indicating the device associated with these nodes

node # the node number on the real device

attr a string that specifies node attributes for devices with programmable nodes. Valid strings are listed in Table 7-1 later in this chapter.

Remarks

The node declaration statement indicates the correspondence between identifiers used in a module and the internal nodes of a real device. Nodes are "pseudo-pins": internal signals that are not accessible on the device's external pins, but that are needed to program otherwise inaccessible fuses. The device nodes supported by ABEL-PLD are listed in the appendix, "Programmable Logic Device Information." The declaration can also specify node attributes for devices with programmable node characteristics.

When lists of *node_id* and *node #* are used in one node declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one node number associated with each identifier listed.

When more than one device is declared in one module, the optional IN portion of the node declaration must be used so that the nodes are associated with the proper device.

A device declaration must be made before the node declarations. The ending semicolon is required. The following example declares three nodes A, B, and C in the device U15.

```
A, B, C NODE IN U15 21,22,23 ;
```

The node attribute string, attr, specifies node attributes. Attributes can be defined in this way or with the ISTYPE statement. The ISTYPE statement and attributes are discussed later in this chapter.

The node declaration,

```
B NODE 22 = 'pos,com' ;
```

specifies that node 22 has positive polarity and is a combinational node.

If shorthand notation is used to refer to internal device nodes, it is not necessary to declare the node in a declaration statement. For examples of shorthand notation of nodes, refer to the chapter "Advanced Features." The appendix "Programmable Device Information" lists the shorthand notation for all supported nodes.

Constant Declaration Statement

Syntax

id [, id]... = expr[, expr]... ;

id an identifier naming a constant to be used within a module
expr an expression defining the constant value

Remarks

The constant declaration statement defines constants to be used within a module. A constant is an identifier that retains a constant value throughout a module.

The identifiers listed on the left side of the equals sign in the constant declaration statement are assigned the values listed on the right side of the equals sign.

There is a one-to-one correspondence between the identifiers and the expressions listed and there must be one expression for each identifier.

The ending semicolon is required.

Constants are useful when a value must be used many times throughout a module and especially when the value is used many times but might need to be changed during the logic design process. Rather than changing the value throughout the module, the value can be changed once in the declaration of the constant.

Constant declarations may not be self-referencing; for example:

X = X;

will cause errors, as will the declarations

a = b;
b = a;

Some examples of valid constant declarations follow:

ABC = 3 * 17;	" ABC is assigned the value 51
Y = 'Bc' ;	" Y = + H4263 ;
X =.X.;	" X means 'don't care'
ADDR = [1,0,15];	" ADDR is a set with 3 elements
A,B,C = 5,[1,0],6;	" 3 constants declared here
D pin 6;	" see next line
E = [5 * 7,D];	" signal names can be included
G = [1,2]+[3,4];	" set operations are legal
A = B & C;	" operations on identifiers are valid
A = [B,C];	" set and identifiers on right

Macro Declaration Statement and Macro Expansion

Syntax

macro_id MACRO [(*dummy_arg* [,*dummy_arg*]...)] {*block*} ;

macro_id an identifier naming the macro
dummy_arg a dummy argument
block a block

Remarks

The macro declaration statement defines a macro. Macros are used to include ABEL-PLD code in a source file without typing or copying the code everywhere it is needed. A macro is defined once in the declarations section of a module and then used anywhere within the module as frequently as needed. Macros can be used only within the module in which they are declared.

Wherever the *macro_id* occurs, the text in the block associated with that macro will be substituted. With the exception of dummy arguments, all text in the block (including spaces, end-of-lines, etc.) is substituted exactly as it appears in the block.

When debugging your source file, you can use the -e and -p flags to examine macro statements. The -e option causes the parsed and expanded source code to be written to the listing file (or to the display if the -l option is omitted). In addition to the listing information provided by the -e option, the -p flag lists the directives that caused code to be added to the source.

The dummy arguments used in the declaration of the macro allow different actual arguments to be used in the macro each time it is invoked in the module. Within the macro, dummy arguments are preceded by a "?" to indicate that an actual argument will be substituted for the dummy by the ABEL-PLD parser. This is best shown by example.

The equation,

NAND3 MACRO (A,B,C) { !(?A & ?B & ?C)} ;

declares a macro named NAND3 with the dummy arguments A, B and C. The macro defines a three-input NAND gate. When the macro identifier occurs in the source, actual arguments for A, B and C will be supplied.

For example, the equation,

D = NAND3 (Clock,Hello,Busy) ;

brings the text in the block associated with NAND3 into the code, with Clock substituted for ?A, Hello for ?B and Busy for ?C. This results in:

D = !(Clock & Hello & Busy) ;

which is the three-input NAND.

The macro NAND3 has been specified by an equation, but it could have been specified using another ABEL-PLD construct, such as the truth table shown here:

```
NAND3 MACRO (A,B,C,Y)

{ TRUTH_TABLE ( [ ?A, ?B,?C ] -> ?Y )

[ 0 , .X., .X.] -> 1 ;
[ .X., 0 , .X.] -> 1 ;
[ .X., .X., 0 ] -> 1 ;
[ 1 , 1 , 1 ] -> 0 ; } ;
```

In this case, the line,

```
NAND3 (Clock,Hello,Busy,D)
```

causes the text,

```
TRUTH_TABLE ( [Clock,Hello,Busy] -> D )
[ 0 , .X., .X.] -> 1 ;
[ .X., 0 , .X.] -> 1 ;
[ .X., .X., 0 ] -> 1 ;
[ 1 , 1 , 1 ] -> 0 ;
```

to be substituted into the code. This text is a truth table definition of D, specified as the function of three inputs, Clock, Hello and Busy. This is the same function as that given by the equation above. The truth table format is discussed later in this chapter.

Other examples of macros:

```
A macro { W = S1 & s2 & s3 ; } ; "macro w/no dummy args
```

```
B MACRO (d) { !?d } ; "macro w 1 dummy argument
```

and when macros are called in logic descriptions:

```
A
X = W + B (inp) ;
Y = W + B( )C ; "note the blank actual argument
```

resulting in:

```
"note leading space from block in A
W = S1 & S2 & S3 ;
X = W + ! inp ;
Y = W + ! C ;
```

Circular macro references (when a macro refers to itself within its own definition) cause the PARSE program to terminate abnormally with errors. This error can often be detected by examining the PARSE listing file. If errors appear after the first use of a macro, and the errors cannot be easily explained otherwise, check for a circular macro reference.

ISTYPE Declaration Statement

Syntax

signal [,signal]... ISTYPE [IN device_id] 'attr [,attr]...';

signal	a pin or node identifier
device_id	a declared device name identifier indicating the device associated with these nodes
attr	a string that specifies attributes for the signal(s). Valid strings are listed in Table 7-1 later in this chapter.

Remarks

The ISTYPE declaration defines attributes or characteristics of pins and nodes for devices with programmable characteristics. The attributes are used to form correct logic for the device and to optimize equations for the device. If no attributes are defined for a device with programmable characteristics the device defaults are applied. If attributes are defined for a device without programmable characteristics, an error is reported by the language processor. An example of ISTYPE declaration usage is presented in the design example file altera.abl.

When more than one signal is listed on the left side of the ISTYPE declarations, all attributes listed on the right side of the ISTYPE declaration are applied to each of the signals.

- When more than one device is declared in one module, the optional IN DEVICE_id portion of the ISTYPE must be used so the signals (pins or nodes) are associated with the correct device.
- A device declaration must be made before the ISTYPE declaration appears in the source file.
- Declarations of the pin and node names used in the ISTYPE declaration must be made before the ISTYPE declaration.

An example of the ISTYPE declaration follows:

```
F0, A istype 'neg, latch' ;
```

This declaration statement defines F0 and A as negative polarity latches. Both F0 and A had to have been defined previously in the module.

Attribute Definitions

Definitions of each of the attributes follows. All attributes may be entered in uppercase, lowercase, or mixed-case letters. Table 7-1 summarizes the attributes and their use. Each attribute is then discussed in more detail.

See the section "Polarity Control" in the chapter "Design Considerations" for more information on polarity control.

The allowed register attributes are given below. The columns marked "Pin," "Node" and "Istype" indicate whether the attribute is supported for pin, node and istype statements.

Table 7-1
Attributes for Pin, Node and ISTYPE

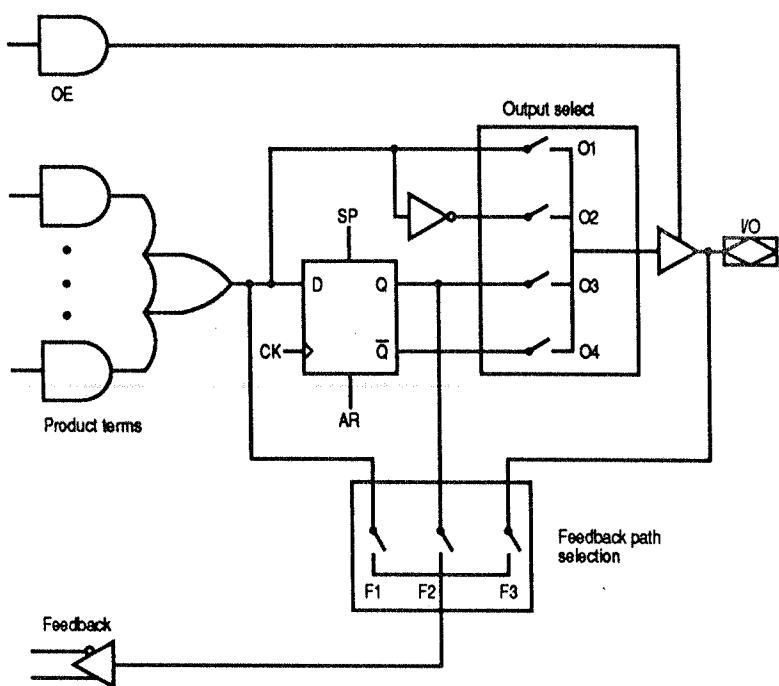
Description	Attr.	Pin	Node	Istype
positive polarity	pos	●	●	●
negative polarity	neg	●	●	●
registered signal	reg	●	●	●
D-type register	reg_d	●	●	●
JK-type register	reg_jk	●	●	●
RS-type register	reg_rs	●	●	●
T-type register	reg_t	●	●	●
combinational signal	com	●	●	●
feedback from pin	feed_pin	●	●	●
feedback from register	feed_reg	●	●	●
feedback from OR-gate	feed_or	●	●	●
single fuse node	fuse		●	●
pin controlled node	pin		●	●
array controlled node	sgn		●	●
product term sharing	share			●

Pos (Positive Polarity)	Pos indicates that the associated input or output has positive polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for that polarity.
Neg (Negative Polarity)	Neg indicates that the associated input or output has negative polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for negative polarity. Neg may be entered in uppercase, lowercase, or mixed-case letters.
Reg (Registered Signal)	Reg indicates that the associated input or output is registered rather than combinational. The device will be programmed for this condition, and the signal will change on application of the clock.
Reg_type (Registered Type)	This attribute indicates the type of register to use for a signal that has programmable register types. The device will be programmed for the indicated register type.
Com (Combinational Signal)	Com indicates that the associated input or output is a combinational or combinatorial signal. The device will be programmed for this condition.
Feed_pin, Feed_reg, and Feed_or (Feedback Specifications)	Some available devices allow definition of the internal feedback paths. Feedback can occur from the output pin, the output of an internal register, or from the OR-gate output, as shown in Figure 7-2. The feed_pin, feed_reg, or feed_or attributes specify which feedback path to use: from the pin, the register output, or the OR-gate, respectively. Only one of the feedback attributes can be in effect at any given time; if more than one feedback attribute is specified for the same signal, the last attribute specified takes effect. Also, feedback attributes are valid only for devices with feedback paths that can be defined by the user. Feedback attributes are valid only for outputs. The attribute may be specified in uppercase, lowercase, or mixed-case characters.
Pin, Eqn, or Fuse (selectable node type)	Some devices allow internal device features, such as output enables or clocks to be configured in one of several ways. The Pin, Eqn, and Fuse attributes specify which type of feature is desired for the indicated node. For example,

Q0.oe istype pin

specifies that the output enable is pin controlled.

Figure 7-2
Feedback Paths for an E0310



F1, feedback from OR-gate; selected by Feed_or
 F2, feedback from register; selected by Feed_reg
 F3, feedback from output pin; selected by Feed_pin

Library Statement

Syntax

LIBRARY 'name'

name a string that specifies the name of file, excluding the file extension

Remarks

The LIBRARY statement causes the contents of the indicated file to be inserted in the ABEL-PLD source file. The insertion begins at the LIBRARY statement.

The file extension of '.inc' is appended to the name specified, and the resulting filename is searched for. If no file is found, ABEL-PLD will attempt to find the file in the abel3lib.inc library file. Refer to "Utilities" in the chapter "Using ABEL-PLD" for more information on libraries.

Equations Statement

Syntax

EQUATIONS [IN *device_id*]

device_id a previously declared device identifier that indicates the device associated with these equations

Remarks

The equations statement defines the beginning of a group of equations associated with a device. Equations specify logic functions with Boolean algebra.

The equations statement is followed by the equations associated with the device indicated by the device identifier. In modules with only one device, the device identifier is optional; in all other cases it must be specified so that the equations will be applied toward the correct device.

The equations following the equation statement are any valid ABEL-PLD equations as described in the chapter "Language Elements."

A sample equations section follows:

```
EQUATIONS IN IC13
A = B & C # A ;
[W,Y] = 3 ;
IF = B == C ;
```

Truth Table Statement

Truth tables are another way to describe logic designs with ABEL-PLD and may be used in lieu of, or in addition to, equations and state diagrams. Truth tables specify outputs as functions of different input combinations in a tabular form. A truth table is specified with a header describing the format of the table and with the table itself.

Header Syntax

```
TRUTH_TABLE [ IN device_id ] ( inputs -> outputs )
or
TRUTH_TABLE [ IN device_id ] ( inputs > reg_outs )
or
TRUTH_TABLE [ IN device_id ] ( inputs > reg_outs -> outputs )
```

device_id	an identifier naming the device associated with the truth table
inputs	the inputs to the logic function
outputs	the outputs from the logic function
reg_outs	the registered (clocked) outputs
->	the input to output function for combinational outputs.
>	the input to output function for registered outputs.

Remarks

The truth table header can have one of the three forms shown above, depending on whether the device has registered or combinational outputs or both.

In all three forms, the device identifier is required when more than one device is declared in a module. An example of truth table usage is presented in the design example file led7.abl.

The inputs and outputs (both registered and combinational) of a truth table are either single signals, or, more frequently, sets of signals. If only one signal is used as either the input or output, its name is specified. Sets of signals used as inputs or outputs are specified in the normal set notation with the signals surrounded by brackets and separated by commas (see "Sets" in the chapter "Language Elements").

The syntax shown in the first form defines the format of a truth table with simple combinational outputs. The values of the inputs determine the values of the outputs.

The second form describes a format for a truth table with registered outputs. The symbol ":" preceding the outputs distinguishes these outputs from the combinational ones. Again the values of the inputs determine the values of the outputs, but now the outputs are registered or clocked: they will contain the new value (as determined by the inputs) after the next clock pulse. The third form is more complex, defining a table with both combinational and registered outputs. It is important in this format to make sure the different specification characters "-" and ":" are used for the different types of outputs.

Truth Table Format

The truth table is specified according to the form described within the parentheses in the header. The truth table is a list of input combinations and resulting outputs. All or some of the possible input combinations may be listed.

As an example, the following truth table defines an exclusive-OR function with two inputs (A and B), one enable (en), and one output (C):

TRUTH_TABLE IN IC16 ([en,A,B] -> C)

```
[0,,X,,X,] -> .X; " don't care w/enable off
[1, 0 , 0 ] -> 0 ;
[1, 0 , 1 ] -> 1 ;
[1, 1 , 0 ] -> 1 ;
[1, 1 , 1 ] -> 0 ;
```

All values specified in the table must be constants, either declared, numeric, or the special constant, ".X." Each line of the table (each input/output listing) must end with a semicolon.

The header defines the names of the inputs and outputs; the table defines the values of inputs and the resulting output values.

The following example shows a truth table description of a simple state machine with four states and one output. The current state is described by signals A and B, which are put into a set. The next state is described by the registered outputs C and D, which are also collected into a set. The single combinational output is signal E. The machine simply counts through the different states, driving the output E low when A equals 1 and B equals 0.

TRUTH_TABLE IN IC17 ([A,B] :> [C,D] -> E)

```
0 : 1 -> 1 ;
1 : 2 -> 0 ;
2 : 3 -> 1 ;
3 : 0 -> 1 ;
```

Note that the input and output combinations are specified by a single constant value rather than by set notation. This is equivalent to

```
[0,0] > [0,1] -> 1;  
[0,1] > [1,0] -> 0;  
[1,0] > [1,1] -> 1;  
[1,1] > [0,0] -> 1;
```

Programmable Polarity Registers

When using state diagrams and truth tables for programmable polarity devices, the default polarity used by ABEL-PLD software (if no polarity is specified via the ISTYPE statement) is negative. Existing ABEL-PLD source files that assume positive polarity for these devices may require the addition of an ISTYPE statement to force positive polarity. See example led1.abl.

State Descriptions

An alternative to describing logic with equations or truth tables is to use a state description. The state description easily describes the operation of a sequential state machine implemented with programmable logic.

The specification of a state description requires the use of the STATE_DIAGRAM construct, which defines the state machine, and the IF-THEN-ELSE, CASE, and GOTO statements which determine the operation of the state machine.

The STATE_DIAGRAM construct is discussed first, and then the syntaxes of the IF-THEN-ELSE, CASE, and GOTO statements are presented.

Syntax of the WITH-ENDWITH statement, which permits the specification of equations in terms of transitions, is also presented.

STATE_DIAGRAM Statement

Syntax

```
STATE_DIAGRAM [ IN device_id ] state_reg  
[ -> state_out ]  
[ STATE state_exp : [ equation ]  
[ equation ]  
:  
:  
:  
trans_stmt ... ]
```

device_id	an identifier specifying the associated device
state_reg	an identifier or set of identifiers specifying the signals that determine the current state of the machine.
state_out	an identifier or set of identifiers that determine the next state of the machine (for designs with external registers)
state_exp	an expression giving the current state
equation	a valid equation that defines the state machine outputs
trans_stmt	an IF-THEN-ELSE, CASE or GOTO statement, optionally followed by WITH-ENDWITH transition equations.

Remarks

The STATE_DIAGRAM construct defines a state machine associated with a device in a module. The state machine starts in one of the states indicated by *state_exp*. The equations listed after that *state_exp* are evaluated, and the transition statement (*trans_stmt*) is evaluated after the next clock, causing the machine to advance to the next state.

Equations associated with a state are optional. Each state must have a transition statement. If none of the transition conditions for a state is met, the next state is undefined. (For some devices, undefined state transitions cause a transition to the cleared register state.) As many states as are needed can be specified.

When there is more than one device declared in a module, the "IN *device_id*" section is required.

Following is an example of a simple state machine that advances from one state to the next, setting the output to the current state, and then starting over again. Note that the states do not need to be specified in any particular order. Note also that state 2 is identified by an expression rather than by a constant. The state register is composed of the signals a and b.

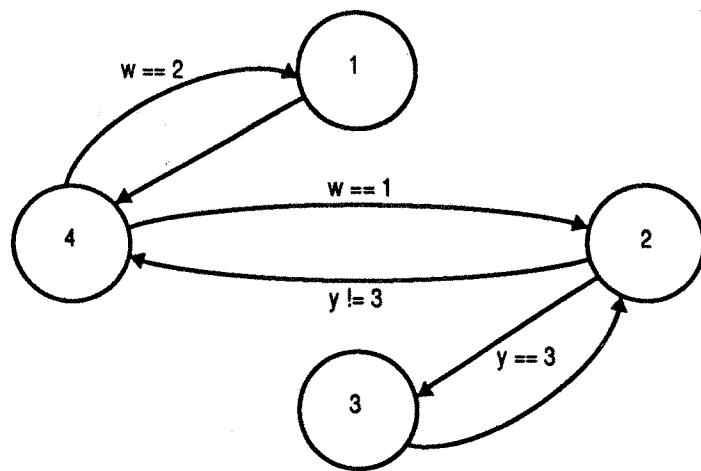
```
state_diagram in u15 [a,b]
state 3 : y = 3 ;
    goto 0 ;
state 1 : y = 1 ;
    goto 2 ;
state 0 : y = 0 ;
    goto 1 ;
state 1 + 1 : y = 2 ;
    goto 3 ;
```

The next state diagram specifies a more complex state machine where the *state_reg* is specified with a constant set containing the signals a and b. Assuming that the state machine starts in state 1 (a = 0,b = 1), the sequence of states will be

```
1,4,2,3,2,4,
1,4,2,3,2,4,
1...
current_state = [a, b] "constant declaration
STATE_DIAGRAM current_state
state 1: w = 1 ;
    y = 1 ; GOTO 4 ;
state 2 : IF y==3 THEN 3
    ELSE 4 ;
state 3 : w = 2 ;
    y = w ;
    GOTO 2 ;
state 4 : y = 3 ;
    CASE w==1: 2;
    w==2: 1;
    ENDCASE ;
```

Figure 7-3 shows the pictorial state diagram for this state machine.

Figure 7-3
Pictorial State Diagram



IF-THEN-ELSE Statement

Syntax

`IF expression THEN state_exp [ELSE state_exp] ;`

expression

any valid expression

state_exp

an expression identifying the next state,
optionally followed by WITH-ENDWITH
transition equations.

Remarks

The IF-THEN-ELSE statement is an easy way to describe the progression from one state to another in a state machine. The expression following the IF keyword is evaluated, and if the result is true, the machine goes to the state indicated by the *state_exp* following the THEN keyword. If the result of the expression is false, the machine advances to the state indicated by the ELSE keyword.

Any number of IF statements may be used in a given state, and the use of the ELSE clause is optional.

Examples:

```

if A==B then 2 ; "if A equals B goto state 2
if x-y then j else k; "if x-y is not 0 goto j, else goto k
if A then b*c; "if A is true (non-zero) goto state b*c

```

Chained IF-THEN-ELSE Statements

Syntax

```
IF expression THEN state_expression
ELSE IF expression THEN state_expression
ELSE IF expression THEN state_expression
ELSE state_expression ;
```

Remarks

Additional IF-THEN-ELSE statements can be chained to the ELSE clause of an IF-THEN-ELSE statement. Any number of IF-THEN-ELSE statements can be chained, but the final statement must end with a semicolon. An example of chained IF-THEN-ELSE statements follows:

```
if a then 1
else
if b then 2
else
if c then 3
else 0 ;
```

Often, chains of mutually exclusive IF-THEN-ELSE statements can be more clearly expressed with a CASE statement. The chained IF-THEN-ELSE statement is intended for situations (such as the preceding example) where the conditions are not mutually exclusive.

See above for examples of IF-THEN-ELSE usage as they are used in a state diagram.

CASE Statement

Syntax

```
CASE [ expression : state_exp; ]
[ expression : state_exp; ]
[ expression : state_exp; ]
:
ENDCASE ;
```

expression

any valid ABEL-PLD expression

state_exp

an expression identifying the next state,
optionally followed by WITH-ENDWITH
transition equations.

Remarks

The CASE statement is an easy way to indicate the transitions of a state machine when there are multiple possible conditions that affect the state transitions.

The expressions contained within the CASE-ENDCASE keywords must be mutually exclusive, meaning that only one of the expressions can be true at any given time. If two or more expressions within the same CASE statement are true, the resulting equations are undefined.

The state machine will advance to the state indicated by *state_exp* following the expression that produces a true value. If no expression is true, the result is undefined, and the resulting action depends on the device being used. (For devices with D flip-flops, the next state will be the cleared register state.) For this reason, you should be sure to cover all possible conditions in the CASE statement expressions. If the expression produces a numeric rather than a logical value, 0 is false, and any non-zero value is true.

Example:

```
case a == 0 : 1 ;
a == 1 : 2 ;
a == 2 : 3 ;
a == 3 : 0 ;
endcase ;
```

More CASE statement examples are presented above.

GOTO Statement

Syntax	<code>GOTO state_exp ;</code>
	state_exp an expression identifying the next state, optionally followed by WITH-ENDWITH transition equations.

Remarks

The GOTO statement causes an unconditional transition to the state indicated by *state_exp*.

Example:

```
GOTO 0 ; "goto state 0
GOTO x+y ; "goto the state x + y
```

WITH-ENDWITH Statement

Syntax	<i>transition_stmt</i> <i>state_exp</i> WITH <i>equation</i> <i>[equation]</i> ; <i>ENDWITH</i> ;
	transition_stmt IF, ELSE, or CASE statement
	state_exp the next state
	equation an equation for state machine outputs

Remarks

The WITH-ENDWITH statement, when used in conjunction with the IF-THEN or CASE statement, allows output equations to be written in terms of transitions; for example:

```
state 5 : IF a == 1 then 1
  WITH x := 1 ;
    y := 0 ;
  ENDWITH;
ELSE 2 WITH
  x := 0 ;
  y := 1 ;
ENDWITH ;
```

XOR_Factor**Syntax**

XOR_Factor
equation

Remarks

The XOR_Factor feature in ABEL-PLD 3.2 software is a technique for converting a sum of products (SOP) equation into an exclusive OR (XOR) equation. The resulting equation contains the sum of product functions, which when exclusive ORed together have the same function as the original equation. The XOR_Factor equation you provide will be divided into the original equation, placing the factor (or its complement) on one side of the XOR and the remainder on the other.

The sum of products form isn't the only way an equation may be expressed. Another method is to express the function in groups that cannot be factored into smaller parts. For example,

$$F = A \& B \# C \& B = (A \# C) \& B.$$

The factors ($A \# C$) and (B) are known as "level 0 kernels," meaning there are no additional factors. An analogy is that the number 30 can be expressed with the prime number factors $2 * 3 * 5$, prime numbers being analogous to level 0 kernels.

Another way to find initial candidate factors is to use a programmable polarity device such as a P22V10. Run the design through ABEL-PLD Reduce with both pos and neg and look at the reduced equations.

After deciding the best XOR_Factor, remember to revise the source file to use the XOR device for the final design.

Example 1

As an example of factors, consider the following:

```
!Q16 =   A & B & !D
        # A & B & !C
        # !B & C & D
        # !A & C & D;
```

Reordering the product terms as shown below indicates that (A & B) and (C & D) are factors. It is not necessary to factor the equation completely into level 0 kernels, but to find some good candidates, then try them in an ABEL-PLD file. A workable solution should turn up after a few iterations.

```
!Q16 = A & B & (!C # !D)
# (!A # !B) & C & D;
```

If we run the following source file through ABEL-PLD software, the program will reduce the equations according to the XOR_Factor given, A & B.

```
module XORfact

    xorfact device 'P20X10';

    Clk,OE    pin 1,13;
    A,B,C,D  pin 2,3,4,5;
    Q16       pin 16;

    XOR_Factor
    - Q16 = A & B;

    equations
        !Q16 := A & B &      !D
                  # !B & C & D
                  # !A &      C & D
                  # A & B & !C;

    end
```

Using A & B as the XOR_Factor, the reduced equations will be as shown.

```
!Q16 := ((A & B) $ (C & D));
```

Example 2

The example below uses a more complex high level equation:

```

Data  = [D7,D6,D5,D4,D3,D2,D1,D0];
Count = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];

Mode  = [I1,I0];
Clear = [ 0, 0];
Hold   = [ 0, 1];
Load   = [ 1, 0];
Inc    = [ 1, 1];

equations
  Count := (Count + 1) & (Mode == Inc) & !CarryIn
           # (Count      ) & (Mode == Inc) & CarryIn
           # (Count      ) & (Mode == Hold)
           # (Data       ) & (Mode == Load)
           # (0          ) & (Mode == Clear);

```

When reduced for an active high or pos device, the following sum of products equation is produced.

```

Q7 :=
  Q7 & I0 & !I1
  # Q7 & I0 & CarryIn
  # Q7 & !Q0 & I0
  # Q7 & !Q1 & I0
  # Q7 & !Q2 & I0
  # Q7 & !Q3 & I0
  # Q7 & !Q4 & I0
  # Q7 & !Q5 & I0
  # Q7 & !Q6 & I0
  # D7 & !I0 & I1
  # !Q7 & Q6 & Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & I0 &
  I1 & !CarryIn;

```

Upon inspection, a good level 0 kernel is ($Q7 \& I0$).

```

Q7 :=
  (!I1
  # CarryIn
  # !Q0
  # !Q1
  # !Q2
  # !Q3
  # !Q4
  # !Q5
  # !Q6) & (Q7 & I0))
  # D7 & !I0 & I1
  # !Q7 & Q6 & Q5 & Q4 & Q3 & Q2 & Q1 & Q0 & I0 &
  I1 & !CarryIn;

```

Using ($Q7 \& I0$) as the XOR_Factor, the reduced equation is

```

!Q7 := ((!I0 # !Q7) $ (D7 & !I0 & I1 # Q6 & Q5 & Q4 &
Q3 & Q2 & Q1 & Q0 & I0 & I1 & !CarryIn));

```

Fuses Section

Syntax

```
FUSES [ IN device_id ]  
fuse_number = fuse_value ;  
or  
fuse_number_set = fuse_value ;
```

device_id	identifier specifying associated device
fuse_number	fuse number obtained from logic diagram of device
fuse_number_set	set of fuse numbers contained in square brackets
fuse_value	number indicating state of fuse(s)

Remarks

The FUSES section of the source file provides a means for explicitly declaring the state of any fuse in the associated device. For example:

```
FUSES  
3552 = 1 ;  
[3478...3491] = ^Hff;
```

Fuse values appearing on the right side of the = symbol can be any number. In the case of only a single fuse number being specified on the left side of the = symbol, the least significant (LSB) bit of the fuse value is assigned to the fuse; a 0 indicates a fuse intact, and a 1 indicates a fuse blown. In the case of multiple fuse numbers, the fuse value is expanded to a binary number and truncated or given leading zeros to obtain fuse values for each fuse number.

When fuse states are specified using the FUSES-section syntax, the resulting fuse values supercede the fuse values obtained through the use of equations, truth tables and state diagrams, and will effect device simulation accordingly. While a high number of fuses can be specified in the FUSES section (typically over 2,000) there is some limit. The maximum number of fuses you can specify varies but is well beyond what is practical when using the FUSES section.

The PC/MS-DOS versions have a limit of 15 fuses per statement (e.g. FUSES[1000..1015] = 0;).

Test Vectors

Syntax

```
TEST_VECTORS [ IN device_id ] [note]
  (inputs -> outputs)
```

```
[invalues -> outvalues; ]
  :
```

device_id	an identifier indicating the device associated with this table
note	a string used to describe the test vectors
inputs	identifier or set of identifiers specifying the names of the input signals to the device, or feedback output signals
outputs	identifier or set of identifiers specifying the output signals from the device
invalues	input value or set of input values
outvalues	output value or set of output values resulting from the given inputs

Remarks

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs. Test vectors are used for simulation of an internal model of the device and functional testing of the programmed device.

A special simulation utility, Simulate, is provided as part of the ABEL-PLD software package. Simulate simulates the operation of the device model by applying the inputs specified in the test vectors to the fuse states created by the language processor. Simulate is discussed further in the chapters "Understanding ABEL-PLD," "Using ABEL-PLD," and in "Advanced Features."

Functional testing of the real device is performed by a logic programmer after a device has been programmed. This can be done because the test vectors become part of the programmer load file that is loaded into the logic programmer.

Test vectors are written for each different device in the module, so that the different characteristics of each device can be taken into account separately during simulation.

Test vectors for a device are specified in a table. The table consists of a header and the vectors themselves. The header defines the format of the table. The vectors specify the input-to-output function.

The form of the test vectors is determined by the header. Each vector is specified in the format described within the parentheses in the header statement. An optional note string can be specified in the header. This note string is often used to describe what the vectors test, and is included as output in the simulation output file, the document output file, and the JEDEC programmer load file.

The table lists input combinations and their resulting outputs. All or some of the possible input combinations can be listed. All values specified in the table must be constants, either declared, numeric or the special constant, ".X.". Each line of the table (each input/output listing) must end with a semicolon.

Following is a simple test vectors table:

```
TEST_VECTORS
( [A,B] -> [C, D] )

[0,0] -> [1, 1] ;
[0,1] -> [1, 0] ;
[1,0] -> [0, 1] ;
[1,1] -> [0, 0] ;
```

The following test vector table is equivalent to the table specified above because values for sets can be specified with numeric constants.

```
TEST_VECTORS
( [A,B] -> [C,D] )

0 -> 3 ;
1 -> 2 ;
2 -> 1 ;
3 -> 0 ;
```

If the signal identifiers used in the test vector header were declared as active-low in the declaration section, then constant values specified in the test vectors will be inverted accordingly.

Chapter 8

Directives

Directives affect the contents of a source file when it is processed. Sections of ABEL-PLD source code can be included conditionally, code can be brought into a source file from another file, and messages can be printed during the processing of a source file. Table 8-1 lists the available directives.

Directives are for the designer who understands the basics of ABEL-PLD and who wants to use more complex structures. Examples of directive use to create test vectors are given in the chapter "Advanced Features."

Some of the directives use arguments to determine conditions. The arguments can be actual arguments, or dummy arguments preceded by question marks. The rules applying to actual and dummy arguments are presented in the chapter "Language Elements."

When debugging your source file, you can use the -e and -p options to examine statements. The -e option causes the parsed and expanded source code to be written to the listing file. Text included by directives is also written. The -p option gives the listing information provided by -e, and also lists the directives that caused code to be added to the source file.

Table 8-1
Directives

@ALTERNATE	@IFDEF	@INCLUDE	@RADIX
@CONST	@IFIDEN	@IRP	@REPEAT
@EXPR	@IFNB	@IRPC	@STANDARD
@EXIT	@IFNDEF	@MESSAGE	@IF
@IFNIDEN	@PAGE	@IFB	

@ALTERNATE Directive

Syntax	@ALTERNATE
Remarks	<p>@ALTERNATE brings an alternate set of operators into effect that duplicate the normal ABEL-PLD operators. This is for users who feel more comfortable with the alternate set because of their familiarity with operators used in other languages.</p> <p>The alternate operators remain in effect until the @STANDARD directive is issued or the end of the module is reached.</p> <p>The alternate operator set is listed in Table 8-2.</p>
Table 8-2 Alternate Operator Set	

ABEL-PLD Operator	Alternate Operator	Description
!	/	NOT
&	*	AND
#	+	OR
\$: + :	XOR
!\$: * :	XNOR

Note that the use of the alternate operator set precludes use of the ABEL-PLD addition, multiplication and division operators because they represent the OR, AND and NOT logical operators in the alternate set.

@CONST (Constant) Directive

Syntax	@CONST <i>id</i> = <i>expression</i> ;	
	<i>id</i>	a valid identifier
	<i>expression</i>	a valid expression
Remarks	<p>@CONST allows new constant declarations to be made in a source file outside the normal (and required) declarations section, before the constants are used in the source file.</p> <p>The @CONST directive is intended to be used inside macros to define internal constants. Constants defined with @CONST override previous constant declarations. Declaring an identifier as a constant in this manner constitutes an error if the identifier was used earlier in the source file as something other than a constant (i.e., a macro, pin, device).</p>	
	<p>Example:</p> <pre>@CONST count = count + 1;</pre>	

@EXIT Directive

Syntax	@EXIT
Remarks	The @EXIT directive causes PARSE to abort processing of the source file with error bits set. (Error bits allow the operating system to determine that a processing error has occurred.)

@EXPR (Expression) Directive

Syntax

`@EXPR [block] expression ;`

block a block

expression a valid expression

Remarks

@EXPR evaluates the given expression, and converts it to a string of digits in the default base numbering system. This string and the block are then inserted into the source file at the point at which the @EXPR directive occurs. The expression must produce a valid number.

Example:

`@expr {ABC} ^B11 ;`

Assuming that the default base is base ten, this example causes the text ABC3 to be inserted into the source file.

@IF Directive

Syntax

`@IF expression block`

expression a valid expression

block a valid block of text

Remarks

@IF is used to include sections of ABEL-PLD source code based on the value resulting from an expression. If the expression is non-zero (logical true), the block of code is included as part of the source.

Dummy argument substitution is valid in the expression.

Example:

`@IF (A > 17) { C = D $ F ; }`

@IFB (If Blank) Directive

Syntax	<code>@IFB (<i>arg</i>) <i>block</i></code>				
	<table> <tr> <td>arg</td><td>an actual argument or a dummy argument preceded by a "?"</td></tr> <tr> <td>block</td><td>a valid block of text</td></tr> </table>	arg	an actual argument or a dummy argument preceded by a "?"	block	a valid block of text
arg	an actual argument or a dummy argument preceded by a "?"				
block	a valid block of text				
Remarks	@IFB includes the text contained within the block if the argument is blank (has 0 characters).				
Examples:					
	<pre> @IFB () { text here will be included with the rest of the source file. } @IFB (hello) { this text will not be included } @IFB (?A) { this text will be included if no value is substituted for A. } </pre>				

@IFDEF (If Defined) Directive

Syntax	<code>@IFDEF <i>id</i> <i>block</i></code>				
	<table> <tr> <td>id</td><td>an identifier</td></tr> <tr> <td>block</td><td>a valid block of text</td></tr> </table>	id	an identifier	block	a valid block of text
id	an identifier				
block	a valid block of text				
Remarks	@IFDEF includes the text contained within the block if the identifier is defined. For example,				
	<pre> A pin 5 ; @ifdef A { Base = ^hE000 ; } "the above assignment is made "because A was defined </pre>				

@IFIDEN (If Identical) Directive

Syntax	@IFIDEN (<i>arg1,arg2</i>) <i>block</i>	
	arg1,2	actual arguments, or dummy argument names preceded by "?"
	block	a valid block of text
Remarks	The text in the block is included in the source file if <i>arg1</i> and <i>arg2</i> are identical.	
	Example:	
	<pre>@ifiden (?A,abcd) { ?A device 'P16R4'; }</pre>	
	A device declaration for a P16R4 is made if the actual argument substituted for A is identical to abcd.	

@IFNB (If Not Blank) Directive

Syntax	@IFNB (<i>arg</i>) <i>block</i>	
	arg	an actual argument, or a dummy argument name preceded by a "?"
	block	a valid block of text
Remarks	@IFNB includes the text contained within the block if the argument is not blank, meaning that it has more than 0 characters.	
	Examples:	
	<pre>@IFNB () { ABEL-PLD source here will not be included with the rest of the source file. }</pre>	
	<pre>@IFNB (hello) { this text will be included }</pre>	
	<pre>@IFNB (?A) { this text will be included if a value is substituted for A}</pre>	

@IFNDEF (If Not Defined) Directive

Syntax	@IFNDEF <i>id block</i>	
	<i>id</i>	an identifier
	<i>block</i>	a valid block of text
Remarks	@IFNDEF includes the text contained within the block if the identifier is undefined. Thus, if no declaration (pin, node, device, macro or constant) has been made for the identifier, the text in the block will be inserted into the source file.	
	Example:	
	<pre>@ifndef A{Base=^hE000;}</pre> <p>"if A is not defined, the block is inserted in the text"</p>	

@IFNIDEN (If Not Identical) Directive

Syntax	@IFNIDEN (<i>arg1,arg2 block</i>)	
	<i>arg1,2</i>	actual arguments, or dummy argument names preceded by "?"
	<i>block</i>	a valid block of text
Remarks	The text in the block is included in the source file if arg1 and arg2 are not identical.	
	Example:	
	<pre>@ifniden (?A,abcd) { ?A device 'P16R8'; }</pre> <p>A device declaration for a P16R8 is made if the actual argument substituted for A is not identical to abcd.</p>	

@INCLUDE Directive

Syntax

`@INCLUDE filespec`

filespec

a string specifying the name of a file, where the specification follows the rules of the operating system being used

Remarks

`@INCLUDE` causes the contents of the file identified by the file specification to be placed in the ABEL-PLD source file. The inclusion will begin at the location of the `@INCLUDE` directive. The file specification can include an explicit drive or path specification that indicates where the file is to be found. If no drive or path specification is given, the file is expected to be on either the default drive or path, or on the drive or path specified by the `-H` parameter.

Example:

`@INCLUDE 'macros.abl'` "file specification

@IRP (Indefinite Repeat) Directive

Syntax

`@IRP dummy_arg (arg [,arg]...) block`

dummy_arg	a dummy argument
arg	an actual argument, or a dummy argument name preceded by a "?"
block	a block of text

Remarks

`@IRP` causes the block to be repeated in the source file *n* times, where *n* equals the number of arguments contained in the parentheses. Each time the block is repeated, the dummy argument takes on the value of the next successive argument.

For example:

`@IRP A (1, ^H0A,0) {B = ?A ; }`

results in:

```
B = 1 ;
B = ^H0A ;
B = 0 ;
```

which is inserted into the source file at the location of the `@IRP` directive. Multiple assignments to the same identifier cause an implicit OR to occur.

Note that if the directive is specified like this:

`@IRP A (1,^H0A,0) {B = ?A ; }`

the resulting text would be:

```
B = 1 ; B = ^H0A ; B = 0 ;
```

The text appears all on one line because the block in the `@IRP` definition contains no end-of-lines. Remember that end-of-lines and spaces are significant in blocks.

@IRPC (Indefinite Repeat, Character) Directive

Syntax	<code>@IRPC <i>dummy_arg</i> (<i>arg</i>) <i>block</i></code>
	<i>dummy_arg</i> a dummy argument
	<i>arg</i> an actual argument, or a dummy argument name preceded by a "?"
	<i>block</i> a block

Remarks @IRPC causes the block to be repeated in the source file n times, where n equals the number of characters contained in arg. Each time the block is repeated, the dummy argument takes on the value of the next successive character.

For example:

```
@IRPC A (Cat)
{B = ?A ;
}
```

results in:

```
B = C ;
B = a ;
B = t ;
```

which is inserted into the source file at the location of the @IRPC directive.

@MESSAGE Directive

Syntax	<code>@MESSAGE '<i>string</i>'</code>
	<i>string</i> any valid string

Remarks @MESSAGE prints a message specified in string at the terminal. This can be used to monitor the progress of the PARSE step of the language processor.

Example:

```
@message 'Includes completed'
```

@PAGE Directive

Syntax	<code>@PAGE</code>
Remarks	Send a form feed to the parser listing file. If no listing is being created, @PAGE has no effect.

@RADIX Directive

Syntax	<code>@RADIX <i>expr</i> ;</code>
	<i>expr</i> a valid expression that produces the number 2, 8, 10 or 16 to indicate a new default base numbering.
Remarks	@RADIX is used to change the default base numbering system. The normal default is base 10 (decimal). This directive is useful when many numbers need to be specified in a base other than 10, say base 2. The @RADIX directive can be issued and all numbers that do not have their base explicitly stated are assumed to be in the new base, in this case, base 2. (See the chapter "Language Elements.")
	The newly specified default base stays in effect until another @RADIX directive is issued or until the end of the module is reached.
	When the default base is set to 16, all numbers in that base that begin with an alphabetic character must begin with leading zeroes.
Examples:	<pre>@radix 2 ; "change default base to binary @radix 11011/11 + 1 ; "change back to decimal</pre>

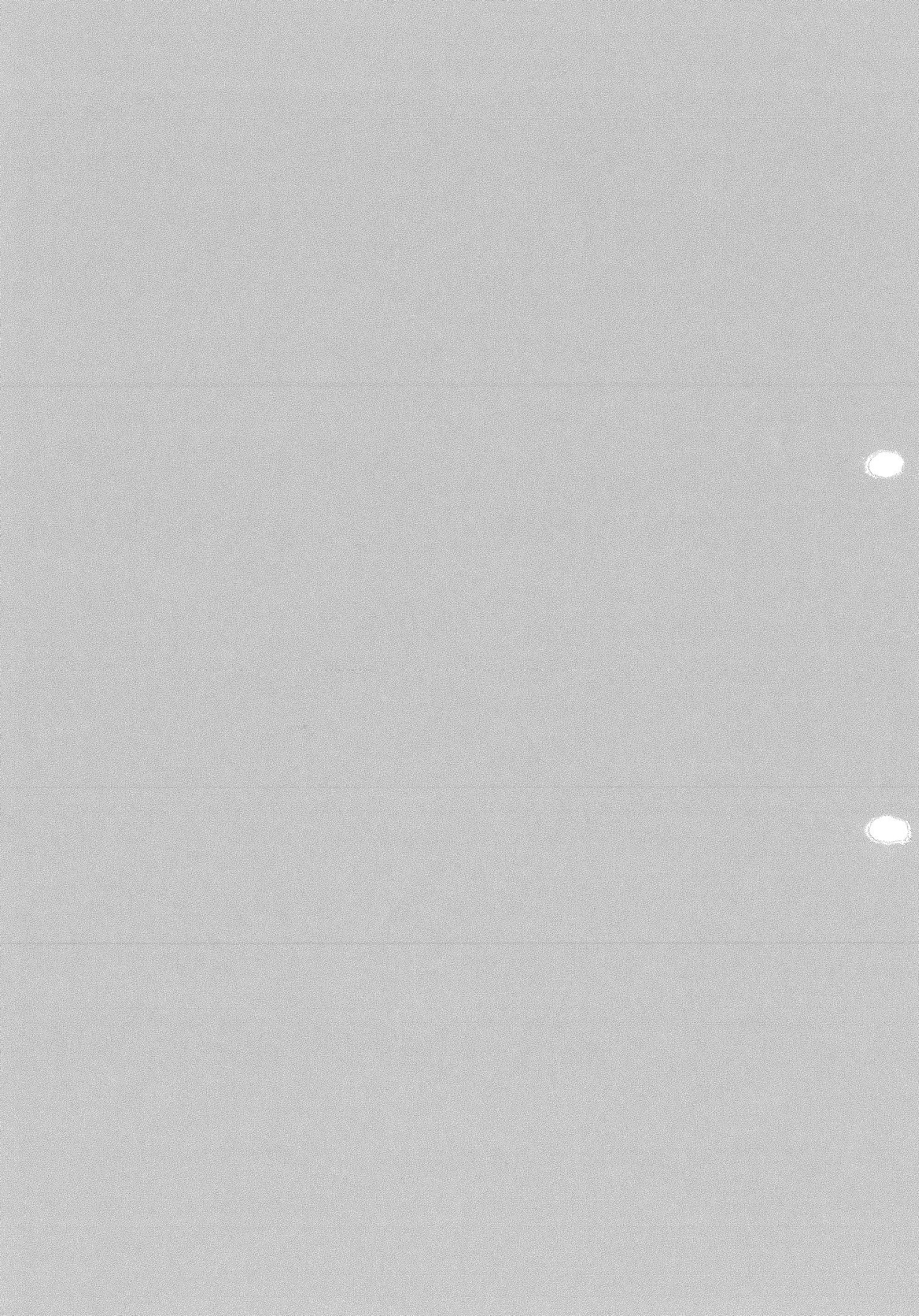
@REPEAT Directive

Syntax	<code>@REPEAT <i>expr</i> <i>block</i></code>
	<i>expr</i> a valid expression that produces a number
	<i>block</i> a block
Remarks	<p>@REPEAT causes the block to be repeated n times, where n is specified by the constant expression.</p> <p>The following use of the repeat directive,</p> <pre>@repeat 5 {H,}</pre> <p>results in the text "H,H,H,H,H," being inserted into the source file. The @REPEAT directive is useful in generating long truth tables and sets of test vectors. Examples of @REPEAT usage can be found in the Applications Guide.</p>

@STANDARD Directive

Syntax	<code>@STANDARD</code>
Remarks	<p>@STANDARD switches the operators in effect back to the ABEL-PLD standard operators from the alternate set. The alternate set is chosen with the @ALTERNATE directive.</p>

Applications Guide



Chapter 9

Design Considerations

This chapter contains general information and considerations that you may find useful as you design logic with ABEL-PLD. This chapter contains

- Exclusive OR Equations
- Polarity Control
- Polarity Control in State Machines
- Register Preloads in the Simulator
- Assigning Pins
- State Machines

Exclusive OR Equations

Previous versions of ABEL-PLD required the user to supply equations in the exact polarity required for the XOR device. If the equations were not in the correct form they would be converted into a sum of products equation with no XORs. ABEL-PLD 3.2 will now map any polarity of XOR equation into any XOR device.

For example, the following active high equation will not map into an active low XOR device:

(1) Q17 := (A & B # C) \$ (D & E);

However, the equation can be transformed by exclusive ORing both sides of the equations with 1, allowing it to be mapped into an active low XOR device.

- (2) $Q17 \$ 1 := ((A \& B \# C) \$ (D \& E)) \$ 1;$
- (3) $\overline{Q}17 := (A \& B \# C) \$ (\overline{D} \& \overline{E});$
- (4) $\overline{Q}17 := (A \& B \# C) \$ (\overline{D} \# \overline{E});$

In equation (3), the second part of the equation was complemented (XOR with 1). The active low form shown in equation (4) performs the same function as equation (1). Equation (4) has two product terms XORed with two product terms and will fit into an active low P20X10. See Chapter 4, "Advanced Features" for more information on selecting an XOR Factor.

Note: *The Fusemap module will automatically swap the parts of an XOR equation to fit devices with an unequal number of product terms.*

Polarity Control

A powerful feature in ABEL-PLD is that the user can write active high equations and let ABEL-PLD convert the logic function so it fits in an active low device. (This also works for active low equations in active high devices.) The following will help explain what happens in this process.

A single logic function may be expressed with many different equations. For example, all three equations below for F1 are equivalent.

- (1) $F1 = (A \& B);$
- (2) $\overline{F}1 = !(A \& B);$
- (3) $\overline{F}1 = \overline{A} \# \overline{B};$

In the example above, equation (3) uses two product terms, while equation (1) requires only one. This logic function will use fewer product terms in an active high device such as the P10H8 than in an active low device such as the P10L8. The logic function performed from input pins to output pins will be the same for both polarities.

Not all logic functions are best optimized to active high. For example, the active low form of F2, equation (3), uses fewer product terms than equation (2).

- (1) $F2 = (A \# B) \& (C \# D);$
- (2) $F2 = (A \& C) \# (A \& D) \# (B \& C) \# (B \& D);$
- (3) $\overline{F}2 = (\overline{A} \& \overline{B}) \# (\overline{C} \& \overline{D});$

Programmable polarity devices are popular because they can provide a mix of active low and active high outputs to achieve the best fit.

Polarity Control in ABEL-PLD

In ABEL-PLD software, the polarity of the equation is controlled with the pos or neg attribute used with the pin assignment or with an istype statement. The pos and neg only control the sum of products conversion in the Reduce module; the logic function of the equation remains the same. An istype neg will force the polarity of the output (left side of the equation) to be active low. The pos forces an active high polarity.

*Note: The pos and neg have no direct control of polarity in Fusemap.
The polarity on the left side of the reduced equation controls the setting of the polarity in programmable polarity devices.*

In an active high device (P16H8), all equations will be active high (pos by default).

$F2 = (A \& C) # (A \& D) # (B \& C) # (B \& D);$

In an active low device (P16L8), all equations will be active low (neg by default).

$\text{!}F2 = (\text{!}A \& \text{!}B) # (\text{!}C \& \text{!}D);$

In a programmable polarity device, the pos or neg can be used to force the equations into the desired form. If pos or neg are not used, the polarity depends on the placement of ! (not operator) on the left side of the equation in Transfor (use the -Q1 option in Document to get the Transfor equation). Since the polarity produced by Transfor is difficult to determine, pos or neg should always be specified.

Polarity Control in State Machines

In addition to controlling the polarity of the sum of product conversion in Reduce, the pos and neg attributes control the state diagram equations generated by Parse.

The following examples show the equations generated by various combinations of polarity and state assignment.

Example 1: Active High Device

```

Q1,Q0 istype 'reg_D,pos';
state_diagram [Q1,Q0]

State 0: IF (In2) THEN 1 ELSE 3;
State 1: GOTO 0;
State 3: GOTO 0;

```

Parse will generate equations similar to the following:

Present State	Inputs	Next State
$[Q_1, Q_0] := ([Q_1, Q_0] == 0) \& (In_2) \& [0,1];$	"IF IN2 THEN 1	
$[Q_1, Q_0] := ([Q_1, Q_0] == 0) \& (!In_2) \& [1,1];$	"IF !IN2 THEN 3	
$[Q_1, Q_0] := ([Q_1, Q_0] == 1) \&$	$[0,0];$	"GOTO 0
$[Q_1, Q_0] := ([Q_1, Q_0] == 3) \&$	$[0,0];$	"GOTO 0

Because a D flip flop defaults to 0 if none of its input product terms are true, only the Next State bits that have a value of 1 need a product term. Q1 needs a product term from the second equation and Q0 needs product terms for the first two equations. For active high or pos state machines, Next State bits equal to 0 are free, while next state bits equal to 1 require a product term.

Transformed Equations for active high device

$$\begin{aligned} Q_1 &:= IQ_1 \& IQ_0 \& !In_2; \\ Q_0 &:= (IQ_1 \& IQ_0 \& In_2 \# IQ_1 \& IQ_0 \& !In_2); \end{aligned}$$

Reduced Equations for active high device

$$\begin{aligned} Q_1 &:= (IQ_1 \& IQ_0 \& !In_2); \\ Q_0 &:= (IQ_1 \& IQ_0); \end{aligned}$$

After reduction, the In2 variable is removed from the Q0 equation and the state machine requires a total of 2 product terms.

Example 2: Active Low Device

```
Q1, Q0 istype 'reg_D, neg';
state_diagram [Q1, Q0]
State 0: IF (In2) THEN 1 ELSE 3;
State 1: GOTO 0;
State 3: GOTO 0;
```

Parse will generate equations similar to the following:

Present State	Input	Next State
$[!Q_1, !Q_0] := ([Q_1, Q_0] == 0) \& (In_2) \& [1,0];$	"IF IN2 THEN 1	
$[!Q_1, !Q_0] := ([Q_1, Q_0] == 0) \& (!In_2) \& [0,0];$	"IF !IN2 THEN 3	
$[!Q_1, !Q_0] := ([Q_1, Q_0] == 1) \&$	$[1,1];$	"GOTO 0
$[!Q_1, !Q_0] := ([Q_1, Q_0] == 3) \&$	$[1,1];$	"GOTO 0

The ABEL-PLD state diagram syntax produces equations that place the state values on the device pins. The active low device has an inverter between the sum of products OR gate and the output pin. Notice that all the Next State bits are inverted, that is, Next State 0 is [1,1]. This means that Q1 needs three product terms and Q0 needs two product terms. In an active low device, Next State bits of 1 in the ABEL-PLD source file are free, Next State bits of 0 will require a product term.

Transformed Equations for active low device

```
!Q1 := ((!Q1 & !Q0 & !In2 # !Q1 & Q0) # Q1 & Q0);
!Q0 := (!Q1 & Q0 # Q1 & Q0);
```

Reduced Equations for active low device

```
!Q1 := (Q0 # !Q1 & !In2);
!Q0 := (Q0);
```

After reduction, the state machine requires a total of 3 product terms.

Example 3: Active Low Device with New State Assignment

```
Q1,Q0 istype 'reg_D,neg';
state_diagram [Q1,Q0]
State 3: IF (In2) THEN 1 ELSE 0;
State 1: GOTO 3;
State 0: GOTO 3;
```

If the actual state values at the pins are not critical, this state machine can use two product terms in an active low device.

Equations for active low device

Present State	Input	Next State
[!Q1,!Q0] := ([Q1,Q0] == 3) & (In2) & [1,0]; "IF IN2 THEN 1		
[!Q1,!Q0] := ([Q1,Q0] == 3) & (!In2) & [1,1]; "IF !IN2 THEN 0		
[!Q1,!Q0] := ([Q1,Q0] == 1) & [0,0]; "GOTO 3		
[!Q1,!Q0] := ([Q1,Q0] == 0) & [0,0]; "GOTO 3		

Transformed Equations for active low device

```
!Q1 := (Q1 & Q0 & In2 # Q1 & Q0 & !In2);
!Q0 := (Q1 & Q0 & !In2);
```

Reduced Equations for active low device

```
!Q1 := (Q1 & Q0);
!Q0 := (Q1 & Q0 & !In2);
```

Register Preloads in the Simulator

When using a preload vector as the first test vector in the simulator, and the device being used has asynchronous presets, it may be desirable to use a "dummy" vector before the preload vector (i.e. all "don't cares"). The dummy vector prevents the possibility of the simulator initialization destroying the preload data. Care should be taken when setting don't cares to 1; when preloading registers to the 1 state, be sure to define values for the resets of the preloaded registers. If the resets are not defined, the registers will be reset after the preload operation, and the preloaded data will be lost.

Assigning Pins

The following are pin assignment errors that cause errors in ABEL-PLD software:

- Assigning the same pin number to two signals (A1, A0 pin 17,17;).
- Assigning invalid pin numbers (Q3 pin 28 in a 20 pin device).
- Assigning an input pin number to an output signal. (In this case, the output signal assignment is ignored.)

Using State Machines

State machine designs are widely used in PLDs for sequential control logic, which forms the core of many digital systems. Control functions can be as simple as a toggle flip-flop or as complex as a supercomputer.

A state machine is a digital device which traverses through a predetermined sequence of states. A state is a stage through which a sequential circuit advances. In each state the circuit stores a recollection of its past history so it can know what to do next. In PLD's output registers and buried registers are used to store the history of the states it has traversed through.

State machines are one of the more powerful features of ABEL-PLD, but some care should be taken in organizing the state diagram and in ordering the states within it. This section describes some of the steps you can take to make state diagrams easy to read and maintain, and to avoid some potential problems. The most common problem encountered with state machines is that too many product terms are created for the chosen device. This problem arises because state machines often have many different states and complex state transitions. The topics discussed in the following subsections will help you avoid this problem by reducing the number of required product terms.

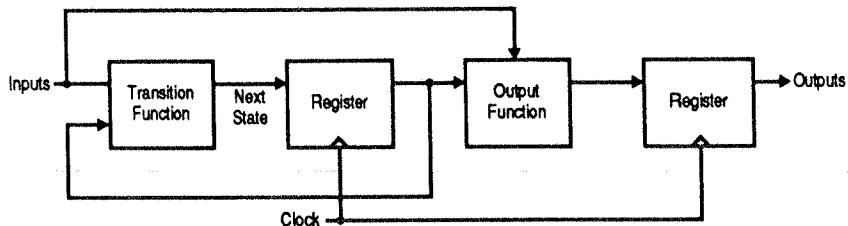
The following subsections provide information related to state machines:

- Mealy and Moore State Machines
- Debugging State Machines with Simulate
- Use Identifiers Rather Than Numbers for States
- "Power On" Register States
- Designing with Programmable Polarity Outputs
- Unsatisfied Transition Conditions, D-Type Flip-Flops
- Unsatisfied Transition Conditions, Other Flip-Flops
- Number Adjacent States for One Bit Changes
- Use State Register Outputs to Identify States

Mealy and Moore

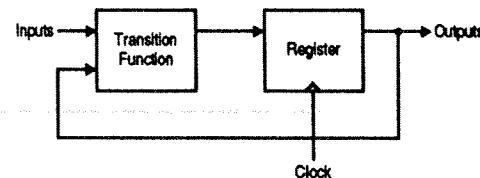
State machines can be classified into two categories, each with different output characteristics. In a Mealy machine (Figure 9-1), the outputs at any given time are a function of its present state and current inputs. As a result, multiple output combinations are possible for any given state.

Figure 9-1
Flow Diagram for a Mealy State Machine



In a Moore machine (Figure 9-2) the outputs are a function of only the present state. Moore machines are a subclass of Mealy machines. An equivalent Moore machine can be defined for any Mealy machine. The following ABEL-PLD designs implement a sequence detector in state machine syntax. The first design shown in Figure 9-5 implements the sequence detector using a Mealy machine. The second design shown in Figure 9-6 is the equivalent Moore machine.

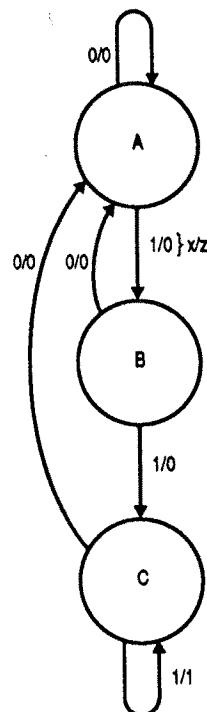
Figure 9-2
Flow Diagram for a Moore State Machine



The sequence detector is designed using a state diagram syntax to describe the behavioral logic of the system. The design of the system will output ($Z = 1$) when three successive 1s have been detected on input (X).

The state diagram for the Mealy sequence detector is given in Figure 9-3 which is to be compared with the equivalent Moore sequence detector in Fig 9-4. In both cases, state A is a state that has been arrived at after $X = 0$, so that any sequence of input 1s has been interrupted and Z cannot be equal to 1.

Figure 9-3
Mealy State Diagram of a
Sequence Detector



Note that in the Moore system the output Z is entered in the circle which represents the state. Such an entry can be made because Z depends only on the state. The symbolism for the Mealy system is different. Here starting in state A, when X = 1, we go to a new state B since the system must remember that a first step toward a successful sequence has occurred. The arrow leading from state A to state B has the notation 1/0. The entry to the left of the line stands for X = 1, and the entry to the right of the line stands for Z = 0. The state C in the Mealy diagram is a state in which it is remembered that there have been two successive 1s. In the interval of that same state, an X = 1 will be acknowledged as a third 1, and the output Z will equal 1. Note that the Mealy system has only three states and the Moore system has four states.

Figure 9-4
Moore State Diagram of a
Sequence Detector

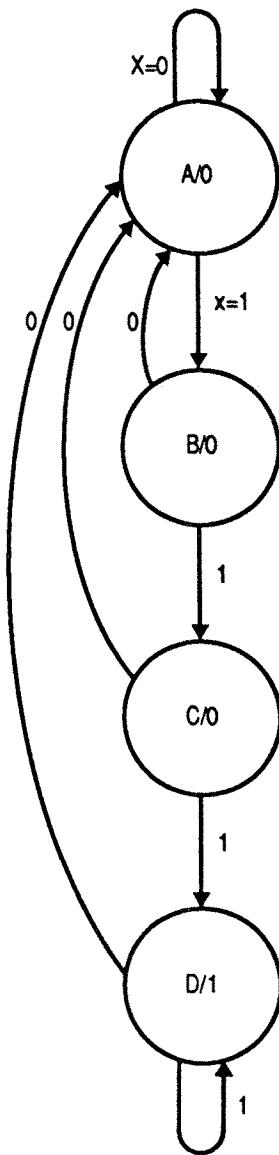


Figure 9-5
Source File for a Mealy State Machine

```

module mealy      flag '-r4'
title 'Sequence detector using a Mealy machine
Jeffrey Davis    Data I/O Corp.'

      _mealy    device  'F167';
"Inputs
      clk      pin      1;
      PR       pin      16 ;
      X        pin      8 ;
"Output
      Z        pin      15;
      Z.pr    istype 'pin';"initialization
"State registers
sreg    = [P2,P3]; "use buried registers for state registers
A      = [0 ,0 ]; "use one bit change for better reduction
B      = [0 ,1 ];
C      = [1 ,1 ];

Equations
      P3.pr = PR ; " on power up goto state B

"This state machine detects a sequence of 3 successive 1s on
"input X.

state_diagram sreg

state A:
      if X then B  WITH
            Z.r = 1 ;
            ENDWITH ;
      else A   WITH
            Z.r = 1 ;
            ENDWITH ;

state B:
      if X then C
      else A ;

state C:
      Z := X ;
      if X then C
      else A   WITH
            Z.r = 1 ;
            ENDWITH ;

test_vectors
(([clk,PR, X ] -> [ sreg , Z ])
[.c.,1 , 0 ] -> [ B , .x. ]; "power up
[.c.,0 , 0 ] -> [ A , 0 ];
[.c.,0 , 1 ] -> [ B , 0 ];
[.c.,0 , 1 ] -> [ C , 0 ];
[.c.,0 , 1 ] -> [ C , 1 ];
[.c.,0 , 0 ] -> [ A , 0 ];
[.c.,0 , 1 ] -> [ B , 0 ];
[.c.,0 , 0 ] -> [ A , 0 ];
[.c.,0 , 1 ] -> [ B , 0 ];
[.c.,0 , 1 ] -> [ C , 0 ];
[.c.,0 , 1 ] -> [ C , 1 ];
[.c.,0 , 0 ] -> [ A , 0 ];
[.c.,0 , 1 ] -> [ B , 0 ];
[.c.,0 , 1 ] -> [ C , 0 ];
[.c.,0 , 1 ] -> [ C , 1 ];

end

```

Figure 9-6

Source File for a Moore State Machine

```

module moore      flag '-r3'
title 'Sequence detector using a Moore machine
Jeffrey Davis    Data I/O Corp.'

      _moore    device  'P22V10';

"Inputs
      clk      pin      1 ;
      PR       pin      2 ;
      X        pin      3 ;
"Output
      Z        pin      23 ;
      Q0,Q1   pin     21,22 ;

      Z,Q0,Q1 istype 'pos,reg';
"State registers
sreg    = [Q1,Q0,Z];
A       = [0 ,0 ,0]; "use one bit change for better reduction
B       = [0 ,1 ,0];
C       = [1 ,1 ,0];
D       = [1 ,1 ,1];

Equations
      [ Q1.ar,Q0.ar,Z.ar] = PR ;"power up to state A

"This state machine detects a sequence of 3 successive 1's on
"input X.

state_diagram sreg

state A:
      if X then B  else A ;
state B:
      if X then C  else A ;
state C:
      if X then D  else A ;
state D:
      if X then D  else A ;

test_vectors
(([clk,PR, X ] -> [ sreg ])
[ 0 ,1 , 0 ] -> [  A   ]; "power reset
[ .c.,0 , 0 ] -> [  A   ];
[ .c.,0 , 1 ] -> [  B   ];
[ .c.,0 , 1 ] -> [  C   ];
[ .c.,0 , 1 ] -> [  D   ];
[ .c.,0 , 0 ] -> [  A   ];
[ .c.,0 , 1 ] -> [  B   ];
[ .c.,0 , 1 ] -> [  C   ];
[ .c.,0 , 1 ] -> [  D   ];
[ .c.,0 , 1 ] -> [  D   ];
[ .c.,0 , 0 ] -> [  A   ];
[ .c.,0 , 1 ] -> [  B   ];
[ .c.,0 , 1 ] -> [  C   ];
[ .c.,0 , 1 ] -> [  D   ];

end

```

Debugging State Machines with Simulate

To test and debug larger state machines:

- Write small sets of test vectors that test individual functions of the state machine, and gradually add them to the simulation.
- Add test vectors that periodically force the machine to known states to eliminate cascading of errors.

Multiple Test Vector Tables

State machines can be difficult to debug once an error occurs because each state is dependent on previous states and points to next state. A simple error in the description of one state transition can cause an incorrect output that cascades through the state machine, causing so many errors during simulation that the original error is difficult to isolate. Therefore, when you debug state machines of any size, you should periodically force (with your test vectors) the machine to a known state and then let the state transitions take place. In this manner, you limit the cascading of errors to a smaller number of states and make it much easier to find initial errors.

It is also helpful with large state machines to start with a small set of test vectors that tests only part of the state machine's function. When operation of that function has been verified, add a set of test vectors that test another function, then add another, and so on, until you have tested the full function of the state machine. Combining this technique of gradual simulation with the forcing vectors discussed above greatly simplifies the testing of large state machines.

More than one set of test vectors can be used to simulate the function of a device. This may be useful in the representation of the test in the source file. Take for example, the source file presented in Figure 9-6 that describes AND and NAND gates implemented on the same device as well as the test vectors used to simulate the operation of that device. The test vectors are described in two separate sections. The first test vectors section lists the test vectors that simulate the operation of the AND portion of the design, and the second section tests the NAND function. With the test vectors written as they are, in two separate sections, the correspondence between test vectors and the function being tested is readily apparent in the source file.

In a similar manner, any time you have two or more distinct functions being performed by the same device, you may want to describe the vectors in separate sections for each function. You are not, however, required to do so.

Figure 9-7
Source File With Multiple Test Vectors Sections

```

module simple
  title 'Simple ABEL-PLD example'
  Dan Poole  Data I/O Corp'

  U7      device  'P14H4';
  A1,A2,A3      pin 1,2,3;
  N1,N2,N3      pin 4,5,6;
  AND,NAND      pin 14,15;

  equations
    AND      = A1 & A2 & A3;
    !NAND    = N1 & N2 & N3;

  test_vectors 'Test And Gate'
    ( [A1,A2,A3] -> AND )
    [ 0, 0, 0 ] -> 0;
    [ 1, 0, 0 ] -> 0;
    [ 0, 1, 0 ] -> 0;
    [ 0, 0, 1 ] -> 0;
    [ 1, 1, 1 ] -> 1;

  test_vectors 'Test Nand Gate'
    ( [N1,N2,N3] -> NAND )
    [ 0, 0, 0 ] -> 1;
    [ 1, 0, 0 ] -> 1;
    [ 0, 1, 0 ] -> 1;
    [ 0, 0, 1 ] -> 1;
    [ 1, 1, 1 ] -> 0;
end simple

```

Express State Registers as a Set

ABEL-PLD requires that state registers in a state machine are expressed as a set in order to automatically select the flip flop type (D, RS, JK or T). If the state register is a single register not in brackets, the flip flop defaults to D.

Use Identifiers Rather Than Numbers for States

A state machine has different "states" that describe the outputs and transitions of the machine at any given point. Typically, each state is given a name, and the state machine is described in terms of transitions from one state to another. In a real device, such a state machine is implemented with registers that contain enough bits to assign a unique number to each state. The states are actually bit values in the register, and these bit values are used along with other signals to determine state transitions.

As you develop a state diagram, you need to label the various states and state transitions. It is best to label the states with identifiers that have been assigned constant values rather than labeling the states directly with numbers. This allows you to change the state transitions easily or to change the state register values associated with each state.

In writing a state diagram with ABEL-PLD, you should follow the same design procedure of first describing the machine with names for the states, and then assigning state register bit values to the state names.

For an example, see Figure 9-8, which lists the source file for a state machine named "sequence." (This state machine is also discussed in the design examples.) In the state diagram, identifiers (A, B, and C) are used to specify the states. These identifiers are assigned a constant decimal value in the declaration section of the source file that identify the bit values in the state register for each state. Note that A, B, and C are only identifiers, they do not indicate the bit pattern of the state machine. It is their declared values that define the value of the state register (sreg) for each state. The declared values are 0, 1, and 2.

Figure 9-8
Source File: State Machine

```

module sequencer
flag '-r3'
title 'State machine example
D. B. Pellerin - Data I/O';

d1          device 'p16r4';

q1,q0      pin  14,15;
clock,enab,start,hold,reset  pin  1,11,4,2,3;
abort       pin  17;
in_B,in_C  pin  12,13;
sreg        =    [q1,q0];

"State Values...
A = 0;           B = 1;           C = 2;
X = .X.

state_diagram sreg;
State A:          " Hold in state A until start is active.
    in_B = 0;
    in_C = 0;
    IF (start & !reset) THEN B WITH abort := 0;
    ELSE A WITH abort := abort;

State B:          " Advance to state C unless reset
    in_B = 1;      " or hold is active. Turn on abort
    in_C = 0;      " indicator if reset.
    IF (reset) THEN A WITH abort := 1;
    ELSE IF (hold) THEN B WITH abort := 0;
    ELSE C WITH abort := 0;

State C:          " Go back to A unless hold is active
    in_B = 0;      " Reset overrides hold.
    in_C = 1;
    IF (hold & !reset) THEN C WITH abort := 0;
    ELSE A WITH abort := 0;

test_vectors([clock,enab,start,reset,hold,sreg]->[sreg,abort,in_B,in_C])
    [ .p. , 0 , 0 , 0 , 0 , A ]->[ X , 0 , 0 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 , X ]->[ A , 0 , 0 , 0 ];
    [ .c. , 0 , 1 , 0 , 0 , X ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 , X ]->[ C , 0 , 0 , 1 ];

    [ .c. , 0 , 1 , 0 , 0 , X ]->[ A , 0 , 0 , 0 ];
    [ .c. , 0 , 1 , 0 , 0 , X ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 1 , 0 , X ]->[ A , 1 , 0 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 , X ]->[ A , 1 , 0 , 0 ];

    [ .c. , 0 , 1 , 0 , 0 , X ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 1 , X ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 1 , X ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

end

```

Powerup Register States

Your state machine design might depend on a specific starting state, and, therefore, you must know what the powerup (or power on) state of a device's register is. If the register powers on in a state that is not defined in the state diagram description of the state machine, the next state is undefined and unknown unless your design specifically accounts for this situation. Therefore, you should account for the power-on state of the device you are using or make sure that your design goes to a known state at powerup.

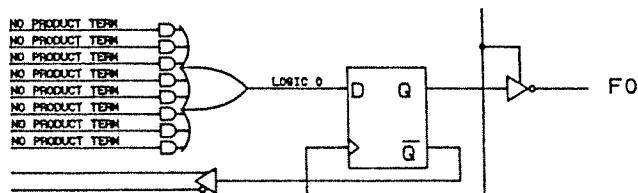
Designing with Programmable Polarity Outputs

With devices featuring programmable polarity on the outputs of the registers, keep in mind that the register outputs may be programmed to the complement of the device outputs. This is important because all the sets and clears are on the registers rather than on the programmable device outputs, and because you are concerned with the states (register outputs) themselves rather than with the outputs at the device pins. The final outputs can be handled separately.

Unsatisfied Transition Conditions, D-Type Flip-Flops

For each state described in a state diagram, you specify the transitions to the next state and the conditions that influence those transitions. For devices with D-type flip-flops, if none of the stated conditions is met, the state register, shown in Figure 9-9, is cleared to all 0s on the next clock pulse. This action causes the state machine to go to the state that corresponds to the cleared state register. This can either be used to your advantage, or cause problems, depending on your design.

Figure 9-9
D-Type Register with False
Inputs



You can use this action to eliminate some conditions in your state diagram and some product terms in the converted design. Rather than having a condition for a transition to the cleared-register state, that transition can be left implicit. If none of the other transition conditions is met, the machine will go to the cleared state. This same fact can cause problems if the cleared state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine will go to the undefined cleared state, and remain in that state.

In general, this means that you should always have a state assigned to the cleared-register state. Or, you must define every possible condition so that some condition is always met for each state. But, the automatic transition to the cleared-register state can also be used to your advantage by eliminating product terms and explicit definitions of transitions.

Unsatisfied Transition Conditions, Other Flip-Flops

If none of the state conditions is met in a state machine that employs J-K, R-S, and T-type flip-flops, the state machine does not advance to the next state as it should, but holds its present state. This is due to the low input to the register from the OR array output. In such a case, the state machine can "get stuck" and will not change state. This holding action can be used to advantage in some designs.

To prevent this condition from occurring, you can use the complement array provided in some devices, such as the F105, to detect a "no conditions met" situation and reset the state machine to a known state.

Number Adjacent States for One-Bit Changes

The number of product terms produced by a state diagram can be reduced greatly by a careful choice of state register bit values. Your state machine should be described with symbolic names for the states, as described above. Then if you assign the numeric constants to these names so that the state register bits change by only one bit at a time as the state machine goes from state to state, the number of product terms required to describe the state transitions is reduced.

For example, take the states, A, B, C, and D, which go from one state to the other alphabetically. The simplest choice of bit values for the state register is a simple numeric sequence. The simplest choice is not, however, the most efficient. Take, for example, the following simple and preferred choices for bit value assignments:

State	Simple Bit Values	Preferred Bit Values
A	00	00
B	01	01
C	10	11
D	11	10

Notice that the preferred bit values cause a change of only one bit as the machine moves from state B to C, whereas the simple choices for the bit values cause a change in both bit values for the same transition. The preferred bit values will produce fewer product terms.

If the language processor reports that too many product terms were produced for one of the state register bits, you should reorganize the bit values so that as the state machine moves from state to state, the bit for which there are too many terms changes in value as few times as possible.

Obviously, the choice of optimum bit values for specific states can require some tradeoffs; you may have to optimize for one bit, and, in the process, increase the value changes for another. The overall object should be to eliminate as many product terms as necessary to fit the design into the device.

Use State Register Outputs to Identify States

Sometimes it is necessary to identify specific states of a state machine, and signal an output that the machine is in one of these specific states. Equations and outputs can be saved if you organize the state register bit values so that one bit in the state register determines whether the machine is in a state of interest. Take, for example, the following sequence of states in which it is required that the C_n states are identified:

State Register Bit Values:

State Name	Q3	Q2	Q1
A	0	0	0
B	0	0	1
C ₁	1	0	1
C ₂	1	1	1
C ₃	1	1	0
D	0	1	0

This choice of state register bit values allows Q3 to be used as a flag to indicate when the machine is in any of the C_n states. Whenever Q3 is high, the machine is in one of the C_n states. Q3 can be assigned directly to an output pin on the device. Notice also that these bit values change by only one bit as the machine cycles through the states, as is recommended in section above.

Chapter 10

Design Examples

The logic designs and design features discussed in the following sections are contained on the Design Examples disk you received with the ABEL-PLD package; and are listed in Table 10-1.

You can process these design examples with ABEL-PLD software, either as they stand, or with your own modifications, to create programmer load files. Many of the design examples listed in Table 10-1 are described in detail within this manual; design examples not described in the manual, but listed in this table, can be examined as the need arises.

Table 10-1 lists the design examples alphabetically by filename of the ABEL-PLD source file and contains columns for

- THE NAME OF THE SOURCE FILE. The source file uses the .abl file extension (e.g., ADD5 is add5.abl). The ABEL-PLD batch file requires that the file extension not be entered for the source file. The command line for any of individual programs defaults to .abl for a source file extension.
- PROGRAMMABLE LOGIC DEVICE TYPE. The type of programmable logic device type used in the design example is listed to assist you in locating meaningful design examples.
- TYPE OF EXAMPLE. A brief description of the design example.

The following logic design examples are believed to be representative of typical programmable logic applications. These serve to illustrate significant ABEL-PLD features. Use them to learn more about ABEL-PLD and to start designing with ABEL-PLD. We encourage you to use these examples directly by modifying them to suit your needs, or by incorporating them into larger system designs.

The titles of the designs indicate the logic design problem being solved, the major features of ABEL-PLD used to solve the problem, and the type of programmable logic device used. Each design has accompanying block diagrams and source file listings.

Your ABEL-PLD distribution disk contains all the examples presented in this section, plus additional example files not discussed. A list and brief description of all examples can be found in the file named examples.txt on your distribution disk. You can use the operating system to display the list of design examples contained in examples.txt.

Table 10-1
*Design Examples Supplied
with ABEL-PLD*

Filename (*.abl)	Device Type	Type of Example
BARREL	P20R8	Sets
BCD7PAL	P16L8	PALASM example for TOABEL
BCD7ROM	RA5P8	Truth table
BGATES	P12H6	ABEL-PLD version of PALASM basic gates
BGEQN	F100	@repeat directive used w/equations
BINBCD	P16L8	Truth table
BJACK	P16R6	State machine
COMP4A	F153	Relational operators
COUNT4	P16R4	Equations
COUNT4A	P16R4	Multiple equations for one output
DECADE	F105	Transition equations complement array
DMUX1T8	P16L8	Sets
FEEDBACK	P16HD8	Simulator example
GATES	P12H6	PALASM example for TOABEL
LCUCROM	RA8P8	Macros
M6809A	P14L4	Sets
M6809B	P14L4	Macros with test vectors
M6809C	P14L4	Macros with test vectors
M6809D	P16L8	Device type from command line
M6809ER	P14L4	Demonstrates simulation error
M6809ERR	P14L4	Demonstrates syntax error
MUXADD	P22V10	Ripple adder
MUX12T4	P14H4	Sets

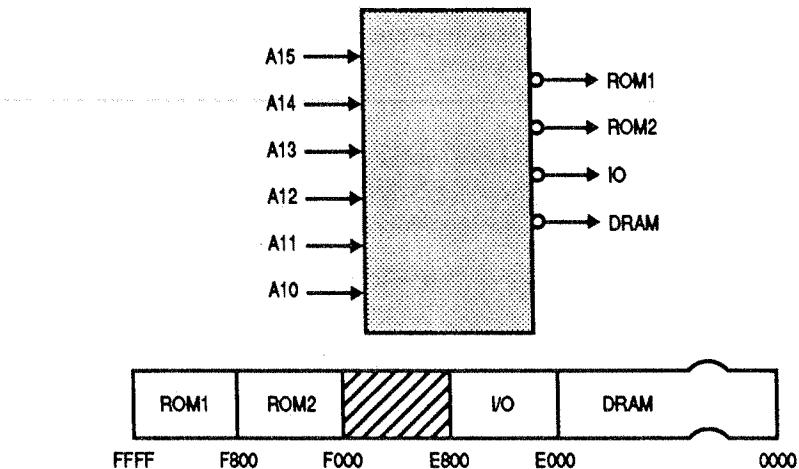
This table is a list of examples available at the time of printing. An updated listing is found in the examples.txt file.

6809 Memory Address Decoder

Equations, P14L4

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL-PLD implementation of such a design.

Figure 10-1
Block Diagram: 6809 Memory
Address Decoder



Design Specification

Figure 10-1 shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (DRAM), I/O (I/O), and two sections of ROM (ROM1 and ROM2). The purpose of this decoder is to monitor the 6 high-order bits (A15-A10) of a sixteen-bit address bus and select the correct section of memory based on the value of these address bits. To perform this function, a simple decoder with six inputs and four outputs is designed with a P14L4 PAL.

The address ranges associated with each section of memory are shown below. These address ranges can also be seen in Figure 10-2.

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

Design Method

Figure 10-3 shows a simplified block diagram for the address decoder. The address decoder is implemented with simple equations employing both relational and logical operators as shown in Figure 10-2. A significant amount of simplification is achieved by grouping the address bits into a set named Address. The lower-order ten address bits that are not used for the address decode are given "don't care" values in the address set. In this way, the designer indicates that the address in the overall design (that beyond the decoder) contains sixteen bits, but that bits 0 to 9 do not affect the decode of that address. This is opposed to simply defining the set as

$$\text{Address} = [A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}]$$

which ignores the existence of the lower-order bits. Specifying all 16 address lines as members of the address set also allows full 16-bit comparisons of the address value against the ranges shown above.

Figure 10-2
6809 Memory Address
Decoder Source File

```
module m6809a
  title '6809 memory decode'
  Jean Designer    Data I/O Corp Redmond WA'

  U09a      device 'P14L4';
  A15,A14,A13,A12,A11,A10  pin 1,2,3,4,5,6;
                           ROM1,IO,ROM2,DRAM      pin 14,15,16,17;

  H,L,X    = 1,0,.X.;
  Address  = [A15,A14,A13,A12,A11,A10,X,X,X,X,X,X,X,X];

  equations
    !DRAM    = (Address <= ^hDFFF);
    !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
    !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
    !ROM1   = (Address >= ^hF800);

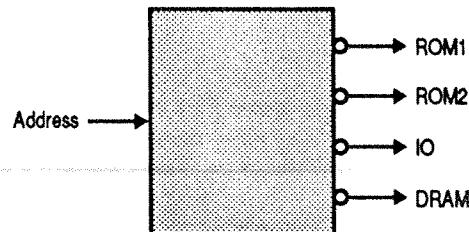
  test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
    ^h0000 -> [ H , H , H , L ];
    ^h4000 -> [ H , H , H , L ];
    ^h8000 -> [ H , H , H , L ];
    ^hC000 -> [ H , H , H , L ];
    ^hE000 -> [ H , H , L , H ];
    ^hE800 -> [ H , H , H , H ];
    ^hF000 -> [ H , L , H , H ];
    ^hF800 -> [ L , H , H , H ];

end m6809a
```

Test Vectors

In this design, the test vectors are a straightforward listing of the values that must appear on the output lines for specific address values. The address values are specified in hexadecimal notation.

Figure 10-3
*Simplified Block Diagram: 6809
Memory Address Decoder*



12 to 4 Multiplexer

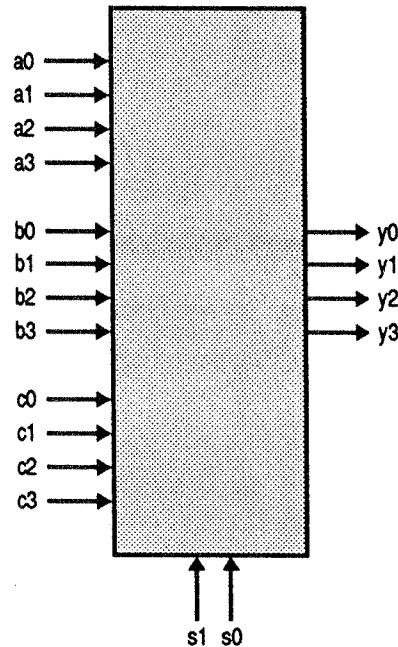
Equations, P14H4

The following describes the implementation of a 12-input to 4-output multiplexer using high level equations.

Design Specification

Figure 10-4 shows the block diagram for this design. The multiplexer selects one of three sets of four inputs and routes that set to the outputs. The inputs are a_0-a_3 , b_0-b_3 , and c_0-c_3 . The outputs are y_0-y_3 . The routing of inputs to outputs is straightforward: a_0 or b_0 or c_0 is routed to the output y_0 , a_1 or b_1 or c_1 is routed to the output y_1 , and so on with the remaining outputs. Decoding of which set is routed to the output is controlled by the input set "select," consisting of s_0 and s_1 .

Figure 10-4
Block Diagram: 12 to 4
Multiplexer



Design Method

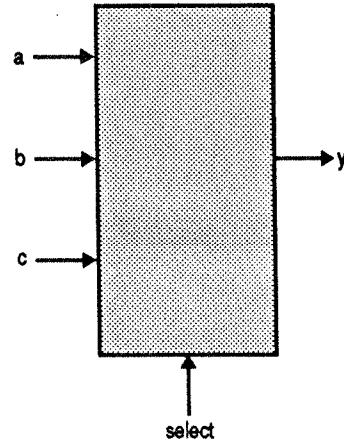
Figure 10-5 shows a block diagram for the same multiplexer after sets have been used to group the signals. All of the inputs have been grouped into the sets a, b, and c; the outputs and select lines are grouped into the sets, y and *select*, respectively. This grouping of signals into sets takes place in the declaration section of the source file listed in Figure 10-6.

Once the sets have been declared, specification of the design is made with the following four equations that use WHEN-THEN statements.

```
when (select == 0) then y = a;
when (select == 1) then y = b;
when (select == 2) then y = c;
when (select == 3) then y = d;
```

The relational expression (==) inside the parentheses produces an expression that evaluates to true or false value depending on the values of s1 and s0.

Figure 10-5
Simplified Block Diagram: 12 to
4 Multiplexer



In the first equation, this expression is then ANDed with the set a which contains the four bits, a0-a3, and could be written as

$$y = (\text{select} == 0) \& a$$

Assume that select is equal to 0 (s1 = 0 and s0 = 0) so that a true value is produced. The true is then ANDed with the set a on a bit by bit basis, which in effect sets the product term to a. If select were not equal to 0, the relational expression inside the parentheses would produce a false value, which when ANDed with anything, would give all zeroes.

The other product terms in the equation work in the same manner. Because select takes on only one value at a time, only one of the product terms passes the value of an input set along to the output set; the others contribute 0 bits to the ORs.

Test Vectors

The test vectors for this design are specified in terms of the input, output and select sets. Note that the values for a set can be specified by decimal numbers and by other sets. The constants H and L used in the test vectors were declared as four bit sets containing all ones or all zeroes.

Figure 10-6
Source File: 12 to 4
Multiplexer

```

module _mux12t4
title '12 to 4 multiplexer
Charles Olivier & Dave Pellerin Data I/O Corp. Redmond WA'

mux12t4          device 'P14H4';

a0,a1,a2,a3      pin    1,2,3,4;
b0,b1,b2,b3      pin    5,6,7,8;
c0,c1,c2,c3      pin    9,11,12,13;
s1,s0            pin    18,19;
y0,y1,y2,y3      pin    14,15,16,17;

H     = [1,1,1,1];
L     = [0,0,0,0];
X     = .x.;
select = [s1, s0];
y     = [y3,y2,y1,y0];
a     = [a3,a2,a1,a0];
b     = [b3,b2,b1,b0];
c     = [c3,c2,c1,c0];

equations
when (select == 0) then y = a;
when (select == 1) then y = b;
when (select == 2) then y = c;
when (select == 3) then y = c;

test_vectors
([select, a, b, c] -> y)
[0      , 1, X, X] -> 1; "select = 0, gates lines a to output
[0      ,10, H, L] -> 10;
[0      , 5, H, L] -> 5;

[1      , H, 3, H] -> 3; "select = 1, gates lines b to output
[1      ,10, 7, H] -> 7;
[1      , L,15, L] -> 15;

[2      , L, L, 8] -> 8; "select = 2, gates lines c to output
[2      , H, H, 9] -> 9;
[2      , L, L, 1] -> 1;

[3      , H, H, 0] -> 0; "select = 3, gates lines c to output
[3      , L, L, 9] -> 9;
[3      , H, L, 0] -> 0;

end _mux12t4

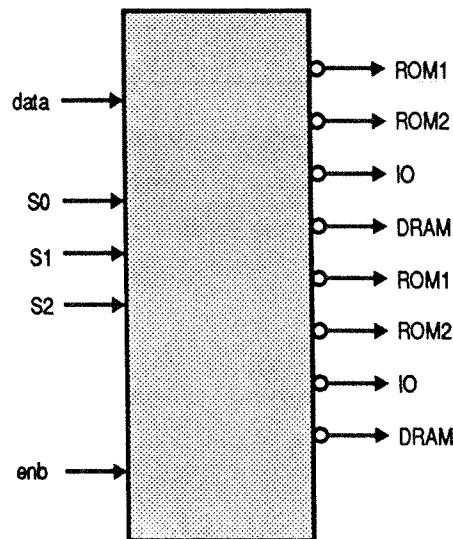
```

1 to 8 Demultiplexer

Equations, P16L8

The following design describes the implementation of a one line to eight line demultiplexer with an enable. The design uses high level equations for a P16L8 PAL.

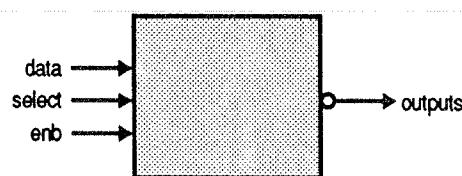
Figure 10-7
Block Diagram: 1 to 8
Demultiplexer



Design Specification

Figure 10-7 shows a block diagram for a one line to eight line demultiplexer with active low outputs, a one bit input, three select lines and an enable. The demultiplexer routes the input named data to one of the eight output lines, y_0-y_7 , according to the value present on the select lines, s_0-s_2 . Because the outputs are inverted, they show the inverse of the input line. The select lines carry a 3-bit binary number ranging from 0 to 7, thus selecting one of the outputs. enb is an enable line. When enb is low (0), the outputs go to high impedance. When enb is high, the outputs are determined by the demultiplexing function.

Figure 10-8
Simplified Block Diagram:
Demultiplexer



Design Method

Figure 10-8 shows a simplified block diagram of the demultiplexer. The select lines and outputs are collected into sets named select and outputs to simplify the equations with which the demultiplexing function is described. The ABEL-PLD description of this logic is shown in Figure 10-9. Each output is determined by ANDing the data line with a relational expression that checks for equivalence between the select lines and one of the eight possible select values. Because select can have only one value at any given time, only one of the outputs is selected for an AND with the data value. The selected output shows the inverse of the data line.

The enable function is implemented with a separate equation using the set outputs along with the .OE dot extension. Thus the equation,

```
outputs.oe = enb ;
```

assigns the enables on each of the output lines to the input, enb.

Test Vectors

The test vectors for this design first select each of the outputs with the data input and enable high, then they select each of the outputs with the data input low and the enable high. Finally, the enable is checked by setting it low; all the outputs go to high impedance.

Figure 10-9
1 to 8 Demultiplexer

```

module dmux1t8
title '1 to 8 line demultiplexer
Mark Kuenster Data I/O Corp.'

DM1      device 'P16L8';

y0,y1,y2,y3,y4,y5,y6,y7 pin 12,13,14,15,16,17,18,19;
s0,s1,s2,data,enb      pin 1,2,3,4,5;

H,L,Z      =      1,0..Z.;
select     =      [s2, s1, s0];
outputs    =      [y7, y6, y5, y4, y3, y2, y1, y0];

equations
!y0      = (select == 0) & data;
!y1      = (select == 1) & data;
!y2      = (select == 2) & data;
!y3      = (select == 3) & data;
!y4      = (select == 4) & data;
!y5      = (select == 5) & data;
!y6      = (select == 6) & data;
!y7      = (select == 7) & data;

outputs.oe = enb;

test_vectors 'Test the demultiplexer with a high input'
([enb,select,data] -> [y7,y6,y5,y4,y3,y2,y1,y0])
[ H , 0 , H ] -> [H, H, H, H, H, H, L]; "Select y0
[ H , 1 , H ] -> [H, H, H, H, H, H, L, H]; "Select y1
[ H , 2 , H ] -> [H, H, H, H, H, L, H, H]; "Select y2
[ H , 3 , H ] -> [H, H, H, H, L, H, H, H]; "Select y3
[ H , 4 , H ] -> [H, H, H, L, H, H, H, H]; "Select y4
[ H , 5 , H ] -> [H, H, L, H, H, H, H, H]; "Select y5
[ H , 6 , H ] -> [H, L, H, H, H, H, H, H]; "Select y6
[ H , 7 , H ] -> [L, H, H, H, H, H, H, H]; "Select y7
[ L , 0 , H ] -> [Z, Z, Z, Z, Z, Z, Z, Z]; "Tristate outputs

test_vectors 'Test the demultiplexer with a low input'
([enb,select,data] -> [y7,y6,y5,y4,y3,y2,y1,y0])
[ H , 0 , L ] -> [H, H, H, H, H, H, H]; "Select y0
[ H , 1 , L ] -> [H, H, H, H, H, H, H]; "Select y1
[ H , 2 , L ] -> [H, H, H, H, H, H, H]; "Select y2
[ H , 3 , L ] -> [H, H, H, H, H, H, H]; "Select y3
[ H , 4 , L ] -> [H, H, H, H, H, H, H]; "Select y4
[ H , 5 , L ] -> [H, H, H, H, H, H, H]; "Select y5
[ H , 6 , L ] -> [H, H, H, H, H, H, H]; "Select y6
[ H , 7 , L ] -> [H, H, H, H, H, H, H]; "Select y7
[ L , 0 , L ] -> [Z, Z, Z, Z, Z, Z, Z, Z]; "Tristate outputs

end dmux1t8

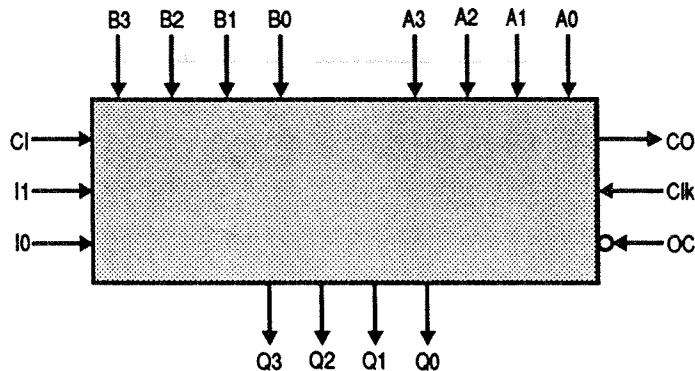
```

4-Bit Counter/Multiplexer

Equations, P16R4

The following design describes the implementation of a 4-bit synchronous counter using a P16R4. Counter features include carry in and carryout, 2 input multiplexing and a hold state. The counter is described by high level equations.

Figure 10-10
Block Diagram: 4-Bit Counter
With 2 Input Multiplexer



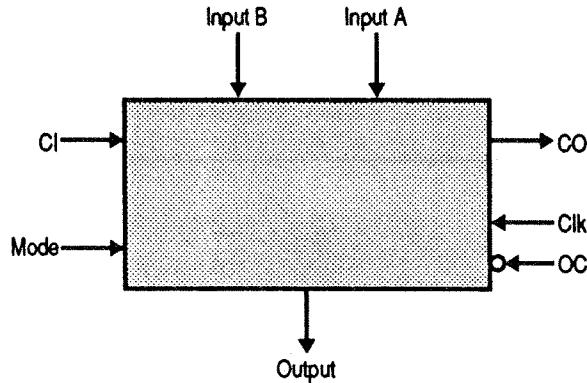
Design Specification

Figure 10-10 shows the counter and its signals. The outputs, Q0, Q1, Q2, and Q3 show the current count, with Q0 being the low-order bit and Q3 being the high-order bit. The counter has four different modes of operation: hold, load A, load B, and increment. The modes are selected by the inputs, I0 and I1, as indicated in the table below. In the hold mode, the current count is retained regardless of clocking. When in load A mode, the counter loads the values on A0-A3 on the next clock pulse. Similarly, the B0-B3 inputs are loaded into the counter on the next clock pulse when the mode is load B. In increment mode, the count is increased by the value on the carry in line CI on a clock pulse. If the count overflows from 15 (hex F) to 0, the carryout line CO, goes to 1. The output control, OC, enables the outputs when low (OC = 0) and forces the outputs to high impedance when high (OC = 1).

Mode	I1	I0	Description
Hold	0	0	count remains unchanged
Load A	0	1	load A0-A3 into the count registers on the clock pulse
Load B	1	0	load B0-B3 into the count registers on the clock pulse
Increment	1	1	increment the count by 1 on clock pulse

The carry in and carryout lines operate such that two or more of the counters can be chained together to form a wider counter. To do this, the carryout of one counter is connected to the carry in of the next counter. Thus, when the first counter counts to 16, it is cleared to 0 and its carryout bit is one, causing the next counter's increment to be one.

Figure 10-11
Simplified Block Diagram: 4-Bit Counter With 2 Input Multiplexer



Design Method

The counter is described with high level equations. Figure 10-11 shows the simplified block diagram corresponding to the ABEL-PLD implementation of the counter design. Figure 10-12 shows the source file.

The design is simplified by grouping the input bits, output bits and mode selectors into sets, so that they can be referenced by name. The inputs A0-A3 are assigned to the set InputA, the inputs B0-B3 to the set InputB, the outputs Q0-Q3 and CO to the set Output, and the mode selectors I0 and I1 to the set Mode.

Notation in the source file can be further simplified by some simple constant assignments. H represents a 1, L represents a 0, X represents the special constant .X. (don't care), and so on. The four possible modes are also assigned as constants: Hold=0, LoadA=1, LoadB=2, and Incr=3. These assignments correspond to the decimal equivalent of the two bit binary number formed of inputs I1 and I0, as already shown in table 10-2. Note also that this two bit binary number is the set Mode.

Thus, in the equations section of the source file, the set Mode is compared to the different possible modes — by name. Take for example, the expression

Mode==LoadA

This is equivalent to

$[I1,I0]==1$

And this is equivalent to

$$(I1 == 0) \& (I0 == 1)$$

which can be written as

$$I1 \& I0$$

This means "if I1 is low and I0 is high." By using sets and constant assignments, the source file becomes more meaningful. The expression, Mode==LoadA, can be read as, "if the mode is LoadA."

The equations section of the source file simply describes the high level equations for a 4-bit adder with carry in, carryout and multiplexed load.

Test Vectors

The advantages of set notation become even more apparent in the test vectors section of the source file. In the test vectors section, it is necessary to show the required output for various given inputs. Rather than listing the outputs and inputs bit by bit, advantage is taken of sets, constants, and hexadecimal notation to simplify the vectors.

For example, because LoadA was assigned the constant value, 1, the name, LoadA, can be used directly in the test vectors as a value for Mode. Thus, it is unnecessary to remember that Mode is made up of I1 and I2 and that LoadA corresponds to I1=0, I0=1.

The test vectors shown in Figure 10-12 test for proper loading of InputA and InputB, for increments after loads, for hold states, for correct operation of the carryout, and for normal increment mode. Refer to the comments beside the test vectors for examples of each type of test.

The PAL 16R4 and many other devices have a dedicated output enable pin. This pin must be held at the proper level (0 or 1) during simulation to observe the outputs. The test vectors in Figure 10-12 include the output enable pin (OC).

Figure 10-12

Source file: 4-bit Counter with 2
Input Mux

```

module count4
  title '4-bit counter with 2 input mux
  based on an example by Birkner/Coli in the MMI PAL Handbook
  Dan Burrier, Mike McGee & Lisa Matheson    Data I/O Corp.'

  P7022      device 'P16R4';

  Clk,OC,CO,I1,I0,CI      pin 1,11,12,13,18,19;
  A0,A1,A2,A3,B0,B1,B2,B3  pin 2,3,4,5,6,7,8,9;
  Q3,Q2,Q1,Q0              pin 14,15,16,17;
  H,L,X,Z,C                = 1,0, .X.,.Z.,.C.:
  InputA                   = [A3,A2,A1,A0];
  InputB                   = [B3,B2,B1,B0];
  Output                    = [CO,Q3,Q2,Q1,Q0];
  Mode                      = [I1,I0];
  Hold,LoadA,LoadB,Incr    = 0,1,2,3; " define Modes

  equations
    !Q0      := (Mode==Hold) & !Q0
    # (Mode==LoadA) & !A0
    # (Mode==LoadB) & !B0
    # (Mode==Incr) & !CI & !Q0          "Hold if no carry
    # (Mode==Incr) & CI & Q0;

    !Q1      := (Mode==Hold) & !Q1
    # (Mode==LoadA) & !A1
    # (Mode==LoadB) & !B1
    # (Mode==Incr) & !CI & !Q1          "Hold if no carry
    # (Mode==Incr) & !Q0 & !Q1          "Hold if Q0=L
    # (Mode==Incr) & CI & Q0 & Q1;

    !Q2      := (Mode==Hold) & !Q2
    # (Mode==LoadA) & !A2
    # (Mode==LoadB) & !B2
    # (Mode==Incr) & !CI & !Q2          "Hold if no carry
    # (Mode==Incr) & !Q0 & !Q2          "Hold if Q0=L
    # (Mode==Incr) & !Q1 & !Q2          "Hold if Q1=L
    # (Mode==Incr) & CI & Q0 & Q1 & Q2;

    !Q3      := (Mode==Hold) & !Q3
    # (Mode==LoadA) & !A3
    # (Mode==LoadB) & !B3
    # (Mode==Incr) & !CI & !Q3          "Hold if no carry
    # (Mode==Incr) & !Q0 & !Q3          "Hold if Q0=L
    # (Mode==Incr) & !Q1 & !Q3          "Hold if Q1=L
    # (Mode==Incr) & !Q2 & !Q3          "Hold if Q2=L
    # (Mode==Incr) & CI & Q0 & Q1 & Q2 & Q3;

    !CO      = !CI # !Q0 # !Q1 # !Q2 # !Q3;

```

```

@PAGE
test_vectors ' test Load A and B'
([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
[ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
[ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;
[ C, L, LoadA, ^h1 , ^h7 ,X ] -> ^h1;
[ C, L, LoadB, ^h1 , ^h7 ,X ] -> ^h7;
[ C, L, LoadA, ^h2 , ^hB ,X ] -> ^h2;
[ C, L, LoadB, ^h2 , ^hB ,X ] -> ^hB;
[ C, L, LoadA, ^h4 , ^hD ,X ] -> ^h4;
[ C, L, LoadB, ^h4 , ^hD ,X ] -> ^hD;
[ C, L, LoadA, ^h8 , ^hE ,X ] -> ^h8;
[ C, L, LoadB, ^h8 , ^hE ,X ] -> ^hE;
[ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
[ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;

test_vectors ' test increment'
([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
[ C, L, LoadB, X , ^h1 ,X ] -> ^h1;
[ C, L, Incr , X , X ,H ] -> ^h2;
[ C, L, LoadB, X , ^h3 ,X ] -> ^h3;
[ C, L, Incr , X , X ,H ] -> ^h4;
[ C, L, LoadA, ^h7 , X ,X ] -> ^h7;
[ C, L, Incr , X , X ,H ] -> ^h8;
[ C, L, LoadA, ^hF , X ,L ] -> ^hF;
[ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
[ C, L, LoadB, X , ^hC ,X ] -> ^hC;
[ C, L, Incr , X , X ,H ] -> ^hD;
[ C, L, Hold , X , X ,H ] -> ^hD;

test_vectors ' test carry'
([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
[ C, L, Incr , X , X ,H ] -> ^hE;
[ C, L, Incr , X , X ,H ] -> ^h1F;    "carryout
[ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
[ C, L, Incr , X , X ,H ] -> ^h1;
[ C, L, Incr , X , X ,L ] -> ^h1;      "no CarryIn
[ C, L, Incr , X , X ,H ] -> ^h2;
[ L, H, Hold , X , X ,X ] -> [X,Z,Z,Z,Z];
end count4

```

Multiple Equations to the Same Signal

When a signal (output pin or node) name appears on the left side of more than one equation, the two equations are ORed together to produce an equation that fully describes the logic function. You can use this ORing of equations to split a design description into functional pieces, each of which describes a distinct part of the design; these individual pieces are ORed together to describe the whole design.

Figure 10-13 shows the same 4-bit counter with 2-input multiplexer, but described by two separate equations sections. (Equations sections are begun by the keyword EQUATIONS.) The first equations section describes the multiplexing function, and the second describes the count function. Notice, however, that both groups of equations are written for the same outputs, Q0-Q3. The multiplexing equations for Q0 are ORed with the count equations for Q0, and together they describe the total function for that output. The same operation is performed for the other outputs to completely describe the design.

Figure 10-13
Multiple Equations Sections,
4-Bit Counter

```

module count4a
title '4-bit counter with 2 input mux
based on an example by Birkner/Coli in the MMI PAL Handbook
Dan Burrier, Mike McGee & Lisa Matheson Data I/O Corp.'

P7022A device 'P16R4';

Clk,OC,CO,I1,I0,CI      pin 1,11,12,13,18,19;
A0,A1,A2,A3,B0,B1,B2,B3 pin 2,3,4,5,6,7,8,9;
Q3,Q2,Q1,Q0              pin 14,15,16,17;

H,L,X,Z,C                = 1,0,.X.,.Z.,.C.;
InputA                   = [A3,A2,A1,A0];
InputB                   = [B3,B2,B1,B0];
Output                    = [CO,Q3,Q2,Q1,Q0];
Mode                      = [I1,I0];
Hold,LoadA,LoadB,Incr    = 0,1,2,3;      " define Modes

equations      " input multiplexer

  !Q0      := (Mode==Hold) & !Q0
            # (Mode==LoadA) & !A0
            # (Mode==LoadB) & !B0;

  !Q1      := (Mode==Hold) & !Q1
            # (Mode==LoadA) & !A1
            # (Mode==LoadB) & !B1;

  !Q2      := (Mode==Hold) & !Q2
            # (Mode==LoadA) & !A2
            # (Mode==LoadB) & !B2;

  !Q3      := (Mode==Hold) & !Q3
            # (Mode==LoadA) & !A3
            # (Mode==LoadB) & !B3;

  " 4 bit counter

  !Q0      := (Mode==Incr) & !CI & !Q0 "Hold if no carry
            # (Mode==Incr) & CI & Q0;

  !Q1      := (Mode==Incr) & !CI & !Q1 "Hold if no carry
            # (Mode==Incr) & !Q0 & !Q1 "Hold if Q0=L
            # (Mode==Incr) & CI & Q0 & Q1;

  !Q2      := (Mode==Incr) & !CI & !Q2 "Hold if no carry
            # (Mode==Incr) & !Q0 & !Q2 "Hold if Q0=L
            # (Mode==Incr) & !Q1 & !Q2 "Hold if Q1=L
            # (Mode==Incr) & CI & Q0 & Q1 & Q2;

  !Q3      := (Mode==Incr) & !CI & !Q3 "Hold if no carry
            # (Mode==Incr) & !Q0 & !Q3 "Hold if Q0=L
            # (Mode==Incr) & !Q1 & !Q3 "Hold if Q1=L
            # (Mode==Incr) & !Q2 & !Q3 "Hold if Q2=L
            # (Mode==Incr) & CI & Q0 & Q1 & Q2 & Q3;

  !CO      = !CI # !Q0 # !Q1 # !Q2 # !Q3;

```

```

test_vectors ' test Load A and B'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
  [ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;
  [ C, L, LoadA, ^h1 , ^h7 ,X ] -> ^h1;
  [ C, L, LoadB, ^h1 , ^h7 ,X ] -> ^h7;
  [ C, L, LoadA, ^h2 , ^hB ,X ] -> ^h2;
  [ C, L, LoadB, ^h2 , ^hB ,X ] -> ^hB;
  [ C, L, LoadA, ^h4 , ^hD ,X ] -> ^h4;
  [ C, L, LoadB, ^h4 , ^hD ,X ] -> ^hD;
  [ C, L, LoadA, ^h8 , ^hE ,X ] -> ^h8;
  [ C, L, LoadB, ^h8 , ^hE ,X ] -> ^hE;
  [ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
  [ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;

test_vectors ' test increment'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, LoadB, X , ^h1 ,X ] -> ^h1;
  [ C, L, Incr , X , X ,H ] -> ^h2;
  [ C, L, LoadB, X , ^h3 ,X ] -> ^h3;
  [ C, L, Incr , X , X ,H ] -> ^h4;
  [ C, L, LoadA, ^h7 , X ,X ] -> ^h7;
  [ C, L, Incr , X , X ,H ] -> ^h8;
  [ C, L, LoadA, ^hF , X ,L ] -> ^hF;
  [ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
  [ C, L, LoadB, X , ^hC ,X ] -> ^hC;
  [ C, L, Incr , X , X ,H ] -> ^hD;
  [ C, L, Hold , X , X ,H ] -> ^hD;

test_vectors ' test carry'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, Incr , X , X ,H ] -> ^hE;
  [ C, L, Incr , X , X ,H ] -> ^h1F;      "carry out
  [ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
  [ C, L, Incr , X , X ,H ] -> ^h1;
  [ C, L, Incr , X , X ,L ] -> ^h1;      "no carry in
  [ C, L, Incr , X , X ,H ] -> ^h2;
  [ L, H, Hold , X , X ,X ] -> [X,Z,Z,Z,Z];
end count4a

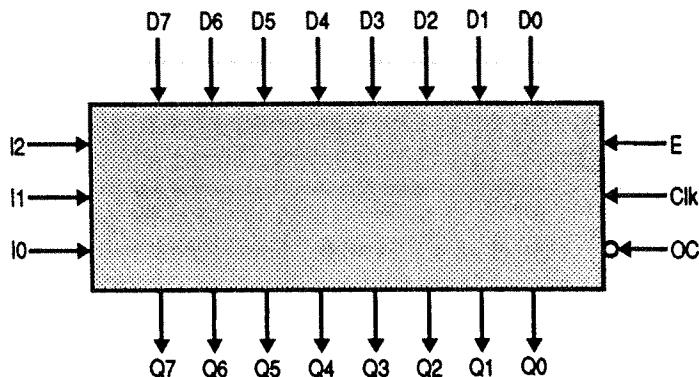
```

8-Bit Barrel Shifter

Equations, P20R8

The following design describes the implementation of an 8-bit barrel shifter including a shift amount selector, an output control, and a device enable. The design is specified with one high level equation using a P20R8.

Figure 10-14
Block Diagram: 8-Bit Barrel Shifter



Design Specification

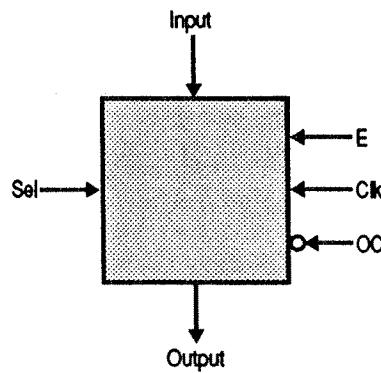
Figure 10-14 shows a block diagram for the barrel shifter. The shifter has eight inputs (D0-D7), eight outputs (Q0- Q7), three select lines (I0-I2), a clock (Clk), an output control (OC), and an enable (E). On each clock pulse when E is high, the outputs show the inputs shifted by n bits to the right, where n is specified by the select lines. The bit shifted out of the barrel shifter on the right is shifted in on the left, actually performing a rotate. When E is low, the shifter outputs are preset to 1.

The output control, when high, sets all outputs to high impedance, without affecting the shift. This means that if a shift is selected while the output control is high, the shift still occurs, but it is not seen at the outputs. If the OC is then set low, the shifted data will appear on the outputs.

Design Method

Figure 10-15 and Figure 10-16 show the simplified block diagram and the source file listing for this design. Pins are assigned so that the shifter outputs are associated with the registered outputs on the PAL. The inputs, outputs, and select lines are then assigned to sets to simplify notation.

Figure 10-15
Simplified Block Diagram: 8-Bit
Barrel Shifter



One high level equation is used to describe the entire function of the barrel shifter. The equation is expressed in sum of products form and assigns a value to the output set. Each product in the equation corresponds to one of the possible shifts and defines the outputs for that shift. Thus, the product term,

$$(Sel == 0) \& \\ [(D7, D6, D5, D4, D3, D2, D1, D0)]$$

defines that for a shift of 0, the inputs are transferred without a shift directly to the outputs. Similarly, the product term,

$$(Sel == 5) \& \\ [(D4, D3, D2, D1, D0, D7, D6, D5)]$$

defines that for a shift of 5, output Q7 gets the value of input D4, Q6 gets the value of D3, and so on, corresponding to the correct shift of 5 places. Notice that the low-order input bits have been "wrapped around," shifted out of the right side and into the left side.

Sel can have only one value at a time, thus only one of the "Sel == " relational statements can be true at a given time, and only one of the product terms contributes to the sum of products. The OR of all the product terms is ANDed with the enable E so that when E is low, all the outputs are preset to 1.

Both the output set on the left side of the equation and the inputs on the right side of the equation are expressed as negative logic, which, in effect, gives active high logic. This is done to compensate for the P20R8's inverted outputs.

Test Vectors

The test vectors check the shift, enable and output control functions of the barrel shifter. To test the shift function, OC is set low, E is set high, the clock is applied and different Sel values are chosen. The shift is first tested with one input bit set high and the rest of the inputs set low. Then, one input bit is set low and the remaining inputs are set high. In both cases, the bits are shifted through one full cycle plus one additional shift so the wraparound shift from Q0 to Q7 is tested.

The preset is tested by setting E low; all inputs should go high. The output control is tested by setting OC high; all outputs should go to high impedance. The single Z in the last test vector expands to cover all outputs: the Z becomes [Z,Z,Z,Z,Z,Z,Z] to cover all eight outputs.

Figure 10-16

Source File: 8-Bit Barrel Shifter

```

module barrel
title '8-bit barrel shifter
Gerrit Barrere Data I/O Corp Redmond WA'

P7095 device 'P20R8';

D7,D6,D5,D4,D3,D2,D1,D0      Pin 2,3,4,5,6,7,8,9;
Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0      Pin 15,16,17,18,19,20,21,22;
Clk,OC,E,I2,I1,I0              Pin 1,13,23,10,11,14;

Input      = [D7,D6,D5,D4,D3,D2,D1,D0];
Output     = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
Sel        = [I2,I1,I0];
H,L,C,Z   = 1,0,.C..Z.;

equations
Output.oe = !OC;

!Output := E & ( (Sel == 0) & !(D7,D6,D5,D4,D3,D2,D1,D0)
# (Sel == 1) & !(D0,D7,D6,D5,D4,D3,D2)
# (Sel == 2) & !(D1,D0,D7,D6,D5,D4,D3)
# (Sel == 3) & !(D2,D1,D0,D7,D6,D5,D4,D3)
# (Sel == 4) & !(D3,D2,D1,D0,D7,D6,D5,D4)
# (Sel == 5) & !(D4,D3,D2,D1,D0,D7,D6,D5)
# (Sel == 6) & !(D5,D4,D3,D2,D1,D0,D7,D6)
# (Sel == 7) & !(D6,D5,D4,D3,D2,D1,D0,D7)) ;

test_vectors
([Clk,OC, E, Sel, Input]    -> Output)
[ C, L, H, 0, ^b10000000]    -> ^b10000000;      " Shift 0
[ C, L, H, 1, ^b10000000]    -> ^b01000000;      " Shift 1
[ C, L, H, 2, ^b10000000]    -> ^b00100000;      " Shift 2
[ C, L, H, 3, ^b10000000]    -> ^b00010000;      " Shift 3
[ C, L, H, 4, ^b10000000]    -> ^b00001000;      " Shift 4
[ C, L, H, 5, ^b10000000]    -> ^b00000100;      " Shift 5
[ C, L, H, 6, ^b10000000]    -> ^b00000010;      " Shift 6
[ C, L, H, 7, ^b10000000]    -> ^b00000001;      " Shift 7
[ C, L, H, 0, ^b01111111]    -> ^b01111111;      " Shift 0
[ C, L, H, 1, ^b01111111]    -> ^b10111111;      " Shift 1
[ C, L, H, 3, ^b01111111]    -> ^b11101111;      " Shift 3
[ C, L, H, 7, ^b01111111]    -> ^b11111110;      " Shift 7
[ C, L, H, 1, ^b00000001]    -> ^b10000000;      " Shift 1/Wrap
[ C, L, H, 1, ^b11111110]    -> ^b01111111;      " Shift 1/Wrap
[ C, L, L, 0, ^b00000000]    -> ^b11111111;      " Preset
[ C, H, H, 0, ^b00000000]    -> Z;                  " Test High Z
end

```

7-Segment Display Decoder

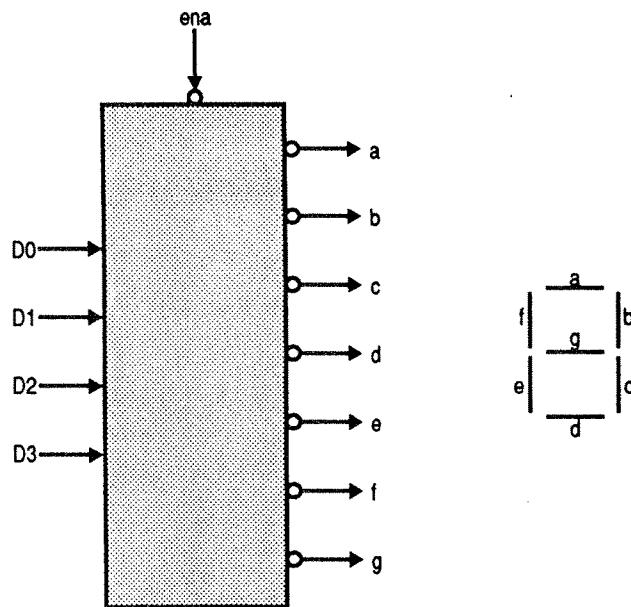
Truth Table, RA5P8

This display decoder decodes a four-bit binary number to display the decimal equivalent on a seven segment LED display. The design incorporates a truth table.

Design Specification

Figure 10-17 shows a block diagram for the design of a seven-segment display decoder and a drawing of the display with each of the seven segments labeled to correspond to the decoder outputs. To light up any one of the segments, the corresponding line must be driven low. Four input lines D0-D3 are decoded to drive the correct output lines. The outputs are named a, b, c, d, e, f, and g corresponding to the display segments. All outputs are active low. An enable, ena, is provided. When ena is low, the decoder is enabled; when ena is high, all outputs are driven to high impedance.

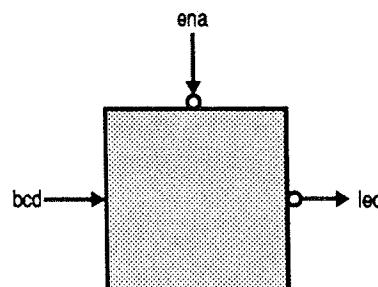
Figure 10-17
Block Diagram: Seven-Segment
Display Decoder



Design Method

Figure 10-18 and Figure 10-19 show the simplified block diagram and the source file for the ABEL-PLD implementation of the display decoder. The FLAG statement is used to make sure that the programmer load file is in the Motorola Exorciser format. The binary inputs and the decoded outputs are grouped into the sets bcd and led to simplify notation. The constants ON and OFF are declared so that the design can be described in terms of turning a segment on or off. To turn a segment on, the appropriate line must be driven low, thus we declare ON as 0 and OFF as 1.

Figure 10-18
Simplified Block Diagram:
Seven-Segment Display Decoder



The design is described in two sections, an equations section and a truth table section. The decoding function is described with a truth table that specifies the outputs required for each combination of inputs. The truth table header names the inputs and outputs. In this example, the inputs are contained in the set named bcd and the outputs are in led. The body of the truth table defines the input to output function.

Figure 10-19

Source file: 4-bit Counter with 2 Input Mux

```

module bcd7rom flag '-d82'
title 'seven segment display decoder'
Walter Bright Data I/O Corp Redmond WA'
"      a
"      --- BCD-to-seven-segment decoder similar to the 7449
"      f| g |b
"      --- segment identification
"      e| d |c
"      ---
U6      device 'RA5P8';

D3,D2,D1,D0      pin 10,11,12,13;
a,b,c,d,e,f,g    pin 1,2,3,4,5,6,7;
ena                pin 15;

bcd      = [D3,D2,D1,D0];
led      = [a,b,c,d,e,f,g];
ON,OFF   = 0,1;           " for common anode LEDs
L,H,X,Z = 0,1..X...Z.;

truth_table (bcd - led)
"      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 ([ena,bcd] - led)
"      a   b   c   d   e   f   g
[L,1] -> [OFF,  ON,  ON, OFF, OFF, OFF, OFF];
[L,2] -> [ ON,  ON, OFF,  ON,  ON, OFF,  ON];
[L,3] -> [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
[L,4] -> [OFF,  ON,  ON, OFF, OFF,  ON,  ON];
[L,5] -> [ ON, OFF,  ON,  ON, OFF,  ON,  ON];
[L,6] -> [ ON, OFF,  ON,  ON,  ON,  ON,  ON];
[L,7] -> [ ON,  ON,  ON, OFF, OFF, OFF, OFF];
[L,8] -> [ ON,  ON,  ON,  ON,  ON,  ON,  ON];
[L,9] -> [ ON,  ON,  ON,  ON, OFF,  ON,  ON];
[L,0] -> [ ON,  ON,  ON,  ON,  ON,  ON, OFF];
[H,5] -> [ Z,   Z,   Z,   Z,   Z,   Z,   Z];
end
bcd7rom

```

Because the design decodes a number to a seven segment display, values for bcd are expressed as decimal numbers, and values for led are expressed with the constants ON and OFF that were defined in the declarations section of the source file. This makes the truth table easy to read and understand; the incoming value is a number and the outputs are on and off signals to the LED.

The input and output values could have just as easily been described in another form. Take for example the line in the truth table:

5 -> [ON, OFF, ON , ON, OFF, ON, ON]

This could have been written in the equivalent form:

[0, 1, 0, 1] -> 36

In this second form, 5 was simply expressed as a set containing binary values, and the LED set was converted to decimal. (Remember that ON was defined as 0 and OFF was defined as 1.) Either of the two forms is valid, but the first is more appropriate for this design. The first form can be read as, "the number five turns on the first segment, turns off the second, . . ." whereas the second form cannot be so easily translated into terms meaningful for this design.

Test Vectors

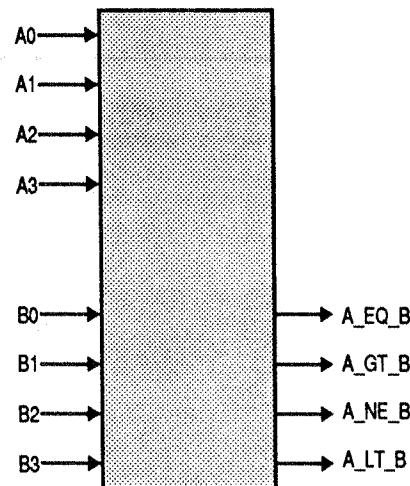
The test vectors for this design test the decoder outputs for the ten valid combinations of input bits. The enable is also tested by setting ena high for the different combinations. All outputs should be at high impedance whenever ena is high.

4-Bit Comparator

Macros, Equations, F153

This is a design for a 4-bit comparator that provides an output for "equal to", "less than", "not equal to", and "greater than", as well as intermediate outputs. The design is implemented with high level equations.

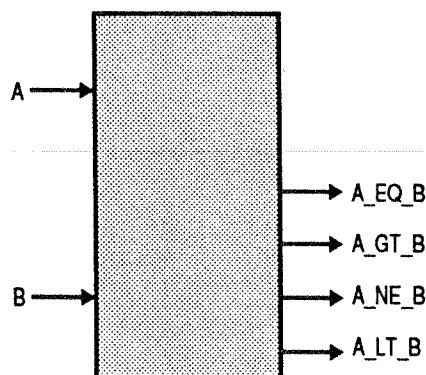
Figure 10-20
Block Diagram: 4-Bit
Comparator



Design Specification

The comparator, as shown in Figure 10-20, compares the values of two four-bit inputs (A0-A3 and B0-B3) and determines whether A is equal to, not equal to, less than, or greater than B. The result of the comparison is shown on the output lines, A_EQ_B, A_GT_B, A_NE_B, and A_LT_B.

Figure 10-21
Simplified Block Diagram: 4-bit
Comparator



Design Method

Figure 10-21 and Figure 10-22 show the simplified block diagram and source file listing for the comparator. The inputs A0-A3 and B0-B3 are grouped into the sets A and B. YES and NO are defined as 1 and 0, to be used in the test vectors.

The equations section of the source file contains the following equations:

```
A_EQ_B = A == B;
A_NE_B = !(A == B);
A_GT_B = A > B;
A_LT_B = !(A > B) # (A == B);
```

You could also use the following equations for the design of this comparator, however, many more product terms are used in the FPLA:

```
A_EQ_B = A == B;
A_NE_B = A != B;
A_GT_B = A > B;
A_LT_B = A < B;
```

The first set of equations takes advantage of product term sharing within the target FPLA, while the latter set requires a different set of product terms for each equation. For example, the equation

`A_NE_B = !(A == B);`

uses the same 16 product terms as the equation

`A_EQ_B = A == B;`

thereby reducing the number of product terms. In a similar manner, the equation

`A_LT_B = !(A > B) # (A == B);`

uses the same product terms as equations

```
A_EQ_B = A == B;
A_GT_B = A > B;
```

whereas the equation

`A_LT_B = A < B;`

in the second set of equations requires the use of additional product terms. Sharing product terms in devices that allow this type of design architecture can serve to fit designs into smaller and less expensive logic devices.

Figure 10-22
Source File: 4-Bit Comparator

```

module _comp4a
flag '-r4'
title '4-bit look-ahead comparator'
Steve Weil & Gary Thomas Data I/O Corp.

comp4a device 'F153';

A3,A2,A1,A0      pin 1,2,3,4;      A = [A3,A2,A1,A0];
B3,B2,B1,B0      pin 5,6,7,8;      B = [B3,B2,B1,B0];

A_NE_B,A_EQ_B,A_GT_B,A_LT_B      pin 14,15,16,17;
No,Yes = 0,1;

equations
  A_EQ_B = A == B;
  A_NE_B = !(A == B);
  A_GT_B = A > B;
  A_LT_B = !((A > B) # (A == B));

test_vectors 'test for A = B'
  ([ A, B ] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
  [ 0, 0 ] -> [ Yes, No, No, No ];
  [ 1, 1 ] -> [ Yes, No, No, No ];
  [ 2, 2 ] -> [ Yes, No, No, No ];
  [ 5, 5 ] -> [ Yes, No, No, No ];
  [ 8, 8 ] -> [ Yes, No, No, No ];
  [10,10] -> [ Yes, No, No, No ];
  [15,15] -> [ Yes, No, No, No ];

test_vectors 'test for A > B'
  ([ A, B ] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
  [ 1, 0 ] -> [ No, Yes, No, Yes ];
  [ 2, 1 ] -> [ No, Yes, No, Yes ];
  [ 4, 3 ] -> [ No, Yes, No, Yes ];
  [ 8, 7 ] -> [ No, Yes, No, Yes ];
  [15,14] -> [ No, Yes, No, Yes ];
  [ 6, 2 ] -> [ No, Yes, No, Yes ];
  [ 5, 0 ] -> [ No, Yes, No, Yes ];

test_vectors 'test for A < B'
  ([ A, B ] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
  [ 3, 9 ] -> [ No, No, Yes, Yes ];
  [14,15] -> [ No, No, Yes, Yes ];
  [ 7, 8 ] -> [ No, No, Yes, Yes ];
  [ 3, 4 ] -> [ No, No, Yes, Yes ];
  [ 2, 8 ] -> [ No, No, Yes, Yes ];

end _comp4a

```

Test Vectors

Three separate test vectors sections are written to test three of the four possible conditions. (The fourth and untested condition of NOT EQUAL TO is simply the inverse of EQUAL TO.) Each test vectors table includes a test vector message that helps make output from the documentation generator (DOCUMENT) and the simulator (SIMULATE) easier to read.

The three tested conditions are not mutually exclusive, so one or more of them can be met by a given A and B. In the test vectors table, the constants YES and NO are used rather than 1 and 0, just for ease of reading. YES and NO are declared in the declaration section of the source file.

Blackjack Machine

This section contains an advanced logic design described with ABEL-PLD and builds upon examples and concepts presented in the earlier sections of this manual. This design, a blackjack machine, is the combination of more than one basic logic design. Design specification, methods, and complete source files are given for all parts of the blackjack machine example, which contains the following logic designs:

- Multiplexer
- 5-bit adder
- Binary to BCD converter
- State machine

This example is a classic blackjack machine based on C.R. Clare's design in *Designing Logic Systems Using State Machines* (McGraw Hill, 1972). The blackjack machine plays the dealer's hand, using typical dealer strategies to decide, after each round of play, whether to draw another card or stand.

The blackjack machine consists of these functions: a card reader that reads each card as it is drawn, control logic that tells it how to play each hand (based on the total point value of the cards currently held), and display logic that displays scores and status on the machine's four LEDs. (For the purposes of this example, we are assuming that the two digital display devices used to display the score have built-in seven-segment decoders.)

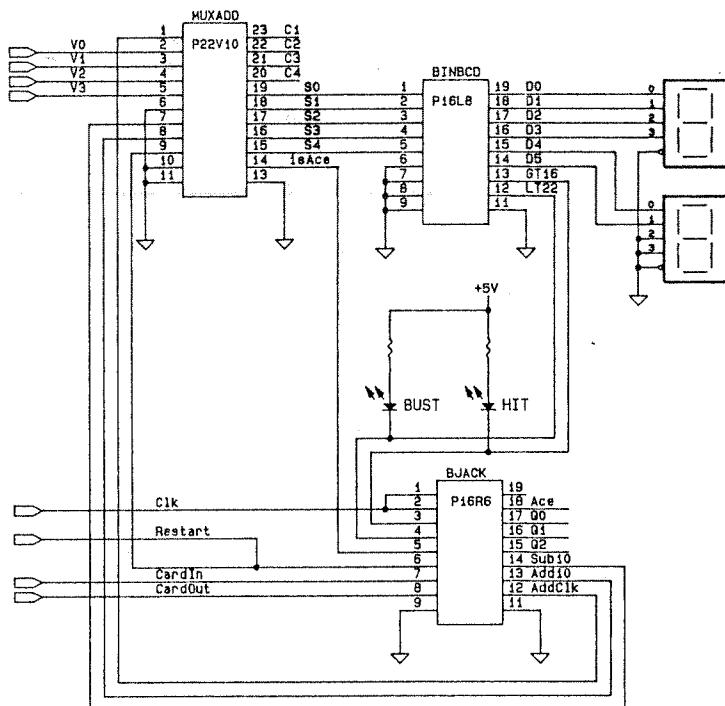
To operate the machine, you insert the dealer's card into the card reader. The machine reads the value and, in the case of later card draws, adds it to the values of previously read cards for that hand. (Face cards are valued at 10 points, non-face cards are worth their face value, and aces are counted as either 1 or 11, whichever count yields the best hand.) If the point total is 16 or less, the GT16 line will be asserted (active low) and the Hit LED will light up. This indicates that the dealer should draw another card. If the point total is greater than 16 but less than 22, no LEDs will light up (indicating that the dealer should draw no new cards). If the point total is 22 or higher, LT22 will be asserted (active low) and the Bust LED will light (indicating that the dealer has lost the hand).

As Figure 10-23 shows, the blackjack machine is implemented in three PLDs:

1. A multiplexer-adder-comparator, which adds the value of the newly drawn card to the existing hand (and indicates an ace to the state machine);
2. A binary to binary-coded-decimal (BCD) converter, which takes in the five-bit binary score and converts it to two-digits of BCD for the digital display; and

3. The blackjack controller, a state machine that contains the game logic. This logic includes instructions that determine when to add a card value, when to count an ace as 1, and when to count an ace as 11.

Figure 10-23
Schematic of a Blackjack
Machine Implemented in Three
PLDs



Circuits that are a straightforward function of a set of inputs and outputs are often most easily expressed in equations; the adder is such a circuit. The PLD for the adder function (identified as MUXADD in Figure 10-23) includes three elements: a multiplexer, the adder itself, and a comparator. The multiplexer selects either the value of the newly dealt card or one of the two fixed values used for the ace (ADD10 or SUB10). The adder adds the value selected by the multiplexer to the previous score when triggered by the clock signal, ADDCLK. The comparator detects when an ace is present and passes this information on to the blackjack controller, BJACK.

Designs with outputs that do not follow a specific pattern are most easily expressed as truth tables. Such is the case with the binary-to-BCD converter (identified in the schematic, Figure 10-23), as BINBCD). This PLD converts five bits of binary input to BCD output for two digital display elements.

The following text describes the internal logic design necessary to keep the card count, to control the play sequence, and to show the count on the digital display, or the state on the Hit and Bust LEDs. Neither the card reader nor the physical design is discussed here. It is assumed that the card reader provides a binary value representative of the card read.

The design has eight inputs, four of which are the binary encoded card values, V0-V3. The remaining four inputs are signals that indicate that the machine is to be restarted (Restart), that a card is in the reader (CardIn), that no card is in the reader (CardOut), and a clock signal (Clk) to synchronize the design to the card reader. CardIn, CardOut, and Clk are provided by the card reader. Restart is provided by a switch on the exterior of the machine.

Device	Function in the Blackjack Machine
P22V10	Multiplexer/Adder/Comparator
P16L8	Binary-BCD converter
P16R4	State machine

Design Specification — MUXADD

MUXADD consists of an input multiplexer, an adder, and a comparator. The multiplexer determines what value is added to the current score (by the adder); the value being

1. The contents of the external card reader (V0-V1 declared as Card);
2. A numeric value of +10; or
3. A numeric value of -10.

Inputs Add10 and Sub10 from the controller (state machine) BJACK determine which of the three values the multiplexer selects for application to the adder. Card is applied to the adder when Add10 and Sub10 are active high, as generated by the BJACK controller. When Add10 becomes active low, 10 is added to the current score (to count an ace as 11 instead of 1), and when Sub10 is active low, -10 is added to the current score (to count an ace as 1 instead of 11).

The adder provides an output named Score (S0-S4, which is the sum of the current adder contents and the value selected by the input multiplexer; i.e., the card reader contents, +10, or -10. The comparator monitors the contents of the external card reader (Card) and generates an output, is_Ace, to the BJACK controller that signifies that an ace is present in the card reader.

Design Method — MUXADD

MUXADD is implemented in a P22V10, and consists of a three-input multiplexer, a five-bit ripple adder, and a five-bit comparator. These circuit elements are defined in the equations shown in Figure 10-24. For the multiplexer inputs, a set named Card defines inputs V0 through V4 as the value of the card reader, while inputs Add10 and Sub10 are used directly in the following equations to define the multiplexer. The multiplexer output to the adder is named Data and is defined by the equations

```
Data      = Add10 & Sub10 & Card
# !Add10 & Sub10 & ten
# Add10 & !Sub10 & minus_ten;
```

The adder (MUXADD) contained in the P22V10 is a five-bit binary ripple adder that adds the current input from the multiplexer to the current score, with carry. The adder is clocked by a signal (AddClk) from the BJACK controller and is described with the following equations:

```
Score    := Data $ Score $ CarryIn;
CarryOut = Data & Score # (Data # Score) & CarryIn;
Reset   = !Clr;
```

In the above equations, Score is the sum of Data (the card reader output, value of ten, or value of minus ten), Score (the current or last calculated score), and CarryIn (the shifted value of CarryOut described below). The new value of Score appears at the S0 through S4 outputs of MUXADD at the time of the AddClk pulse generated by the BJACK controller (state machine).

Before the occurrence of the AddClk clock pulse, an intermediate adder output appears at combinatorial outputs of the P22V10 labeled C0 through C4 and defined as the set named CarryOut shown below. A second set named CarryIn defines the same combinatorial outputs as CarryOut, but with the outputs shifted one bit to the left as shown below.

```
CarryIn  = [C4,C3,C2,C1, 0];
CarryOut = [ X,C4,C3,C2,C1];
```

That is, the set declarations define CarryIn as CarryOut but with the required shift to the left for application back to adder input. At the time of the AddClk pulse from the BJACK controller, CarryIn is added to Score and Data by an exclusive-or operation.

The comparator portion of MUXADD is defined with

```
is_Ace  = Card == 1;
```

which provides an input to the BJACK controller whenever the value provided by the card reader is 1.

Test Vectors — MUXADD

The test vectors shown in Figure 10-24 verify operation of MUXADD by first clearing the adder so that Score is zero, then adding card reader values 7 and 10. The test vectors then input an ace (1) from the card reader (Card) to produce a Score of 1 and pull the is_Ace output high. Subsequent vectors verify the -10 function of the input multiplexer and adder.

Figure 10-24

Source File:

Multiplexer/Adder/Comparator

```

module _MuxAdd
flag '-r3'
title '5-bit ripple adder with input multiplex
Michael Holley    Data I/O Corp.    Redmond WA

MuxAdd device 'P22V10';

AddClk,Clr,Add10,Sub10,is_Ace    pin 1, 9, 8, 7,14;
V4,V3,V2,V1,V0                  pin 6, 5, 4, 3, 2;
S4,S3,S2,S1,S0                  pin 15,16,17,18,19;
C4,C3,C2,C1                     pin 20,21,22,23;
Reset node 25;

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

Card      = [V4,V3,V2,V1,V0];
Score     = [S4,S3,S2,S1,S0];
CarryIn   = [C4,C3,C2,C1, 0];
CarryOut  = [ X,C4,C3,C2,C1];
ten       = [ 0, 1, 0, 1, 0];
minus_ten = [ 1, 0, 1, 1, 0];

" Input Multiplexer
Data      = Add10 & Sub10 & Card
           # !Add10 & Sub10 & ten
           # Add10 & !Sub10 & minus_ten;
equations
Score    := Data $ Score $ CarryIn;

CarryOut = Data & Score # (Data # Score) & CarryIn;

Reset    = !Clr;          "Async reset node for registers

is_Ace   = Card == 1;

test_vectors
([Ad dClk,Clr,Add10,Sub10,Card] -> [Score,is_Ace])
[ L , L , H , H , X ] -> [ 0 , L ]; "Clear
[ C , H , H , H , 7 ] -> [ 7 , L ];
[ C , H , H , H , 10 ] -> [ 17 , L ];
[ L , L , H , H , X ] -> [ 0 , L ]; "Clear
[ C , H , H , H , 1 ] -> [ 1 , H ];
[ C , H , L , H , 1 ] -> [ 11 , H ]; "Add 10
[ C , H , H , H , 4 ] -> [ 15 , L ];
[ C , H , H , H , 8 ] -> [ 23 , L ];
[ C , H , H , L , 8 ] -> [ 13 , L ]; "Subtract 10
[ C , H , H , H , 5 ] -> [ 18 , L ];
end _MuxAdd

```

Design Specification — BINBCD

For display of the Score appearing at the output of MUXADD, a binary to bcd converter is implemented in a P16L8. It is the function of the converter to accept the four lines of binary data generated by MUXADD and provide two sets of binary coded decimal outputs for two bcd display devices; one to display the units of the current score, and the other to display the tens. The four-bit output bcd1 (D0-D3) contains the units of the current score, and is connected to the high-order display digit. The two-bit output bcd2 (D4 and D5) contains the tens, and is fed to the low-order display digit.

BINBCD also provides a pair of outputs to light the Bust and Hit LEDs. Bust is lit whenever Score is 22 or greater; while Hit is lit whenever Score is 16 or less.

Design Method — BINBCD

The design of BINBCD is shown in the source file of Figure 10-25. The design of the converter is easily expressed with a truth table that lists the value of Score (inputs S0 through S4 are declared as Score) for values of bcd1 and bcd2. bcd1 and bcd2 are sets that define the outputs that are fed to the two digital display devices. The truth table lists Score values up to decimal 31.

The truth table represents a method of expressing the design "manually." You could use a macro to create the truth table such as

```
clear(binary);
@repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

As indicated in Figure 10-25 and described in the next paragraph, this macro is used to generate the test vectors for the converter. You can examine the result of the macro by running the design with ABEL-PLD and the -e (expand) option. The generated *.LST and *.OUT files show the truth table created from the macro.

The BINBCD design also provides the outputs LT22 and GT16 to control the Bust and Hit LEDs. A pair of equations generate an active-high LT22 signal to turn off the Bust LED whenever Score is less than 22, and an active-high GT16 signal to turn off the Hit LED whenever Score is greater than 16.

Test Vectors — BINBCD

The test vectors shown in Figure 10-25 verify operation of the LT22 and GT16 outputs of the converter by assigning various values for Score and checking for the corresponding outputs.

The test vectors for the binary to bcd converter are defined by means of the following macro:

```
test_vectors ( score - [bcd2,bcd1])
    clear(binary);
    @repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

This macro generates a test vector with the variable binary set to 0 by the macro (a) {@const ?a=0}; (contained in the binbcd.abl source file shown in Figure 10-25), followed by additional vectors provided by the @repeat directive. The latter vectors are generated by incrementing the value of the variable binary by a factor of 1 (see inc macro (a) {@const ?a=?a+1;}); in Figure 10-25 for each vector. On the output side of the test vectors, the division arithmetic operation (/) is used to create the output for bcd2 (tens display digit), while the remainder from (modulus) operator is used to create the output for bcd1 (units display digit). See the binbcd.tmv file after running binbcd.abl through Parse with the -e option.

The use of macros and directives to create test vectors is described in more detail in the chapter "Advanced Features."

Figure 10-25
Source file: 4-bit Counter with 2
Input Mux

```

module _binbcd
flag '-r3'
title 'comparator and binary to bcd decoder for Blackjack Machine
Michael Holley Data I/O Corp'

" The 5 -bit binary (0 - 31) score is converted into two BCD outputs
.
" The integer division '//' and the modulus operator '%' are used to
" extract the individual digits from the two digit score.
" 'Score % 10' will yield the 'units' and
" 'Score / 10' will yield the 'tens'
"
"The 'GT16' and 'LT22' outputs are for the state machine controller.

binbcd device 'P16L8';

S4,S3,S2,S1,S0 pin 5,4,3,2,1;
score           = [S4,S3,S2,S1,S0];

LT22,GT16      pin 12,13;

D5,D4          pin 14,15;
bcd2           = [D5,D4];

D3,D2,D1,D0    pin 16,17,18,19;
bcd1           = [D3,D2,D1,D0];

" Digit separation macros
binary          = 0;      "scratch variable
clear  macro (a) {@const ?a=0};
inc    macro (a) {@const ?a=?a+1;};

equations
LT22   = (score  22);    "Bust
GT16   = (score  16);    "Hit / Stand

test_vectors ( score -> [GT16,LT22])
1     -> [ 0 , 1 ];
6     -> [ 0 , 1 ];
8     -> [ 0 , 1 ];
16    -> [ 0 , 1 ];
17    -> [ 1 , 1 ];
18    -> [ 1 , 1 ];
20    -> [ 1 , 1 ];
21    -> [ 1 , 1 ];
22    -> [ 1 , 0 ];
23    -> [ 1 , 0 ];
24    -> [ 1 , 0 ];

```

```

@page truth_table ( score -> [bcd2,bcd1])
    0 -> [ 0 , 0 ];
    1 -> [ 0 , 1 ];
    2 -> [ 0 , 2 ];
    3 -> [ 0 , 3 ];
    4 -> [ 0 , 4 ];
    5 -> [ 0 , 5 ];
    6 -> [ 0 , 6 ];
    7 -> [ 0 , 7 ];
    8 -> [ 0 , 8 ];
    9 -> [ 0 , 9 ];
   10 -> [ 1 , 0 ];
   11 -> [ 1 , 1 ];
   12 -> [ 1 , 2 ];
   13 -> [ 1 , 3 ];
   14 -> [ 1 , 4 ];
   15 -> [ 1 , 5 ];
   16 -> [ 1 , 6 ];
   17 -> [ 1 , 7 ];
   18 -> [ 1 , 8 ];
   19 -> [ 1 , 9 ];
   20 -> [ 2 , 0 ];
   21 -> [ 2 , 1 ];
   22 -> [ 2 , 2 ];
   23 -> [ 2 , 3 ];
   24 -> [ 2 , 4 ];
   25 -> [ 2 , 5 ];
   26 -> [ 2 , 6 ];
   27 -> [ 2 , 7 ];
   28 -> [ 2 , 8 ];
   29 -> [ 2 , 9 ];
   30 -> [ 3 , 0 ];
   31 -> [ 3 , 1 ];

" This truth table could be replaced with the following macro.
"     clear(binary);
"     @repeat 32 {
"         binary -> [binary/10,binary%10]; inc(binary);}

"
" The test vectors will demonstrate the use of the macro.

test_vectors ( score -> [bcd2,bcd1])
    clear(binary);
    @repeat 32 {
        binary -> [binary/10,binary%10]; inc(binary);}
end _binbcd

```

Design Specification — BJACK

BJACK (the blackjack controller) is technically a state machine; that is, a circuit capable of storing an internal state reflecting prior events. State machines use sequential logic, branching to new states and generating outputs on the basis of both the stored states and external inputs.

In the case of the controller, the state machine stores states that reflect the following blackjack machine conditions:

- the value of Score (in one of the following ranges of decimal values): 0 to 16, 17 to 21, or 22+
- the status of the card reader (card in or card out)
- ace present in the card reader.

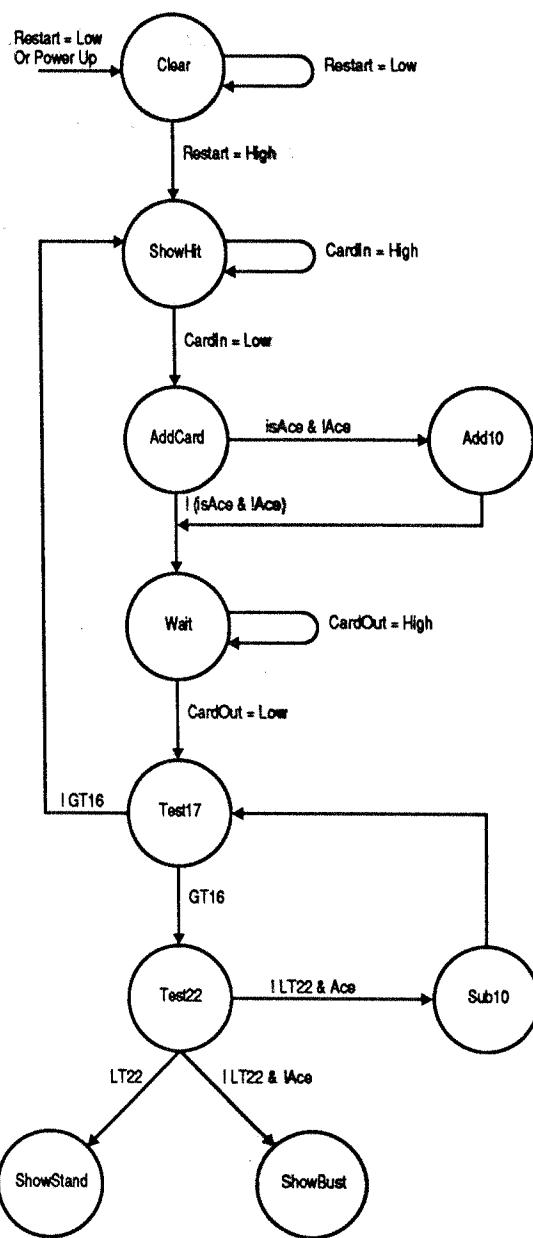
On the basis of these stored states, plus input from each newly drawn card, the blackjack controller decides whether or not a +10 or -10 value is to be sent to the adder.

Design Method — BJACK

The ability of ABEL-PLD to accept design input in a variety of forms is especially helpful in the case of state machines. The easiest way to express a state machine design is with a state diagram. Any other form of design expression would be tedious to develop and would be likely to contain errors.

In describing a state machine, the first step is to develop a state diagram. Figure 10-26 shows a pictorial state diagram for the controller that indicates state transitions and the conditions that cause those transitions. Transitions are shown by arrows, and the conditions causing the transitions are written alongside the arrow.

Figure 10-26
Pictorial State Diagram:
Blackjack Machine



You must then express the state diagram in the form shown in state_diagram portion of Figure 10-27. There is a one-to-one correlation between the pictorial state diagram and the state diagram described in the source file (Figure 10-27). The table below provides a more detailed description of the state identifiers (state machine states) illustrated in the state diagram and listed in the source file.

State Identifier	Description
Clear	Clear the state machine, adder, and displays.
ShowHit	Indicate that another card is needed. Hit indicator is lit.
AddCard	Add the value present at the adder input to the current count.
Add10	Add the fixed value 10 to the current count, effectively giving an ace a value of 11.
Wait	Wait until a card is taken out of the reader.
Test17	Test the current count for a value less than 17.
Test22	Test the current count for a value less than 22.
Sub10	Add the fixed value -10 to the current count, effectively subtracting 10 and restoring an ace to 1 after it was an 11.
ShowBust	Indicate that no more cards are needed. Bust indicator is lit.
ShowStand	Indicate that no more cards are needed. Neither Hit nor Bust indicators are lit.

Note that in Figure 10-27, each of the state identifiers (For example, Clear and ShowHit) are defined as sets having binary values. These values were chosen to minimize the number of product terms used in the P16R6.

Operation of the state machine proceeds as follows if no aces are drawn: If a card is needed from the reader, the state machine goes to state ShowHit. When CardIn goes low, meaning that a card has been read, a transition to state AddCard is made. The card value is added to the current score or count value. The machine goes to Wait state until the card is withdrawn from the reader. The machine goes to Test17 state. If the score is less than 17, another card is drawn. If the score is greater than or equal to 17, the machine goes to state Test22. If the score is less than 22, the machine goes to the ShowStand state. If the score is 22 or greater, a transition is made to the ShowBust state. In either the ShowStand or ShowBust state, a transition is made to Clear (to clear the state register and adder) when Restart goes low. When Restart goes back to high, the state machine returns to ShowHit state and the cycle begins again.

Operation of the state machine when an ace is drawn is essentially the same. A card is drawn and the score is added. If the card is an ace and no ace has been drawn previously, the state machine goes to state Add10, and ten is added to the count, in effect making the ace an 11. Transitions to and from Test17 and Test22 proceed as before. However, if the score exceeds 21, and an ace has been set to 11, the state machine goes to state Sub10, 10 is subtracted from the score, and the state machine goes to state Test17.

Test Vectors — BJACK

Figure 10-27 shows three sets of test vectors; each set represents a different "hand" of play (as described above the set of vectors) and tests the different functions of the design. The Restart function is used to set the design to a known state between each hand and the state identifiers are used instead of the binary values which they represent.

Figure 10-27
Source File: State Machine
(Controller)

```

module _Bjack
flag '-r3'
title 'BlackJack state machine controller
Michael Holley & Kyu Lee Data I/O Corp. 23 Mar 1987'

    Bjack    device  'P16R6';

"Inputs
    Clk,ClkIN      pin 1,2;
"System clock
    GT16,LT22      pin 3,4;
"Score less than 17 and 22
    is_Ace         pin 5;
"Card is ace
    Restart        pin 6;
"Restart game
    CardIn,CardOut pin 7,8;
"Card present switches
    Ena            pin 11;

    Sensor          = [CardIn,CardOut];
    In              = [ 0 , 1 ];
    InOut           = [ 1 , 1 ];
    Out             = [ 1 , 0 ];

"Outputs
    AddClk         pin 12;
"Adder clock
    Add10          pin 13;
"Input Mux control
    Sub10          pin 14;
"Input Mux control
    Q2,Q1,Q0       pin 15,16,17;
    Ace            pin 18;
"Ace Memory

    High,Low        = 1,0;
    H,L,C,X        = 1,0,.C.,.X.;

"test vector characters

    Qstate          = [Add10,Sub10,Q2,Q1,Q0];
    Clear           = [ 1 , 1 , 1 , 1 , 1 ];
    ShowHit         = [ 1 , 1 , 1 , 1 , 0 ];
    AddCard         = [ 1 , 1 , 0 , 0 , 0 ];
    Add_10          = [ 0 , 1 , 0 , 0 , 0 ];
    Wait            = [ 1 , 1 , 0 , 0 , 1 ];
    Test_17         = [ 1 , 1 , 0 , 1 , 0 ];
    Test_22         = [ 1 , 1 , 0 , 1 , 1 ];
    ShowStand       = [ 1 , 1 , 1 , 0 , 0 ];
    ShowBust        = [ 1 , 1 , 1 , 0 , 1 ];
    Sub_10          = [ 1 , 0 , 0 , 0 , 1 ];

equations
    Qstate := Clear & !Restart;

```

```
@page
state_diagram Qstate

State Clear:    AddClk      = !ClkIN;
                Ace        := Low;
                goto ShowHit;

State ShowHit:   AddClk      = Low;
                Ace        := Ace;
                if (CardIn==Low) then AddCard else ShowHit;

State AddCard:   AddClk      = !ClkIN;
                Ace        := Ace;
                if (is_Ace & !Ace) then Add_10 else Wait;

State Add_10:    AddClk      = !ClkIN;
                Ace        := High;
                goto     Wait;

State Wait:      AddClk      = Low;
                Ace        := Ace;
                if (CardOut==Low) then Test_17 else Wait;

State Test_17:   AddClk      = Low;
                Ace        := Ace;
                if !GT16 then ShowHit else Test_22;

State Test_22:   AddClk      = Low;
                Ace        := Ace;
                case     LT22 : ShowStand;
                !LT22 & !Ace : ShowBust;
                !LT22 & Ace  : Sub_10;
                endcase;

State Sub_10:    AddClk      = !ClkIN;
                Ace        := Low;
                goto Test_17;

State ShowBust:  AddClk      = Low;
                Ace        := Ace;
                goto ShowBust;

State ShowStand: AddClk      = Low;
                Ace        := Ace;
                goto ShowStand;
```

```

@page test_vectors 'Assume two cards that total between 16 and 21'
([Ena,Clk,ClkIN,GT16,LT22,is_Ace,Restart,Sensor] -> [Ace,Qstate,AddClk])
[ L , C , L , L , H , L , L , Out ] -> [ X ,Clear , H ];
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];

>
[ L , C , L , L , H , L , H , InOut ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , H , H , L , H , _In ] -> [ L ,Wait , L ];
[ L , C , L , H , H , L , H , InOut ] -> [ L ,Wait , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,Test_17 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,Test_22 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,Clear , H ];

test_vectors 'Assume 2 Aces and another card that total between 16 and 21'
([Ena,Clk,ClkIN,GT16,LT22,is_Ace,Restart,Sensor] -> [Ace,Qstate,AddClk])
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];

[ L , C , L , L , H , H , H , InOut ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,Add_10 , H ];
[ L , C , L , L , H , H , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];

[ L , C , L , L , H , H , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , L , H , H , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];

[ L , C , L , L , H , L , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , H , H , L , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , H , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,Test_22 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,Clear , H ];

```

```
@page
test_vectors 'Assume an Ace and 2 cards that total between 16 and 21'
([Ena,Clk,ClkIN,GT16,LT22,is_Ace,Restart,Sensor] -> [Ace,Qstate,AddClk])
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , H , H , InOut ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,Add_10 , H ];
[ L , C , L , L , H , H , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];

[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , L , H , L , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];

[ L , C , L , L , H , L , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , H , L , L , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , H , L , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , H , L , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , H , L , L , H , Out ] -> [ H ,Test_22 , L ];
[ L , C , L , H , L , L , H , Out ] -> [ H ,Sub_10 , H ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,Test_17 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,Test_22 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , H , H , L , L , Out ] -> [ L ,Clear , H ];
end
```

Chapter 11

Advanced Devices

This chapter contains information that will help you use the features of the more advanced programmable logic devices. This section includes devices with enable-type outputs, configurable output macro cells, and the use of internal device nodes.

Output Enables

There are three types of output enables used in programmable logic devices: pin controlled, term controlled, and configurable. Each of these output enables, and how it is implemented in ABEL-PLD, is discussed in the following paragraphs.

Pin Controlled Output Enable

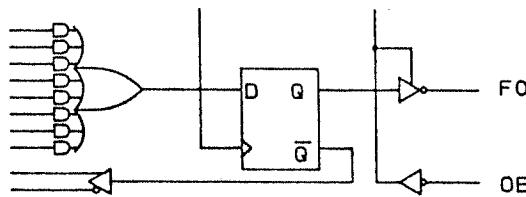
The pin controlled output enable is illustrated in Figure 11-1. Here, the output enable input is a specific device pin; F0 is high-impedance for OE, and is active for !OE. The pin controlled output enable is not included in the design equations but should be included in all test vectors.

Term Controlled Output Enable

An example of a term controlled output enable is shown in Figure 11-2. In this example, the product term that enables pin F0 is A&B. The equation in the ABEL-PLD source file would be

$$F0.OE = A \& B;$$

Figure 11-1
Output Enable Controlled by
Device Pin

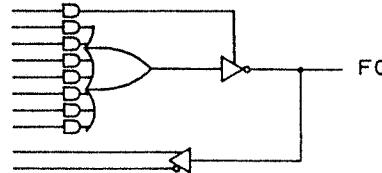


See the chapter "Advanced Features" for more information on dot extension notation.

For devices that have a negative output enable, a ! can be placed on either side of the equation as follows

$$\begin{aligned} \text{!F0.OE} &= A \& B; \\ \text{F0.OE} &= !(A \& B); \end{aligned}$$

Figure 11-2
Output Enable Controlled by
Product Term



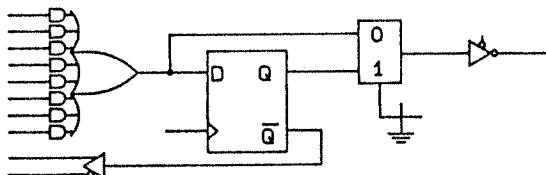
Configurable Output Enable

The configurable output enable is one that, through the use of a device feature called a 'select multiplexer' can be made to operate in any of four distinct modes:

- All related outputs are controlled by a device pin.
- All related outputs are controlled by a device term.
- All related outputs are enabled at all times.
- All related outputs are disabled at all times.

Figure 11-3 shows a typical select multiplexer that can select any one of the four modes, by means of a pair of fuses. For devices with this type of output enable, you may specify the type of enable you wish to have on the device with a combination of ISTYPE statements and equations.

Figure 11-6
Macro Cell, Configurable to
Combinatorial or Registered
Output



Controlling Register Type

For devices with selectable register types, the register type may be selected by using one of the attributes "reg_D", "reg_JK", "reg_RS", "reg_T" or "reg_G" to indicate which register type is desired. For example, to select a RS type register for an output in the F105, use the statement:

```
F0 istype 'reg_RS';
```

The selection of a register type may be made independently of whether the register has been bypassed (istype 'com') or selected (istype 'reg').

Selectable Product Term Sharing

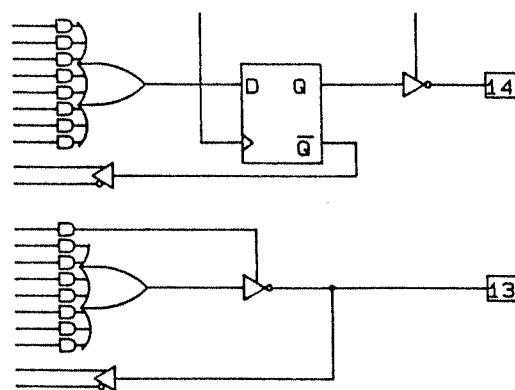
Some devices allow you to specify whether an output may share product terms from another output or internal node. To specify that the terms are to be shared, use the statement

```
F0,F1 istype 'share';
```

More On Feedback

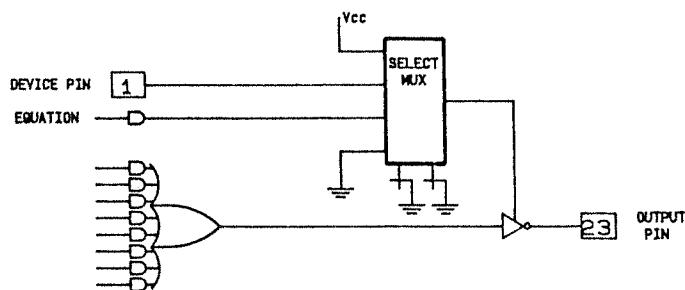
Most programmable logic devices provide feedback from some or all of their outputs. In simple cases, you need only refer to the output pin name as an input in your design equations, and the feedback will be properly used. Figure 11-7 shows a portion of a P16R4 logic diagram and illustrates how the feedback is routed in the device for registered and combinatorial output pins. We have seen in earlier examples such as the 4-bit counter/multiplexer described in the design examples, how the feedback is used for devices such as this. In some cases, however, you will need to use more explicit syntax to indicate what kind of feedback you require.

Figure 11-7
Registered and Combinatorial
Feedback (P16R4)



Pin	To configure the output enable for pin F0 to be controlled by a device pin, add the following ISTYPE statement to the declarations section of your source file: <code>F0.OE istype 'pin';</code>
	and, if the device allows you to choose a pin or pin polarity for the enable, write an equation to specify which pin is to be used, and its polarity: <code>F0.OE = IOE;</code>
Term	To specify that an output enable is to be controlled by a term in the device, declare the enable as follows: <code>F0.OE istype 'eqn';</code>
	and write an equation for the enable: <code>F0.OE = A & B;</code>
Enable or Disable	To specify that an output is to be always enabled or disabled, declare the enable with <code>F0.OE istype 'fuse';</code>
	and indicate the desired state of the output enable by writing an equation of the form <code>F0.OE = 1;</code>
	or <code>F0.OE = 0;</code>
	For most devices and designs, the use of the ISTYPE statement to configure the selectable enable is optional since ABEL-PLD can normally determine the type of output enable required from the form of equation you write.
	More detailed information on the use of select multiplexers can be found in "Using Select Multiplexers" later in this chapter.

Figure 11-3
Typical Multiplexer for Output
Enable Modes



Output Macro Cell Control with ISTYPE

As we have seen, the ISTYPE statement can be used to specify the exact configuration of programmable output enables. The ISTYPE statement is also used to precisely configure other device features, such as output polarity, selectable feedback, register bypass and type, and other macro cell features. Figure 11-4 shows how various ISTYPE statements affect the macro cell configuration.

Controlling Macro Cell Polarity

Output polarity is controlled by entering "pos" or "neg" in the ISTYPE statement. The ISTYPE statement for Q18 (see Figure 11-5) is an example of controlling the output polarity to negative. For a positive output, this statement would be

```
Q18 IsType 'pos,reg,feed_reg';
```

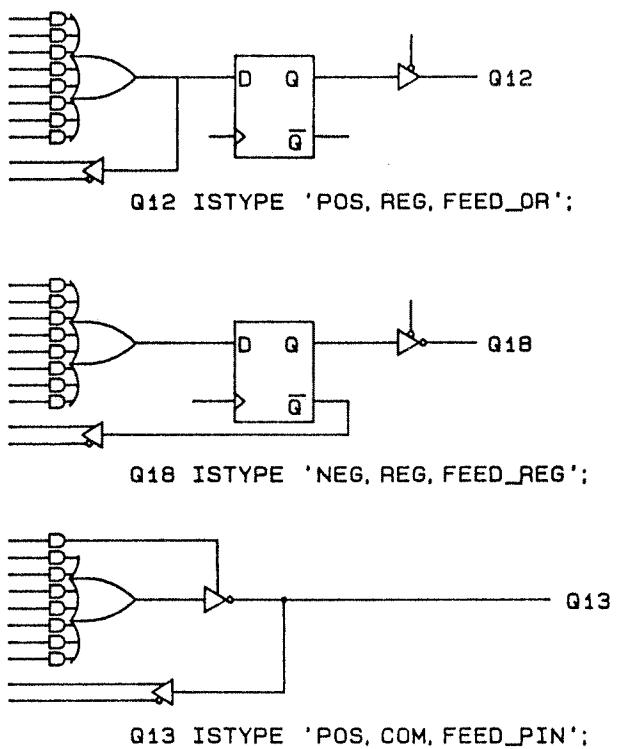
The output polarity (positive or negative) of the macro cell device will affect the DeMorgan's equation inversion, and can result in an excess of product terms. For example, a typical equation, such as

$$Y1 = (A \# B) \& (C \# D) \& (E \# F);$$

will reduce to the following three product terms for a negative output.

$$\begin{aligned} \text{!}Y1 = & (\text{!}A \& \text{!}B \\ & \# \text{!}C \& \text{!}D \\ & \# \text{!}E \& \text{!}F); \end{aligned}$$

Figure 11-4
Controlling Macro Cells with
ISTYPE



However, if a positive output is specified, the same equation requires eight product terms, as follows

```

Y1 = (A & C & E
#A & C & F
#A & D & E
#A & D & F
#B & C & E
#B & C & F
#B & D & E
#B & D & F);

```

When dealing with programmable polarity devices, where the output can be programmed as negative or positive, some consideration should be given to polarity. For most programmable polarity devices, ABEL-PLD defaults to a negative selection when the polarity is not specified. In some devices this can mean a design will not fit into the device (causing a "too many terms for output" error message), although it would if positive polarity was specified for the outputs.

Controlling Macro Cell Feedback Point

Figure 11-5 also shows that the ISTYPE statement can be used to explicitly state the macro cell configuration, as in output Q12. Or, the ISTYPE statement can be used to state the feedback point only, leaving the polarity and register/combinatorial selection to be determined by ABEL-PLD from the equations. In the example listing, the ISTYPE statement defines the Q14 output as a feedback from the OR function only. The equation

$\text{IQ14} := \text{D2 \& D3}$

further defines Q14 as a negative, registered output.

Figure 11-5
Controlling Macro Cells with
ISTYPE — Source File

```

module mc flag '-f'
title 'Controlling output macro cells
Brian Durwood Data I/O'

mc device 'E0310';

Clk,D2,D3,D4,D5,D6,D7    pin 1,2,3,4,5,6,7;
D8,D9,D11,Q12,Q13,Q14    pin 8,9,11,12,13,14;
Q15,Q16,Q17,Q18,Q19      pin 15,16,17,18,19;

"Explicitly state macro cell configuration
Q12           IsType   'pos,reg,feed_or';
Q13           IsType   'pos,com,feed_pin';
Q18           IsType   'neg,reg,feed_reg';

"Let equations determine register/combinatorial and polarity
Q14           IsType   'feed_or';
Q15,Q16,Q17 IsType   'feed_pin';

Ck,X,Z = .C., .X., .Z.;

equations
  Q12.RE = D6;      "Async Reset
  Q12.PR = D7;      "Sync Preset

  test_vectors ([Clk,D6,D7] -> [Q12,Q14,Q18])
  [ 0, 1, 0] -> [ 0 , 1 , 1 ]; "Reset
  [ 0, 0, 1] -> [ 0 , 1 , 1 ];
  [ Ck, 0, 1] -> [ 1 , 0 , 0 ]; "Preset with clock
  [ 0, 1, 0] -> [ 0 , 1 , 1 ]; "Reset

  equations
    Q12 := D2 & D3; "Feedback from the OR
    Q13 = Q12 & D4;

    test_vectors ([Clk,D2,D3,D4] -> [Q12,Q13])
    [ Ck, 1, 1, 1] -> [ 1 , 1 ];
    [ 0, 1, 0, 1] -> [ 1 , 0 ];
    [ Ck, 0, 1, 1] -> [ 0 , 0 ];
    [ 0, 1, 1, 1] -> [ 0 , 1 ];

```

```

@page

equations
  !Q14 := D2 & D3; "Registered Negative
  Q15 = D3 & D4; "Combinatorial Positive

test_vectors ([Clk,D2,D3,D4] -> [|Q14,Q15|])
  [ Ck, 1, 1, 1] -> [ 1 , 1 ];
  [ Ck, 1, 0, 1] -> [ 0 , 0 ];
  [ Ck, 1, 1, 0] -> [ 1 , 0 ];
  [ Ck, 0, 1, 1] -> [ 0 , 1 ];

equations      "bidirectional buffer
  Q16 = Q17;      Q16.OE = D4;
  Q17 = Q16;      Q17.OE = !D4;

test_vectors ([D4,Q16,Q17] -> [Q16,Q17])
  [ 1, X, 0] -> [ 0, X];
  [ 1, X, 1] -> [ 1, X];
  [ 0, 0, X] -> [ X, 0];
  [ 0, 1, X] -> [ X, 1];

equations
  Q18 := D5 & !Q18;
  Q19 = Clk;

test_vectors ([Clk,D5] -> [Q18,Q19])
  [ Ck, 0] -> [ 0, 0 ];
  [ Ck, 1] -> [ 1, 0 ];
  [ Ck, 1] -> [ 0, 0 ];
  [ 0 , 1] -> [ 0, 0 ];
  [ 1 , 1] -> [ 1, 1 ];

end

```

Selecting or Bypassing Device Registers

Figure 11-6 shows an output macro cell that allows the specification of whether the output register is bypassed to provide a combinatorial output, or selected to provide a registered output (other selectable features have been omitted for clarity). ABEL-PLD will normally choose whether to bypass or select the register based on the form of equation written for that output; the use of "=" in the equation indicates that the output is to be combinational (register bypassed,) and the use of ":" indicates that the output is to be registered.

For situations where you need to explicitly declare the configuration of the register (as in the case of devices with multiple feedback or configurable input registers) you can use the statement

`F0 istype 'reg';`

to indicate that the register for pin F0 is to be selected, or

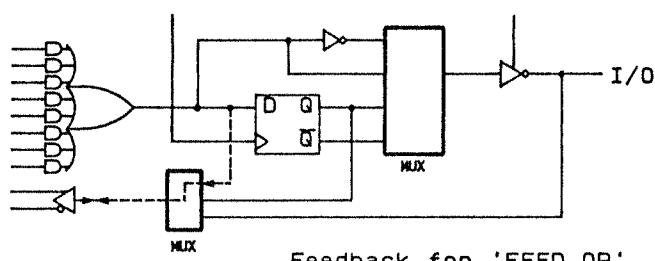
`F0 istype 'com';`

to indicate that the register is to be bypassed.

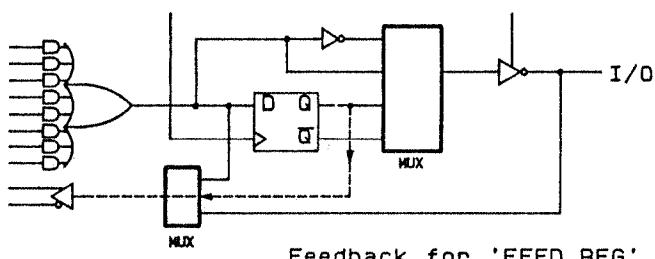
Selectable Feedback Type

Some devices, such as the E0310, allow the selection of which feedback path you wish to use for each device output. Figure 11-8 shows the logic diagram for one output of a E0310 and illustrates the three selectable feedback paths. The section "Controlling Macro Cell Feedback Point" describes how to configure the feedback for devices such as this using the ISTYPE statement. Once the feedback type has been specified in your design, no special equation syntax is required to use the feedback in your equations.

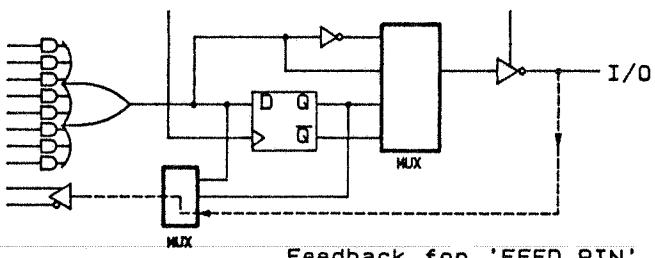
Figure 11-8
Selectable Feedback Paths
(E0310)



Feedback for 'FEED_DR'



Feedback for 'FEED_REG'

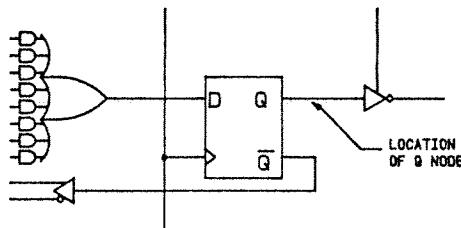


Feedback for 'FEED_PIN'

The .Q Dot Extension

Register feedback is indicated in a design by using the .Q dot extension when referring to a signal as an input in the equations. Figure 11-9 shows a typical registered output, and where you should assume that the .Q signal originates when you use it in equations; although this is not necessarily where the device feedback actually comes from.

Figure 11-9
Location of the Q Signal in
Registered Devices



In most devices, as in Figure 11-9, the register feedback does not actually originate at the Q output of the register, but at the complement output. The PARSE module will invert the feedback if it comes from the complement output. This can be seen in the "original equations" section of the DOCUMENT output (*.DOC) when the -q0 parameter is specified on the ABEL-PLD command line.

For the purpose of writing equations, however, you should assume the polarity of the .Q signal is the same as the polarity of the Q output of the register, regardless of any inversion the output may have after the register.

Using Complement Arrays

The complement array is a unique feature that is found in some logic sequencers. The following example shows a typical use in termination of a counter sequence.

Transition equations may be used to express the design of counters and state machines in some devices that contain J-K and/or R-S flip-flops. A transition equation expresses a state of the circuit as a variation of, or an adjustment to, the previous state. This type of equation eliminates the need to specify each node of the circuit, but only those that require a transition to the opposite state.

An example of transition equations usage is shown in Figure 11-10, a source file for a decade counter having a single (clock) input and a single latched output. The purpose of this counter is to divide the clock input by a factor of ten and generate a 50% duty-cycle squarewave output. The device used for this design is an F105 FPLS. In addition to its registered outputs, this device contains a set of "buried" (or feedback) registers whose outputs are fed back to the product term inputs. COMP, P0, P1, P2, and P3 are default names for the outputs that are defined in the F105 device file for nodes 49 and 37 through 40. If no other name for these nodes is declared (and none other is in Figure 11-10) the default names are used.

Before examining the equations in Figure 11-10, note that the decade counter design is shown in Figure 11-11. Figure 11-11 is an abbreviated logic diagram of the F105 that shows the "buried" (no output pins) registers arranged as a counter having feedback paths back to the inputs by means of product terms. The counter registers are designated P0 through P3 by default in the F105. These node designations can then be used in the equations. The output register is declared as F0.

Node 49, the complement array feedback, is declared (as COMP) so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero in the event an illegal counter state is encountered. Any illegal state (and also state 9) will result the absence of an active product term to hold node 49 at a logic low. When node 49 is low, Figure 11-11 shows that product term 9 resets each of the feedback registers so that the counter is set to state zero. (To simplify the following description of the equations in Figure 11-10, node 49 and the complement array feedback are temporarily ignored.)

The first equation states that the F0 (output) register is set (to provide the counter output) and the P0 register is set when registers P0, P1, P2, and P3 are all reset (counter at state zero) and the clear input is low. Figure 11-11 shows how the fuses are blown to fulfill this equation; the complemented outputs of the registers (with the clear input low) form product term 0. Product term 0 sets register P0 to increment the decade counter to state 1, and sets register F0 to provide an output at pin 18.

Figure 11-10
Transition Equations for a
Decade Counter

```

module decade
  title 'Decade Counter - Uses Complement Array and Default Node Names
  Michael Holley    Data I/O Corp'

  TE10      device 'F105';

  Clk,Clr,F0,PR    pin 1,8,18,19;
  _State          = [P3,P2,P1,P0];  "State Registers
  H,L,Ck,X        = 1, 0, .C., .X.;

  equations
    [P3.AP,P2.AP,P1.AP,P0.AP,F0.AP] = PR;      "Async Preset Option

    "Output           Next State     Present State     Input
    [F0 , COMP,       P0 ] := !P3 & !P2 & !P1 & !P0 & !Clr; "0 to 1
    [   COMP,         P1 ,P0.R] := !P3 & !P2 & !P1 &  P0 & !Clr; "1 to 2
    [   COMP,         P0 ] := !P3 & !P2 &  P1 & !P0 & !Clr; "2 to 3
    [   COMP,         P2 ,P1.R,P0.R] := !P3 & !P2 &  P1 &  P0 & !Clr; "3 to 4
    [   COMP,         P0 ] := !P3 &  P2 & !P1 & !P0 & !Clr; "4 to 5
    [F0.R, COMP,      P1 ,P0.R] := !P3 &  P2 & !P1 &  P0 & !Clr; "5 to 6
    [   COMP,         P0 ] := !P3 &  P2 &  P1 & !P0 & !Clr; "6 to 7
    [   COMP,P3,P2.R,P1.R,P0.R] := !P3 &  P2 &  P1 &  P0 & !Clr; "7 to 8
    [   COMP,         P0 ] :=  P3 & !P2 & !P1 & !P0 & !Clr; "8 to 9
    [   P3.R,P2.R,P1.R,P0.R] := !COMP; "Clear

  "After Preset, inhibit clock until a High-to-Low clock transition.

  test_vectors ([Clk,PR,Clr] -> [_State,F0])
    [ 1 , 0, 0 ] -> [^b1111, H]; " Power-on Preset
    [ Ck, 0, 1 ] -> [  0 , H]; " Clear to 0
    [ Ck, 0, 0 ] -> [  1 , H];
    [ Ck, 0, 0 ] -> [  2 , H];
    [ 1 , 1, 0 ] -> [^b1111, H]; " Preset high
    [ 1 , 0, 0 ] -> [^b1111, H]; " Preset low
    [ Ck, 0, 0 ] -> [  0 , H]; " COMP forces to State 0
    [ Ck, 0, 0 ] -> [  1 , H];
    [ Ck, 0, 0 ] -> [  2 , H];
    [ Ck, 0, 0 ] -> [  3 , H];
    [ Ck, 0, 0 ] -> [  4 , H];
    [ Ck, 0, 0 ] -> [  5 , H];
    [ Ck, 0, 0 ] -> [  6 , L];
    [ Ck, 0, 0 ] -> [  7 , L];
    [ Ck, 0, 0 ] -> [  8 , L];
    [ Ck, 0, 0 ] -> [  9 , L];
    [ Ck, 0, 0 ] -> [  0 , L];
    [ Ck, 0, 0 ] -> [  1 , H];
    [ Ck, 0, 0 ] -> [  2 , H];
    [ Ck, 0, 1 ] -> [  0 , H]; " Clear
end

```

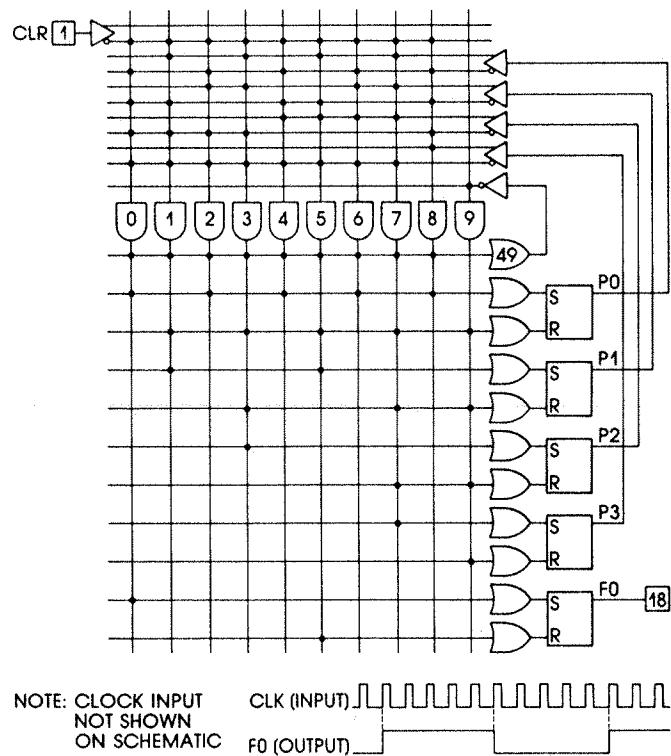
The second equation performs a transition from state 1 to state 2 by setting the P1 register and resetting the P0 register. (The .R dot extension is used to define the reset input of the registers.) In state 2, the F0 register remains set, maintaining the high output. The third equation again sets the P0 register to achieve state 3 (P0 and P1 both set), while the fourth resets P0 and P1, and sets P2 for state 4, and so on.

Wraparound of the counter from state 9 to state 0 is achieved by means of the complement array node (node 49). The last equation defines state 0 (P3, P2, P1, and P0 all reset) as equal to !COMP, that is, node 49 at a logic low. When this equation is processed by ABEL-PLD, the fuses are blown as indicated in Figure 11-11. Figure 11-11 shows that state 9 (P0 and P3 set) provides no product term to pull node 49 high. As a result, the !COMP signal is true to generate product term 9 and reset all the "buried" registers to zero.

Equations for XOR PALs

When writing equations for XOR PALs, you should use parentheses to group those parts of the equation that go on either side of the XOR. This is because the XOR operator (\$) and the OR operator (#) have the same priority in ABEL-PLD.

Figure 11-11
Abbreviated F105 Schematic



Chapter 12

Downloading to Programmers

Downloading to a Model 29 with LogicPak

The following steps transfer a programmer load file from an IBM PC/XT/AT (or compatible) to a Data I/O Model 29 programmer with a LogicPak installed:

1. Connect the Model 29 cable to the COM1: port on the IBM PC and to the serial interface of the Model 29 (see Figures 12-20 and 12-21).
2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual). Parity should be set with the power off.
3. Configure the IBM PC for 4800 baud (9600 baud could also be used) with no parity by issuing the following command at the DOS prompt on the PC:
`MODE COM1:4800,n,8`
4. Enter the family code and pinout code of the device to be programmed into the Model 29 as follows:
`VERIFY RAM DEVICE START
XXXX`

where XXXX is the 4-digit family and pinout code. A list of these codes may be found in the LogicPak manual or on the Data I/O Wallchart of Programmable Devices.

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

`SELECT E B START`

The Model 29 will display a stationary action symbol.

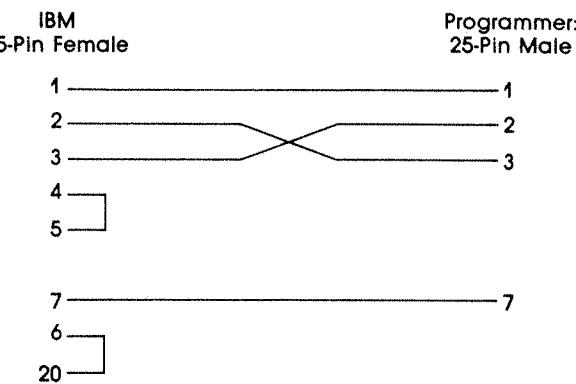
6. At the PC, execute the command:

```
COPY d:filename.ext COM1:
```

where *d* is the drive specification, and *filename* and *ext* are the filename and extension of the programmer load file. The load file extension is .jed. The Model 29 action symbol should rotate when the file is being transferred.

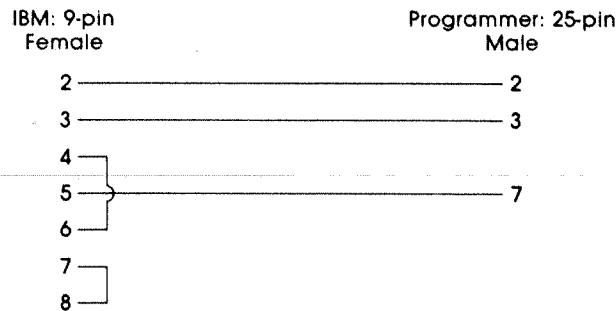
7. When the transfer is complete, the Model 29 will display the fuse checksum signifying that the programmer load file is now resident in programmer memory.

Figure 12-1
Cable Configuration for Transfer
Between an IBM-XT and a Data
I/O Programmer



NOTE: The 25-pin male connector is also known as DB25P, the female is a DB25S.

Figure 12-2
Cable Configuration for Transfer
Between an IBM-AT and a Data
I/O Programmer



Downloading to a UniSite™ Universal Programmer

Refer to Chapter 2 of your UniSite Operator's Manual for a complete sample procedure.

PROM Download (Model 29/UniPak2)

Unless you are using PROMlink, the following steps are necessary to transfer a programmer load file from the IBM PC to a DATA I/O Model 29 programmer with a UniPak2:

1. Connect the Model 29 cable to the COM1: port on the IBM PC and to the serial interface of the Model 29 (see Figures 12-20 or 12-21).
2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual). Parity should be set with the power off.
3. Configure the IBM PC for 4800 baud with no parity by issuing the following command at the DOS prompt on the PC:

MODE COM1:4800,n,8

4. Select the microprocessor transfer format of the load file. A list of supported microprocessor formats may be found in table 4-1. For example, the code for the Motorola Exorciser format is 82. To select this format enter the following at the 29:

SELECT 8 2 START START

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

COPY PORT RAM START

The Model 29 will display a stationary action symbol.

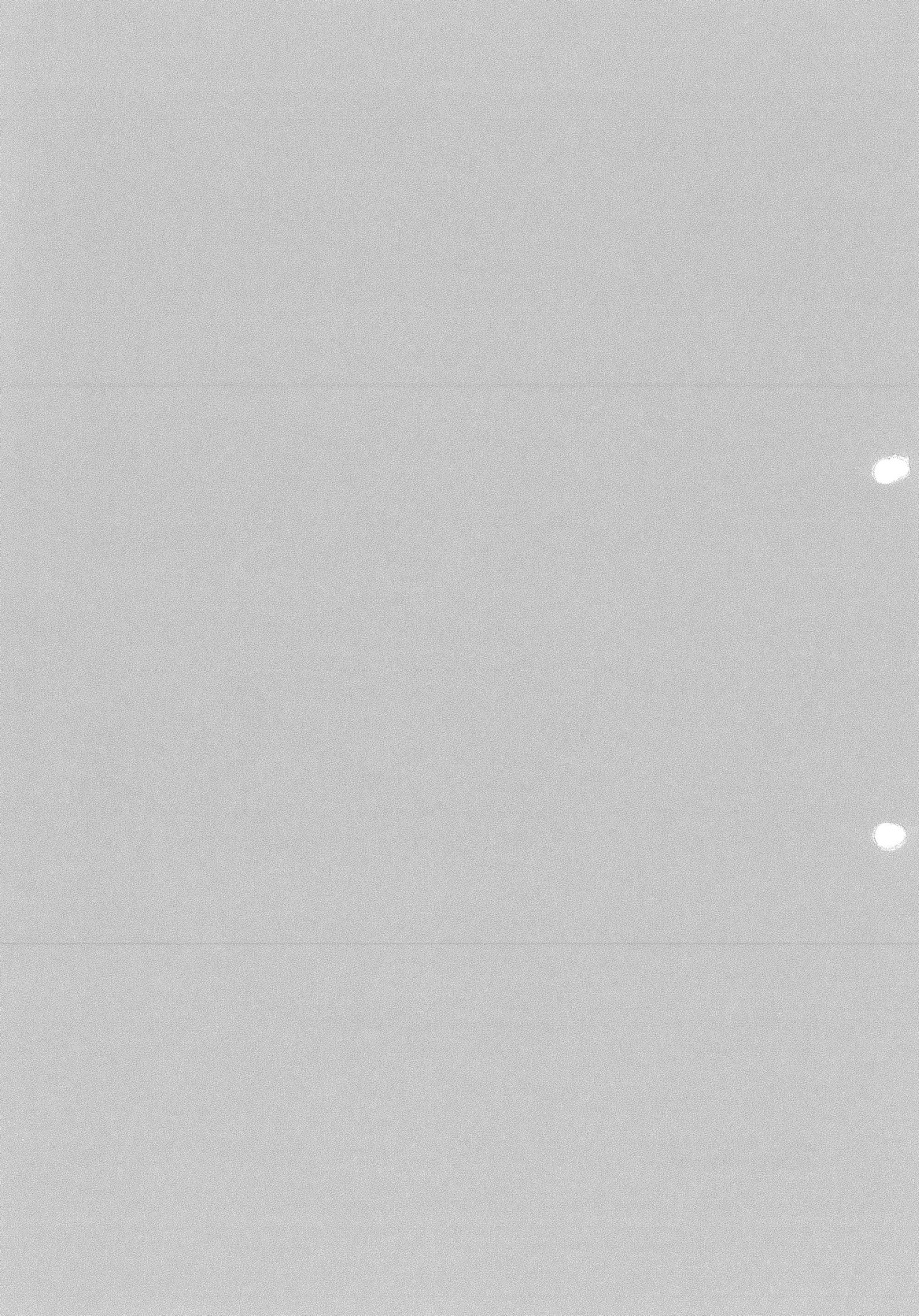
6. At the PC, execute the command:

COPY d:filename.ext COM1:

where d is the drive specification, and filename and ext are the filename and extension of the programmer load file. The load file extension for PROMs is .Pxx where xx is the format code (see Table 12-7). The Model 29 action symbol should rotate when the file is being transferred.

7. When the transfer is complete, the Model 29 will display the fuse RAM sumcheck.

Appendices



Appendix A

Error Messages

This appendix contains listings of the error messages you may encounter while using the ABEL-PLD language processor or any of the utilities provided with the ABEL-PLD software package. The error messages are divided into four subsections:

1. General Error Messages
2. Non-fatal Simulation Error Messages
3. TOABEL Error Messages
4. ABELLIB Error Messages

Within the subsections, the actual error messages appear in smaller typeface. Most of the error messages are listed with a brief description; however those that are self-explanatory are just listed. The symbol 'xxx' in an error message indicates that some actual text or numeric value appears in that location in the message. That text or information may vary depending on the condition causing the error.

Additional information regarding a particular error message can usually be obtained by using the index located at the back of the manual.

General Error Messages

The following error messages may be displayed on the screen during execution of ABEL-PLD programs.

Command Line Errors

Command line errors indicate that an error was made on the command line when one of the language processor programs was invoked. These errors can occur when a program is invoked directly, or from a batch file.

Global Command Line errors

illegal argument 'xxx'

An invalid argument (flag) has been specified on the command line.

unable to open input file 'xxx'

unable to open output file 'xxx'

The output file specified can not be created, due to an invalid file name, path or drive specification, or lack of disk space.

input and output file names may not be the same

All file names specified to ABEL-PLD programs must have unique names to avoid overwriting input files.

illegal value for flag 'xxx'

A command line argument (flag) that requires a numeric value has been specified with an invalid value.

list and output file names may not be the same

All file names specified to ABEL-PLD programs must have unique names to avoid overwriting input files.

only one input file allowed, 'xxx'

You have specified more than one input file, either by using the '-I' command line flag, or by omitting the dash ('-') character from another command line argument. Any command line argument not prefixed with a dash is assumed to be an input file name.

only one output file allowed, 'xxx'

More than one output file name was specified using the '-O' flag.

only one listing file allowed, 'xxx'

More than one list file was specified using the '-L' flag.

file name too long

File names specified to ABEL-PLD programs must not exceed the file name length limits of the operating system, and the full file name (including path and drive specification) must not exceed 64 characters.

Fusemap Command Line Errors

illegal value for unused terms 'xxx', valid = y,n

The value specified using the '-k' flag is invalid; valid values are 'y' or 'n'.

illegal value for checksum 'xxx', valid = 0,1 or 2

The value specified using the '-c' flag is invalid; value may be '0', '1', or '2'.

illegal type of output file 'xxx', valid = 0,82,83,87 or 88

The value specified using the '-d' flag is invalid.

illegal type of output file for wide PROM's with wordsize8

The value specified using the '-d' flag is not appropriate for the device specified in the design.

Simulate Command Line Errors

no device specified

The device type must be specified to SIMULATE by using the '-n' command line flag.

illegal trace level 'xxx', values = 0, 1, 2, 3, 4 or 5

The value specified using the '-t' flag is invalid.

illegal default X value 'xxx', values = 0 or 1

The value specified using the '-x' flag is invalid.

illegal default Z value 'xxx', values = 0 or 1

The value specified using the '-z' flag is invalid.

illegal breakpoint

The breakpoint information specified using the '-b' flag is incorrect. On MS-DOS systems, you must use a period ('.') to separate breakpoint fields, instead of a comma.

Fatal Errors

Can't find device file 'xxx'

The specified device file cannot be found. ABEL-PLD programs attempt to find device files by first searching for the device file itself, and if no file is found, attempt to find the device file in the device file library 'abel3lib.dev'.

The search is performed first in the current (default) directory, then in the directory indicated in the 'abel3dev' variable and, if the device file is still not found, all directories in your 'path' variable.

Equation for 'xxx' needs reduction

Use the next highest level of reduction to eliminate this error.

Equation for 'xxx' is not sums-of-products

There are two reasons for these errors:

1. The input file to FUSEMAP is not the output file from REDUCE.
2. You used operators other than &, # and ! (and \$ for PROMs) and did only a "-R0" reduction in REDUCE. Run REDUCE with a stronger reduction level.

Error writing 'xxxx' file

The output file can no longer be written to. This can arise when a disk is too full, or if the disk data has been damaged.

Illegal operator in equation for 'xxx'

Use the next highest level of reduction to eliminate this error.

Illegal term element at 'xxx'

Use the next highest level of reduction.

Memory overflow

Memory overflow can occur for any of the following reasons:

1. Too many equations or test vectors.
2. Equations are too complex.
3. Equations have too many XOR (\$) operators.
4. Insufficient free memory available.
5. Too many symbols.

(Refer to the section "Number Adjacent States for One-bit Changes" in Chapter 9 for information on reducing the number of product terms.)

PALs and FPLAs can only be output to a JEDEC file

The output file type must be specified with the "-d0" option (JEDEC format) or left unspecified for PALs and FPLAs.

Premature end of source file

The input file specified is incomplete, possibly due to errors in a previous ABEL-PLD program.

PROMs can not be output to a JEDEC file

For PROMs, the output file type must be specified with the "-dxx" option where xx is not 0.

Too many errors

PARSE terminates execution when many errors are detected in a source file.

Wrong version of intermediate file

The intermediate file specified as input to an ABEL-PLD program was generated by an incompatible (older) version of an ABEL-PLD program.

Intermediate File Errors

Intermediate files are created by one program in the ABEL-PLD language processor and used by one or more of the other programs. If there are errors in an intermediate file, it is likely that subsequent programs needing that file will not operate correctly.

File not produced by PARSE

The intermediate file specified as input to TRANSFOR was not produced by PARSE.

File not produced by TRANSFOR

The intermediate file specified as input to REDUCE was not produced by TRANSFOR.

File not produced by FUSEMAP

The intermediate file specified as input to SIMULATE was not produced by FUSEMAP.

File not produced by REDUCE

The intermediate file specified as input to FUSEMAP was not produced by REDUCE.

File not produced by ABEL-PLD 3.x

The file was not produced by ABEL-PLD 3.x versions of the ABEL-PLD programs.

Intermediate file error: TGxxx

An error was detected in the input to TRANSFOR. This may be caused by:

1. An input file generated by PARSE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: RGxxx

An error was detected in the input to REDUCE. This may be caused by:

1. An input file generated by PARSE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: FGxxx

An error was detected in the input to FUSEMAP. This may be caused by:

1. Using an input file generated by REDUCE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: SGxxx

These errors occur in SIMULATE for the following reasons:

1. Error in a previous pass.
2. Hardware failure.
3. The file has been modified.

Intermediate file error: DGxxx

These errors in DOCUMENT mean one of the following:

1. Error in an earlier pass.
2. Hardware failure.
3. The file has been modified.

Logical Errors

Logical errors are device-related errors and point to elements of your design that cannot be implemented on the chosen device, or which are implemented incorrectly.

Logical errors may be created by mistakes in the source file or by errors in your design.

Attribute for 'xxx' is 'latch' but pin is register

This input has the wrong attribute assigned to it. Rewrite your attribute definitions or use a different part.

Attribute for 'xxx' is 'register' but pin is latch

This input has the wrong attribute assigned to it. Rewrite your attribute definitions or use a different part.

can't map equation for 'xxx' with ISTYPE 'xxx'

The device has a select multiplexer type internal node that can not be programmed using the combination of attribute and equation specified in the design.

can't map equation for select mux 'xxx' - use ISTYPE

The device has a select multiplexer type internal node that can not be programmed with the equation specified in the design. Use an ISTYPE statement to configure the multiplexer properly, or rewrite the equation to match the type of node available in the device.

Can't use an XOR for 'xxx'

The XOR operator is not supported for pin 'xxx'. Use a stronger reduction to eliminate the XORs.

Enable for 'xxx' is a pin, not equation

Enable for this pin is not term controllable.

Enable of 'xxx' can only be a constant or a pin

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Enable of 'xxx' has already been programmed

The enable for this pin is also the enable for other pins and it has already been programmed. Rewrite your equations to eliminate repeat programming of the enable.

Enable of 'xxx' has already been set for this group

The enable for this pin can only be done in groups and the enable for the group that includes this output has already been set. Rewrite your equations so that the enable of this pin is not reprogrammed.

Illegal constant for output 'xxx'

The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Illegal pin for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Inverted input not allowed from 'xxx'

The pin (or node) 'xxx' does not allow the given type of input. Rewrite your equations to use the other type.

Node number 'xxx' is not defined for this device

You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL-PLD package to identify device pins and nodes.

Number must be 1 or !0 for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Output 'xxx' used more than once

Some pins can be accessed in more than one way. You have equations referencing this pin in more than one way. Rewrite your equations to use only one method.

Pin can only be negative logic for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Pin number 'xxx' is not defined for this device

You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL-PLD package to identify device pins and nodes.

Pin or node 'xxx' is already defined as 'xxx'

You have defined the pin or node with multiple names.

Positive input not allowed from 'xxx'

The input from this pin/node can be only of one type. Rewrite your equations to use the other type.

Select mux 'xxx' respecified

The indicated select multiplexer type internal node has been specified with a type that conflicts with a previous usage. Examine the logic diagram for the device and use ISTYPE statement or rewrite equations to properly configure the select multiplexer node.

Signal 'xxx' is not in device 'xxx'

You have specified a pin or node that is not declared in the device.

Signal 'xxx' used or defined as combinational

The indicated signal has been declared as a combinatorial signal, or is a programmable type and has been used previously as a combinatorial signal.

Signal 'xxx' used or defined as registered

The indicated signal has been declared as a registered signal, or is a programmable type and has been used previously as a registered signal.

Single fuse equation for 'xxx' must be a constant

The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Status of 'xxx' register has already been set for this group.

The selection of registered/non- registered pins in this part can only be done in groups, and it has already been done for the group that includes this pin. Rewrite your equations so that all the outputs in this group are of the same type ('=' or ':=').

Too many terms for FPLA array at output 'xxx'

The equation for 'xxx' required too many terms programmed into the FPLA. Use a stronger reduction.

Too many terms for output 'xxx'

There are not enough logic terms in the part to program your equation. Either use a stronger reduction (-r2 for example) or rewrite your equations to use fewer terms.

'xxx' attribute not allowed on this device

You have specified a programmable attribute that is not supported on the device.

'xxx' cannot be enabled

You have written an enable equation for a pin that cannot have an enable equation.

'xxx' cannot be programmed inverted

The equation for 'xxx' has the wrong polarity for this pin and the pin can not be programmed to change polarities. Either rewrite your equations or use a stronger reduction which will automatically convert your equations to the correct polarity.

'xxx' has no enable

'xxx' is pin or node identifier.

'xxx' has no registered feedback

You have used pin 'xxx' as an input or feedback pin but it does not have those features. Rewrite your equations so they do not use this pin as an input.

'xxx' is a registered pin

You have used the wrong type of assignment for this pin. Use ":=" for registered pins and "=" for non- registered pins.

'xxx' is an invalid output

'xxx' is pin or node identifier

'xxx' is not an input pin

Your equation attempts to make use of input or feedback from a pin in the part that is not an input pin and/or has no feedback.

'xxx' is not an input/programmable pin

You have used a pin, that is not an input, as an input in an equation.

'xxx' is not an output pin

You have specified a pin that is not an output pin as an output pin.

'xxx' is not a registered pin

You have used the wrong type of assignment for this pin. Use ":=" for registered pins and "=" for non- registered pins.

'xxx' is not in sums-of-products form

Specify a stronger reduction level and process again.

Preprocessor Errors

Preprocessor errors are syntax errors detected by the preprocessor step of PARSE. The ABEL-PLD source is preprocessed before it is actually parsed; the preprocessor expands macros and acts on directives. To correct a preprocessor error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below. Refer to the ABEL-PLD Language Reference Manual for the correct syntax for the ABEL-PLD source files.

```
'(' expected
',' expected
'=' expected
'xxx' actual arguments expected
Block expected
Constant label expected
Dummy argument expected
Identifier expected
Label expected
Number expected
Number is too large
Radix 'xxx' is not one of 2, 8, 10 or 16
String expected
```

Syntax Errors

Syntax errors are detected by PARSE and indicate missing, incorrect, or incomplete elements and structures in the ABEL-PLD source file. Correct syntax errors by eliminating the error in the source file. The PARSE listing file may aid you in this. The listing file contains pointers that indicate approximately where in a line the error occurs, followed by the error message describing the error. To correct a syntax error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below, with a brief explanation where the message may not be self-explanatory. Refer to the Language Reference for the correct syntax for the ABEL-PLD source files.

'**)**' expected

'**-**' expected

'**:**' expected

'**:+:**' or '**:-:**' expected

'**:**' expected

'**:**' or '**-**' expected

'**;**' expected

'**=**' or '**:=**' expected

'**]**' expected

'THEN' expected

'xxx' actual arguments specified on command line

Number of arguments found after the MODULE keyword is less than that on the command line.

Actual argument length exceeds 'xxx' chars

Element 'xxx' in range

An intermediate identifier name produced as a result of a range is undefined.

Block expected

Can't have letters imbedded in a number

Can't map set onto a different sized set

Can't map set onto a non-set element

Sets cannot be assigned to numbers, signals or special constants; they can only be assigned to other sets.

Cannot operate on signal 'xxx'

Can only operate on constants in this context.

Closing " of string not found

sing '}' of block not found

Declaration keyword expected

The valid declaration keywords are PIN, NODE, MACRO,
DEVICE, ISTYPE, and LIBRARY

Device_label expected

Digit not in radix 'xxx'

Don't know what device to use

Either an "IN device_id" is necessary or no device was ever
specified.

Dummy argument 'xxx' not recognized

The dummy argument found after the "?" was not specified
previously.

Element expected

A set, identifier, number, special constant, or parenthetical
expression was expected here.

Enables are not registered

'GOTO', 'IF' or 'CASE' expected

Fuse number expected

Identifier expected

Identifier length exceeds 'xxx' chars

Illegal character 'xxx' in source file

'xxx' is the decimal value of the bad character.

Illegal operation on special constant **Illegal set in state diagram header**

Check for illegal nested sets in header

Inconsistency in number of parameters

The number of labels on the left side of a declaration keyword did
not match the number of declarations on the right side of the
declaration keyword.

Industry part number string expected

Valid equation for 'fuse' type output, must specify value

Invalid equation for 'pin' type output, must specify signal name

You have written an equation for a select multiplexer type node
that does not match the type specified in an ISTYPE statement.

Invalid flag string 'xxx'

Invalid set range

Keyword 'xxx' is out of context

Label 'xxx' is already defined

Label expected

Max of 'xxx' elements in set was exceeded

Mismatch in number of set elements

Sets with different numbers of elements in them cannot be operated on with a
binary operator.

'MODULE' expected

Module label doesn't match 'xxx'

If a label is used after the END statement at the end of a module,
that label must match the label used with the MODULE statement
that began the module.

Multiple mapping to signal 'xxx'

An attempt was made to assign more than one test condition to a
single signal.

Negative declaration not allowed

Negative declarations are only allowed for pins and nodes.

No more than 'xxx' args are declared

More actual arguments are used than there were dummy
arguments specified.

Number expected

Numeric overflow

A number which requires more than 32 bits to represent it was specified.

Obsolete dot extension - use pin attribute instead

You have used the '.M' dot extension, which has been obsoleted by the ISTYPE 'reg_D' or 'reg_JK' pin attribute. Use an ISTYPE statement.

Only ^B, ^D, ^H, or ^O radix allowed

Only one label allowed

Operator not allowed

Section keyword or 'END' expected

A section begins with one of the following keywords:
EQUATIONS, TRUTH_TABLE, STATE_DIAGRAM,
TEST_VECTORS, FUSES

Signal not allowed

Signal number for 'xxx' is way too large

Signal or .X. expected

Source line length exceeds 'xxx' chars

Special constant must end with a ''

Special constant not allowed

String expected

String length exceeds 'xxx' chars

Suffix xxx not legal in this context

A shorthand node suffix has been specified that is not supported for the signal specified.

Undefined compiler directive '@'xxx"

Undefined label 'xxx'

Undefined label 'xxx', maybe 'xxx' was meant

Undefined operation on sets

Undefined operation on signal

Undefined special constant 'xxx'

Undefined token

A character was found outside of a valid context in the source file.

Unrecognized attribute 'xxx'

Unrecognized industry part number 'xxx'

The indicated device is not supported by ABEL-PLD.

'#' for OR instead of '|'

Device File/Internal Errors

Device file and internal errors appear as follows:

Device file error: 'xxx'

"xxx" is a number indicating the type of device file error. Any of these errors point to an error in a device specification file, to a hardware error, or possibly to a bug in the program. If a device file error occurs, note the number of the device file error and contact your Data I/O representative.

Internal error: YYxxx

YY is a two character code indicating the program that failed and xxx is the number of the internal error. If a device file error occurs, and all hardware is operating normally, note the number of the device file error and contact your Data I/O representative.

Internal error: 'xxx', line 'xxx'

If this error occurs, note the file and line numbers indicated and contact your Data I/O representative.

Non-Fatal Simulation Errors

Non-fatal simulation errors indicate that something is not quite right in the design, but the condition is not severe enough to terminate the simulation process. Pay close attention to these errors, they may point to errors in the design that could cause problems in a programmed device.

Device unstable

The device did not stabilize within 20 iterations.

Both inputs high on RS flip-flop : 'xxx'

Both 'R' and 'S' inputs to the flip-flop were true, resulting in unknown output value.

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

Flip-flop inputs during load : 'xxx'

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

K input high while flip-flop in D mode : 'xxx'

The 'K' input to a JK-D type flip-flop was found to be true when the flip-flop was operating in D mode.

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

Output not High-Z during load : 'xxx'

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

TOABEL Error Messages

Fatal error 701 : Invalid command line argument
Fatal error 702 : Duplicate input and output file name
Fatal error 703 : Error opening input file
Fatal error 704 : Error opening output file
Fatal error 705 : Device not supported
Fatal error 706 : Unexpected end of file encountered
Fatal error 707 : Error writing output file
Syntax error 710 : Invalid symbol ':'
Syntax error 711 : Not a valid pin identifier
Syntax error 712 : Duplicate pin identifier
Syntax error 713 : No device found in input file
Syntax error 714 : Missing '('
Syntax error 715 : Too many elements in enable expression
Syntax error 716 : Expected pin identifier
Syntax error 717 : ';' expected
Syntax error 718 : Undefined pin identifier
Syntax error 719 : Invalid symbol in test vector

ABELLIB Error Messages

bad library -- no directory count

bad library -- no directory offset

bad library -- directory entry

The library file specified contains invalid data. Possible causes include: using a library file that was generated on a different operating system; using a library that has been modified; hardware (disk) problems.

can't create file 'xxxxxxx'

The indicated file could not be created when attempting to extract it from the library, due to a lack of disk space, or hardware problems.

can't read file 'xxxxxxx'

The indicated file could not be found or read when attempting to add it to the library.

can't create library file

can't write library file

ABELLIB was unable to create or write the file due to a lack of disk space, hardware problems, or incorrectly specified library filename or path.

file 'xxxxx' not in library

The indicated file was not found in the library file.

Appendix B

JEDEC Standard Number 3A

Introduction

This appendix defines a format for the transfer of information between a data preparation system and a logic device programmer. Jedec format provides for, but is not limited to, the transfer of fuse, test, identification, and comment information in an ASCII representation, and defines the "intermediate code" between device programmers and data preparation systems. It does not define device architecture nor does it define programming algorithms or the device specific information for accessing the fuses or cells.

Note: This appendix represents a Data I/O-specific implementation of JEDEC Standard No. 3-A.

The standard includes a transmission protocol based on traditional PROM formats that allow a device programmer to share a computer serial port with a terminal. The protocol is not a complete communications protocol and does not do retries or error correction. This protocol is not required if the device programmer has local storage, such as a floppy disk.

Field programmable logic devices may require more testing than programmable memories, so the standard defines a functional testing format. This test vector format is not a general purpose parametric test language. Figure B-1 provides an example of a PLD Data File.

Figure B-1
Example of a PLD Data File

Summary of Programming and Testing Fields

The programming and testing information is contained in various fields. To comply with the standard, the device programmer, tester, and development system must provide and recognize certain fields. Table B-1 lists the field identifiers and descriptions.

Table B-1
Field Identifiers and Descriptions

Identifier	Description
(n/a)	Design specification
N	Note
QF	Number of fuses in device
QP	Number of pins in test vectors ***
QV	Maximum number of test vectors ***
F	Default fuse state *
L	Fuse list *
C	Fuse checksum
X	Default test condition **
V	Test vectors **
P	Pin sequence **
D	Device (obsolete)
G	Security fuse
R,S,T	Signature analysis
A	Access time
* Programmer must recognize	
** Tester must recognize	
*** Development system must provide	

Special Notations and Definitions

Notational Conventions

In addition to the descriptions and examples, this appendix uses the Backus-Naur Form (BNF) to define the syntax of the data transfer format. BNF is a shorthand notation that follows these rules:

- "`::=`" means "is defined as".
- Characters enclosed by single quotes are literals (required).
- Angle brackets enclose identifiers.
- Square brackets enclose optional items.
- Braces (curly brackets) enclose a repeated item. The item may appear zero or more times.
- Vertical bars indicate a choice between items.
- Repeat counts are given by a `:n` suffix. For example, a six-digit number would be defined as "`<number>` ::= `<digit>:6`".

For example, in words, the definition of a person's name reads:

The full name consists of an optional title followed by a first name, a middle name, and a last name. The person may not have a middle name or may have several middle names. The titles consist of: Mr., Mrs., Ms., Miss, and Dr.

BNF syntax:

```
full name> ::= [<title>] <fn.name> {<mn.name>} <ln.name>  
<title> ::= 'Mr.' | 'Mrs.' | 'Ms.' | 'Miss' | 'Dr.'
```

Examples:

Miss Mary Ann Smith

Mr. John Jacob Joseph Jones

Tom Anderson

BNF Rules and Definitions

The following standard definitions are used throughout the rest of this appendix:

```
<digit> ::= '0' | '1' | '2' | '3' | '4'  
          | '5' | '6' | '7' | '8' | '9'
```

```
<hex-digit> ::= <digit> | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
```

```

<binary-digit> ::= '0' | '1'
<number> ::= <digit> (<digit>)
<del> ::= <space> | <carriage return>
<delimiter> ::= <del> (<del>)
<printable character> ::= <ASCII 20 hex ... 7E hex>
<control character> ::= <ASCII 00 hex ... 1F hex>
| <ASCII 7F hex>
<STX> ::= <ASCII 02 hex>
<ETX> ::= <ASCII 03 hex>
<carriage return> ::= <ASCII 0D hex>
<line feed> ::= <ASCII 0A hex>
<space> ::= <ASCII 20 hex> | ''
<valid character> ::= <printable character>
| <carriage return> | <line feed>
<field character> ::= <ASCII 20 hex ... 29 hex>
| <ASCII 2B hex ... 7E hex>
| <carriage return> | <line feed>

```

Transmission Protocol

Protocol Syntax

This STX-ETX protocol is based on traditional PROM formats that allow a device programmer to share a serial computer port with a terminal. The transmission consists of a start-of-text (STX) character, various fields, and end-of-text (ETX) character, and a transmission checksum. The character set consists of the printable ASCII characters and four control characters (STX, ETX, CR, LF). Other control characters should not be used because they can produce undesirable side-effects in the receiving equipment.

Syntax of the transmission protocol:

<format> ::= <STX> {<field>} <ETX> <xmit checksum>

Computing the Transmission Checksum

The transmission checksum is the 16 bit sum (that is, modulo 65,535) of all ASCII characters transmitted between and including the STX and ETX. (see Figure B-2.) The parity bit is excluded in the calculation.

Syntax of the transmission checksum:

`<xmit checksum> ::= <hex-digit>:4`

Figure B-2
Computing the Transmission Checksum

random text <return><line feed>	= 0000
<STX>TEST*<return><line feed>	02+54+45+53+54+2A+0D+0A = 0183
QF0384*<return><line feed>	51+46+30+33+38+34+2A+0D+0A = 01A7
F0*<return><line feed>	46+30+2A+20+20+0D+0A = 00F7
L10 101*<return><line feed>	4C+31+30+20+31+30+31+2A+0D+0A = 01A0
<ETX>05C4 <return> random text	03 = 0003

	05C4

Disabling the Transmission Checksum

Some computer operating systems do not allow the user to control what characters are sent, especially at the end of a line. The receiving equipment should always accept a dummy value of "0000" as a valid checksum. This dummy checksum is a method of disabling the transmission checksum.

Data Fields**General Field Syntax**

In general, each field in the format starts with an identifier, followed by the information, and terminated with an asterisk. For example, "C1234**" specifies that the checksum of the fuse data is 1234. The design specification header does not have an identifier and must be the first field in the transmission, immediately followed by the STX character.

Syntax of fields:

```

<field>    ::=  [<delimiter>] <field identifier>
                  {<field character>} ""
<field identifier> ::=  | 'A' | 'C' | 'D' | 'F' | 'G'
                           | 'L' | 'N' | 'P' | 'Q' | 'R'
                           | 'S' | 'T' | 'V' | 'X'
<reserved identifier> ::=  | 'B' | 'E' | 'H' | 'I' | 'J'
                           | 'K' | 'M' | 'O' | 'U' | 'W'
                           | 'Y' | 'Z'

```

Field Identifiers

Each field begins with a single character identifier that identifies the field type. Multiple character identifiers can be used to create subfields (that is, "A1", "A\$", or "AB3"). The field is terminated with an asterisk. Therefore, asterisks cannot be embedded within the field. While not required, carriage returns and line feeds should be used to improve the readability of the format. Reserved identifiers currently have no function and are reserved for future use. Receiving equipment should ignore fields starting with reserved identifiers. The meanings of the field identifiers are given in Table B-2.

Table B-2
Field Identifiers

A-	Access Time	N	Note
B-	*	O	*
C-	Checksum	P	Pin sequence
D-	Device type	Q	Value
E-	*	R	Resulting vector
F-	Default fuse state	S	Starting vector
G-	Security fuse	T	Test cycles
H-	*	U	*
I-	*	V	Test vector
J-	*	W	*
K-	*	X	Default test condition
L-	Fuse list	Y	*
M-	*	Z	*

* reserved for future use

Comment and Definition Fields

Design Specification Field

The design specification is the first field in the format. It must be included and it does not have an identifier to signal its start. An asterisk terminates the field. The contents of the design specification are not defined but should consist of:

- User's name and company
- Date, part number, and revision
- Manufacturer's device number
- Other information

Syntax of the Design Specification:

```
<design specification> ::= {<field character>} **
```

Example:

```
File for PLD 12S8  
Created on 8-Feb-85 3:05PM  
6809 memory decode 123-0017-001  
Joe Engineer Advance Logic Corp *
```

If none of the above information is required, a blank field consisting of the terminating asterisk is a valid design specification.

Note Field (N)

The note field is used to place notes and comments in the data file. The note field(s) may appear anywhere in the file and the receiving equipment may ignore this field.

Syntax of the Note Field:

```
<note> ::= 'N' <field characters> **
```

Example:

```
N Following vectors were modified for ACME 123 tester*
```

Device Definition Field (D) (Obsolete)

This field is now obsolete. It has been eliminated to ensure that the format is device and technology independent.

**Value Field
(QF, QP, QV)**

The Q field expresses values or limits which must be provided to the receiving equipment. The following three subfields are defined:

- The F subfield for the number of fuses
- The P subfield for the number of pins or test conditions in the test vector
- The V subfield for the maximum number of test vectors

These values enable the receiving device to efficiently allocate memory and perform certain calculations. The QF field tells the receiving equipment how much memory to reserve for fuse data, the number of fuses to set to the default condition, and the number of fuses to include in the fuse checksum.

The value fields must occur before any device programming or testing fields in the data file. Files with only testing fields do not require the QF field and files with only programming fields do not require the QP and QV fields.

Syntax for Value Fields:

```

<fuse limit>      ::= 'QF' <number> ""
<number of pins> ::= 'QP' <number> ""
<vector limit>   ::= 'QV' <number> ""

```

Example:

QF1024*	(Indicates device has 1024 fuses)
QP24*	(Indicates device has 24 pins)
QV250*	(Indicates a maximum of 250 test vectors)

Device Programming Fields

Syntax and Overview

Each fuse or cell of a device is assigned a decimal number and has two possible states: a zero, specifying a low resistance link (a logical connection between two points); or a one, specifying a high resistance link (no logical connection between two points). The fuse numbers start at zero and are consecutive to the maximum fuse number. For example, a device with 2048 fuses would have fuse numbers between 0 and 2047. Fuse information describing the state of each fuse in the device is given by three fields. All user programmable fuses or cells may be specified with an L field. There are no separate fields for control terms or architecture fuses.

Syntax of Fuse Information fields:

```

<use information> ::= [<default state>] <fuse list>
                     [<fuse list>] [<fuse checksum>]

<default state>  ::= 'F' <binary-digit> ""

<fuse list>       ::= 'L' <number> <delimiter>
                     [<binary-digit> [<delimiter>]] ""

<fuse checksum>  ::= 'C' <hex-digit>;4 ""

```

Example:

```

F0*
L0000 01001110 00001000 11110000 11111111 01010001*
C021A*

```

Fuse Default States Field (F)

The F field defines the states of fuses that are not explicitly defined in the L fields. If no F field is specified, all fuse states must be defined after the QF field and before the first L field.

Example:

F0* *(Set default to 0)*

Fuse List Field (L)

The L field starts with a decimal fuse number and is followed by a stream of fuse states (0 and 1). The fuse number may include leading zeros (that is, L12 and L0012 are the same). A space and/or a carriage return must separate the fuse number from the fuse states. The stream of fuse states can be as long as desired (up to the maximum allowable fuse number).

If the state for a fuse is specified more than once, the last state replaces all previous states specified for that fuse. This allows a file to be modified or "patched" by appending new fuse states to the file.

Example:

```
L0000
11111011111111111111111111111111
10111111111111111111111111111111
11101111111111111111111111111111
00000000000000000000000000000000*
```

Example:

```
L0000
11111011111111111111111111111111011111111
1111111111111111
111011111111
1111111111111111
00000000000000000000000000000000*
```

Example:

```
L00  11111011111111111111111111111111*
L28  10111111111111111111111111111111*
L56  11101111111111111111111111111111*
L84  00000000000000000000000000000000*
```

Fuse Checksum Field

The fuse information checksum field is used to detect transmitting and receiving errors. The checksum is for the entire device (fuse number 0 to the maximum fuse number set by the QF field), not just the fuse states sent. If multiple C fields are received only the last is significant.

The field contains the 16-bit sum (that is, modulo 65,535) of the 8-bit words containing the fuse states for the entire device. The 8-bit words are formed as shown in Figure B-3 and the computation of the fuse checksum is as shown in Figure B-4.

Figure B-3
8-bit Words Formed from
Fuse States for Checksum

word 00 msb	1	1	1	1	1	1	1	0
Fuse No.	7	6	5	4	3	2	1	0
word 01 msb	1	1	1	1	1	1	1	1
Fuse No.	15	14	13	12	11	10	9	8
<hr/>								
word 62 msb	1	1	1	1	1	1	1	lsb
Fuse No.	-	-	-	-	499	498	497	496

Figure B-4
Computing the Fuse Checksum

QF500*		
F0*	L0000 01001110 00001000 11110000 11111111 01010001*	
C0212A*		
Fuse		
Number	MSB	LSB
0000	0 1 1 1 0 0 1 0	72
0008	0 0 0 1 0 0 0 0	10
0016	0 0 0 0 1 1 1 1	0F
0024	1 1 1 1 1 1 1 1	FF
0032	1 0 0 0 1 0 1 0	8A
0040	0 0 0 0 0 0 0 0	00
0048	0 0 0 0 0 0 0 0	00
<hr/>		
0488	0 0 0 0 0 0 0 0	00
0496	- - - 0 0 0 0	00
Fuse checksum 021A		

Device Testing Fields

Syntax and Overview

Functional test information is specified by test vectors containing test conditions for each device pin.

Syntax of Functional Test Information:

```
<function test> ::= [<default test condition>]
                    [<pin list> <test vector>
                     {<test vector>}]  

<default test condition> ::= 'X' <binary digit> "*"  

<pin list> ::= 'P' <pin number>:N "*"  

<pin number> ::= <delimiter> <number>  

N ::= number of pins on device  

<test vector> ::= 'V' <number> <delimiter>
                    <test condition>:N "*"  

<test condition> ::= <digit> | 'B' | 'C' | 'F' | 'H' | 'K'  

                    | 'L' | 'N' | 'P' | 'X' | 'Z'  

<reserved condition> ::= 'A' | 'D' | 'E' | 'G' | 'T' | 'J'  

                    | 'M' | 'O' | 'Q' | 'R' | 'S' | 'T'  

                    | 'U' | 'V' | 'W' | 'Y' | 'Z'
```

Table B-3
Test Conditions

0	-	Drive input low
1	-	Drive input high
2-9	-	Drive input to super voltage 2-9
B	-	Buried register preload
C	-	Drive input low, high, low
F	-	Float input or output
H	-	Test output high
K	-	Drive input high, low, high
L	-	Test output low
N	-	Power pins and outputs not tested
P	-	Preload registers
X	-	Output not tested, input default level
Z	-	Test input or output for high impedance

Lisa: the anchor frame name is "testcond". Couldn't figure out where to place. Caption frame is anchored (inappropriately) to beginning line of table (above anchor line--ick!). No call out reference is set.

Default Test Condition Field (X)

The X field defines the input logic level for test vectors not explicitly defined for the "don't care" test condition. The X field will set test vectors 1 through the maximum (set by QV) to the default input test condition. If the X field is used, it must be specified after the QV and QP fields and before the first test vector.

Example:

X1* *(Set default test condition to 1)*

In the following example vectors 2 and 5 would default to the "don't care" value of 0 and no outputs would be tested for vectors 2 and 5.

Example:

```

QV5*
QP20*
X0*
V0001 101010000N0ZLLHHZ11N*
V0003 111XXXXXXN0ZHLLZ11N*
V0004 011XXXXXXN0ZLHLHZ11N*

```

Test Vectors

Each test vector contains N test conditions where N is the number of pins on the device. Table A-3 lists the conditions that can be specified for device pins.

The V field starts with a decimal vector number, followed by a space, then by a series of test conditions for each pin, and terminated by an asterisk. The vector number may include leading zeros.

Example:

```
V0001 000000XXXNXXXHHHLXXN*
V0002 010000XXXNXXXHHHLXXN*
V0003 100000XXXNXXXHHHLXXN*
V0004 110000XXXNXXXHHHLXXN*
```

The vectors are applied in numerical order to the device being tested. The highest numbered vector to be applied is defined by the QV field. If a vector is not specified during a data transfer, the default value or a vector from a previous transfer will be used. If the same numbered vector is specified more than once, the data in the last vector replaces any data contained in previous vectors with that number. This allows the set of test vectors to be modified or "patched" without transferring the entire set.

Pin Sequence

The conditions contained in test vectors are applied to the device pins in numerical order from left to right unless specified otherwise with the P field. (The left most condition is applied to pin 1, and the right most condition is applied to pin 20 of a 20 pin device, for example. If the timing sequence is not defined, a test condition may be applied to pin 5 before or after pin 4.) The P field indicates an alternative correspondence between the test conditions and the pin numbers.

Example:

```
P 1 2 3 4 5 6 14 15 16 17 7 8 9 10 11 12 13 18 19 20*
V0001 111000HLHHNNNNNNNNNNNN*
V0002 100000HHHLNNNNNNNNNN*
```

Vector 1 will apply 111000 to pins 1 through 6 and HLHH to pins 14 through 17. Pins 7 through 13 and 18 through 20 are not tested (N).

Test Conditions

The test condition logic levels are defined by the device technology (for example TTL, CMOS, ECL). The 0 and 1 test conditions apply a steady state logic level to the device pin. The device tester should allow the applied input conditions to be overridden by bidirectional (input/output) device pins. The X or "don't care" test condition applies the default level defined by the X field. The F test condition applies a high impedance to the device pin.

The sequence that the input conditions are applied to the device is not defined, so multiple vectors should be used when the sequence is important. The following example ensures that pin 4 transitions to a logic level 1 before pin 3.

```
V01 XX00XXXXNXXXXXXXXXN*
V02 XX01XXXXNXXXXXXXXXN*
V03 XX11XXXXNXXXXXXXXXN*
```

The test conditions 2 through 9 apply a non-standard or super voltage to the device. This may be used to access special test modes. The levels are defined for each device and test vectors utilizing super voltages could damage "second source" devices.

The C test condition applies a logic level 0 until all other inputs are stable (and device timing specifications are met) then switches to a logic level 1 and returns to a logic level 0 before the outputs are tested. The K test condition goes from 1 to 0 to 1 in a similar manner. For devices more than one clock input, multiple test vectors should be used to ensure the proper clocking sequence. The N test condition is used for power pins and other outputs not tested.

After all inputs have stabilized, including clock, the output tests are performed. The L test for a logic level 0 and the H test for a logic level 1.

The Z test condition tests that an output is in a high impedance condition.

Register Preload

Register Preload means forcing or "jam loading" a register to a known state. Three types of register preloading, "in- circuit," "output register," and "buried register" are defined. The "in-circuit" preload is accomplished with dedicated input pins or internal control logic and uses normal in-circuit logic levels. The standard input and clock test conditions may be used to preload the registers in these devices. The "output register" and "buried register" preload use non-standard levels or "super voltages" to access special modes to preload the registers.

Because super voltages are unique for each device, the following generic methods will allow one set of test vectors to work with "second source" devices. The device programmer/tester will apply the specific super voltage algorithm for each device type.

The "output register" method is used for devices with registers connected to device pins. A P test condition is used to "jam load" registers to a desired state. When the P test condition is applied to the clock pin, the logic level on the register output pin is loaded into the register according to the logic configuration of the device. During preload certain device pins may have to be in a defined state, such as an output enable control pin.

For devices with separate banks of registers, the P test condition is applied to the each clock pin. For example, if pin 1 clocks bank A and pin 2 clocks bank B, a P on pin 2 would preload bank B.

The 0 and 1 input conditions should be used instead of the L and H output test conditions. If the preload must be verified, use a separate test vector to test the outputs.

Example: (16R4 type programmable array logic device)

```
V1 PXXXXXXXXXN1XX1101XXN*  Preload  
V2 0XXXXXXXXXN0XXHHLHXXN*  Test (don't clock)  
V3 CXXXXXXXXXN0XXHLHLXXN*  Next State
```

The "buried register" method can be used for devices with internal registers not connected to device pins. This may also be used for registers connected to device pins. The preload test vector has a B in the first position followed by a single digit, then followed by the register states and terminated with an asterisk. The preload test vector is the same length as the other test vectors and the unused positions are filled with don't cares. The device registers to be preloaded are assigned an index number starting at 1.

```
<test vector> ::= 'V' <number> <delimiter>  
'B' <digit> <test conditions>:N-2 **
```

In the following example a 20 pin device with 6 buried registers is preloaded to "110100."

```
V27 B0110100XXXXXXXXXXXX* (preload vector)  
V28 010101001N0XXHHLHXXN* (normal vector)
```

The digit in the second position of the preload test vector is used to allow more registers than pins. A 20 pin device with 30 registers would require two preload vectors.

```
V05 B01101001010101010* (preload first 18)  
V06 B111010010001XXXXXXX* (preload next 12)
```

The H and L test conditions can be used to verify the state of the buried registers.

Programmer/Tester Options

Security Fuse (G)

The security fuse(s) of certain logic devices may be enabled for programming by sending a 1 in the G field. The security fuse prevents the reading of the fuse states. Syntax for the Security Fuse field:

<security fuse> ::= 'G' <binary-digit> ''

Example:

G1* (*Enable security fuse programming*)

Signature Analysis Test (S, R, T)

Signature Analysis tests are specified by the S, R, and T fields. The S field defines the starting vector for the test. The possible states are 0 and 1. The R field contains the resulting vector or test-sum. The T field denotes the number of test cycles to be run.

Syntax for Signature Analysis Test:

<starting vector> ::= 'S' <test condition>:N ''
 <resulting vector> ::= 'R' <hex-digit>:8 ''
 <test cycles> ::= 'T' <number> ''

N ::= number of pins on device

Example:

S010001000011100011110110*
 R5BCD34A7*
 T01*

Access Time (A)

The A field defines the propagation delay for test vectors in one nanosecond increments. This field may include optional subfields.

Syntax for Access Time:

<access time> ::= 'A'{<field characters>} <number> ''

Example:

A25*
 APD25*

Examples

Data File Examples

Example 1

Minimum file for device programmer as defined Jedec Standard No. 3,
October 1983.

Example 2

Data file for device programming.

Example 3

Data file for device testing.

File for PLD 12S8 Created on 8-Feb-85 3:05PM
6809 memory decode 123-0017-001
Joe Engineer Advanced Logic Corp *
QP20* QV8*
V0001 00000XXXNXXXXHHHLXXN*
V0002 010000XXXNXXXXHHHLXXN*
V0003 100000XXXNXXXXHHHLXXN*
V0004 110000XXXNXXXXHHHLXXN*
V0005 111000XXXNXXXXHHLHHXXN*
V0006 111010XXXNXXXXHHHHHXXN*
V0007 111100XXXNXXXXHHHLHXXN*
V0008 111110XXXNXXXXLHHHHXXN*

Example 4**Data file for programming and testing with options.**

```
File for PLD 12S8 Created on 8-Feb-85 3:05PM
6809 memory decode 123-0017-001
Joe Engineer Advanced Logic Corp *
QP20* N Number of pins*
QF0448* N Number of fuses*
QV8* N Number of vectors*
G1* N Program security fuse*
F0* N Default fuse state*
X0* N Default test condition*

N Fuse RAM Data*
L0000
11111011111111111111111111111111
10111111111111111111111111111111
11101111111111111111111111111111*
L0112
010101110111101111111111111111*
L0224
010101111011101111111111111111*
L0336
010101110111011111111111111111*
N Test Vectors*
V0001 000000XXXNXXXXHHHLXXN*
V0002 010000XXXNXXXXHHHLXXN*
V0003 100000XXXNXXXXHHHLXXN*
V0004 110000XXXNXXXXHHHLXXN*
V0005 111000XXXNXXXXHHHLHXXN*
V0006 111010XXXNXXXXHHHHXXN*
V0007 111100XXXNXXXXHHHLHXXN*
V0008 111110XXXNXXXXLHHHXXN*
N Fuse RAM checksum*
C124E*
N Signature Analysis test information*
T01*
S00000000000000000000000000000000*
R95E4B822*
```

Example 5

Data file showing position independence of fields.

File for PLD 12S8 Created on 8-Feb-85 3:05PM
6809 memory decode 123-0017-001
Joe Engineer Advanced Logic Corp *
QP20*
QE448* QV8* F0*
V1 000000000N000HHHL00N*
V2 010000000N000HHHL00N*
V3 100000000N000HHHL00N*
V4 110000000N000HHHL00N*
L0 11111011111111111111111111111111*
L28 10111111111111111111111111111111*
L56 11101111111111111111111111111111*
L84 00000000000000000000000000000000*
L112 01010111011110111111111111111111*
L224 01010111011101111111111111111111*
L336 01010111011101111111111111111111*
L140 11111111111111111111111111111111*
L140 0000000000000000000000000000000000*
C124E*
V8 111111111N111HHHL11N*
V6 111010000N000HHHH00N*
V7 111100000N000HHHL00N*
V5 111000000N000HLHH00N*
V8 111110000N000LHHH00N*

Appendix C

Programmable Logic Device Information

This appendix provides information that pertains to specific types of programmable logic devices. The types of information provided in this appendix include

- Programming support for specific devices
- Special ABEL-PLD considerations for certain devices
- Device node numbers (assigned by Data I/O) and their functions

PROM Support

ABEL-PLD supports PROMs based on memory size (32 x 8 or 1024 x 4). Ignoring the pin numbers on the chip diagram, it is possible to use ABEL-PLD's RA9P8 file (512 x 8) for AMD's 27S15 (24 pins), 27S27 (22 pins) or 27S29 (20 pins). The chip diagram will match the 20 pin device. See the cnt10rom.abl example on disk or tape for a state diagram design. This example also shows how to program the powerup initialize words available on the RA10R8 and RA11R8.

Table C-1
TOABEL Supported Devices

P10H8	P16H2	P16R8	P20R4
P10L8	P16H6	P16RP4	P20R6
P12H10	P16H8	P16RP6	P20R8
P12H6	P16HD8	P16RP8	P20RS10
P12L10	P16L2	P18H4	P20RS4
P12L6	P16L6	P18L4	P20RS8
P14H4	P16L8	P20C1	P20X10
P14H8	P16LD8	P20H2	P20X4
P14L4	P16P8	P20L10	P20X8
P14L8	P16R4	P20L2	P22V10
P16C1	P16R6	P20L8	

Altera and Intel EPLDs

The following devices are supported.

Table C-2
Signetics Supported Devices

F100	F162	F253
F103	F163	F2605
F105	F167	F273
F153	F168	F839
F161	F173	

ABEL-PLD	Altera	Intel	Turbo Bits	Miser Bits
E0310	EP300		—	—
E0310	EP310	5C031	—	—

AMD

Due to conflicting device names, the following AMD parts have been assigned alternative names for use with ABEL-PLD:

AMD Name	ABEL-PLD Name
22P10	P22AP10
22XP10	P22XP10
20RP4	P20ARP4
20XRP4	P20XRP4
20RP6	P20ARP6
20XRP6	P20XRP6
20RP8	P20ARP8
20XRP8	P20XRP8
20RP10	P20ARP10
20XRP10	P20XRP10

Lattice, SGS Thomson, National, VTI and AMD GALs

The 16V8, 16Z8, and 20V8 GALs have three architecture modes: Register, Complex, and Simple. ABEL-PLD uses three device files to support each GAL. The architecture mode suffix is added to the device name. For example, to use a 16V8 in the Register mode, use the device name P16V8R.

16V8	16Z8	Control Bits		Mode	AC0	SYN
		20V8				
P16V8R	P16Z8R	P20V8R	Register	1	0	
P16V8C	P16Z8C	P20V8C	Complex	1	1	
P16V8AS*	P16Z8AS	P20V8AS*	Simple	0	1	

* The original Lattice GALs did not have I/O feedback in the Simple mode. The new 16V8A and 20V8A now have I/O feedback. To differentiate the two models, a new device type was created. The P16V8AS and P20V8AS are Simple mode parts with I/O feedback.

The new *Logic Diagram Package* reflects these three operational modes.

MMI P16X4

The example, limit.abl, demonstrates the use of macros to control the input pairs for arithmetic PALs, such as the P16X4.

Signetics

Signetics has changed their product numbering system. All 82 device prefixes were replaced with PL. For example, 82S153 became PLS153 with the numbering change. For ABEL-PLD 3.2, the numbering of these device has been changed further. In ABEL-PLD 3.2, the 82S has been dropped from the device name so that, for example, the device file for the F82S153 is now named F153. To support the old numbering system (that may appear in existing ABEL-PLD source files) each of the following devices is provided with duplicate device files, one for each device number:

Old Device Name	New Device Name
F82S100	F100
F82S103	F103
F82S153	F153
F82S173	F173

All other devices with the F82S prefix (such as the F82S153) are no longer supported by ABEL-PLD 3.2 and only the new device files are provided, such as F153, F163, etc.

Node Extension Notation

Load	.L
J input	.J
Preset	.PR
K input	.K
Reset	.RE
Mode Fuse	.M
Enable	.OE

The complement term may be addressed with nodes shown below:

Node	Number	Name
F473	28	comp
F105	49	comp
F167	43	comp
F168	45	comp

The complement term may be referred to in your ABEL-PLD source file with the name COMP, with no need to declare COMP in the declarations section of the source file.

For the F105 family of devices, the buried registers may be referred to as P0 through P5, with no node declaration required.

Signetics 16V8 and 20V8

The Signetics 16V8 and 20V8 parts use the device files P16SV8 and P20SV8, respectively.

Device Nodes

Table C-1 lists the nodes used by ABEL-PLD along with the names given to those nodes by the manufacturer. Refer also to the manufacturer's logic diagrams or those supplied in the Programmable Logic Device Schematics publication. Device names are given in boldface followed by the number of pins on the device; node numbers follow. Detailed information about device nodes used in an ABEL-PLD design can be obtained by specifying the DOCUMENT flags -S and -F1 on the ABEL-PLD command line. Refer also to the ABEL-PLD Language Reference and Applications Guide for information on the use of nodes.

Table C-3
Nodes Used by ABEL-PLD

Node No.	Notation	Name	Function
Device: F105 (28 Pins)			
29	.R	RF0	R f/f input for F0 (pin 18)
30	.R	RF1	R f/f input for F1 (pin 17)
31	.R	RF2	R f/f control for F2 (pin 16)
32	.R	RF3	f/f input for F3 (pin 15)
33	.R	RF4	R f/f input for F4 (pin 13)
34	.R	RF5	R f/ input for F5 (pin 12)
35	.R	RF6	R f/f input for F6 (pin 11)
37	-	P0	S f/f input (and output) for P0
38	-	P1	S f/f input (and output) for P1
40	-	P3	S f/f input (and output) for P3
41	-	P4	S f/f input (and output) for P4
42	-	P5	S f/f input (and output) for P5

Node No.	Notation	Name	Function
43	.R	RP0	R f/f input for P0
44	.R	RP1	R f/f input for P1
45	.R	RP2	R f/f input for P2
46	.R	RP3	R f/f input for P3
47	.R	RP4	R f/f input for P4
48	.R	RP5	R f/f input for P5
	-	COMP	49 Complement
Device: F167 (24 Pins)			
25	.R	RF0	R f/f input for F0 (pin 9)
26	.R	RF1	R f/f input for F1 (pin 10)
27	.R	RF2	R f/f input for F2 (pin 11)
28	.R	RF3	R f/f input for F3 (pin 13)
29	.R	RP0	R f/f input for P0 (pin 14)
30	.R	RP1	R f/f input for P1 (pin 15)
31	-	P7	S f/f input (and output) for P7
32	-	P6	S f/f input (and output) for P6
33	-	P5	S f/f input (and output) for P5
34	-	P4	S f/f input (and output) for P4
35	-	P3	S f/f input (and output) for P3
36	-	P2	S f/f input (and output) for P2
37	.R	RP7	R f/f input for P7
38	.R	RP6	R f/f input for P6
39	.R	RP5	R f/f input for P5
40	.R	RP4	R f/f input for P4
41	.R	RP3	R f/f input for P3
42	.R	RP2	R f/f input for P2
43	-	COMP	Complement term at offset 45
Device: EP300			
21	-	R	Reset term at fuse 2628
22	-	P	Preset term at fuse 2592
Device: P22V10 (24 Pins)			
25	-	R	Reset term at fuse 0000
26	-	P	Preset term at fuse 5764

User Electronic Signature Word

The P16V8, P16Z8, P18V10, P20V8, P22V10G, and P26CV12 devices all have bits reserved for a User Electronic Signature Word. The library files for these devices have a macro for setting these bits. To use the macro, use a library statement to include the file, then call the macro with

SIGNATURE(WXYZ)

after the declarations section. See the example gals.abl for sample usage. An example of the electronic signature word macro for the P16V8 is shown below.

```
" P16V8 Electronic Signature Word
SIGNATURE MACRO (string)
{
    @CONST cnt = 1;
    @CONST r1 = 2056; " First address of signature word
    @IRPC A (?string)
    {
        @IF (cnt 8) { @MESSAGE 'WORD TOO LONG'; @EXIT; }
        FUSES [@EXPR r1;..@EXPR (r1+7);] = @EXPR '?A';
        @CONST r1 = r1 + 8;
        @CONST cnt = cnt + 1;
    }
}
```

Turbo Bits

The E0310 device file has a macro to control the turbo bits. Turbo bits allow selection of either speed or power consumption within the device. The default mode is to program the turbo bits, which will make the device respond faster to changes on the inputs.

Macro files in the library control the correct fuses for each device. To use the macro, use a library statement to include the correct macro file, then call the macro with

TURBO(0)

To put the device in the low power mode.

Appendix D

ASCII Table

ASCII in Decimal and Hexadecimal

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	/
7	07	BEL	50	32	2	93	5D]
8	08	BS	51	33	3	94	5E	^
9	09	HT	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	,
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	:	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	SP	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	\$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	-
39	27	'	82	52	R	125	7D	}
40	28	(83	53	S	126	7E	~
41	29)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

Appendix E

Set Operations

Set operations are applied according to normal rules of Boolean algebra, as described below. In the rules, uppercase letters are used to denote set names, and lowercase letters are used to denote elements of a set. The letters, k and n are used as subscripts to the elements and to the sets. A subscript following a set name (uppercase letter) indicates how many elements the set contains. Thus, the notation, A_k , indicates that set A contains k elements. a_{k-1} is the $(k-1)$ th element of set A. a_1 is the first element of set A.

Set Operation Rules

$|A_k ::= [l_{ak}, l_{ak-1}, \dots, l_{a_1}]$

$-A_k ::= |A_k + 1$

$A_k.OE ::= [a_k.OE, l_{ak-1}.OE, \dots, a_1.OE]$

$a_k \ A_k \ a_{k-1} \ a_1$

$A_k \ \& \ B_k ::= [a_k \ \& \ b_k, a_{k-1} \ \& \ b_{k-1}, \dots, a_1 \ \& \ b_1]$

$A_k \ # \ B_k ::= [a_k \ # \ b_k, a_{k-1} \ # \ b_{k-1}, \dots, a_1 \ # \ b_1]$

$A_k \ $ \ B_k ::= [a_k \ $ \ b_k, a_{k-1} \ $ \ b_{k-1}, \dots, a_1 \ $ \ b_1]$

$A_k \ !$ \ B_k ::= [a_k \ !$ \ b_k, a_{k-1} \ !$ \ b_{k-1}, \dots, a_1 \ !$ \ b_1]$

$A_k == B_k ::= (a_k == b_k) \ \& \ (a_{k-1} == b_{k-1}) \ \& \ \dots \ \& \ (a_1 == b_1)$

$A_k != B_k ::= (a_k != b_k) \ # (a_{k-1} != b_{k-1}) \ # \dots \ # (a_1 != b_1)$

$A_k + B_k ::= D_k$

where:

$d_n ::= a_n \ $ \ b_n \ $ \ c_{n-1}$

$c_n ::= (a_n \ $ \ b_n) \ # (a_{n-1} \ & \ c_{n-1}) \ # \ b_n \ \& \ c_{n-1}$

$c_0 ::= 0$

$A_k - B_k ::= A_k + (-B_k)$

$A_k < B_k ::= c_k$

where

$c_n ::= (l_{an} \ \& \ (b_n \ # \ c_{n-1}) \ # \ a_n \ \& \ b_n \ \& \ c_{n-1})! = 0$

$c_0 ::= 0$

$A_k = B_k ::= !(B_k \ A_k)$

Appendix F

The Terminal Emulator Utility

Term-100 is a VT-100 terminal emulation program written to run on an IBM PC/XT/AT or compatible computer. It is a program which makes the PC act like a VT-100 terminal and allows the user to communicate with other computers or machines (such as UniSite and other Data I/O logic programmers) and transfer files between them. Term-100 supports operation up to 19,200 baud and software handshaking (Xon/Xoff). It also supports 8-bit data transfers which permits the efficient transfer of files using any of the binary data formats supported by the programmer. Term-100 has two operating modes, General (G) and the programmer (U). General mode causes Term-100 to perform like a VT-100 terminal, while the programmer mode adds some additional features. These features allow easier operation when controlling a programmer from the PC. This mode automatically opens and closes files when transferring data files with the programmer, which frees the user from having to direct the terminal emulator to do so at the appropriate times.

Term-100 Features

TERM-100.EXE is a VT100-like terminal emulator program that provides

- Control of the Programmer on any PC or compatible
- Use of file transfer operations between the programmer and the host PC without leaving the programmer menus
- Binary file transfer between the programmer and a PC

- Automatic file transfer operations that eliminate the need for direct user interaction with the terminal emulator
- Configuration file to specify UniSite/General operation mode and to set communication parameters
- Send/receive data in 7- or 8-bit mode
- Automatic file closure at the end of upload and download operations
- Reports of data communication errors
- Commands for displaying and changing the current directory

The Term-100 Files

The following files accompany the Term-100 program and perform the indicated functions.

UNISITE.BAT	Batch file used to invoke Term-100 (UniSite mode)
UNI9600.CFG	Configuration file (UniSite mode, 9600 baud)
UNI19200.CFG	Configuration file (UniSite mode, 19200 baud)
TERM.BAT	Batch file used to invoke TERM-100 (General Mode)
GEN9600.CFG	Configuration file (General mode, 9600 baud)
GEN19200.CFG	Configuration file (General mode, 19200 baud)
TERM-100.EXE	Terminal emulator program

Modifying the Batch Files

There are two batch files provided with TERM-100. By default, these files use 9600 baud configuration files. (Configuration files are discussed in "The Term-100 Configuration File"). For 19200 baud operation, you can edit these files and change the configuration file name to UNI19200.CFG, which is the name for the configuration file that specifies 19200 baud operation to TERM-100. For example:

**UNISITE.BAT
(original file)**

```
Rem: Term-100 will use GEN9600.CFG from ABEL3DEV directory.
if not (%ABEL3DEV%) == () TERM-100
%ABEL3DEV%\UNI9600.CFG
```

**UNISITE.BAT
(modified file)**

```
Rem: Term-100 will use GEN9600.CFG from ABEL3DEV directory.
if not (%ABEL3DEV%) == () TERM-100
%ABEL3DEV%\UNI19200.CFG
```

```
Rem: Term-100 will use GEN9600.CFG from local directory if
Rem: ABEL3DEV not defined
if (%ABEL3DEV%) == () TERM-100 UNI19200.CFG
```

Modify the TERM.BAT file in a similar fashion if you will be using Term-100 in the General mode (not necessary for operation with UniSite only).

Running Term-100

After installing the Term-100 files, simply enter

UniSite GENERAL	<i>(for UniSite mode)</i> <i>(for General mode)</i>
--------------------	--

to start Term-100. When Term-100 is started, it will read a configuration file to set the operating mode (General or UniSite) and set the various communication port parameters (baud rate, parity, etc.).

The Term-100 Configuration File

Term-100 uses a configuration file to specify the mode (UniSite/General) and the communication parameters. This file is read when Term-100 is invoked. These files are expected to be in the directory pointed to by the environment variable ABEL3DEV (normally the \DATAIO\LIB3 directory). If this variable is not defined, then the files are expected to be in the current directory. If Term-100 does not find the configuration file, the message

Unable to Use Config File

will be printed on the screen and Term-100 will use the default settings (refer to Default Settings).

Four configuration files are included with Term-100, and the user should create any additional configuration files desired. The configuration file must conform to the following specifications:

1. The first line of the file must specify the mode: either General (G) or UniSite (U). Upper or lower case may be used and only the first character of the word is significant.
2. The second line must specify the baud rate. The complete number is required, ie. 9600, not 96.
3. The third line specifies the parity (N/E/O). Upper or lower case may be used and only the first character of the word is significant.
4. The fourth line specifies the number of bits per character (7 or 8).
5. The fifth line specifies the number of stop bits (1 or 2).

Note: Comments can follow the first word (or character) as long as they are separated from the word by a space. The line must end with a line feed. Pressing <Enter> is optional.

Default Settings

The following factory default settings are used if no configuration file is present:

Parameter	Factory Default Setting
Mode	G (General)
Baud Rate	9600
Parity	N (None)
Bits per Character	8
Stop Bits	1

Term-100 Commands and Functions

Term-100 has two help screens. One can be invoked when in the General (G) mode and the other can be invoked when in the UniSite (U) mode. Table 4-1 describes the menu and functions applicable to the General (G) mode. Table 4-2 lists the functions that can be invoked when in the UniSite (U) mode.

Table F-1
Term-100 General Mode

Command	Function
Shift F1	Transfer file from PC to COM1 serial port.
Shift F2	Transfer file from COM1 serial port to PC.
Shift F3	Close file (to be used in conjunction with Shft F1 and Shft F2)
Shift F4	Toggle local echo on/off (off - full duplex, on - half duplex)
Shift F5	Toggle online / offline
Alt F1	Terminates the Term-100 program and returns to DOS (closes any open files).
Alt F2	Switches between text mode and binary mode for transferring files to COM1. Ctrl-Z, Ctrl-C, Line-Feed, and Null characters are stripped from the files sent to COM1: in text mode. When transferring a file from the PC to a host while in text mode, an EOF character is sent after the data to indicate the end of the file. In binary mode, all characters are transmitted. Binary mode is the recommended mode for operation with UniSite. <i>Caution: Do not use text mode to transmit files in JEDEC or binary format. Filtering of control characters may result.</i>
Alt F3	Displays the Term-100 Help screen.
Alt F4	Displays the current end-of-file character for text mode file transfers. To change this character, enter the hexadecimal value of the desired control character (ASCII 1 through 1F). Press <Enter> to select the new character. The default setting is Ctrl-Z (1A Hex). This parameter only applies to text mode and the EOF character must match that selected on the UniSite Communications Parameters screen. <i>Note: Term-100 operating in the U (UniSite) mode automatically closes the file after UPLOAD and DOWNLOAD operations.</i>
Alt F5	Change directory: Allows the user to change directories from within Term-100. Enter the path for the directory you want to change to. You may change to a different drive also by entering the drive letter followed by a colon (A:\UTIL).
Alt F6	Display current directory: displays the current directory without exiting Term-100.

Table F-2
Term-100 UniSite Mode

Command	Function
Alt F1	Terminates the Term-100 program and returns to DOS (closes any open files).
Alt F2	Switches between text mode and binary mode for transferring files to COM1: Control-Z, Control-C, Line-Feed, and Null characters are stripped from the files sent to COM1: in text mode. When transferring a file from the PC to a host while in text mode, an EOF character will be sent after the data to indicate the end of the file. In binary mode, all characters are transmitted. Binary mode is the recommended mode for operation with UniSite.
	<i>Caution: Do not use text mode to transmit files in JEDEC or binary format. Filtering of control characters may result.</i>
Alt F3	Displays the Term-100 Help screen.
Alt F4	Displays the current end-of-file character for text mode file transfers. To change this character, enter the hexadecimal value of the desired control character (ASCII 1 through 1F). Press <Enter> to select the new character. The default setting is Control-Z (1A Hex). This parameter only applies to text mode. If text mode is used, the EOF character must match that which is selected on the UniSite Communications Parameters screen.
	<i>Note: Term-100 operating in the U (UniSite) mode automatically closes the file after UPLOAD and DOWNLOAD operations.</i>
Alt F5	Change directory: Allows the user to change directories from within Term-100. Enter the path for the directory you want to change to. You may change to a different drive also by entering the drive letter followed by a colon (A:\UTIL).
Alt F6	Display current directory: displays the current directory without exiting Term-100.

The status line at the bottom of the screen displays the current settings of various Term-100 parameters and also indicates how to access the Term-100 help screen ALT F3. The fields on the status line are shown below.

[Caps]	The Caps-Lock key is enabled
[Num]	The Num-Lock key is in the Number mode
[Text]	The file-transfer mode is set to Text mode
[Bin]	The file-transfer mode is set to Binary mode
[EOF]	Indicates the end of file character (only used for file transfers in text mode)

Transferring Files, General Mode

JEDEC File Transfer, PLDS to Computer

In this example, a JEDEC format device file must be loaded into the programmer RAM (part of the Programmable Logic Development System, PLDS) and be ready to be downloaded to the computer.

1. Set the Model 29 programmer to full-duplex (echo) operation by entering the following key sequence from the front panel of the programmer:

SELECT C E START 0 START

2. Enter the logic terminal mode (TRC) from the Model 29 programmer by entering:

SELECT E 1 START

3. Run the TERM-100 program by typing the following (from the computer keyboard):

TERM

Your computer is now emulating a dumb (VT52) terminal and the TERM-100 program menu is displayed on the screen.

4. Prepare TERM-100 to receive a file by selecting the full-duplex mode from the menu (TERM-100 defaults to full-duplex at start-up):

F7

(toggles between full-duplex and half-duplex operation)

5. Select the Create Disk File function by entering:

F4 *filename*

to receive the transmitted file. You can also specify a disk drive and path, if desired.

6. Prepare the programmer to transmit the JEDEC file by typing (from the computer keyboard):

C

You should see the device file scroll down the screen as it is being written into the disk file.

7. Using the TERM-100 Terminate File Transfer function, close the file when transmission is complete by entering:

F2

This example is concluded and the JEDEC file transfer from the programmer to your computer is complete. There are other steps in the sequence of downloading a file from the programmer to the computer using TERM-100. The preceding method is the most direct, and requires the least number of steps.

JEDEC File Transfer, Computer to PLDS

In this example, a JEDEC format device file must be created and saved to disk. If you are performing these examples in sequence, you can transfer the same device file from Example 1 back to the programmer (part of the Programmable Logic Development System, PLDS). Ensure that the COM1: serial port is correctly configured.

1. Run the TERM-100 program by typing the following characters from the computer keyboard:

TERM

2. Ensure that the Model 29 is in full-duplex mode. If not, enter the following key sequences from the programmer front panel:

SELECT C E START 0 START

3. Enter the Logic Terminal mode (from the front panel of the Model 29 programmer) by entering:

SELECT E 1 START

The logic programmer menu will be displayed, including the Command prompt. Most of this menu will overwrite the TERM-100 program menu on the screen, except for the status block. The TERM-100 status block and the Command prompt indicate that you can interact with the TERM-100 program and the programmer through the terminal emulator.

4. At the Command prompt (from the computer) type:

Command:1 xxxxx

where 1 is the menu selection that allows you to specify the family/pinout code for the device to be programmed. The **xxxxx** represents, and is replaced by, the device family/pinout code number that you supply.

5. Prepare the programmer to receive a JEDEC file. At the Command prompt (from the computer) type:

Command:B

6. To recall the complete TERM-100 menu, press:

F3

The TERM-100 menu will be displayed.

7. Prepare TERM-100 to transmit a file by specifying the appropriate transmission mode (use binary mode to transmit JEDEC files). From the TERM-100 menu, press:

F6

(this toggles between Binary and Text modes)

See the status field at the bottom of the screen--TERM-100 defaults to Binary mode at start-up.

8. Set TERM-100 for full-duplex (TERM-100 defaults to full-duplex at start-up):

F7

(toggles between full- and half-duplex operation)

9. Select the Send Disk File function from the TERM-100 menu:

F5 filename RETURN

to transmit the file. The file being transmitted must be for the same device family/pinout code as that previously entered from the menu. You can also specify a disk drive and path, if desired.

When the file transfer is complete, the Command prompt will be displayed.

10. Using the TERM-100 Terminate File Transfer function, close the file by pressing

F2

This concludes the example of the JEDEC file transfer from your computer to the programmer.

The examples provided in this section describe only two of the possibilities for using Term-100. Note that both examples make use of features or capabilities that originate from programmer menu option selections and programmer/pak Select Code functions.

When your programmer is set in Terminal Remote Control mode and TERM-100 has converted your computer into a programmer terminal, all of these programmer/pak features or capabilities are accessible. Refer to the Model 60A/H and/or LogicPak manuals for more information on the use of these features and capabilities.

Transferring Files, UniSite Mode

A special host command named Transfer is used by UniSite to communicate with Term-100.

Only the first two characters (TR) in either upper- or lowercase are necessary, or you may type TRANSFER. The Transfer command should be followed by a filename and ended with a carriage return. There should be at least one blank to separate the command and the filename. Extra blanks are ignored. To transfer a file between the PC and UniSite, enter the command Transfer

followed by the filename on the host command line of the programmer. The file will be automatically opened and closed on the PC at the appropriate times.

Upload Operation

1. When you are in the More Commands/Transfer Data/Upload menu, the destination field must reflect which serial port the PC is connected to.
2. In the Upload Data to Host field at the bottom of the screen, enter

transfer *filename*

or alternately

tr *filename*

The file will be transferred.

Download Operation

1. When you are in the More Commands/Transfer Data/Download menu, the Source field must reflect which serial port the PC is connected to. (Terminal or Remote; Terminal is used for most operations with a PC.)
2. In the Download Data from Host field at the bottom of the screen, enter

transfer *filename*

or alternately

tr *filename*

Recapturing the UniSite Screen

Anytime you want to repaint the UniSite screen, press

<Ctrl> R

Term-100 Error Messages

Term-100 will report the following errors:

Parity Error

The parity of the data received did not correspond with the parity setting of the UART in the PC. Make sure the parity settings in the PC and the programmer are the same.

Framing Error

The received character did not contain a valid stop bit. Make sure the baud rates of the PC and the programmer match.

Overrun Error

The UART received another character before the previous one was read by the processor, causing data to be lost. Try using a slower baud rate or eliminate any background programs currently running on the PC.

Transfer Error

This indicates Term-100 did not close the file on the PC automatically. Make sure the Term-100 mode is set to UniSite (U). If the mode is correct, try using the pacing delay feature on the programmer, or a slower baud rate.

Unable to use config file

Term-100 either couldn't find the configuration file or there was an error detected in the configuration file. During installation of ABEL, the configuration files are loaded into the LIB3 directory. Make sure the environment variable ABEL3DEV is defined and points to this or some other directory that contains the configuration files. Or, you can put the files in the directory you are currently running from. Check the contents of the configuration file used (UNI9600.CFG by default) to ensure there are no syntax errors. If this error occurs, Term-100 will use the default parameters.

Default comm param used

Term-100 detected a syntax error in the communication parameters specified in the configuration file. Term-100 will use the mode specified in the file. However, it will use the default values for the communication parameters. Make sure the syntax and values specified in the configuration file are correct.

Not enough memory for Term-100

This indicates there is not enough free memory available in the PC. Term-100 requires a minimum of 48K bytes of RAM. Remove other programs from memory or add more RAM.

Troubleshooting

If you cannot invoke Term-100 or are having difficulty with the transfer command, check the following:

1. Invoke the Term-100 Help screen and make sure the mode of Term-100 is set to UniSite (U). (This is determined by the configuration file used).
2. When you are using the TR (Transfer) command, make sure you are using the correct syntax on the host command line of the programmer.
3. If the UniSite screen is not present on the PC screen, press **<Ctrl> R** to repaint the UniSite screen.
4. Be sure to type **UNISITE** (*uppercase*) to invoke Term-100. *Do not type*

TERM-100

since this will prevent use of the UNISITE.BAT batch file which is necessary for setting the configuration parameters.

Make sure the UniSite terminal port baud rate matches the Term-100 baud rate. The UniSite factory default setting is 9600 baud.

Index

- (dash)

- a, PARSE, 3-14
- b, SIMULATE, 3-33
- c, FUSEMAP, 3-28
- d, FUSEMAP, 3-27
- e, PARSE, 3-14
- f, DOCUMENT, 3-39
- h, PARSE, 3-14
- i
 - DOCUMENT, 3-38
 - FUSEMAP, 3-27
 - PARSE, 3-14
 - REDUCE, 3-22
 - SIMULATE, 3-32
 - TOABEL, 3-44
 - TRANSFOR, 3-19
- j, FUSEMAP, 3-29
- k, FUSEMAP, 3-28
- l, PARSE, 3-14
- n, SIMULATE, 3-32
- o
 - DOCUMENT, 3-38
 - FUSEMAP, 3-27
 - PARSE, 3-14
 - REDUCE, 3-22
 - SIMULATE, 3-32
 - TOABEL, 3-44
 - TRANSFOR, 3-19
- p, PARSE, 3-14
- q, DOCUMENT, 3-38
- r, REDUCE, 3-22
- s, DOCUMENT, 3-39

-t, SIMULATE, 3-34
-u, SIMULATE, 3-35
-w, SIMULATE, 3-33
-x, SIMULATE, 3-35
-y, PARSE, 3-15
-z, SIMULATE, 3-35

. (period)

.abl file
 overview, 2-2
.AP, 4-3
.AR, 4-3
.C, 4-3
.J, 4-3
.K, 4-3
.L, 4-3
.OE, 4-3
.PR, 4-3
.Q, 4-3
 register feedback, 11-12
.R, 4-3
.RE, 4-3
.S, 4-3

@

@CONST, 8-3
@EXIT, 8-3
@EXPR, 8-4
@IF, 8-4
@IFB, 8-5
@IFDEF, 8-5
@IFIDEN, 8-6
@IFNB, 8-6
@IFNDEF, 8-7
@IFNIDEN, 8-7
@INCLUDE, 8-8
@IRP, 8-9
@IRPC, 8-10
@MESSAGE, 8-10
@PAGE, 8-11
@RADIX, 8-11
@REPEAT, 8-12
@STANDARD, 8-12

1 (numbers)

1 to 8 demultiplexer, example, 10-9
 12 to 4 multiplexer, example, 10-6
 4-bit comparator, example, 10-26
 4-bit counter/multiplexer, example, 10-12
 6809 memory address decoder, example, 10-3
 7-segment display decoder, example, 10-22
 8-bit barrel shifter, example, 10-19
 @ALTERNATE, 8-2

A

Abellib, 3-44
 Altera, supported devices, C-2
 AMD
 alternate filenames, C-3
 GALs, architecture modes, C-3
 Architecture modes, C-3
 Arguments, 6-21
 -a, 3-14
 actual, 6-22
 dummy, 6-21
 See also Options
 Arithmetic operators, 6-8
 Arrays, complement, 11-12
 ASCII characters, 6-2
 ASCII table, D-1
 Assignment operators, 6-11
 Assignments
 multiple, 10-16
 to identifiers, 6-14
 to sets, 6-17
 Asynchronous circuits, and simulation, 4-27
 Attributes, 7-13
 com, 7-14
 eqn, 7-14
 feed_, 7-14
 for nodes, 7-14
 fuse, 7-14
 neg, 7-14
 pin, 7-14
 polarity control, 9-3
 pos, 7-14
 reg, 7-14
 reg_, 7-14

B

Base number
 changing, 8-11
Batch files
 calling ABEL-PLD from, 3-10
 creating own, 3-11
Binary to bcd converter, example, 10-35
Blocks, 6-21
Boolean set operations, E-1
Breakpoints
 advanced use of, 4-30
 SIMULATE, 3-33
Buffered outputs
 and simulation, 4-38

C

Case statement, 7-23
 example, 7-24
Characters
 valid, 6-2
Checksum
 -c, 3-28
Chip diagram, 3-39
Circuits, asynchronous, 4-27
Cleanup, 3-46
Clock inputs, 4-29
 See also feedback
com, 7-14
Combinational signal, 7-14
Command line, 3-10
 -a, PARSE, 3-14
 -b, SIMULATE, 3-33
 -c, FUSEMAP, 3-28
 -d, FUSEMAP, 3-27
 -e, PARSE, 3-14
 -f, DOCUMENT, 3-39
 -h, PARSE, 3-14
 -i, DOCUMENT, 3-38
 -i, FUSEMAP, 3-27
 -i, PARSE, 3-14
 -i, REDUCE, 3-22
 -i, SIMULATE, 3-32
 -i, TRANSFOR, 3-19
 -k, FUSEMAP, 3-28
 -l, PARSE, 3-14
 -n, SIMULATE, 3-32
 -o, DOCUMENT, 3-38
 -o, FUSEMAP, 3-27

-o, PARSE, 3-14
-o, REDUCE, 3-22
-o, SIMULATE, 3-32
-o, TRANSFOR, 3-19
-p, PARSE, 3-14
-q, DOCUMENT, 3-38
-s, DOCUMENT, 3-39
-t, SIMULATE, 3-34
-u, SIMULATE, 3-35
-w, SIMULATE, 3-33
-x, SIMULATE, 3-35
-y, PARSE, 3-15
-z, SIMULATE, 3-35
DOCUMENT, 3-36
FUSEMAP, 3-24
PARSE, 3-12
REDUCE, 3-20
SIMULATE, 3-30
TRANSFOR, 3-18
Comment fields, B-7
Comments, 6-5
Complement arrays, 11-12
Configurable output enable, 11-2
Constant declarations, 5-7, 7-9
Constants
 special, 6-7

D

Data fields, B-6
Debugging state machines, 4-10, 9-13
Decimal, changing numbering system, 8-11
Declarations, 7-5
 keyword, 7-6
 making within source file, 7-6
Definition fields, B-7
Design considerations, 9-1
Device declaration, 5-6
Device declarations, 7-6
Device library, 3-6
 path to, 3-15
Device nodes, C-5
 controlling, 4-1
Device programming fields, B-9
Device testing fields, B-12
Device type
 -n, 3-32
 variable, 4-7
Devices

finding a specific, 3-43
registers, 11-8
specific information, C-1
See also User Notes
with clock inputs, 4-29

DIP diagram, 3-39

Directives

- @ALTERNATE, 8-2
- @CONST, 8-3
- @EXIT, 8-3
- @EXPR, 8-4
- @IF, 8-4
- @IFB, 8-5
- @IFDEF, 8-5
- @IFIDEN, 8-6
- @IFNB, 8-6
- @IFNDEF, 8-7
- @IFNIDEN, 8-7
- @INCLUDE, 8-8
- @IRP, 8-9
- @IRPC, 8-10
- @MESSAGE, 8-10
- @PAGE, 8-11
- @RADIX, 8-11
- @REPEAT, 8-12
- @STANDARD, 8-12
- block in, 6-21
- expand, 3-14
- if blank, 8-5
- if defined, 8-5
- if identical, 8-6
- if not blank, 8-6
- if not defined, 8-7
- if not identical, 8-7
- indefinite repeat, 8-9
- indefinite repeat, character, 8-10
- overview, 8-1
- to create test vectors, 4-12

Document

- f, 3-39
- i, 3-38
- o, 3-38
- q, 3-38
- s, 3-39
- equations, 3-38
- fusemap, 3-39
- operation, 3-36
- overview, 2-11
- symbols, 3-39

Document output file, 2-5

Don't Care values
and simulation, 4-16
SIMULATE, 3-35
Dot extensions, 4-2
.Q, 11-12
table of, 4-3
Downloading
to Model 29 with LogicPak, 12-1
to Model 29 with UniPak2, 12-3
to UniSite, 12-3
Dummy arguments, 6-21
-a, 3-14
passing from the command line, 4-7

E

Electronic signature word, C-7

Enable

See also .OE
configurable, 11-2
output, 11-1
pin controlled, 11-1
term controlled, 11-1

eqn, 7-14
Equations, 6-13

DOCUMENT, 3-38

Equations statement, 5-8, 7-16

Errors, A-1

in pin assignment, 9-6

Espresso, 3-23

Espresso by pin, 3-23

Examples

1 to 8 demultiplexer, 10-9
12 to 4 multiplexer, 10-6
4-bit comparator, 10-26
4-bit counter/multiplexer, 10-12
6809 memory address decoder, 10-3
7-segment display decoder, 10-22
8-bit barrel shifter, 10-19
binary to bcd converter, 10-35
blackjack machine, 10-30
design, 10-1
equations, 10-3, 10-6, 10-9, 10-12, 10-19, 10-26
F153, 10-26
macros, 10-26, 10-36
multiplexer/adder/comparator, 10-32
P14H4, 10-6
P14L4, 10-3
P16L8, 10-9, 10-35

P16R4, 10-12
P16R6, 10-38
P20R8, 10-19
P22V10, 10-32
RA5P8, 10-22
state description, 10-38
truth tables, 10-22, 10-35
Exclusive OR equations, 9-1
Expanded text
 -e, 3-14
 -p, 3-14
Expressions, 6-11
 table of, 6-13

F

F153, example, 10-26
feed_or, 7-14
feed_pin, 7-14
feed_reg, 7-14
Feedback, 11-9
 .Q, 11-12
 and simulation, 4-39
 of macro cells, 11-6
 selectable, 11-11
 specifications, 7-14
Fields
 comment, B-7
 data, B-6
 definition, B-7
 device programming, B-9
 device testing, B-12
 programming, B-3
 testing, B-3
Files
 .doc, 2-5
 .jed, 2-4
 .lst, 2-4
 .out, 2-5
 .sim, 2-5
 document output, 2-5
 intermediate, 2-5
 list, 2-4
 programmer load, 2-4
 simulation output, 2-5
 temporary, 3-46
Flag statement, 5-6, 7-4
Flags
 See Options

table of, 3-2
Flip-flops, 9-18
 D-type, 9-17
Format
 of programmer load file, 3-27
fuse, 7-14
Fusemap
 -c, 3-28
 -d, 3-27
 -h, 3-29
 -i, 3-27
 -k, 3-28
 -o, 3-27
 checksum, 3-28
 DOCUMENT, 3-39
 format of programmer load file, 3-27
 operation, 3-24
 overview, 2-7
 unused OR fuses, 3-28
Fuses statement, 7-28
Fuses, unused, 3-28

G

Goto statement, 7-24
example, 7-24

H

High impedance values
SIMULATE, 3-35

I

Identifiers, 6-2
 choosing, 6-4
 in state machines, 9-14
See also keywords
 multiple assignments to, 6-14
If blank, 8-5
If defined, 8-5
If identical, 8-6
If not blank, 8-6
If not defined, 8-7
If not identical, 8-7
If-then-else, 7-22
 chained, 7-23

Include library, 3-7
 path to, 3-14
Indefinite repeat, 8-9
Indefinite repeat, character, 8-10
Input files
 DOCUMENT, 3-38
 FUSEMAP, 3-27
 PARSE, 3-14
 REDUCE, 3-22
 SIMULATE, 3-32
 TRANSFOR, 3-19
Intel, supported devices, C-2
Intermediate files, 2-5
ISTRYPE declaration, 7-12
 controlling macro cells with, 11-4
 example, 7-12

J

JEDEC files, 2-4
 path to, 3-29
JEDEC Notations, B-4
JEDEC Standard Number 3-A, B-1

K

Keywords
 declaration, 7-6
 table of, 6-3

L

Language processor
 DOCUMENT, operation, 3-36
 Document, overview, 2-11
 FUSEMAP, operation, 3-24
 Fusemap, overview, 2-7
 operation of TRANSFOR, 3-18
 overview, 2-5
 PARSE, operation, 3-12
 Parse, overview, 2-6
 REDUCE, operation, 3-20
 Reduce, overview, 2-7
 SIMULATE, operation, 3-30
 Simulate, overview, 2-8
 Transfor, overview, 2-7
Lattice, architecture modes, C-3

Libraries
 device, 3-6
 include, 3-7
 path to device files, 3-15
 path to include files, 3-14
Library manager
 See abellib
Library statement, 7-15
List file, 2-4
 -l, 3-14
 example, 3-15
 syntax errors, 3-15
Logic operators, 6-8

M

Macro cells
 controlling feedback point, 11-6
 controlling polarity of, 11-4
 controlling with ISTYPE, 11-4
Macro declaration, 7-10
Macros
 blocks in, 6-21
 debugging, 7-10
 dummy arguments in, 6-22, 7-10
 example, 7-11, 10-36
 expansion of, 3-14, 7-10
 to create test vectors, 4-12
Mealy, 9-7
Mealy vs. Moore, 9-7
Menus
 DOCUMENT, 3-36
 FUSEMAP, 3-24
 overview, 3-7
 PARSE, 3-12
 REDUCE, 3-20
 Run Series, 3-9
 SIMULATE, 3-30
 TRANSFOR, 3-18
Model 29, downloading to, 12-1, 12-3
Module statement, 5-6, 7-3
Moore, 9-7
Multiplexer/Adder/Comparator, example, 10-32

N

National, architecture modes, C-3
neg, 7-14
Negative polarity, 7-14
Node declarations, 5-6, 7-8
example, 7-8
Node extension notation, Signetics, C-4
Nodes, C-5
controlling, 4-1
dot extensions, 4-2
numbers, 4-1
selectable, 7-14
Notations
JEDEC, B-4
Numbers, 6-5
changing base, 8-11

O

One-bit changes, 9-18
Operation
menus, 3-7
Operators, 6-8
arithmetic, 6-8
assignment, 6-11
logical, 6-8
priority of, 6-11
relational, 6-9
Options
-a, PARSE, 3-14
-b, SIMULATE, 3-33
-c, FUSEMAP, 3-28
-d, FUSEMAP, 3-27
-e, PARSE, 3-14
-f, DOCUMENT, 3-39
-h, PARSE, 3-14
-i, DOCUMENT, 3-38
-i, FUSEMAP, 3-27
-i, PARSE, 3-14
-i, REDUCE, 3-22
-i, SIMULATE, 3-32
-i, TRANSFOR, 3-19
-j, FUSEMAP, 3-29
-k, FUSEMAP, 3-28
-l, PARSE, 3-14
-n, SIMULATE, 3-32
-o, DOCUMENT, 3-38
-o, FUSEMAP, 3-27
-o, PARSE, 3-14

- o, REDUCE, 3-22
- o, SIMULATE, 3-32
- o, TRANSFOR, 3-19
- p, PARSE, 3-14
- q, DOCUMENT, 3-38
- r, REDUCE, 3-22
- s, DOCUMENT, 3-39
- t, SIMULATE, 3-34
- u, SIMULATE, 3-35
- w, SIMULATE, 3-33
- x, SIMULATE, 3-35
- y, PARSE, 3-15
- z, SIMULATE, 3-35
- table of, 3-2
- OR equations
 - exclusive, 9-1
- Original equations, 3-38
- Output enable
 - configurable, 11-2
 - pin controlled, 11-1
 - term controlled, 11-1
- Output files, 2-4
 - DOCUMENT, 3-38
 - FUSEMAP, 3-27
 - PARSE, 3-14
 - REDUCE, 3-22
 - SIMULATE, 3-32
 - TRANSFOR, 3-19
- Outputs
 - buffered, 4-38
 - programmable polarity, 9-17

P

- P14H4, example, 10-6
- P14L4, example, 10-3
- P16L8, example, 10-9, 10-35
- P16R4, example, 10-12
- P16R6, example, 10-38
- P20R8, example, 10-19
- P22V10, example, 10-32
- PALASM
 - converting to ABEL, 3-43
 - create output file in, 3-38
- Parameters
 - See Options
- Parse
 - a, 3-14
 - e, 3-14

-h, 3-14
-i, 3-14
-l, 3-14
-o, 3-14
-p, 3-14
-y, 3-15
operation, 3-12
overview, 2-6
pin, 7-14
Pin declarations, 5-6, 7-7
example, 7-7
Pins
assigning, 9-6
controlling preset/reset, 4-27
PLCC package diagram, 3-39
PLDs, device-specific information, C-1
Polarity
negative, 7-14
positive, 7-14
programmable outputs, 9-17
Polarity control, 9-2
in state machines, 9-3
pos, 7-14
Positive polarity, 7-14
Powerup, 9-17
Powerup state, 4-27
SIMULATE, 3-35
Preload register, B-15
Preload registers, 4-18
supervoltage, 4-21
Preset
controlling with pins, 4-27
controlling with product terms, 4-26
Preset registers, 4-18
special considerations, 4-19
Presto by pin, 3-23
Problems, A-1
with assigning pins, 9-6
Processing source file, 2-5
Product terms, 3-39
controlling preset/reset, 4-26
reducing, 9-18
selectable sharing of, 11-9
Program Series
See Run Series
Programmable polarity outputs, 9-17
Programmer load files, 2-4
format of, 3-27
path to, 3-29
Programmer/tester options, B-17

Programmers
 downloading to, 12-1
 Programming fields, B-3
 PROMs, C-1
 Propagation delays
 See feedback
 Protocol, transmission, B-5
 PSF menus
 See Menus

R

RA5P8, example, 10-22
 Reduce
 -i, 3-22
 -o, 3-22
 -r, 3-22
 operation, 3-20
 overview, 2-7
 solving timing problems with, 4-4
 Reduced equations, 3-38
 Reduction level, 3-22
 Espresso, 3-23
 Espresso by pin, 3-23
 Presto, 3-23
 Presto by pin, 3-23
 simple, 3-22
 Redundancy
 intentional, 4-4
 reg, 7-14
 reg_d, 7-14
 reg_jk, 7-14
 reg_rs, 7-14
 reg_t, 7-14
 Register
 preset and preload, 4-18
 Register preload, B-15
 Registered signal, 7-14
 Registers
 bypassing, 11-8
 controlling type of, 11-9
 powerup states, 9-17
 preload, 9-6
 selecting, 11-8
 type, 7-14
 Relational operators, 6-9
 Reset, 4-18
 controlling with pins, 4-27
 controlling with product terms, 4-26

S

- Set operations, 6-16, E-1
- Sets, 6-15, 6-17
 - errors, 6-20
 - example, 6-15
 - expressing state registers as, 9-14
 - operators used with, 6-16
 - restrictions on, 6-20
- SGS Thomson, architecture modes, C-3
- Sharing product terms, 11-9
- Signals
 - combinational, 7-14
 - multiple assignments, 10-16
 - See also pin or node
 - registered, 7-14
 - See also sets
- Signetics, support information, C-4
- Simulate
 - b, 3-33
 - i, 3-32
 - n, 3-32
 - o, 3-32
 - t, 3-34
 - u, 3-35
 - w, 3-33
 - x, 3-35
 - z, 3-35
 - advanced use of breakpoints, 4-30
 - advanced use of trace levels, 4-30
 - breakpoints, 3-33
 - device type, 3-32
 - don't care values, 3-35
 - high impedance values, 3-35
 - operation, 3-30
 - overview, 2-8
 - powerup state, 3-35
 - test vectors, 4-10
 - trace levels, 3-34
 - watch points, 3-33
- Simulation
 - debugging state machines, 2-13
 - register preloads, 9-6
- Simulation output file, 2-5
- Source file
 - arguments, 6-21
 - basic structure, 7-1
 - blocks, 6-21
 - case statement, 7-23
 - comments, 6-5
 - constant declarations, 5-7, 7-9

declarations, 7-5
device declaration statement, 7-6
device declarations, 5-6
elements of, 5-2
equations, 6-13
equations statement, 5-8, 7-16
examining, 5-4
expressions, 6-11
flag statement, 5-6, 7-4
fuses statement, 7-28
goto statement, 7-24
identifiers, 6-2 – 6-3
if-then-else, 7-22
ISTYPE declaration, 7-12
keywords, 6-3
library statement, 7-15
macro declaration, 7-10
module statement, 5-6, 7-3
node declarations, 5-6, 7-8
numbers, 6-5
operators, 6-8
overview, 2-2
pin declarations, 5-6, 7-7
processing, 5-9
sets, 6-15
special constants, 6-7
state descriptions, 7-20
strings, 6-4
test vectors statement, 5-8, 7-29
title statement, 5-6, 7-5
truth table statement, 7-17
with-endwith, 7-24
XOR Factor, 7-25
State descriptions, 7-20
example, 7-21, 10-38
State machines
debugging, 4-10, 9-13
design considerations, 9-6
identifiers in, 9-14
Mealy vs. Moore, 9-7
powerup register states, 9-17
reducing product terms, 9-18
state registers in, 9-14
using state register outputs, 9-19
State registers, 9-19
in state machines, 9-14
State_Diagram, 7-20
States, identifying, 9-19
Strings, 6-4

Supervoltage preload, 4-21
Symbols, 3-39
Syntax, 6-1
 errors, 3-15
System requirements, 1-2

T

Temporary files
 deleting, 3-46
Term controlled output enable, 11-1
Term-100, F-1
Terminal emulator, F-1
Test vectors
 and simulation, 4-10
 creating, 4-12
 creating, example, 10-36
 multiple tables, 4-11, 9-13
Test vectors statement, 5-8, 7-29
Testing fields, B-3
Timing problems, 4-4
Title statement, 5-6, 7-5
TOABEL, 3-43
Trace level 0, 4-31
Trace level 1, 4-32
Trace level 2, 4-33
Trace level 3, 4-33
Trace level 4, 4-36
Trace level 5, 4-36
Trace levels
 advanced use of, 4-30
 breakpoints, 3-33
 SIMULATE, 3-34
 two, 4-37
 watch points, 3-33
Tranformed equations, 3-38
Transfor
 -i, 3-19
 -o, 3-19
 operation, 3-18
 overview, 2-7
Transition conditions, 9-18
 unsatisfied, 9-17
Transmission protocol, B-5
Trouble, A-1

Truth table statement, 7-17

Truth tables

example, 10-35

format of, 7-18

Turbo bits, C-7

U

UniSite, downloading to, 12-3

Unused OR fuses, 3-28

Use In-File Setting, 3-7

User electronic signature word, C-7

V

VT-100

terminal emulator, F-1

VTI, architecture modes, C-3

W

Watch points

SIMULATE, 3-33

With-endwith, 7-24

X

XOR PALS, 11-15

XOR_Factor, 7-25

example, 7-25

