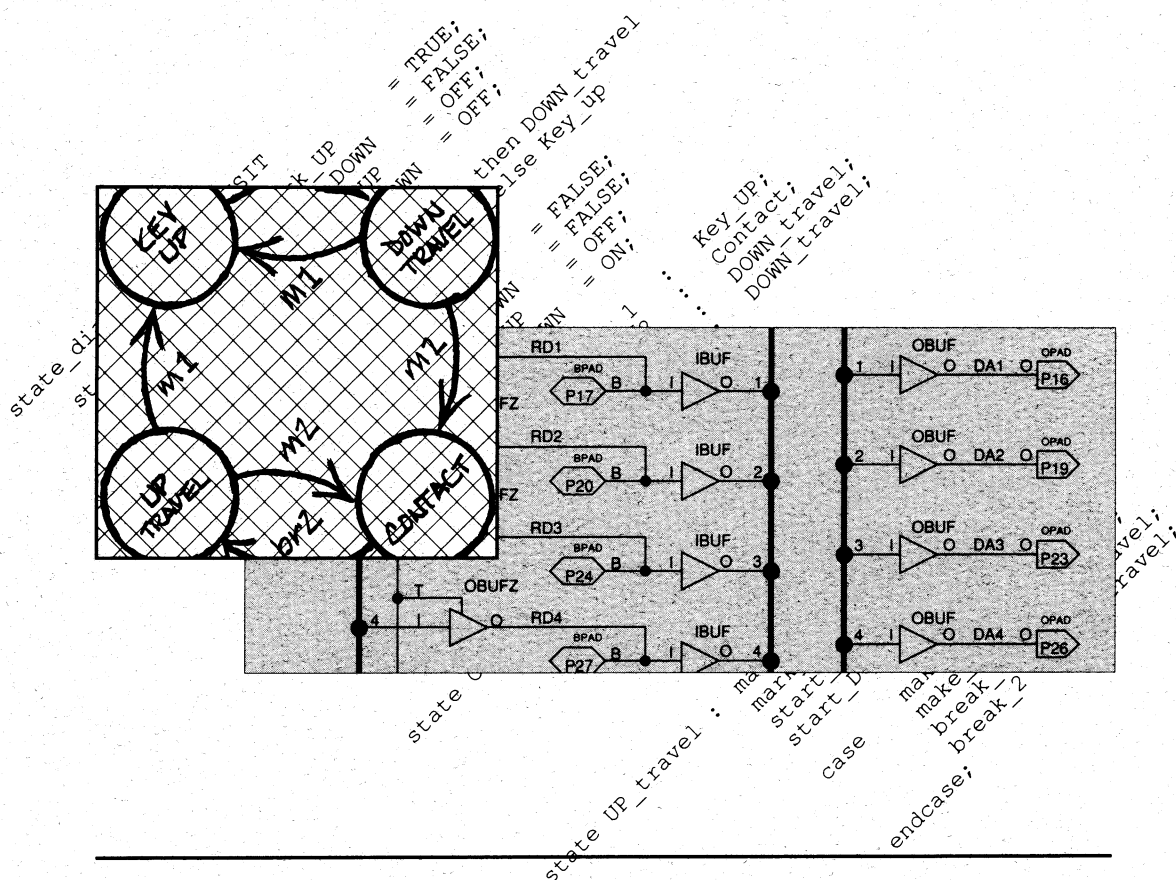


DESIGN SOFTWARE



USER MANUAL





ABELTM

Design Software

User Manual

September 1990

981-0252-001

Data I/O has made every attempt to ensure that the information in this document is accurate and complete. However, Data I/O assumes no liability for errors, or for any damages that result from use of this document or the equipment which it accompanies.

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 is a registered trademark of Data I/O Corporation.
2900, 29B, ABEL, ABEL-HDL, ABEL-PLA, PLDgrade, SmartPart and UniSite are trademarks of Data I/O Corporation.

Altera is a trademark of Altera Corporation

Intel is a trademark of Intel Corporation

Motorola is a trademark of Motorola, Inc.

PALASM is a trademark of Advanced Micro Devices, Inc.

Signetics SNAP is a trademark of Signetics Company

Xilinx is a trademark of Xilinx, Inc.

© 1990 Data I/O Corporation
All rights reserved

Table of Contents

Preface

Technical Questions?	xvii
Customer Support BBS	xviii
End User Registration and Address Change	xviii
Warranty Information	xviii
Description	xviii
Warranty Service	xviii
Data I/O Update Service	xix
Typographic Conventions	xx

Introducing ABEL

1. Understanding ABEL

ABEL Features	1-1
The Open ABEL Concept	1-2
Structure of ABEL	1-3
Source File	1-4
Processing Modules	1-5
Output files	1-7

2. Tutorials

Introduction to a Source File	2-1
Creating a Simple Design	2-3
The Source File	2-3
Processing a Design	2-7
Compiling the Design	2-8

Simulating the Design	2-8
Optimizing the Design	2-8
Generating a Programmer Load File	2-9
Simulating the Device	2-9
Automatic Design Updating	2-10
Creating an Architecture-independent Design	2-11
Declarations	2-12
Equations	2-12
Summary	2-13
Creating a State Description	2-14
Basic Structure	2-15
Considerations	2-15
Creating a Truth Table	2-16
Considerations	2-17

Creating a Design

3. ABEL-HDL Language Structure

Summary	3-1
Introduction to ABEL-HDL	3-2
Basic Syntax	3-3
Valid ASCII Characters	3-3
Identifiers	3-3
Constants	3-5
Blocks	3-6
Comments	3-6
Numbers	3-7
Strings	3-8
Operators, Expressions, and Equations	3-8
Sets	3-13
Arguments and Argument Substitution	3-18
Basic Structure	3-19
Header	3-22
Module Statement	3-22
Options	3-22
Title	3-22
Declarations	3-22
Declarations Keyword	3-22
Device Declaration	3-23
Signal Declarations	3-23
Constant Declarations	3-24

Macro Declarations	3-24
Library Declaration	3-24
Logic Description	3-25
Dot Extensions	3-25
Equations	3-26
Truth Tables	3-26
State Descriptions	3-26
Fuse Declarations	3-27
XOR Factors	3-27
Test Vectors Section	3-27
Test Vectors	3-27
Trace Statement	3-27
End Statement	3-27
Other Elements	3-28
Directives	3-28

4. Language Reference

.ext — Dot Extensions	4-2
= — Constant Declarations	4-8
'attr' — Signal Attributes	4-10
@directive — Directives	4-13
Case	4-23
Declarations	4-25
Device	4-26
End	4-27
Equations	4-28
Fuses	4-29
Goto	4-30
If-Then-Else	4-31
Istype — Attribute Declarations	4-33
Library	4-34
Macro	4-35
Module	4-38
Node	4-39
Options	4-40
Pin	4-41
Property	4-42
State_diagram	4-43
Test_vectors	4-47
Title	4-49
Trace	4-50
Truth_table	4-51

When-Then-Else	4-54
With-endwith	4-55
XOR_Factors	4-56

5. Design Considerations

Architecture-independent Language Features	5-1
Device Independence Vs. Architecture Independence	5-2
Signal Attributes	5-2
Signal Dot Extensions	5-2
Pin-to-pin Vs. Detailed Descriptions for Registered Designs	5-3
Using := for Pin-to-pin Designs	5-3
Detailed Circuit Descriptions	5-4
When to Use Detailed Descriptions	5-7
Using := for Alternative Flip-flop Types	5-8
Using Active-low Declarations	5-8
Flip-flop Equations	5-9
Feedback Considerations — Using Dot Extensions	5-10
Dot Extensions and Architecture-Independence	5-11
Dot Extensions and Detail Design Descriptions	5-13
@DCSET Considerations and Precautions	5-14
Polarity Control	5-16
Automatic Polarity Selection	5-17
Polarity Control in ABEL	5-17
Exclusive OR Equations	5-18
Optimizing XOR Devices	5-18
Using XOR Operators in Equations	5-18
Using Implied XORs in Equations	5-18
Using XORs for Flip-flop Emulation	5-19
State Machines	5-20
Mealy and Moore	5-21
Use Identifiers Rather Than Numbers for States	5-26
Powerup Register States	5-27
Unsatisfied Transition Conditions	5-28
Precautions for Using @DCSET	5-29
Number Adjacent States for One-Bit Change	5-33
Use State Register Outputs to Identify States	5-33
Using Complement Arrays	5-34

6. Source File Examples

Equations	6-2
6809 Memory Address Decoder	6-2

12 to 4 Multiplexer	6-4
1 to 8 Demultiplexer	6-7
4-Bit Counter/- Multiplexer	6-9
8-Bit Barrel Shifter	6-15
Bidirectional Three-State Buffer	6-18
4-Bit Comparator	6-20
Truth Table Examples	6-23
7-Segment Display Decoder	6-23
State Description Examples	6-26
Three-State Sequencer	6-26
Blackjack Machine	6-29

Compiling a Design

7. Using ABEL Processing Modules

Prerequisites	7-1
Program Summary	7-3
Using Help	7-4
Help in the ABEL Design Environment	7-4
Help for the Command Line	7-5
Using the ABEL Design Environment	7-6
Starting the ABEL Design Environment	7-7
Basic Menu Operation	7-7
Automatic Design Updating	7-8
File Management	7-9
System Commands	7-9
Setting Menu Defaults	7-10
Options	7-10
Optional Programs	7-10
Correcting a Source File	7-11
Specifying a Source File from the Command Line	7-12
Compiling a Source File	7-12
Setting Compile Options	7-13
Simulating a Source File	7-14
Simulating Equations	7-15
Simulating Design and Device	7-15
Setting Simulation Trace Options	7-16
Optimizing a Source File	7-20
Setting Optimization Options	7-20
Selecting a Device	7-22
Creating a Fusemap from a Design	7-23

Mapping from a PLA File	7-23
Setting Map Options	7-24
FPGA Translation	7-26
Fault Grading	7-27
PLDgrade	7-27
Viewing Processing Results	7-27
Downloading to Programmers	7-29
Cable Configuration for PCs	7-29
Using the Command Line	7-29
Response File	7-30
Turning Off Options	7-30
Command Line Summary	7-31
Batch File Operation	7-32
Utilities	7-33
Partitioning and Merging PLA Files	7-33
Creating a FutureNet Schematic	7-34
Translating to Other Formats	7-34
Creating Signetics Tables	7-34
Library Management	7-34
Cleaning Up Extra Files	7-36
Finding a Device	7-36

8. Using Advanced Features

Timing Problems and PLAOpt	8-1
Advanced Simulation	8-3
How ABEL Simulates Your Design	8-3
Trace Levels and Break Points	8-5
Simulation and Designs with Buffered Outputs	8-13
Simulation and Unspecified Inputs	8-14
Simulation for Designs with Feedback	8-14
Register Preloads in the Simulator	8-17
Test Vectors and Simulation	8-17
Debugging State Machines	8-17
Multiple Test Vector Sections	8-18
Automatic Signal Selection	8-19
Using Macros and Directives to Create Test Vectors	8-19
Don't Cares in Simulation	8-23
Preset and Preload Registers	8-24
Devices with Clock Inputs	8-33

Program Messages

9. Program Messages

Message Types	9-1
AHDL2PLA Messages	9-2
PLAOpt Messages	9-10
Fuseasm Messages	9-13
PLASim and JEDSim Messages	9-18

Backward Compatibility

10. Backward Compatibility

Syntax Changes	9-1
Multiple Devices in a Module	9-1
Automatic Node Names	9-2
Assignment Operators	9-2
State Machines	9-2
Attribute Changes	9-2
Dot Extension Changes	9-3
New Language Features	9-3
Conceptual Changes	9-4
Architecture Independent Language Features	9-4
PIN and NODE Declarations	9-4
Signal Attributes	9-4
Program Changes	9-5
Option Changes	9-5
Simultaneous Operation with Earlier Versions	9-7
Changes from Versions Prior to ABEL 3.2	9-7

Appendixes

A. ASCII Table

B. JEDEC Standard Number 3A

C. Operator Rules

Glossary

Index

Reference Card

List of Figures

Figure 1-1. The Open ABEL Concept	1-2
Figure 1-2. Program Flow	1-4
Figure 2-1. Source File Structure	2-2
Figure 2-2. Source File: m6809a.abl	2-3
Figure 2-3. Program Flow	2-7
Figure 2-4. JEDEC Simulation Results: m6809a.sim	2-10
Figure 2-5. Source File: par_det.abl	2-11
Figure 2-6. Simple State Diagram Description	2-14
Figure 2-7. Simple Truth Table	2-16
Figure 2-8. Truth Table from State Machine	2-17
Figure 3-1. Source File Structure	3-20
Figure 4-1. Pin-to-pin Dot Extensions in an Inverted Output Architecture	4-5
Figure 4-2. Pin-to-pin Dot Extensions in a Non-inverted Output Architecture	4-5
Figure 4-3. Detailed Dot Extensions for a D-type Flip-flop Architecture	4-5
Figure 4-4. Detailed Dot Extensions for a T-type Flip-flop Architecture	4-5
Figure 4-5. Detailed Dot Extensions for an RS-type Flip-flop Architecture	4-6
Figure 4-6. Detailed Dot Extensions for a JK-type Flip-flop Architecture	4-6
Figure 4-7. Detailed Dot Extensions for a Latch with Active High Latch Enable	4-6
Figure 4-8. Detailed Dot Extensions for a Latch with Active Low Latch Enable	4-7
Figure 4-9. Differences Between MACRO and Declared Equations	4-36
Figure 4-10. Pictorial State Diagram	4-45
Figure 4-11. Architecture-independent State Machine	4-46

Figure 4-12. Source File Using XOR_Factor	4-57
Figure 5-1. Detail Macrocell	5-4
Figure 5-2. Pin-to-pin Macrocell	5-5
Figure 5-3. Dot Extensions and Architecture- Independence: Circuit 1	5-11
Figure 5-4. Pin to Pin One-bit Synchronous Circuit	5-11
Figure 5-5. Dot Extensions and Architecture- Independence: Circuit 2	5-12
Figure 5-6. Dot Extensions and Architecture- Independence: Circuit 3	5-12
Figure 5-7. Detail One-bit Synchronous Circuit with Inverted Qout	5-13
Figure 5-8. Detail One-bit Synchronous Circuit with Non-inverted Qout	5-14
Figure 5-9. Source File Showing @DCSET Optimization	5-15
Figure 5-10. JK Flip-flop Emulation Using T Flip-flop	5-19
Figure 5-11. T Flip-flop Emulation Using D Flip-flop	5-20
Figure 5-12. JK Flip-flop Emulation, D Flip-flop with XOR	5-20
Figure 5-13. Flow Diagram for a Mealy State Machine	5-21
Figure 5-14. Flow Diagram for a Moore State Machine	5-22
Figure 5-15. Mealy State Diagram of a Sequence Detector	5-22
Figure 5-16. Moore State Diagram of a Sequence Detector	5-23
Figure 5-17. Mealy Source File	5-24
Figure 5-18. Moore Source File	5-25
Figure 5-19. Using Identifiers for States	5-27
Figure 5-20. D-type Register with False Inputs	5-28
Figure 5-21. @DCSET-Incompatible State Machine Description	5-29
Figure 5-22. @DCSET-Compatible State Machine Description	5-31
Figure 5-23. Transition Equations for a Decade Counter	5-35
Figure 5-24. Abbreviated F105 Schematic	5-36
Figure 6-1. Block Diagram: 6809 Memory Address Decoder	6-2
Figure 6-2. Simplified Block Diagram: 6809 Memory Address Decoder	6-3
Figure 6-3. 6809 Memory Address Decoder Source File	6-3
Figure 6-4. Block Diagram: 12 to 4 Multiplexer	6-4
Figure 6-5. Simplified Block Diagram: 12 to 4 Multiplexer	6-5

Figure 6-6. Source File: 12 to 4 Multiplexer	6-6
Figure 6-7. Block Diagram: 1 to 8 Demultiplexer	6-7
Figure 6-8. Simplified Block Diagram: Demultiplexer	6-7
Figure 6-9. Source File: 1 to 8 Demultiplexer	6-8
Figure 6-10. Block Diagram: 4-Bit Counter With 2 Input Multiplexer	6-9
Figure 6-11. Simplified Block Diagram: 4-Bit Counter With 2 Input Multiplexer	6-10
Figure 6-12. Source file: 4-bit Counter with 2 Input Mux	6-11
Figure 6-13. Multiple Equations Sections, 4-Bit Counter	6-13
Figure 6-14. Block Diagram: 8-Bit Barrel Shifter	6-15
Figure 6-15. Simplified Block Diagram: 8-Bit Barrel Shifter	6-16
Figure 6-16. Source File: 8-Bit Barrel Shifter	6-17
Figure 6-17. Block Diagram: Bidirectional Three-State Buffer	6-18
Figure 6-18. Simplified Block Diagram: Bidirectional Three-State Buffer	6-18
Figure 6-19. Source File: Bidirectional Three-State Buffer	6-19
Figure 6-20. Block Diagram: 4-Bit Comparator	6-20
Figure 6-21. Simplified Block Diagram: 4-bit Comparator	6-20
Figure 6-22. Source File: 4-Bit Comparator	6-22
Figure 6-23. Block Diagram: Seven-Segment Display Decoder	6-23
Figure 6-24. Simplified Block Diagram: Seven-Segment Display Decoder	6-24
Figure 6-25. Source file: 4-bit Counter with 2 Input Mux	6-25
Figure 6-26. State Diagram: Three-state Sequencer	6-26
Figure 6-27. Source File: Three-state Sequencer	6-28
Figure 6-28. Schematic of a Blackjack Machine Implemented in Three PLDs	6-30
Figure 6-29. Source File: Multiplexer/Adder/Comparator	6-33
Figure 6-30. Source file: 4-bit Counter with 2 Input Mux	6-35
Figure 6-31. Pictorial State Diagram: Blackjack Machine	6-37
Figure 6-32. Source File: State Machine (Controller)	6-39
Figure 7-1. Help Menu	7-4
Figure 7-2. ABEL Design Environment Menu and Program Flow	7-6
Figure 7-3. Main Menu	7-7
Figure 7-4. File Menu	7-9

Figure 7-5. Defaults Menu	7-10
Figure 7-6. Edit Menu	7-11
Figure 7-7. Compile Menu	7-12
Figure 7-8. Compile Options Dialog Box	7-13
Figure 7-9. Simulate Trace Options Dialog Box	7-16
Figure 7-10. Optimize Menu	7-20
Figure 7-11. Optimize Options Dialog Box	7-21
Figure 7-12. PartMap Menu	7-23
Figure 7-13. PartMap Options Dialog Box	7-24
Figure 7-14. View Menu	7-27
Figure 7-15. PC to Programmer Cable Configuration	7-29
Figure 8-1. Circuit Using an Input and Its Complement	8-2
Figure 8-2. Timing Diagram for $F = B \& !C \# !A \& C$	8-2
Figure 8-3. Simulation Processing Flow Diagram	8-5
Figure 8-4. Regfb.abl Source File	8-6
Figure 8-5. No Trace Simulation Output	8-7
Figure 8-6. No Trace Simulation Output Showing Error	8-7
Figure 8-7. Brief Trace Simulation Output	8-7
Figure 8-8. Clock Trace Simulation Output	8-8
Figure 8-9. Detail Table Format Simulation Output	8-9
Figure 8-10. Pins Format Simulation Output	8-10
Figure 8-11. Macro Format Simulation Output	8-11
Figure 8-12. Wave Format Simulation Output	8-13
Figure 8-13. Synchronous Feedback Circuit	8-14
Figure 8-14. Asynchronous Feedback Circuit	8-15
Figure 8-15. Source File: Asynchronous Feedback Circuit	8-16
Figure 8-16. Brief Trace Simulation Output: Asynchronous Feedback Circuit	8-16
Figure 8-17. Detail Trace Simulation Output: Asynchronous Feedback Circuit	8-16
Figure 8-18. Source File with Multiple Test Vector Sections	8-18
Figure 8-19. Simulation Results Showing Automatic Signal Selection	8-19
Figure 8-20. Test Vectors Described With a Macro and @IF and @IRP Directives	8-20

Figure 8-20. Test Vectors Described With a Macro and @IF and @IRP Directives	8-20
Figure 8-21. Simulation Results Showing Test Vectors Created with a Macro and @IF/@IRP Directives	8-21
Figure 8-22. Test Vectors Described with a Macro, @CONST and @REPEAT Directives	8-22
Figure 8-23. Simulation Results Showing Vectors Created with a Macro, @CONST and @REPEAT	8-22
Figure 8-24. Assignment of Don't Care Value (.x.) to Design Outputs	8-24
Figure 8-25. PLASim Results with Outputs Specified as Don't Care	8-24
Figure 8-26. Test Vectors for Special Preset Conditions	8-26
Figure 8-27. Timing Diagram Showing Test Vector Action	8-27
Figure 8-28. Internal Register of the F159	8-28
Figure 8-29. Invoking the TTL Preload Function	8-28
Figure 8-30. Defining Illegal States and Test Vectors for Illegal States	8-30
Figure 8-31. Using Test Vectors to Preload a State Machine	8-31
Figure 8-32. Controlling Reset/Preset by Product Term	8-32
Figure B-1. Example of a PLD Data File	B-2
Figure B-2. Computing the Transmission Checksum	B-4
Figure B-3. 8-bit Words Formed from Fuse States for Checksum	B-8
Figure B-4. Computing the Fuse Checksum	B-9

List of Tables

Table 3-1. Special Constants	3-5
Table 3-2. Number Representation in Different Bases	3-7
Table 3-3. Logical Operators	3-9
Table 3-4. Arithmetic Operators	3-9
Table 3-5. Relational Operators	3-9
Table 3-6. Assignment Operators	3-11
Table 3-7. Operator Priority	3-11
Table 3-8. Valid Set Operations	3-15
Table 3-9. Attributes	3-24
Table 3-10. Dot Extensions	3-25
Table 4-1. Dot Extensions	4-3

Table 4-2. Dot Extensions for Architectures	4-4
Table 4-3. Attributes	4-10
Table 4-4. Alternate Operator Set	4-14
Table 7-1. Program Summary	7-3
Table 7-2. AHDL2PLA Reduction Rules	7-13
Table 7-3. Data Translation Format Codes and File Extensions . . .	7-26
Table 7-4. Command Line Option Summary	7-31
Table B-1. Field Identifiers and Descriptions	B-2
Table B-2. Field Identifiers	B-5
Table B-3. Test Conditions	B-10

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 AM and 5:00 PM 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
- Product serial number (if available)
- Detailed description of the problem you are experiencing
- Error messages (if any)
- Device manufacturer and part number (if device-related)
- Product manual

USA and Canada

800 247-5700
Fax: 206 882-1043

International

Data I/O Japan 03 432-6991

Data I/O Europe +31 (0)20 6622866

Data I/O
Intercontinental 206 881-6444

Written Inquiries

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

Customer Support BBS

You can also call Data I/O's Customer Support Bulletin Board System (BBS). From the Customer Support BBS you can obtain a wide range of information on Data I/O products, including current product information, new revision information, known bugs (and work-arounds), helpful application notes, and other miscellaneous information. In addition, the BBS has a collection of DOS utilities you can download.

The Customer Support BBS also has a message facility which allows you to leave messages for Customer Support Personnel. For example, you could request support for a specific device, or suggest how we can improve our products. Or you could leave a message telling us what you think of Data I/O product(s).

To learn more about the Data I/O Customer Support BBS, call it at 206 882-3211. The protocol is 1200/2400/9600 (Courier HST) baud, 8 data bits, 1 stop bit, and no parity. Online help files are available throughout the BBS to help you learn more about the BBS.

End User Registration and Address Change

If your address has changed since you filled out your Warranty Registration Card, please notify your nearest Data I/O representative. This ensures that you receive information about product enhancements. Be sure to include the product serial number, if available.

Warranty Information

Description

Data I/O Corporation warrants its products against defects in materials and workmanship for a period of ninety (90) days for software and one (1) year for hardware unless specified otherwise. The warranty begins when the product is shipped.

Warranty Service

Data I/O maintains customer support centers throughout the world, each staffed with factory-trained technicians to provide prompt, quality service.

For warranty service, contact your nearest Data I/O Customer Support Center. If you do not have a name or phone number for your nearest office, refer to the list of Data I/O offices below.

United States
Data I/O Corporation
Customer Resource Center
 10525 Willows Road N.E.
 P.O. Box 97046
 Redmond, WA 98073-9746
 Telephone: 800 247-5700
 Fax: 206 882-1043
 Telex: 152167

Data I/O San Jose
 1701 Fox Drive
 San Jose, CA 95131
 Telephone: 408 437-9600
 Fax: 408 437-1218

Data I/O New Hampshire
 20 Cotton Road
 Nashua, NH 03063
 Telephone: 603 889-8511
 800 858-5803 (NJ & NY only)
 Fax: 603 880-0697

Data I/O Intercontinental
 10525 Willows Road N.E.
 P.O. Box 97046
 Redmond, WA USA 98073-9746
 Telephone: 206 881-6444
 Fax: 206 882-1043
 Telex: 4740166

Data I/O Japan
 Sumitomoseimei
 Higashishinbashi Bldg. 8F
 2-1-7, Higashi-Shinbashi
 Minato-Ku, Tokyo 105, Japan
 Telephone: 03 432-6991
 Fax: 03 432-6094 (Sales)
 03 432-6093 (Other)
 Telex: 2522685 DATAIO J

Data I/O Canada
 6725 Airport Road, Suite 302
 Mississauga, Ontario
 L4V 1V2 Canada
 Telephone: 416 678-0761
 Fax: 416 678-7306

Data I/O-Instrumatic Electronic
Systems Vertriebs GmbH
 Lochhammer Schlag 5a
 D-8032 Gräfelfing
 West Germany
 Telephone: 089 858580
 Fax: 089 8585810

Data I/O Europe
 World Trade Center
 Strawinskylaan 633
 1077 XX Amsterdam,
 The Netherlands
 Telephone: +31 (0)20 6622866
 Fax: +31 (0)20 6624427
 Telex: 16616 DATIO NL

Data I/O Update Service

Data I/O Customer Support is committed to providing you with the support programs you need to keep your Data I/O products in optimum operating condition and up to date with the latest, state-of-the-art capabilities.

For more information, or to order a Service Agreement or Update Agreement, call your nearest Data I/O representative. The name and phone number of your nearest Data I/O office can be found in the Warranty Service section of this chapter.

Typographic Conventions

Throughout this manual different typographic conventions represent different cases of input and output.

Keyboard Keys

Keyboard keys may be shown in boxes (for example, **Q**) or as bolded text.

The **Enter** key (or on some keyboards, the **Return** key) is shown as this symbol **↵**.

Key Combinations

Key combinations, such as **Control-Z**, are shown as two key boxes separated by a dash; for example, **Ctrl** – **Z**.

A key combination like **Esc** **Ctrl** – **T** means to press and release **Esc**, then press **Ctrl** and **T** at the same time.

Variable Input

Variable input is italicized and should be replaced with the requested information. For example, enter *copy filename.hex* means type in *copy* just as you see it and replace *filename.hex* with the name of your file.

Optional Input

Optional items of a command are shown in brackets; for example

[option1] [option2]...[optionn]

Items separated by a vertical bar (for example, *OR|OR|...*) are mutually exclusive; that is, only one of the options listed can be specified.

Displayed Messages

Text that appears on the screen will be displayed in a typewriter-like typeface; for example,

You will see this text displayed on the screen.

1 *Understanding ABEL*

This chapter explains the process used by ABEL software to create a programmer load file from your design. First you must describe your design in a source file, then the ABEL software processes the source file to produce a programmer load file, and simulate and document your design.

ABEL Features

The **ABEL Design Environment** provides an integrated approach to processing your design.

ABEL Hardware Description Language (ABEL-HDL) allows you to

- describe your design with equations, truth tables, state diagrams, or any combination of the three
- design, optimize and simulate your design without specifying a device or assigning pins

Standard Format Output Files

The output files produced by ABEL are in standard formats to interface with other tools:

- **ABEL-PLA** output files allow designs to be created, optimized and simulated in ABEL, then moved directly to other software tools, such as LCA and FPGA design tools (see "Open ABEL" below).
- **JEDEC** format output files download directly to programmers.
- **PROM** format output files allow programming of PROMs.

Automatic Device Selection and Fitting (SmartPart)

The optional SmartPart™ package for ABEL offers several options for automatic device selection by user-selected criteria, and device fitting and pin assignment. The standard ABEL-PLA format files can also be used by custom fitters from other vendors.

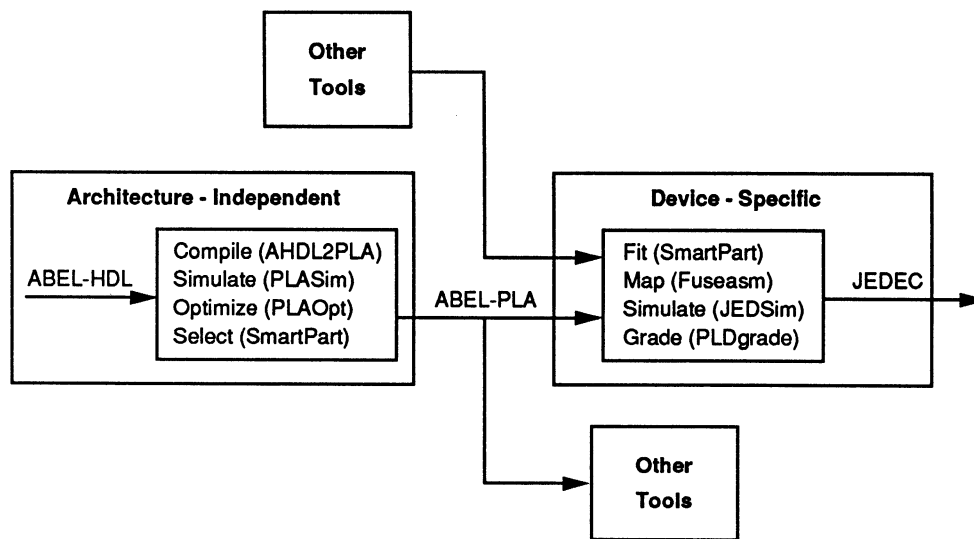
Fault Grading (PLDgrade)

The optional PLDgrade™ package for ABEL provides fault grading and testability analysis for your design.

The Open ABEL Concept

The ABEL-HDL language is architecture-independent. This means that you don't have to specify a particular device in your design, and can defer the implementation details and instead focus on the design itself. The ABEL-HDL compiler and supporting software allow designs written in the ABEL-HDL language to be functionally verified (through simulation), and be implemented in programmable integrated circuits (PICs) or transferred to other design environments through standard format design transfer files. See Figure 1-1.

Figure 1-1
The Open ABEL Concept



095-0694-001

The ABEL-PLA file format is based on the Espresso PLA format developed at the University of California at Berkeley. This format provides a compact way to transfer logic descriptions between various tools. Data I/O has added certain enhancements to the basic PLA format. These enhancements allow for more PLD-specific information and more complete design description of circuits that include flip-flops and other elements not otherwise addressed in the standard format.

The intent of Open ABEL is to allow device vendors to provide specialized tools (typically device fitters) that interface directly to ABEL. Many of the newer, more complex devices benefit from these specialized tools, and the ABEL-PLA file format provides the link between ABEL and these tools.

In addition to the specialized interface tools provided by device vendors, some users may want to develop their own interfaces to transfer design data into or out of the ABEL system. The ABEL-PLA file can be used for this as well. For more information about the ABEL-PLA format and Open-ABEL, contact Data I/O.

Structure of ABEL

Figure 1-2 shows the ABEL design process, which consists of two parts. The first part, shown in the upper portion of the diagram, consists of architecture-independent design entry and optimization. Architecture independence allows you to enter your design, compile that design into an ABEL-PLA file (and optional test vector file), optimize the ABEL-PLA file and use the resulting file as input to the SmartPart device selector or other utilities as required, all before selecting a device. The PLASim simulation module also allows you to verify correct operation of your design before committing to a specific device architecture.

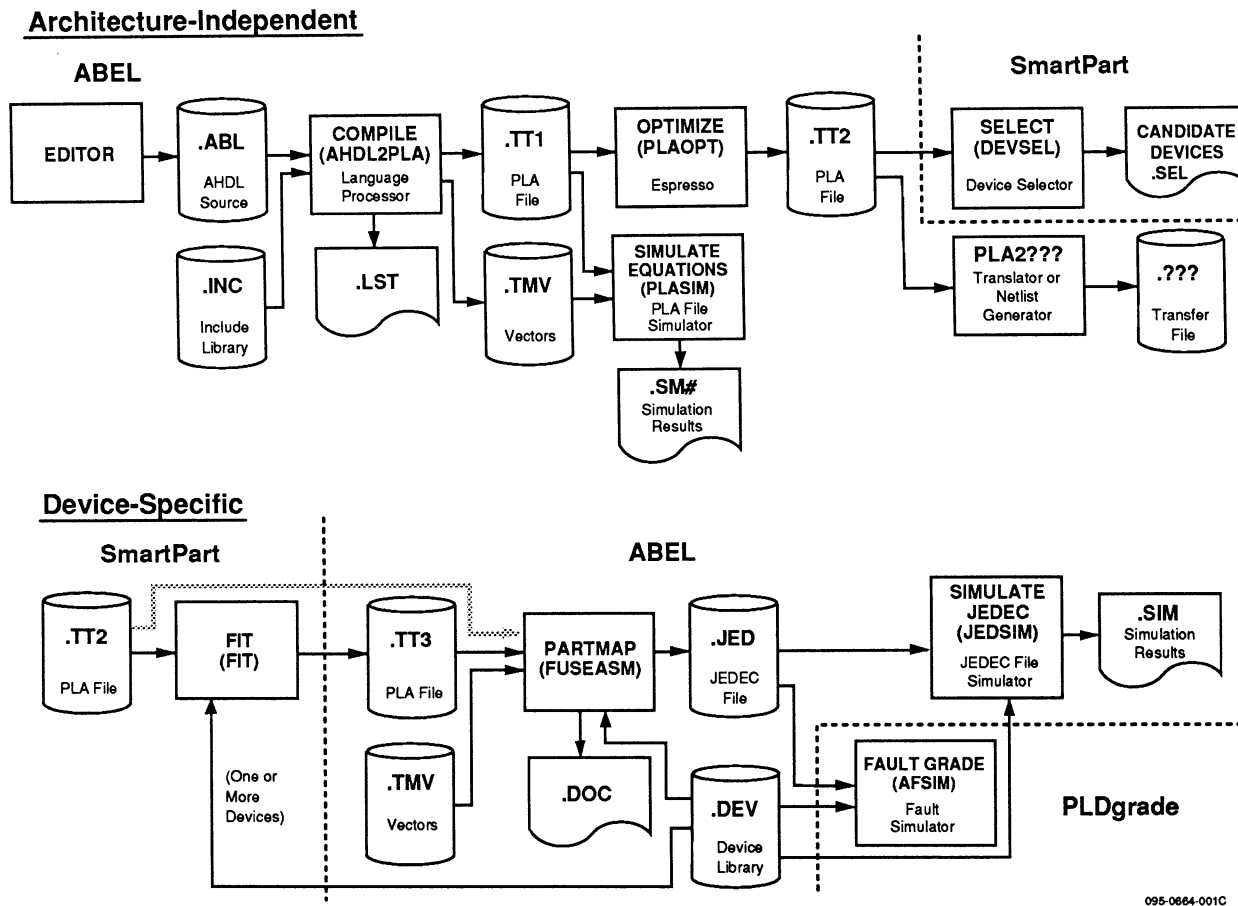
The second part of the design process implements the optimized design (in the ABEL-PLA file) into a target architecture. This part of the design process can include the use of SmartPart device fitters, which perform automatic pin assignment and other synthesis functions, or can proceed directly to the PartMap module, which creates a programmer load file (in JEDEC or other formats) for a specified device architecture. The JEDSim device simulator can then be used to verify that the programmed device will operate as expected.

Figure 1-2 also shows which functions are performed by basic ABEL and the ABEL options, SmartPart and PLDgrade.

The entire design process can be controlled from within the ABEL Design Environment, or can be invoked as stand-alone modules from the command line or a batch file. Other utilities are provided with the ABEL package that provide design documentation and translation functions. These utilities are discussed in the chapter "Using ABEL Processing Modules."

A summary of the functions that each ABEL language processor performs are discussed in this chapter. For information on running the ABEL software, refer to the chapter "Using ABEL Processing Modules."

Figure 1-2
Program Flow



Source File

The ABEL programs take high-level logic descriptions contained in source files and convert them into 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-HDL language.

Processing Modules

The ABEL processing modules synthesize and optimize logic descriptions in the source file and then convert them to programmer load files that can be downloaded to a logic programmer. The processing modules

- check your logic description for proper ABEL-HDL language syntax
- synthesize your design
- simulate your equations
- perform logic reduction
- generate a programmer load file
- simulate the operation of the programmed device
- create design documentation

The following is a summary of each of the modules. The ABEL Design Environment choice is given first, followed by the name of the executable file that performs the function:

1. **Compile (ahdl2pla)** — Compiles the source file, checks for correct syntax, expands macros, acts on directives, and synthesizes the design. Converts the description to PLA format.
2. **Simulate Equations (plasim)** — Simulates equations using a PLA File and test vector file.
3. **Optimize (plaopt)** — Performs logic reduction on the PLA File.
4. **PartMap (fuseasm)** — Creates the programmer load file and design documentation from a PLA file.
5. **Simulate JEDEC (jedsim)** — Simulates the function of a design in a device using JEDEC file and test vector file.
6. **SmartPart (devsel and fit)** — Optional module that selects candidate devices and fits the design into the selected device(s), including assigning pins.
7. **PLDgrade (afsim)** — Optional module that fault grades the design.

ABEL can be run through the ABEL Design Environment or by the command line. A batch file (command script) runs a default pass of the processing modules, or for more control over processing options, you can run each module separately, or use the ABEL Design Environment. The chapter "Using ABEL Processing Modules" discusses how to run the ABEL software using the ABEL Design Environment or the command line. This chapter gives an overview of what happens to your design file during each of the steps listed above. Figure 1-2 shows the processing flow of the modules.

Compile	Compilation is performed by the AHDL2PLA processing module. AHDL2PLA converts state diagrams and truth tables to Boolean equations, translates test vectors, expands macros, and checks for correct syntax. If any errors are found, the approximate place where the error occurs and the type of error are written to a listing file (.lst).
Simulate Equations	Equations are simulated with the PLASim processing module. PLASim applies inputs to the equations in the PLA file using the test vector file produced by AHDL2PLA. This allows you to simulate your design before you choose a device or assign pins. PLASim produces a simulation results file (*.smn).
Optimize	Optimization is performed by the PLAOpt processing module. PLAOpt reduces your logic description so that fewer product terms are used in the programmable logic device. PLAOpt reduces the equations in the ABEL-PLA file and writes the reduced equations to the output file, also in ABEL-PLA format.
PartMap	The Fuseasm processing module uses PLA files to create a design documentation file (*.doc) and programmer load files in various formats (including JEDEC) that are loaded into a programmer to program and test devices (*.jed or *.pxx).
Simulate JEDEC	<p>The device and design are simulated with the JEDSim processing module. JEDSim uses design and device information in the JEDEC file to simulate the operation of the design with a particular programmable device. JEDSIM can use any programmer load file conforming with the JEDEC standard to simulate the operation of PALs, FPLAs, and FPLSs, and a separate test vector file can be specified to simulate internal device nodes.</p> <p>JEDSim does not execute Boolean equations or apply inputs to ABEL-HDL 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.</p>
SmartPart	<p>The optional SmartPart package contains two modules that select candidate devices for a design (Devsel) and fit the design into the device (Fit).</p> <p>Devsel accepts user-specified selection criteria such as power rating, manufacturer and number of pins. The device database can be edited to include price and 3 user-defined fields. Devsel produces a report of devices and chips that meet the selected criteria (including a design file).</p> <p>Fit fits a design into a device and assigns pins. Fitters are device-specific. See the <i>SmartPart User Manual</i> for more information if you have this option.</p>
PLDgrade	The optional PLDgrade package uses the JEDEC file for a particular design and PLD to evaluate the fault coverage provided by the test vectors in the file. This process is referred to as "fault simulation." See the <i>PLDgrade User Manual</i> for more information if you have this option.

Output files

Many types of output files are created during execution of ABEL software. For complete information, see "Viewing Processing Results" in the chapter "Using ABEL."

<i>module_name.ttn</i>	PLA files used by ABEL programs.
<i>source_file.bat*</i>	AHDL2PLA output.
<i>source_file.tmv</i>	Test vector file.
<i>source_file.lst</i>	Compiler listing file.
<i>module_name.fus</i>	Fuses statement from AHDL2PLA.
<i>module_name.smn</i> <i>device_id.sim</i>	Simulation output from PLASim (.smn) and JEDSim (.sim).
<i>source_file.dmc</i>	Design manager control file that associates source file with all ABEL software output files.
<i>source_file.opt</i>	Options chosen for a file in the ABEL Design Environment are stored in this file.
<i>device_id.jed</i> <i>device_id.pxx</i> <i>device_id.pof</i>	Programmer load files (default is JEDEC format).
<i>device_id.doc</i>	Design documentation from Fuseasm.
<i>device_id.fts</i>	Fault grading results from the optional PLDgrade fault simulator.
<i>module_name.sel</i>	Device selector candidate list from the optional SmartPart package.
<i>module_name.fit</i>	Fitter pin assignments from the optional SmartPart package or ad hoc fitter.
<i>module_name.eqn</i>	PLA2EQN equations output.
<i>module_name.pds</i>	PLA2EQN ACTEL, PALASM II and XACT output.
<i>module_name.pin</i> <i>module_name.bee</i>	PLA2EQN Signetics Snap output.
<i>module_name.pla</i>	PLA2EQN PLA file output (same as .ttn).

These working files can be removed after they are no longer needed using the **cleanup4** program provided with the ABEL package. See "Utilities" in the chapter "Using ABEL Processing Modules" for more information.

*Note: The .bat file is created on DOS and UNIX systems only. On VMS-based systems, this file will have a .cmd extension.

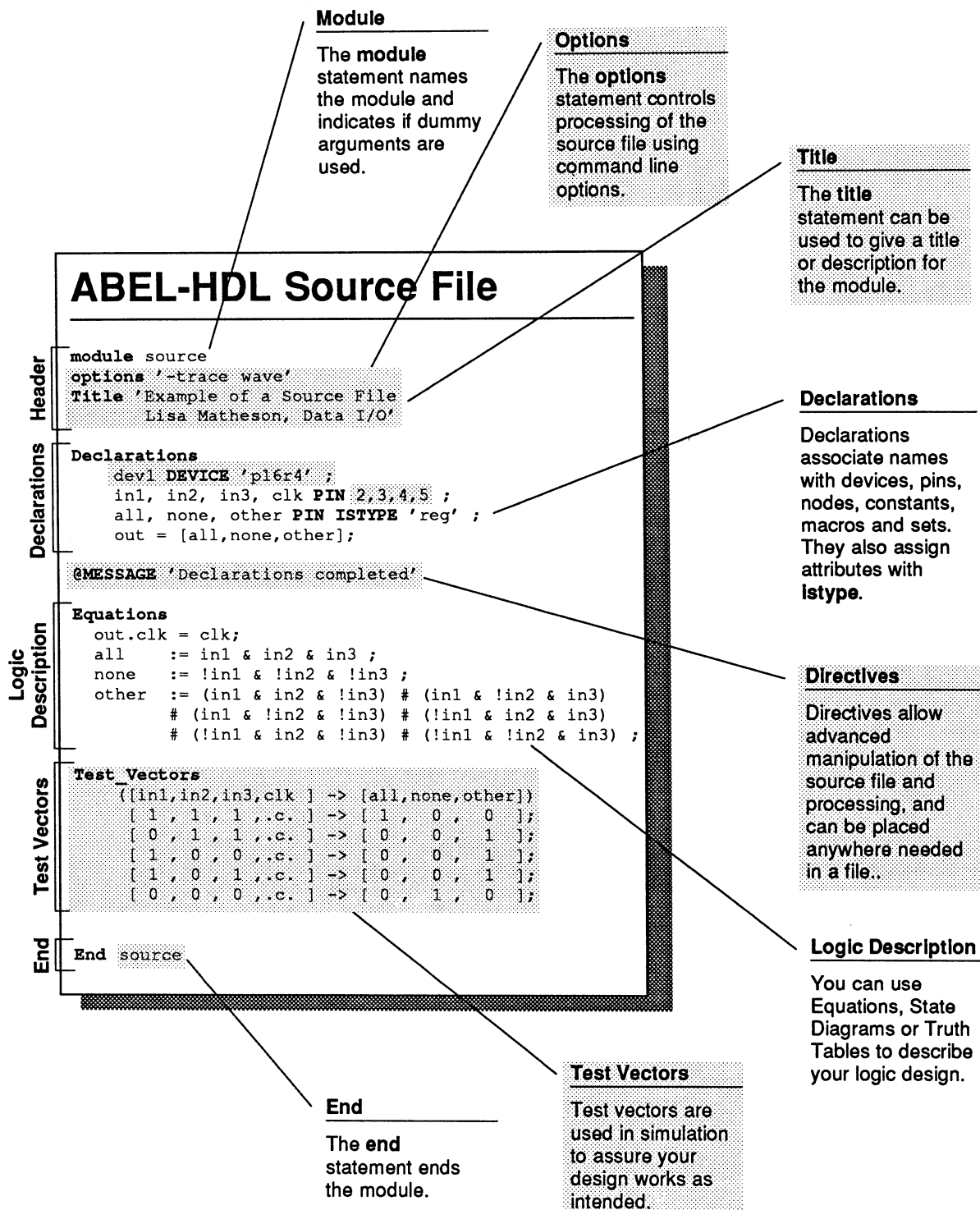
2 *Tutorials*

This chapter assumes you have installed ABEL and the ABEL Design Environment as described in the *Installation Guide*. Not all aspects of the ABEL Hardware Description Language (ABEL-HDL) or the ABEL processing modules are described here; refer to the chapters "Language Reference" and "Using ABEL Processors" for complete information.

Introduction to a Source File

Figure 2-1 shows the basic elements of an ABEL-HDL source file.

Figure 2-1
Source File Structure



Bold denotes ABEL-HDL keywords.
Shading denotes optional items.

Creating a Simple Design

This session will step you through an architecture-specific design. You may also want to look at the tutorials on processing a design, creating architecture-independent designs, creating state descriptions, or creating truth tables.

The source file for a memory decoder, **m6809a.abl**, is provided with the ABEL package. If desired, you can enter the design using the ABEL Design Environment editor, or using any ASCII editor. ABEL-HDL keywords are shown in boldface.

If you want to view the existing m6809a.abl source file instead of entering it, make sure m6809a.abl is in your working directory. Start the ABEL Design Environment by entering

```
abel4 m6809a.abl
```

Main menu selections in the ABEL Design Environment are made by pressing **Alt** and the highlighted letter. Submenu selections are made by pressing the highlighted letter.

The Source File

The m6809a.abl source file is shown in Figure 2-2, then each part of the file is discussed in detail.

Figure 2-2
Source File: m6809a.abl

```

module M6809A
title '6809 memory decode
Jean Designer Data I/O Corp Redmond WA'

                                m6809a 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

```

Module Statement

```
module M6809A
```

The **module** statement is a required element of the source file. It defines the beginning of the module and must be paired with an **end** statement. The module name given determines the name of the PLA file and other output files created during processing by ABEL software. Choose a name that is unique and descriptive; unique to avoid overwriting existing files, and descriptive so that you can easily remember what the files contain. Also, choose a name that is a valid filename for your operating system.

Title Statement

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

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 list 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. The title statement and description are optional, and can be used to provide a quick summary of the file contents.

Device Declaration

```
m6809a device 'P14L4';
```

The device declaration is used to associate the name of the device with the type of programmable logic device the logic design will be programmed into. This declaration is optional for the first three ABEL processing modules (AHDL2PLA, PLAOpt and PLASim). The device name and device type are separated by **device**, the keyword of the Device declaration. For the example above, the identifier, **m6809a**, will be the name of the programmer load file that is created during processing.

Declaring a device does not restrict the design to the device specified. The *device selector* and *fitters* can be used later in the design process to retarget the design to a different device.

Signal and Pin Assignments

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

Note that each declaration line ends with a required semicolon (;). The first declaration assigns the identifiers A15 through A10 to pin numbers 1 through 6, respectively. That is, A15 is assigned to Pin number 1, A14 to Pin 2, and so on. In the second declaration, the identifiers ROM1, IO, ROM2 and DRAM are assigned to pins 14 through 17. Assignment of pin numbers is optional for the first three processing modules, but must be done before processing by Fuseasm. An optional Fitter program can be used to do pin assignments automatically. See the *SmartPart User Manual* for a tutorial on the fitter if you have this option

Constant and Set Declarations

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

This entry assigns constant values to the identifiers **H**, **L** and **X**, and assigns a set to the identifier **Address**. These identifiers can then be used in the source file instead of the constants and set. Constant and set declarations help to make source files more readable, and also allow you to change constant values easily, since only the declaration needs to be changed. Note that each declaration ends with a required semicolon (;).

Equations Section

equations

```
!DRAM   = (Address <= ^hDFFF);

!IO      = (Address >= ^hE000) & (Address <= ^hE7FF);

!ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);

!ROM1    = (Address >= ^hF800);
```

This application is easiest to describe with equations. The equations use the logical operators **!** (NOT) and **&** (AND), the combinatorial assignment operator **=**, the relational operators **>=** (greater than or equal) and **<=** (less than or equal) to describe the function of the circuit.

The first equation,

```
!DRAM   = (Address <= ^hDFFF);
```

states that the signal **DRAM** goes low whenever the set **Address** is less than or equal to the hex value **DFFF**. The signal **DRAM** and the set **Address** have both been defined earlier in the source file.

Note that the equations end with required semicolons (;), and are fully parenthesized to assure proper operation. The second equation states that the signal **IO** goes low whenever **Address** is between the hex values **E000** and **E7FF**. Without the parentheses, the AND operator (**&**) would be performed before the relational operators (**<=** and **>=**), and the equation would be interpreted as if it were parenthesized as follows:

```
!IO = Address >= (^hE000 & Address) <= ^hE7FF);
```

which is not the intended result and would cause errors when compiled.

You can also use truth tables and state descriptions in addition to or instead of equations to suit your application. Note that each equation ends with a required semicolon (;).

Test Vector Section

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 ];
```

These test vectors are used in the ABEL simulators, PLASim and JEDSim to simulate the circuit description. Test vectors describe the expected output conditions for various circuit inputs. Test vectors are always written for *pin-to-pin* behavior. Nodes are considered to be unattached pins. Note that each test vector ends with a required semicolon (;).

End Statement

end M6809A

The end statement indicates the end of the source file for the ABEL programs.

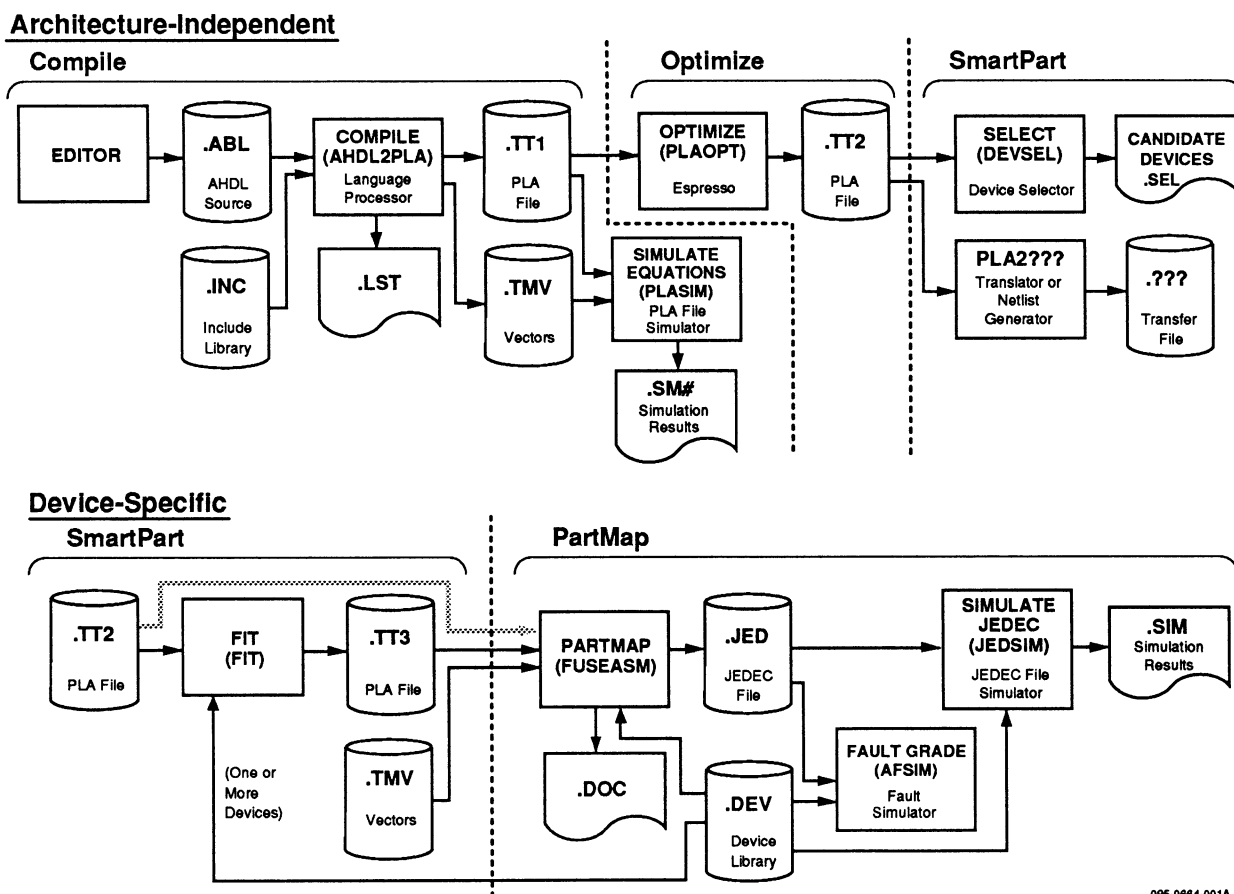
Processing a Design

This section steps through the ABEL processing modules for the device-specific design created in the first tutorial. The steps for an architecture-independent design are exactly the same, except that a device must be selected and pins assigned before the programmer load file can be generated.

This section gives a sampling of options available from the ABEL design environment; it does not demonstrate the use of all available options.

Figure 2-3 shows the program flow.

Figure 2-3
Program Flow



095-0064-001A

Compiling the Design

The ABEL program AHDL2PLA checks your source file for syntax errors and converts it into a PLA format file for processing by subsequent programs.

1. To run the file **m6809a.abl** through AHDL2PLA, make sure the file is in your current directory and start the menus by entering:

abel4 m6809a

2. Select the **Compile** menu.
3. Select **Options...** to call up the Compile Options dialog box.
4. Use **[Tab]** or the arrow keys to place the cursor on **Standard Listing** and press **[Space]** to select it. Then highlight **<OK>** and press **[J]**.
5. Select **Compile** to compile your design. A window will open to display processing messages.

The List File (.lst)

6. If there are any syntax errors during compilation, look at the output file, **m6809a.lst**, by choosing the **View** menu and selecting **Compile Listing**. This file will show error types and locations. Correct any errors before continuing. A common error is the omission of a required semicolon (;).

Simulating the Design

The ABEL package has two ways to simulate a design: by equations using a PLA file and by device using a JEDEC file. The **Simulate Equations** options in the **Compile**, **Optimize** and **SmartPart** menus simulate design equations, and the simulation is of the design only. No device information is used, even if provided. Because **Simulate Equations** is architecture-independent, you can use it to simulate your design before device selection and pin assignment. See later in this tutorial for device simulation.

7. Select **Trace options...** from the **Compile** menu. The **Simulate Trace Options** dialog box will appear.
8. Place the cursor on **Wave format** and press **[Space]** to select it.
9. Press **[F5]**, then highlight **Simulate Equations** and press **[J]** to start simulation. Note that if you have changed the source file since compilation, the ABEL Design Environment will automatically update the compilation run before simulation.

The Simulation
Output File (.sm1)

10. Look at the **m6809a.sm1** file produced by **Simulate equations** by selecting **Simulation results** from the **View** menu.

Optimizing the Design

11. Select **Options...** from the **Optimize** menu, then select **Reduce bypin, auto polarity** by placing the cursor on this selection and pressing **[Space]**. Then select **<OK>**.

12. When you are returned to the **Optimize** menu, select **Reduce** to start reduction. The optimization program reduces the logic according to options you selected. For more information on this option, see "Using ABEL."

Generating a Programmer Load File

The **PartMap** menu selections are the only menu selections that are device-specific. That is, you must select a device and assign pins before processing your design with **PartMap**. In our example, we have already selected a device and assigned pins. If we were to process an architecture-independent design, or if we wanted to target a different device, we would have to either manually select a device and assign pins, or use the **SmartPart** menu (if you have the optional **SmartPart** package) to do device selection and pin assignments. Refer to the tutorial in the *SmartPart User Manual* for information on using these optional programs.

Use the instructions below to generate and view a programmer load file from your PLA format file using the program defaults

13. Select **PLDmap Report** from the **View** menu. ABEL's auto-update feature will automatically run Fuseasm to create the report requested. If you get a "file not found" message, check to make sure **Auto-Update** is checked in the **Defaults** menu and try again.

The **PLDmap** selection from the **PartMap** menu creates a programmer load file, **m6809a.jed** from the design. Different formats for the programmer load file can be selected, but the default is JEDEC format.

Simulating the Device

Another method of simulation simulates a design and device together, using a JEDEC format file.

Use the instructions below to simulate the **m6809a** design. The options shown below are to use the **macro** trace format for Vector 1.

14. Select **Trace Options...** from the **PartMap** menu. The **Simulate Trace Options** dialog box will appear.
15. Place the cursor on **Macrocell** format and press **[Space]** to select it.
16. Highlight the **First Display Vector** field and enter "1".
Then Highlight the **Last Display Vector** field and enter "1". This limits the results file to the first vector.
17. Select **<OK>**. When you're returned to the **PartMap** menu, select **Simulate JEDEC** (or **[F4]**) to begin simulation.

The Simulation Output File (.sim)

18. Look at the **m6809a.sim** file by selecting **Simulation Results** from the **View** menu. The **Simulation Results** selection always displays the results from the last simulation. This file is also shown in Figure 2-4.

Figure 2-4
JEDEC Simulation Results:
m6809a.sim

```

Simulate ABEL 4.00  Date Mon Aug 13 11:12:46 1990

Fuse file: 'm6809a.jed'  Vector file: 'm6809a.tmv'  Part: 'P14L4'

ABEL 4.00 Data I/O Corp. JEDEC file for: P14L4 V8.0
Created on: Mon Aug 13 11:11:59 1990

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

Vector 1
Vector In  [000000.....]

                                ROM1 Pin 14  | \
                                ---| >O--- H   Vec=H
                                | | /
                                |
PT 336 [FFFF] ]---| OR = L |---
                                |
                                ROM2 Pin 16  | \
                                ---| >O--- H   Vec=H
                                | | /
                                |
PT 112 [FFFF] ]---| OR = L |---
                                |
                                IO Pin 15   | \
                                ---| >O--- H   Vec=H
                                | | /
                                |
PT 224 [FFFF] ]---| OR = L |---
                                |
                                DRAM Pin 17  | \
                                ---| >O--- L   Vec=L
                                | | /
                                |
PT   0 [TTTF] ]---| OR = H |---
                                |

Vector Out [.....HHHL...]

8 out of 8 vectors passed.

```

Automatic Design Updating

With automatic design updating, you can set all of your options, then select your target function. All prerequisite functions will be performed automatically. An example of auto-updating was used in "Generating a Programmer Load File" above.

For another example of auto-updating, load a different source file (for example, regfb.abl), and select **Optimized Equations** from the **View** menu. The ABEL Design Environment will do the prerequisite steps to produce the requested file (Compile and Optimize).

Creating an Architecture-independent Design

This section shows you some of the differences between architecture-specific and architecture-independent design, and presents some of the considerations that need to be made when creating a design that will map into many different architectures and devices.

Par_det.abl is a source file for a parity detector provided with the ABEL package. This file is shown in Figure 2-5.

Figure 2-5
Source File: *par_det.abl*

```

module par_det
title 'Architecture independent description of a serial parity detector
      Jeff Davis  Data I/O Corp.   May 17,1990 '

declarations
      clock          pin ;
      serial_in      pin ;
      odd_even       pin ;
      shift          pin ;
      parity         pin istype 'reg';

      ODD  = 1;
      EVEN = 0;

equations
      parity.clk = clock; "Describe clocked input to register

      "toggle parity when odd/even = EVEN and serial_in = 0
      "      and when odd/even = ODD  and serial_in = 1

      parity := ((serial_in !$ odd_even) & shift) $ parity.FB ;

test_vectors ([clock,serial_in,odd_even,shift]->[parity])
      [ .c. , 0 , EVEN , 0 ]->[ .X. ];
      [ .c. , 1 , EVEN , 1 ]->[ EVEN ];
      [ .c. , 0 , EVEN , 1 ]->[ ODD ];
      [ .c. , 1 , EVEN , 1 ]->[ ODD ];
      [ .c. , 1 , EVEN , 1 ]->[ ODD ];
      [ .c. , 0 , EVEN , 1 ]->[ EVEN ];
      [ .c. , 1 , ODD , 1 ]->[ ODD ];
      [ .c. , 0 , ODD , 1 ]->[ ODD ];
      [ .c. , 1 , ODD , 1 ]->[ EVEN ];
      [ .c. , 0 , ODD , 1 ]->[ EVEN ];
      [ .c. , 1 , ODD , 1 ]->[ ODD ];
      [ .c. , 1 , EVEN , 0 ]->[ .X. ];

end

```

Declarations

Pin and node numbers do not need to be assigned in an architecture-independent design. However, assigning pin and node numbers does not mean your design is not architecture-independent, since pin numbers can be reassigned manually or with the optional SmartPart program.

```
clock           pin ;
serial_in       pin ;
odd_even        pin ;
shift           pin ;
```

Note that no device declaration was made as in the architecture-specific design, and the pin number assignments have been left blank. Your design can be compiled (**Compile**), simulated (**Compile** or **Optimize: Simulate equations**) and reduced (**Optimize**) before pin number assignments are made.

A device must be declared and pin number assignments must be specified before processing by Fuseasm and JEDSim. Device selection and pin assignments can be done manually, or the **SmartPart** menu can be used to do automatic device selection, pin assignments and device fitting if you have the optional SmartPart package. Refer to the *SmartPart User Manual* if you have this option.

The last pin declaration

```
parity          pin  istype 'reg';
```

declares the signal **parity** as a clocked memory element. The **reg** assignment is an architecture-independent attribute and allows most device architectures to be used for this design.

Equations

The behavior of the signal **parity** is described in terms of the desired output at the pin for the various input conditions. The dot extension, **.FB**, is used to indicate the registered feedback from the pin with signal name **parity**. The equations do not contain any architecture-specific attributes, and therefore allow for architecture-independence.

```
equations
  parity.clk = clock; "Describe clocked input to register

"toggle parity when odd/even = EVEN and serial_in = 0
"          and when odd/even = ODD  and serial_in = 1

  parity := ((serial_in !$ odd_even) & shift) $ parity.FB ;
```

The comments (preceded by double quotes (")), state the function of the following equation. Fully commenting your file will make the source code easier to understand and modify.

The last equation uses the logical operators **&** (AND), **\$** (Exclusive OR), **!\$** (Exclusive NOR), and the **:=** assignment operator. The **:=** operator is used to specify the behavior of a registered design in terms of its expected outputs. This method of design, referred to as the *pin-to-pin* model, is in contrast to the *detail* model, in which you describe your circuit in terms of its flip-flop stimulus. Note also that the expression is fully parenthesized () to assure proper operation.

Summary

Architecture-independent design descriptions, such as the one shown above, use signal attributes and special signal dot extensions to resolve design ambiguities. It is important to realize, however, that as you begin to use more complex features in your design (such as register reset and preset, or specific register types), you begin to limit the number of different architectures that the design is appropriate for. Also, architecture-independent design descriptions will almost always require more descriptive ABEL-HDL source files, since certain information (such as signal types, or the existence of clock and enable inputs) cannot be inferred by ABEL from a declared device type.

Creating a State Description

A state machine described in ABEL-HDL consists of the following elements:

- State registers
- State values
- State transitions
- Output equations

A simple architecture-independent state description is shown in Figure 2-6:

*Figure 2-6
Simple State Diagram
Description*

```

module sm1

    q0,q1    pin istype 'reg' ; "state registers
    clock    pin;           "global clock input
    start    pin;           "initialization input
    A,B      pin istype 'com'; "combinatorial outputs

equations

    [q1,q0].clk = clock;

state_diagram [q1,q0]

    state [1,1]:
        A = 1;           "A is high in this state
        B = 1;           "B is high in this state
        if (start)
            then [1,1]
        else
            [0,0];

    state [1,0]:
        A = 1;           "A is high in this state
        if (start)
            then [1,1]
        else
            [0,1];

    state [0,0]:
        if (start)
            then [1,1]
        else
            [0,1];

    state [0,1]:
        B = 1;           "B is high in this state
        goto [1,1];

end

```

The design begins with declarations for the state registers (q0 and q1) that are registered output pins, for the state machine inputs (the start input signal and the common clock), and for the state machine's combinatorial outputs (A and B).

The clock equation

```
[q1,q0].clk = clock;
```

indicates how the two state registers are to be clocked.

Basic Structure

The state diagram itself consists of a state diagram header (`state_diagram`) followed by four state descriptions: one for each possible state of the machine. Each state description begins with a description of the state value to be stored in the state registers (`q0` and `q1`). These values can be written as sets of binary values as shown, or can be specified as numbers or given symbolic values through the use of constant declarations.

The state values are followed by an optional sequence of state equations (notice that the third state has no state equations). The state equations are evaluated only in the state in which they are written. ABEL-HDL also allows you to write output equations that are related to state transitions instead of to specific states. The `WITH` statement is used for this purpose. Refer to the chapter "Language Reference" for more information on this feature.

Following the state equations are the transition statements. Each state in a state machine must include at least one transition statement. There are three types of transition statements in ABEL-HDL. Two of these types (the `IF-THEN-ELSE` conditional transition and the `GOTO` unconditional transition) are shown in this example. The `CASE` statement (not shown) can also be used to specify transitions.

Considerations

There are many things to consider when writing complex state diagrams. This simple example demonstrates the basic format of an ABEL-HDL state description. Before beginning to write more complex state machines, review the information in the chapter "Design Considerations" and study the state machine examples provided with the ABEL software.

Creating a Truth Table

Truth tables are an alternative method for describing both combinatorial and sequential circuits. A truth table consists of the following elements:

- A truth table header
- Truth table entries

A simple truth table is shown in Figure 2-7:

Figure 2-7
Simple Truth Table

```

module TT1
    a,b,c    pin;           "design inputs
    AND,OR   pin istype 'com'; "combinatorial outputs

    truth_table ([a,b,c]->[AND,OR])
        [0,0,0]->[ 0 , 0];
        [0,0,1]->[ 0 , 1];
        [0,1,0]->[ 0 , 1];
        [0,1,1]->[ 0 , 1];
        [1,0,0]->[ 0 , 1];
        [1,0,1]->[ 0 , 1];
        [1,1,0]->[ 0 , 1];
        [1,1,1]->[ 1 , 1];

end

```

The design begins with declarations of the inputs and combinatorial outputs of the circuit, then proceeds directly to the truth table that describes the design function. The truth table begins with a header that defines the inputs and outputs of the function. The header is followed by a sequence of truth table entries that define the required output values for each possible set of input values.

As in state diagrams, the values for each input condition and corresponding output can be entered in a number of forms. For example, the above truth table could be rewritten in the form:

```

truth_table ([a,b,c]->[AND,OR])
    0    ->[ 0 , 0];
    1    ->[ 0 , 1];
    2    ->[ 0 , 1];
    3    ->[ 0 , 1];
    4    ->[ 0 , 1];
    5    ->[ 0 , 1];
    6    ->[ 0 , 1];
    7    ->[ 1 , 1];

```

with the same result.

Truth tables can also be written for sequential functions. For example, the state machine design presented earlier in this chapter could be written as a truth table as shown in Figure 2-8:

Figure 2-8
Truth Table from State Machine

```

module sm2
    q0,q1    pin istype 'reg' ; "state registers
    clock    pin;             "global clock input
    start    pin;             "initialization input
    A,B      pin istype 'com'; "combinatorial outputs

    equations

        [q1,q0].clk = clock;

    truth_table ([q1.fb,q0.fb,start]:>[q1,q0]->[A,B])
        [ 1 , 1 , 1 ]:>[ 1, 1]->[1,1];
        [ 1 , 1 , 0 ]:>[ 0, 0]->[1,1];
        [ 1 , 0 , 1 ]:>[ 1, 1]->[1,0];
        [ 1 , 0 , 0 ]:>[ 0, 1]->[1,0];
        [ 0 , 0 , 1 ]:>[ 1, 1]->[0,0];
        [ 0 , 0 , 0 ]:>[ 0, 1]->[0,0];
        [ 0 , 1 , 1 ]:>[ 1, 1]->[0,1];
        [ 0 , 1 , 0 ]:>[ 1, 1]->[0,1];

    end

```

You can see that the `:>` operator is used to describe registered transitions, and the `->` operator is used to describe combinatorial transitions.

Considerations

As with state machines, there are important considerations when writing complex truth tables for sequential circuits. Refer to the chapters "Language Reference" and "Design Considerations" for more detailed information on truth tables.



3 ABEL-HDL Language Structure

ABEL uses logic descriptions contained in a *source file* to create programmer load files. A source file is an ASCII text file in the ABEL Hardware Description Language, ABEL-HDL. A source file may contain any number of modules, each with a logic description. The requirements for ABEL-HDL are described in the following chapters.

This chapter provides the basic syntax and structure of a design description in ABEL-HDL. For usage information on specific elements, refer to the "Language Reference." You can write a source file using any word processor or text editor that produces ASCII files, such as the simple editor in the ABEL Design Environment.

Summary

Introduction — An introduction to ABEL-HDL and to the idea of architecture-independent and architecture-specific logic descriptions.

Basic Syntax of a source file, including

- Valid ASCII Characters
- Identifiers and Keywords
- Constants
- Blocks
- Comments
- Numbers
- Strings
- Operators, Expressions and Equations
 - Logical Operators
 - Arithmetic Operators
 - Relational Operators
 - Assignment Operators
 - Expressions
 - Equations

- Sets and Set Operation
- Arguments and Argument Substitution

Basic Structure of a design description, including

- Header
 - Module
 - Options
 - Title
- Declarations
 - Keyword
 - Device
 - Constant
 - Signal
- Logic Description
 - Equations
 - Truth Tables
 - State Descriptions
 - Fuses
 - XOR Factors
- Test Vectors
- End

Introduction to ABEL-HDL

ABEL-HDL is a hardware description language optimized for, but not limited to, PLD design. ABEL-HDL is based on Data I/O's industry-standard ABEL design language.

ABEL-HDL supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. The ABEL-HDL compiler and supporting software allow designs written in ABEL-HDL to be functionally verified (through simulation), and be implemented in PLDs or transferred to other design environments through standard format design transfer files.

ABEL-HDL allows designs to be entered and verified with little or no concern for the target device architecture.

Architecture-independent design descriptions (those that do not include device declarations and specific pin number declarations) require more comprehensive descriptions than their architecture-specific counterparts. Assumptions that can be made when a particular device is specified are not possible when no device is specified. See the section "Architecture-independent Language Features" in the chapter "Design Considerations" for more information.

Basic Syntax

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

1. A line may be up to 150 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 .
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 uppercase, lowercase or mixed-case.
6. Identifiers (user-supplied names and labels) can be 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 discussed later in this chapter. 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 2), 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. Except for Reserved Identifiers (Keywords), 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)

The keywords listed below are reserved identifiers that are part of ABEL-HDL. Keywords 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.

case	goto	property
declarations	if	state
device	in (obsolete)	state_diagram
else	istype	test_vectors
enable (obsolete)	library	then
end	macro	title
endcase	module	trace
endwith	node	truth_table
equations	options	when
flag (obsolete)	pin	with
fuses		

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 their function. 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

Constants

Constant, non-changing values can be used in ABEL-HDL 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" later in this chapter). Constant values may be either numeric or one of the non-numeric special constant values. The special values are listed in Table 3-1.

Table 3-1
Special Constants

Constant	Description
.C.	clocked input (low-high-low transition)
.D.	clock down edge (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.
.U.	clock up edge (low-high transition)
.X.	don't care condition
.Z.	tristate value

When a special constant is used, it must be entered as shown in Table 3-1 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 uppercase or lowercase.

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" later in this chapter, and in the "Language Reference.")

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.

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 are shown in boldface:

```
MODULE Basic Logic; "gives the module a name
TITLE 'ABEL-HDL 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 required strings, not comments, and are part of the statement.

Numbers

All operations in ABEL-HDL involving numeric values are done with 32-bit accuracy. Thus, valid numeric values fall in the range 0 to 2^{32} minus 1. 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 3-2 gives the four bases supported by ABEL-HDL and their accompanying symbols. The base symbols can be typed in uppercase or lowercase.

Table 3-2
*Number Representation in
Different Bases*

Base Name	Base	Symbol
binary	2	[^] b
octal	8	[^] o
decimal	10	[^] d (default)
hexadecimal	6	[^] 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 the "Language Reference" for more information.

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

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.

Numbers may also be specified by strings of one or more 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

Strings

Strings are series of ASCII characters enclosed by single quotes (apostrophes). Strings are used in the TITLE, MODULE and OPTIONS 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'
```

is the string

```
It's easy to use ABEL
```

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

```
'He\\she can use backslashes in a string'
```

becomes the string

```
He\she can use backslashes in a string
```

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

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 two examples) are indicated by operators within the expression.

ABEL-HDL 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-HDL incorporates the standard logical operators listed in Table 3-3. Logical operations involving operands of more than one bit are performed bit by bit. For alternate operators, refer to the @ALTERNATE directive in the "Language Reference."

Table 3-3
Logical Operators

Operator	Description
!	NOT: ones complement
&	AND
#	OR
\$	XOR: exclusive OR
!\$	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 3-4 lists the arithmetic operators.

Table 3-4
Arithmetic Operators

Operator	Example	Description
-	-A	twos complement (negation)
-	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

Note: 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.

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 3-5 lists the relational operators.

Table 3-5
Relational Operators

Operator	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

All relational operations are unsigned. For example, the expression `!0 > 4` is true since the complement of `!0` is 1111 (assuming 4 bits of data), 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, `!0`, the complement of 0, is 32 bits all set to 1.

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 means 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 priority table later in this chapter). The following equation

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

has the default parentheses grouping

```
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, combinatorial and registered. Combinatorial or immediate assignment occurs without any delay as soon as the equation is evaluated. Registered assignment occurs at the next clock pulse from the clock associated with the output. The `:=` operator is used only when writing pin-to-pin registered equations. Refer to the chapter "Design Considerations." Table 3-6 shows the assignment operators.

Table 3-6
Assignment Operators

Operator	Description
=	Combinatorial assignment
:=	Registered assignment

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 3-7 summarizes the logical, arithmetic and relational operators, presented in groups according to their priority.

Table 3-7
Operator Priority

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 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.

Equations use the two assignment operators = (combinatorial) and := (registered) described above.

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.

See "Equations" and "When-Then-Else" in the "Language Reference."

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.

Equations Found	Equivalent Equation
$A = B;$ $A = C;$	$A = B \# C;$
$A = B;$ $A = C \& D;$	$A = B \# (C \& D);$
$A = !B;$ $A = !C;$	$A = !B \# !C;$
$!A = B;$ $!A = C;$	$A = !(B \# C);$
$!A = B;$ $A = !C;$	$A = !C \# !B;$
$!A = B;$ $!A = C;$ $A = !D;$ $A = !E;$	$A = !D \# !E \# !(B \# C);$

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-HDL 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 decremting 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-HDL source file sets are not mathematical sets.

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 3-8 lists the operators that may be used with sets. The appendix "Operator Rules" 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.

For example, the Boolean 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.

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 Boolean 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. Valid set operations are given in Table 3-8 below. For example,

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

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

The set assignment

```
[a,b] = c & d;
```

is the same as the two assignments

```
a = c & d;
b = c & d;
```

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;"compiler will add leading zero
```

Table 3-8
Valid Set Operations

Operator	Example	Description
=	A = 5	combinatorial assignment
:=	A := [1,0,1]	registered 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 (`==`, `!=`, `>`, `>=`, `<`, `<=`) 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

`[a, b, c] & [1, 0, 1]`
`= [a & 1, b & 0, c & 1]`
`= [a, 0, c]`

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. Relational operators always result in a single bit.

`[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 is a single-bit result which is distributed to both members of the set on the output side of the equation.

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

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

If you have a set assigned to a single value, the value will be padded with 0s and then applied to the set. For example,

```
[A1,A2,A3] = 1
```

is equivalent to

```
A1 = 0
A2 = 0
A3 = 1
```

which may not be the intended result.

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 since relational operators evaluate to a single-bit result:

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

It will be expanded as follows:

```
9 & ( !a & !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.

```
[x1, y1] = [a, b] & 1 & d
          = ([a, b] & 1 ) & d
          = ([a, b] & [0, 1]) & d
          = ([a & 0, b & 1]) & d
          = [ 0 , b ] & d
          = [0 & d, b & d]
          = [0, b & d]
```

```
x1 = 0
y1 = b & d
```

```
[x2,y2] = 1 & d & [a, b]
        = (1 & d) & [a, b]
        = d & [a, b]
        = [d & a, d & b]
```

```
x2 = a & d
y2 = b & d
```

If you are unsure about the interpretation of an equation, try the following hints:

1. Fully parenthesize your equation. Errors can occur if you are not familiar with the precedence rules in Table 3-7.
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.

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-HDL, a distinction must be made between two types of arguments: actual and dummy. Their definitions are given here.

dummy argument	an identifier 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-HDL.

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-HDL 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.

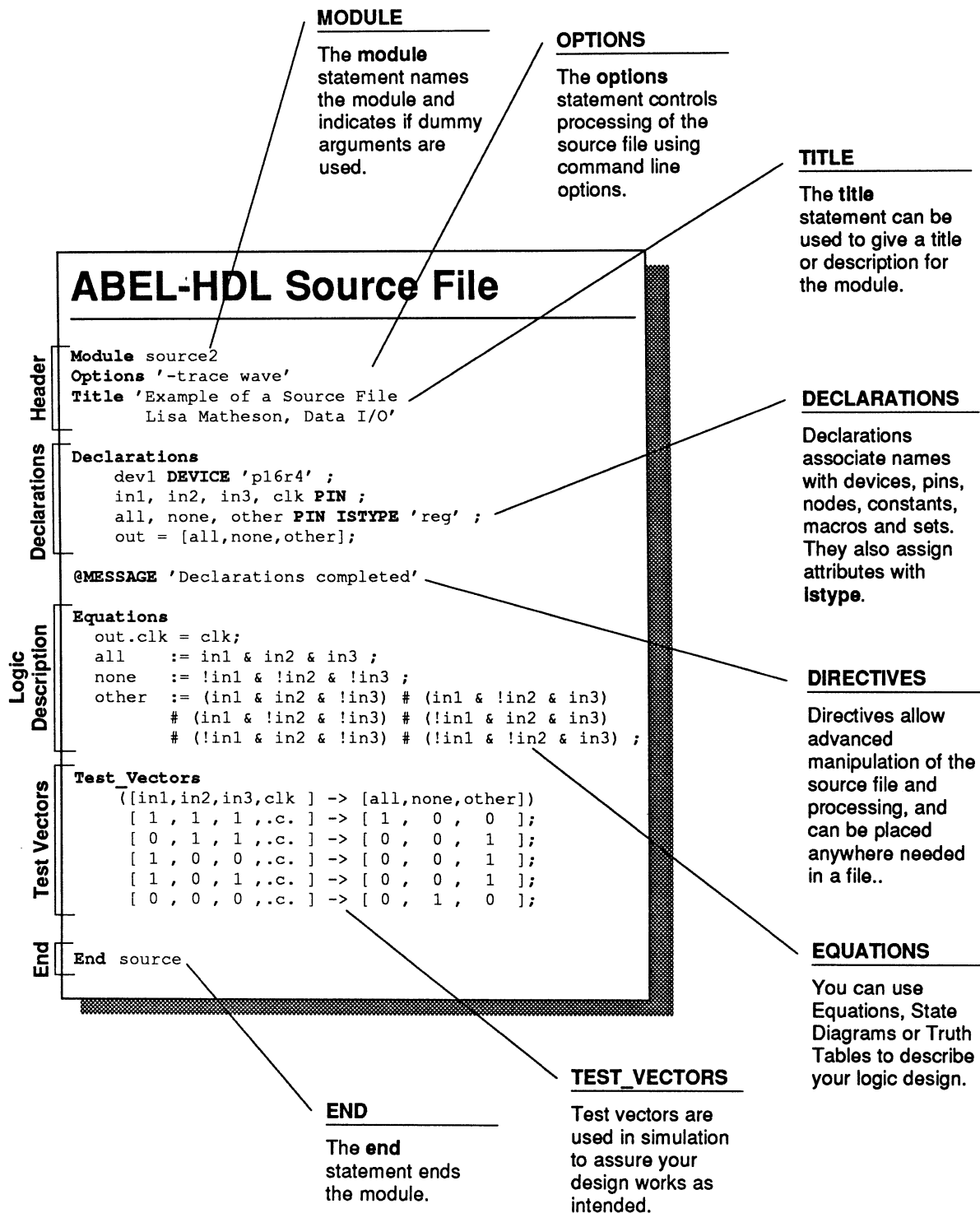
Further discussion and examples of argument usage are given in the "Language Reference" under "Module," "Macro" and "@directive."

Basic Structure

ABEL-HDL source files can be broken into independent parts called modules. Each module contains a complete logic description. At the simplest level, an input file to the ABEL language processors consists of one module; at the most complex level, any number of modules may be combined into one source file and processed at the same time. 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. Only one device can be specified per module.

This section covers the basic elements and keywords that make up an ABEL-HDL source file module. A module can be divided into five sections — Header, Declarations, Logic Description, Test Vectors and End. The elements of the source file are shown in the template in Figure 3-1. There are also directives which can be included in any of the five sections. These sections are presented briefly below, then each element is introduced. Complete information on each element can be found in the Language Reference. Figure 3-1 gives a pictorial representation of a source file.

Figure 3-1
Source File Structure



Bold denotes ABEL-HDL

The following three rules apply to module structure:

1. A module must contain only one header (composed of the Module statement and optional Title and Options statements).
2. Besides the header, all sections of the source file can be repeated in any order. Declarations must either immediately follow the header or the **Declarations** keyword.
3. No symbol (identifier) can be referenced before it is declared.

Header

The Header Section can consist of the following elements:

- Module Statement (required)
- Options
- Title

Declarations

A Declarations Section can consist of the following elements:

- Declarations Keyword
- Device Declaration (one per module)
- Signal Declarations (pin and node numbers optional)
- Constant Declarations
- Macro Declarations
- Library Declarations

Logic Description

One or more of the following elements can be used to describe your design.

- Equations
- Truth Tables
- State Diagrams
- Fuses
- XOR Factors

Test Vectors Section

A Test Vectors Section can consist of the following elements:

- Test vectors
- Trace Statement

End Statement

A module is closed with the end statement:

- End Statement

Other Elements

Directives can be placed anywhere needed in a design:

- Directives

Header

Module Statement

Keyword: `module`

The Module statement is a required element of the source file. The **Module** statement defines the beginning of the module and must be paired with an **End** statement. The **Module** statement also indicates whether any dummy arguments are used.

Options

Keyword: `options`

The **Options** statement can be used to control processing of the source file by the language processor, and is optional.

Title

Keyword: `title`

The title is optional and is used to describe the 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.

Declarations

The declarations section of a module specifies the names of signals used in the design, and an optional device declaration. 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 at least one 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 their respective subsections.

Declarations Keyword

Keyword: `declarations`

This keyword allows declarations (such as sets or other constants) to be declared in any part of the source file.

Device Declaration

Keyword: `device`

```
device_id DEVICE real_device ;
```

The Device declaration is optional, and only one can be made per module. It associates a device identifier with a specific programmable logic device.

Signal Declarations

The Pin and Node declarations are made to declare signals used in the design, and optionally to associate pin and/or node numbers with those signals. Actual pin and node numbers do not have to be assigned until you want to process the design with Fuseasm and/or JEDSim. Attributes can be assigned to signals within pin and node declarations with the Istype statement. Dot extensions can also be used in equations to precisely describe the signals; see "Dot Extensions" under "Logic Descriptions" later in this chapter.

Note: Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later in the process by a fitter (see the SmartPart User Manual if you have this option).

Pin Declarations

Keyword: `pin`

```
[ ! ]pin_id [, [ ! ]pin_id...] PIN [pin# [,pin# ] ]  
[ISTYPE 'attributes' ] ;
```

See "Attribute Assignment" below.

Node Declarations

Keyword: `node`

```
[ ! ]node_id [, [ ! ]node_id...] NODE [node# [,node# ] ]  
[ISTYPE 'attributes' ] ;
```

See "Attribute Assignment" below.

Attribute Assignment

Keyword: `istype`

```
signal [,signal]... ISTYPE 'attributes' ;
```

The ISTYPE statement defines attributes (characteristics) of signals for devices with programmable characteristics or when no device and pin/node number has been specified for a signal. Even when a device has been specified, using attributes will make it more likely that the design will operate consistently if the device is changed later. ISTYPE can be used after pin or node declarations.

Attributes may be entered in uppercase, lowercase or mixed-case letters. Table 3-9 summarizes the attributes and their use. Each attribute is discussed in more detail in the chapter "Language Reference" under 'attr'.

Table 3-9
Attributes

Attribute	Meaning
'buffer'	The target architecture does not have an inverter between the associated flip-flop (if any) and the actual output pin.
'com'	The signal is combinatorial.
'invert'	The target architecture has an inverter between the associated flip-flop (if any) and the actual output pin.
'neg'	Complement signal prior to processing into sum-of-products equations. 'Neg' is typically used in combination with the -reduce fixed option to control the polarity of a target device with programmable polarity.
'pos'	Do not complement signal prior to processing [default].
'reg'	A clocked memory element (generic flip-flop).
'reg_D'	A clocked memory element (D-type flip-flop).
'reg_T'	A clocked memory element (T-type flip-flop).
'reg_SR'	A clocked memory element (SR-type flip-flop).
'reg_JK'	A clocked memory element (JK-type flip-flop).
'reg_G'	A memory element (D-type flip-flop with a gated clock).
'xor'	The target architecture has an XOR gate, so one top-level exclusive-OR operator is retained in the design equations.

Constant Declarations

Keyword: **=**

```
id [, id]... = expr [, expr]... ;
```

A constant is an identifier that retains a constant value through a module. CONSTANT declarations use the = sign for their keyword and must be in a declarations section or after the @CONST directive. Refer to "Special Constants" earlier in this chapter.

Macro Declarations

Keyword: **macro**

```
macro_id MACRO [(dummy_arg [, dummy_arg]... )] {block} ;
```

The macro declaration statement defines a macro. Macros are used to include ABEL-HDL code in a source file without typing or copying the code everywhere it is needed.

Library Declaration

Keyword: **library**

```
LIBRARY 'name'
```

The LIBRARY statement causes the contents of the indicated file to be extracted from the **abel4lib.inc** library and be inserted in the ABEL-HDL source file. The insertion begins where the LIBRARY statement is located.

Logic Description

One or more of the following elements can be used to describe your design.

- Equations
- Truth Tables
- State Descriptions
- Fuses
- XOR Factors

Dot Extensions

Signal dot extensions, like attributes, are a way to more precisely describe the behavior of a circuit in a logic description that may be targeted to a variety of different architectures. The dot extensions supported in ABEL-HDL can be applied in complex language situations such as in nested sets or complex expressions.

Syntax:

signal_name . ext

Dot extensions can be generally classified into those that are architecture-specific, and those that are generalized for all architectures. The dot extensions listed in Table 3-10 are supported in ABEL-HDL:

Table 3-10
Dot Extensions

Dot Extension	Description
Architecture-Independent	
.CLK	Clock input to an edge-triggered flip-flop
.OE	Output enable
.PIN	Pin feedback
.FB	Register feedback
Architecture-specific	
.D	Data input to a D-type flip-flop
.J	J input to a JK-type flip-flop
.K	K input to a JK-type flip-flop
.S	S input to an SR-type flip-flop
.R	R input to an SR-type flip-flop
.T	T input to a T-type (toggle) flip-flop
.Q	Register feedback
.PR	Register preset
.RE	Register reset
.AP	Asynchronous register preset
.AR	Asynchronous register reset
.SP	Synchronous register preset

Dot Extension	Description
.SR	Synchronous register reset
.LE	Latch-enable input to a latch
.LH	Latch-enable (high) to a latch
.LD	Register load input
.CE	Clock-enable input to a gated-clock flip-flop
.FC	Flip-flop mode control

Equations

Keyword: `equations`

Equations

```
[ WHEN condition      THEN ] [ ! ] element=expression;
[ ELSE equation ];
      or
[ WHEN condition      THEN ] equation; [ ELSE equation];
```

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. The equations describe the logic design. (You can also express the design by means of truth tables or state diagram.) See "Operators, Expressions and Equations" earlier in this chapter and "When-Then-Else" in the "Language Reference."

Truth Tables

Keyword: `truth_table`

```
TRUTH_TABLE (inputs -> outputs )
inputs -> outputs ;
:
      or
TRUTH_TABLE (inputs [:> registered outputs] [-> outputs ] )
```

Truth tables specify outputs as functions of different input combinations in a tabular form. (You can also express the design by means of Boolean equations or state diagrams.) See also "@DCSET" under "@directive" in the "Language Reference."

State Descriptions

Keyword: `state_diagram`

```
STATE_DIAGRAM state_reg
[-> state_out]
[STATE state_exp : [equation]
[equation]
:
:
:
trans_stmt ...]
```

The `State_Diagram` section contains the state descriptions required to describe the logic design. (You can also express the design by means of equations or truth tables.)

The specification of a state description requires the use of the `State_diagram` syntax, which defines the state machine, and the `If-Then-Else`, `Case`, and `Goto` statements which determine the operation of the state machine.

See also "With-endwith" in the "Language Reference."

Fuse Declarations

Keyword: `fuses`

FUSES

```
fuse_number = fuse value ;  
or  
fuse_number_set = fuse value ;
```

The FUSES section of the source file provides a means for explicitly declaring the state of any fuse in the associated device.

XOR Factors

Keyword: `XOR_Factors`

XOR_Factors

```
signal name = xor_factors
```

The XOR_Factors section allows you to specify a Boolean expression that is to be factored out of, and XORed with, the sum-of-products reduced equations. This can result in dramatic reductions in the size of the reduced equations when the design is implemented in a device featuring XOR gates.

Test Vectors Section

Test Vectors

Keyword: `test_vector`

```
Test_vectors [note ]  
(inputs -> outputs)  
  
[invalues -> outvalues ; ]  
:
```

Test vectors are optional and are used to verify the functionality of the logic design. Test vectors specify the expected functional operation of a logic device by defining its outputs as a function of its inputs. Design test vectors can be used in conjunction with test vectors generated by PLDtest Plus, which functionally test the programmed device.

Trace Statement

Keyword: `trace`

```
trace (inputs -> outputs) ;
```

The Trace statement is used to control the display features of PLASim and JEDSim.

End Statement

Keyword: `end`

```
end module_name
```

The End statement ends the module.

Other Elements

Directives

Keyword: @

`@directive [options]`

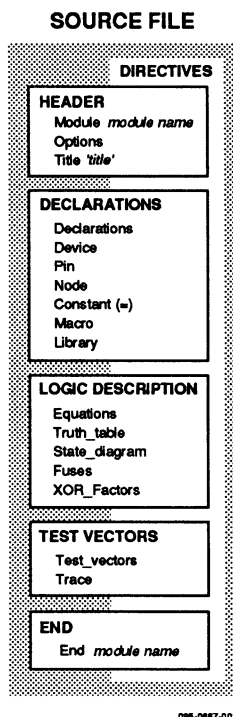
Directives provide many options that can affect the contents of a source file when it is processed. Sections of ABEL-HDL source code can be included conditionally, code can be brought into a source file from another file, and messages can be printed during the processing of a source file.

Some directives take arguments that are used to determine conditions. These arguments can be actual arguments or dummy arguments preceded by a question mark. The rules applying to actual and dummy arguments are presented under "Arguments" earlier in this chapter.

Available directives are listed below. See @ in the "Language Reference" for complete information.

@ALTERNATE	@IFNIDEN
@CONST	@INCLUDE
@DCSET	@IRP
@EXPR	@IRPC
@EXIT	@MESSAGE
@IF	@ONSET
@IFB	@PAGE
@IFDEF	@RADIX
@IFIDEN	@REPEAT
@IFNB	@STANDARD
@IFNDEF	

4 Language Reference



This chapter gives detailed information about each of the language elements in ABEL-HDL. This chapter assumes you are familiar with the basic syntax discussed in the chapter "ABEL-HDL Language Structure." Each entry contains the following sections, if applicable:

- **Source File Map** — This map highlights the section of the source file the element is in. If no section is highlighted, the element is applicable to more than one section. An example map is shown at the left.
- **Syntax** — The required syntax for the element.
- **Purpose** — A brief description of the intended use of the element.
- **Usage** — A discussion of the potential uses of the element, including any considerations that need to be addressed when using the element.
- **Examples** — Examples of the element as used in a design description.
- **See Also** — References to other elements and discussions, and to design examples that demonstrate the use of the element.

Basic syntax information on subjects such as blocks, strings, sets and arguments are provided in the chapter "ABEL-HDL Language Structure."

.ext — Dot Extensions

Syntax

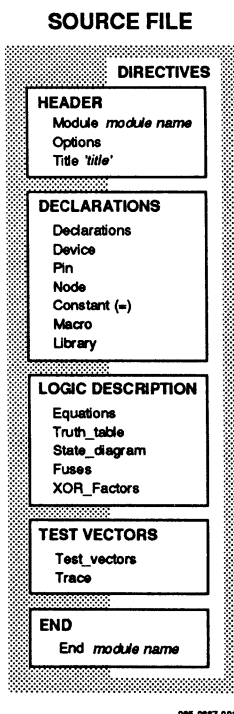
signal_name.ext

Purpose

Dot extensions provide a means to refer specifically to internal signals and nodes associated with a primary signal in your design.

Usage

Signal dot extensions more precisely describe the behavior of a circuit in a logic description. Dot extensions are applied to signals and are used to remove the ambiguities in equations.



The dot extensions supported in ABEL-HDL can be applied in complex language situations such as in nested sets or complex expressions.

Using Detailed vs. Architecture-independent Dot Extensions:

ABEL-HDL contains a comprehensive set of signal extensions (dot extensions) that can be used to more precisely specify the meaning of a design description. These dot extensions allow you to refer to various circuit elements (such as register clocks, presets, feedback and output enables) that are related to a primary signal.

Some of the dot extensions supported in ABEL-HDL are general purpose, and are intended for use with a wide variety of device architectures. These dot extensions are therefore referred to as "architecture-independent". Other dot extensions are intended for specific classes of device architectures, or require specific device configurations. These dot extensions are called "architecture-dependent" or "detailed" dot extensions.

In most cases it is possible to describe a circuit using either detailed or architecture-independent dot extensions. Which form you will use will depend on the application, and whether you want the application to be capable of being implemented in a wide variety of device architectures. The advantages of each method will be discussed later in this section.

Table 4-1 lists the dot extensions supported in ABEL-HDL.
Architecture-independent dot extensions are indicated as such.

Table 4-1
Dot Extensions

Dot Ext.	Arch. Indep. ¹	Description
.AP		Asynchronous register preset
.AR		Asynchronous register reset
.CE		Clock-enable input to a gated-clock flip-flop
.CLK ²	●	Clock input to an edge-triggered flip-flop
.D ²		Data input to a D-type flip-flop
.FB	●	Register feedback
.FC		Flip-flop mode control
.J		J input to a JK-type flip-flop
.K		K input to a JK-type flip-flop
.LD		Register load input
.LE		Latch-enable input to a latch
.LH		Latch-enable (high) to a latch
.OE ²	●	Output enable
.PIN	●	Pin feedback
.PR ²		Register preset (synchronous or asynchronous)
.Q		Register feedback
.R		R input to an SR-type flip-flop
.RE ²		Register reset (synchronous or asynchronous)
.S		S input to an SR-type flip-flop
.SP		Synchronous register preset
.SR		Synchronous register reset
.T		T input to a T-type (toggle) flip flop
¹ Dot extension is architecture-independent if this column is marked. Otherwise it is architecture-specific. ² Example follows.		

Table 4-2 shows the dot extensions allowable and required for different register types.

Table 4-2
Dot Extensions for Architectures

Register Type	Allowable Extensions	Definition
combinatorial logic (no register)	none .oe (opt) .pin (opt)	output output enable pin
D-type flip-flop	.clk (req) .d (req) .fc (opt) .oe (opt) .q (opt) .sp (opt) .sr (opt) .ap (opt) .ar (opt) .pin (opt)	clock data input flip-flop mode control output enable f-f output sync preset sync reset async preset async reset pin
JK-type flip-flop	.clk (req) .j (req) .k (req) .fc (opt) .oe (opt) .q (opt) .sp (opt) .sr (opt) .ap (opt) .ar (opt) .pin (opt)	clock j input k input flip-flop mode control output enable f-f output sync preset sync reset async preset async reset pin
SR-type flip-flop	.clk (req) .s (req) .r (req) .oe (opt) .q (opt) .sp (opt) .sr (opt) .ap (opt) .ar (opt) .pin (opt)	clock set input reset input output enable f-f output sync preset sync reset async preset async preset pin
T-type flip-flop	.clk (req) .t (req) .oe (opt) .q (opt) .sp (opt) .sr (opt) .ap (opt) .ar (opt) .pin (opt)	clock toggle input output enable f-f output sync preset sync reset async preset async reset pin
G-type latch	.d (req) .le or .lh (req) .oe (opt) .q (opt) .pin	data input latch enable output enable f-f output pin

Figures 4-1 through 4-8 show the effect of each dot extension:

Figure 4-1
Pin-to-pin Dot Extensions in an
Inverted Output Architecture

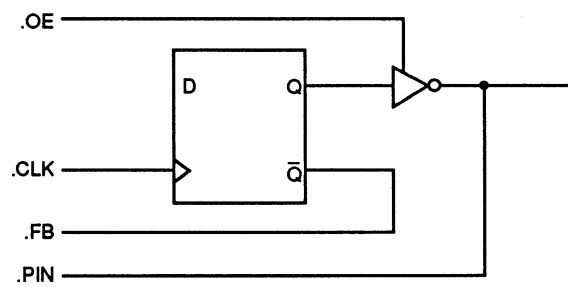


Figure 4-2
Pin-to-pin Dot Extensions in a
Non-inverted Output Architecture

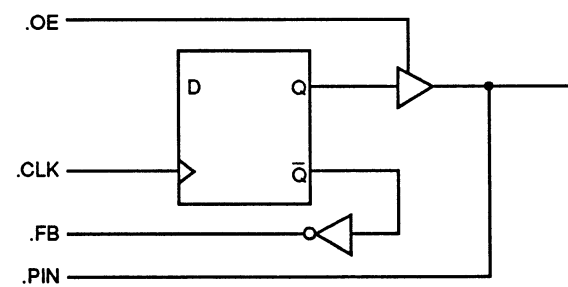


Figure 4-3
Detailed Dot Extensions for a
D-type Flip-flop Architecture

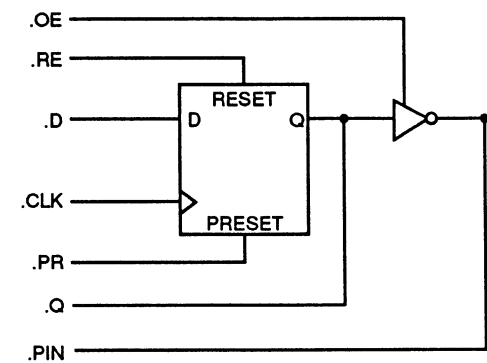


Figure 4-4
Detailed Dot Extensions for a
T-type Flip-flop Architecture

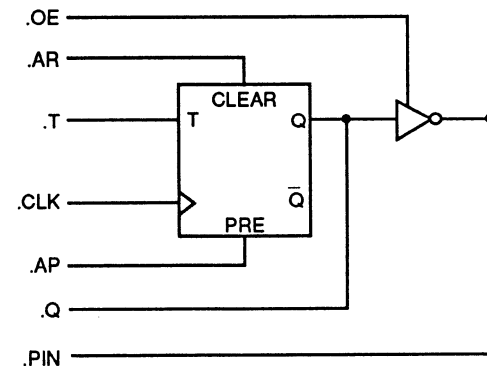


Figure 4-5
Detailed Dot Extensions for an
RS-type Flip-flop Architecture

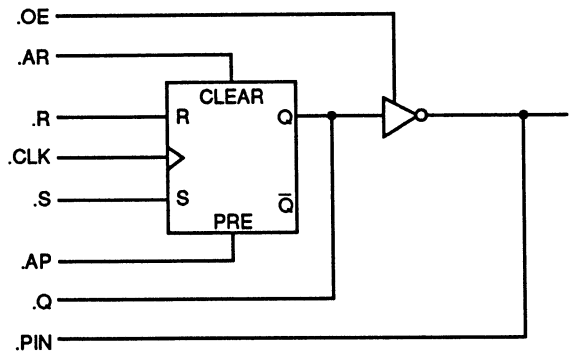


Figure 4-6
Detailed Dot Extensions for a
JK-type Flip-flop Architecture

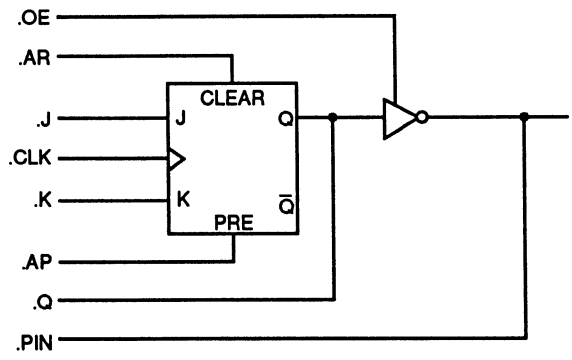
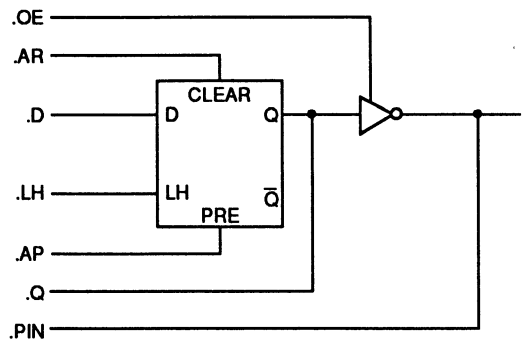
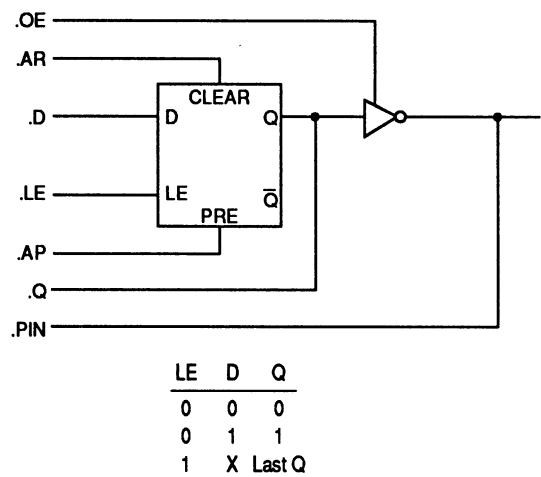


Figure 4-7
Detailed Dot Extensions for a
Latch with Active High Latch
Enable



LH	D	Q
1	0	0
1	1	1
0	X	Last Q

Figure 4-8
Detailed Dot Extensions for a Latch with Active Low Latch Enable



Examples

These equations precisely describe the desired circuit as being a toggling D-type flip-flop that is clocked by the input Clock:

```
foo.clk = Clock;  
foo.D = !foo.Q # Preset;
```

Register preset:

```
Q2.PR = S & !T;
```

Register reset:

```
Q2.RE = R & !T;
```

Three-state Output Enables

Output enables are described in ABEL-HDL with the `.oe` dot extension applied to an output signal name. For example,

```
foo.oe = !enab;
```

The equation specifies that the input signal `enab` is to be used to control the output enable for an output signal named `foo`.

Note that explicitly stating the value of a fixed output enable restricts the device fitters' ability to map the indicated signal to a simple input pin instead of a three-state I/O pin.

See Also

`'attr'`
"Attributes" in the chapter "ABEL-HDL Language Structure"
"Dot Extensions" in the chapter "Design Considerations"

= — Constant Declarations

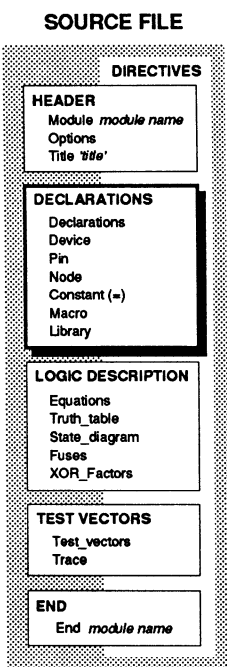
Syntax

```
id [,id ]... = expr [,expr ]... ;
```

Purpose

A constant declaration defines constants used in a module.

Usage



085-0712-001

id an identifier naming a constant to be used within a module

expr an expression defining the constant value

Note: The equal sign (=) used for constant declarations in the Declarations section is also used for equations in the Equations section. See "Operators" in "ABEL-HDL Language Structure."

A constant is an identifier that retains a constant value throughout a module.

The identifiers on the left side of the equals sign are assigned the values listed on the right side. There is a one-to-one correspondence between the identifiers and the expressions listed: there must be one expression for each identifier.

The ending semicolon is required after each declaration.

Constants are useful when a value is used many times in a module, especially when the value may be changed during the 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;
```

An include file, constants.inc, supplied with ABEL software contains definitions for the most frequently used ABEL-HDL constants. This file may be included in your design with the command:

```
Library 'constants' ;
```

Examples

```

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

```


Using Intermediate Expressions

Intermediate (constant) expressions can be used in the declarations section to reduce the number of output pins required to implement multi-level functions. Intermediate expressions may increase the number of product terms required per output. For example,

```
Declarations
  TMP1 = [A3..A0] == [B3..B0];
  TMP2 = [A7..A4] == [B7..B4];
```

```
Equations
  F = TMP1 & TMP2;
```

translates into

```
F = (A7 !$ B7) & (A6 !$ B6) & (A5 !$ B5) & (A4 !$ B4)
    (A3 !$ B3) & (A2 !$ B2) & (A1 !$ B1) & (A0 !$ B0);
```

which requires one output with 16 product terms. However, the following:

```
Declarations
  TMP1,TMP2 pin 18,19

Equations
  TMP1 = [A3..A0] == [B3..B0];
  TMP2 = [A7..A4] == [B7..B4];
  F = TMP1 & TMP2;
```

translates into

```
TMP1 =(A3 !$ B3) & (A2 !$ B2)
      (A1 !$ B1) & (A0 !$ B0);

TMP1 =(A7 !$ B7) & (A6 !$ B6)
      (A5 !$ B5) & (A4 !$ B4)

F = TMP1 & TMP2;
```

which requires three outputs with less than 8 product terms per output.

See Also

Declarations
Equations "Special Constants" in the chapter "ABEL-HDL Language Structure"

'attr' — Signal Attributes

Syntax

```
signal_id [, signal_id ] ISTYPE 'attr';
or
signal_id [, signal_id ] [(PIN|NODE) [## [,## ] ] ]
ISTYPE 'attr';
```

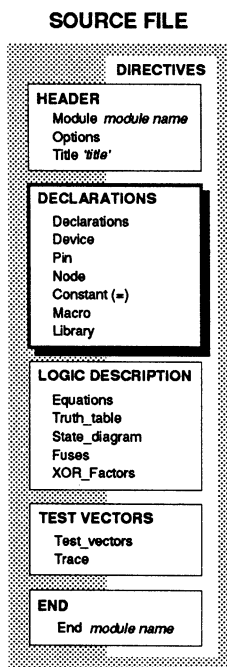
Purpose

Signal attributes remove ambiguities in architecture-independent designs. Even when a device has been specified, using attributes will make it more likely that the design will operate consistently if the device is changed later.

Usage

Table 4-3 summarizes the available attributes.

Table 4-3
Attributes



085-5712-001

Dot Ext.	Arch. Indep.	Description
'buffer'		No Inverter in Target Device
'com'	●	Combinatorial
'invert'		Inverter in Target Device
'neg'	●	Complement Signal
'pos'	●	Positive Polarity
'reg'	●	Clocked Memory Element
'reg_d'		D Flip-flop Clocked Memory Element
'reg_g'		D Flip-flop Gated Clock Memory Element
'reg_jk'		JK Flip-flop Clocked Memory Element
'reg_sr'		SR Flip-flop Clocked Memory Element
'reg_t'		T Flip-flop Clocked Memory Element
'xor'		XOR Gate in Target Device

'buffer'	The target architecture does not have an inverter between the associated flip-flop (if any) and the actual output pin.
'invert'	<p>The target architecture has an inverter between the associated flip-flop (if any) and the actual output pin.</p> <p>Control of output inversion in devices is accomplished through the use of the 'invert' or 'buffer' attributes. These attributes enforce the existence ('invert') or non-existence ('buffer') of a hardware inverter at the device pin associated with the output signal specified.</p> <p>In registered devices, the 'invert' attribute ensures that an inverter will be located between the output pin and its associated register output. This is important because the location of the inverter affects a register's reset, preset, preload and powerup behavior as observed on the associated output pin.</p>
'com'	Specifies a combinatorial symbol.
'neg'	<p>Complement signal prior to processing into sum-of-products equations.</p> <p>'Neg' is typically used in combination with the -reduce fixed option to control the polarity of the final design equations.</p> <p>Designs that take an unusual amount of time to process (in AHDL2PLA's matrix conversion steps) may benefit from the inclusion of the 'neg' attribute for the design output signals.</p>
'pos'	<p>Do not complement signal prior to processing into sum-of-products equations.</p> <p>'Pos' indicates that the associated input or output has positive polarity. The device will be programmed to reflect this condition if -reduce fixed is specified, and any equations associated with this signal will be optimized for that polarity.</p>
'reg'	The signal specified is a registered output. Equations, state diagrams and truth tables will generate logic for a D-type flip-flop, normalized to take into account any inverters in the target device.
'reg_d'	The signal specified is a registered output. Equations, state diagrams and truth tables will generate logic for a D-type flip-flop, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer').
'reg_g'	The signal specified is a registered output. Equations, state diagrams and truth tables will generate logic for a D-type flip-flop, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Equations and truth tables must be written using the .D and .CE dot extensions when this attribute is used.
'reg_jk'	The signal specified is a JK-type registered output. State diagrams will generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Equations and truth tables must be written using the .J and .K dot extensions when this attribute is used.

'reg_sr'	The signal specified is an SR-type registered output. State diagrams will generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Equations and truth tables must be written using the .S and .R dot extensions when this attribute is used.
'reg_t'	The signal specified is a T-type registered output. State diagrams will generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Equations and truth tables must be written using the .T dot extension when this attribute is used.
'xor'	The signal specified will be implemented using an XOR gate fed by two sum-of-products logic circuits. If XOR operators are used in the design equations for this output (or if high-level operators are used that result in XOR operations), then one XOR operator will be retained through optimization. Use this attribute if you will be implementing your design in an architecture featuring XOR gates.

See Also

Istype
 .ext
 "Attributes" and "Dot Extensions" in the chapter "ABEL-HDL Language Structure"
 "Architecture-independent Designs" and "XOR Factor" in the chapter "Design Considerations"

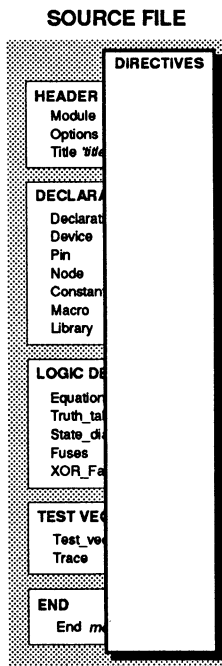
@directive — Directives

Purpose

Directives affect the contents of a source file when it is processed. Sections of ABEL-HDL 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. The available directives are given on the following pages.

Usage

Directives allow you to use more complex structures. See the chapter "Using Advanced Features" for more information.



095-0710-001

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 "ABEL-HDL Language Structure."

When debugging your source file, use the -list and -list expand options to AHDL2PLA to examine compiled and expanded source code created by directives. The -list expand option will show text included by directives and lists the directives that caused the code to be added to the source file.

@Alternate — Alternate Operator Set

Syntax

@alternate

Usage

@ALTERNATE brings an alternate set of operators into effect that replace the normal ABEL-HDL operators. This is for users who feel more comfortable with the alternate set because of their familiarity with operators used in other languages.

The alternate operators remain in effect until the @STANDARD directive is issued or the end of the module is reached.

Note that the use of the alternate operator set precludes use of the ABEL-HDL addition, multiplication and division operators because they represent the OR, AND and NOT logical operators in the alternate set. The !, &, #, \$ and !\$ will still work when @Alternate is in effect.

The alternate operator set is listed in Table 4-4.

Table 4-4
Alternate Operator Set

ABEL-HDL Operator	Alternate Operator	Description
!	/	NOT
&	*	AND
#	+	OR
\$:+:	XOR
!\$:*:	XNOR

See Also

@STANDARD

@Const — Constant Declarations

Syntax

@const **id** = **expression** ;

id a valid identifier

expression a valid expression

Usage

@CONST allows new constant declarations to be made in a source file outside the normal (and required) declarations section.

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 (that is, a macro, pin, device).

Examples

```
@CONST count = count + 1;
```

See Also

= (Constant Declarations)
"Special Constants" in the chapter "ABEL-HDL Language Structure"

@Dcset — Don't Care Set

Syntax `@dcset`

Usage ABEL-HDL uses don't-care conditions to help in the optimization of partially-specified logic functions. Partially specified logic functions are those that have less than 2^n significant input conditions, where n is the number of input signals. The @dcset option allows the optimization to use 1 or 0 for don't cares to optimize these functions.

See Also @ONSET
Truth_table
"@DCSET Considerations and Precautions" in the chapter "Design Considerations"

@Exit — Exit Directive

Syntax `@exit`

Usage The @exit directive causes AHDL2PLA 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

Usage @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.

@Expr can contain variable values and be used in loops with @Repeat.

Examples `@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 — If Directive

Syntax

```
@if expression {block }
```

expression a valid expression

block a valid block of text

Usage

@IF is used to include sections of ABEL-HDL 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.

Examples

```
@if (A > 17) { C = D $ F ; }
```

@Ifb — If Blank Directive

Syntax

```
@IFB (arg) {block }
```

arg an actual or dummy argument preceded by a "?"

block a valid block of text

Usage

@IFB includes the text contained within the block if the argument is blank (has 0 characters).

Examples

```
@IFB ()  
{text here is included with the rest of the source file.}  
  
@IFB ( hello ) { this text will not be included }  
  
@IFB (?A) {this text is included if no value is  
substituted for A. }
```

See Also

"Arguments and Argument Substitution" in the chapter "ABEL-HDL Language Structure"

@Ifdef — If Defined Directive

Syntax

```
@ifdef id {block }
```

id an identifier

block a valid block of text

Usage

@IFDEF includes the text contained within the block if the identifier is defined.

Examples

```
A pin 5 ;  
@ifdef A { Base = ^hE000 ; }  
"the above assignment is made  
because A was defined"
```


@Ifiden — If Identical Directive

Syntax

```
@ifiden (arg1,arg2 ) {block }
```

arg1,2 actual arguments, or dummy argument names preceded by "?"

block a valid block of text

Usage

The text in the block is included if arg1 and arg2 are identical.

Examples

```
@ifiden (?A,abcd) { ?A device 'P16R4'; }
```

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 (arg) {block }
```

arg an actual argument, or a dummy argument name preceded by a "?"

block a valid block of text

Usage

@IFNB includes the text contained within the block if the argument is not blank, meaning that it has more than 0 characters.

Examples

```
@IFNB () { ABEL-HDL source here is not included with the  
rest of the source file. }
```

```
@IFNB ( hello ) { this text will be included }
```

```
@IFNB (?A) {this text is included if a value is  
substituted for A}
```

@Ifndef — If Not Defined Directive

Syntax

```
@ifndef id {block }
```

id an identifier

block a valid block of text

Usage

@IFDEF 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.

Examples

```
@ifndef A{Base=^hE000;}  
"if A is not defined, the block is inserted in the text"
```

@Ifniden — If Not Identical Directive

Syntax

```
@ifniden (arg1,arg2 ) {block }
```

arg1,2 actual arguments, or dummy argument names
 preceded by "?"

block a valid block of text

Usage

The text in the block is included in the source file if arg1 and arg2 are not identical.

Examples

```
@ifniden (?A,abcd) { ?A device 'P16R8'; }
```

A device declaration for a P16R8 is made if the actual argument substituted for A is not identical to abcd.

@Include — 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

Usage

@INCLUDE causes the contents of the file identified by the file specification to be placed in the ABEL-HDL 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 the default drive or path.

Examples

```
@INCLUDE 'macros.abl'     "file specification
```

See Also

Library

@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

Usage

@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.

Examples

```
@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. Note that 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

```
@irpc dummy_arg (arg) {block }
```

dummy_arg a dummy argument

arg an actual argument, or a dummy argument name
 preceded by a "?"

block a block

Usage

@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.

Examples

```
@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 — Message Directive

Syntax

```
@message 'string'
```

string any valid string

Usage

@Message sends the message specified in *string* to your monitor. This can be used to monitor the progress of the parsing step of the language processor in AHDL2PLA, or as an aid to debugging complex sequences of directives.

Examples

```
@message 'Includes completed'
```

@Onset — Don't Care's

Syntax

```
@onset
```

Usage

The @onset directive disables the use of don't care input conditions for optimization.

See Also

@Dcset

@Page — Page Directive

Syntax `@page`

Usage Send a form feed to the parser listing file. If no listing is being created, `@page` has no effect.

@Radix — Default Base Numbering Directive

Syntax `@radix expr ;`
expr a valid expression that produces the number 2, 8, 10 or 16 to indicate a new default base numbering.

Usage The `@Radix` directive is used to change the default base numbering system. The default is base 10 (decimal). This directive is useful when many numbers need to be specified in a base other than 10. 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. (See "Numbers" in the chapter "ABEL-HDL Language Structure.")

The newly specified default base stays in effect until another `@radix` directive is issued or until the end of the module is reached. Note that when a new `@radix` is issued, the specification of the new base must be in the current base format.

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

```
@radix 2 ;           "change default base to binary
@radix 1010 ;        "change from binary to decimal
```

@Repeat — Repeat Directive

Syntax `@repeat expr {block }`
expr a valid expression that produces a number
block a block

Usage `@REPEAT` causes the block to be repeated *n* times, where *n* is specified by the constant expression.

Examples The following use of the repeat directive,

```
@repeat 5 {H,}
```

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 example files.

@Standard — Standard Operators Directive

Syntax

`@standard`

Usage

The `@standard` option resets the operators in effect back to the ABEL-HDL standard operators from the alternate set. The alternate set is chosen with the `@alternate` directive.

Case

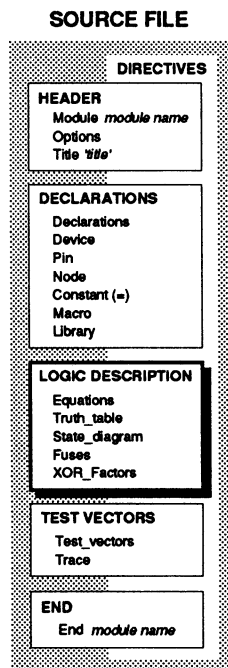
Syntax

```
CASE expression : state_exp;
[ expression : state_exp; ]
[ expression : state_exp; ]
:
ENDCASE ;
```

Purpose

The CASE statement is used under the **State_diagram** section to indicate the transitions of a state machine when there are multiple possible conditions that affect the state transitions.

Usage



expression any valid ABEL-HDL expression

state_exp an expression identifying the next state, optionally followed by WITH-ENDWITH transition equations.

Case statements can be nested with If-Then-Else, GOTO and other CASE statements.

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.

Examples

```
case a == 0 : 1 ;
      a == 1 : 2 ;
      a == 2 : 3 ;
      a == 3 : 0 ;
endcase ;
```

See Also

State_diagram
Goto
If-then-else
With-endwith

Constant Declarations

See =.

Declarations

Syntax

Declarations *declarations*

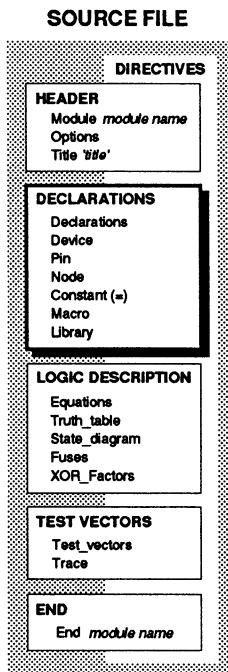
Purpose

The declarations keyword allows declarations (such as sets or other constants) to be declared in any part of the ABEL-HDL source file.

Usage

declarations Any valid declarations can be given after the **Declarations** keyword.

The Declarations keyword is not necessary for declarations immediately following the module, options, and/or title statement(s).



095-0712-001

Examples

An example of declared equations is shown below:

```
module DECLARE
    declare device    'P16V8C';
    A,B,Out1         pin 1,2,15;

    Equations
        Out1 = A & B;

    Declarations
        C,D,E,F,Out2    pin 3,4,5,6,16;
        Temp1 = C & D;
        Temp2 = E & F;

    Equations
        Out2 = Temp1 # Temp2;
end;
```

See Also

demo1800.abl

Device

Syntax

device_id **DEVICE** *real_device* ;

Purpose

The device declaration statement associates the device name used in a module with an actual programmable logic device on which designs are implemented.

Usage

device_id an identifier used for the programmer load filenames

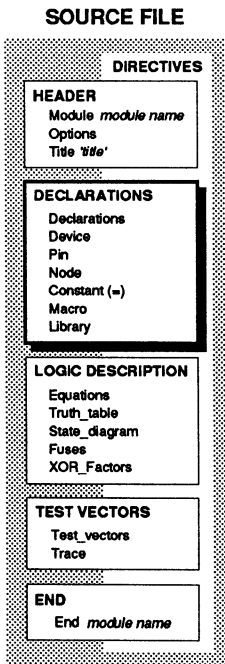
real_device a string describing the architecture name of the real device represented by *device_id*

The device declaration is optional.

Device identifiers used in device declarations should be valid filenames since JEDEC files are created by appending the extension .jed to the identifier. The architecture name of the programmable logic device is indicated by the string, *real_device*.

The device specified by *real_device* must be supported. Use the **finddev4** utility described in the *User Notes* to get the appropriate device name for your device.

The ending semicolon is required.



085-0712-001

Examples

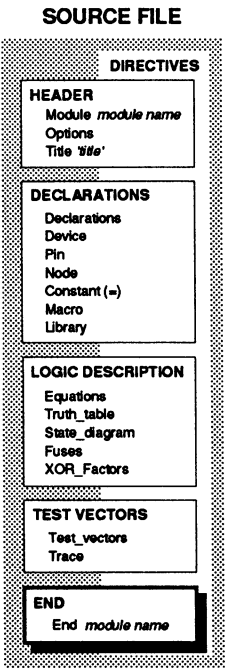
D1 DEVICE 'P16R4' ;

End

Syntax `end module_name`

Purpose The **end** statement denotes the end of the module.

Usage The end statement can be followed by the module name. For multi-module source files, the module name is required.



085-0715-001

Equations

Syntax

```

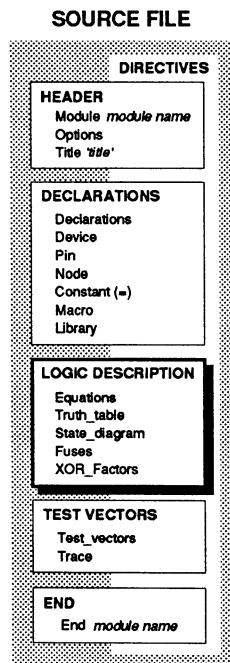
equations
[ WHEN condition THEN ] [ ! ] element=expression;
[ ELSE equation ];
      or
[ WHEN condition THEN ] equation;
[ ELSE equation];

```

Purpose

The equations statement defines the beginning of a group of equations associated with a device.

Usage



085-0713-001

condition any valid expression

element an identifier naming a signal or set of signals, or an actual set, to which the value of the expression will be assigned

expression any valid expression

= and := combinatorial and registered assignment operators

Equations specify logic functions with an extended form of Boolean algebra.

An ending semicolon is required after each equation.

The equations following the equation statement are any valid ABEL-HDL equations as described in "ABEL-HDL Language Structure."

Examples

A sample equations section follows:

```

equations
A = B & C # A ;
[W,Y] = 3 ;
!F = (B == C) ;

```

See Also

When-Then-Else

Module

State_diagram

Truth_table

"Operators, Expressions and Equations" in the chapter "ABEL-HDL Language Structure"

Fuses

Syntax

```
FUSES
fuse_number = fuse_value ;

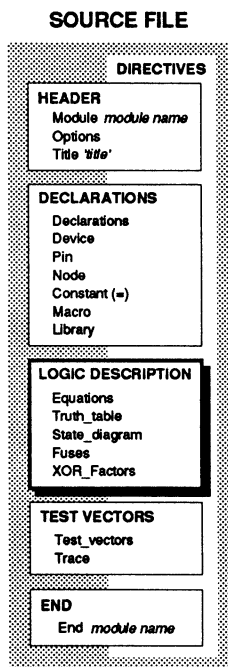
or

fuse_number_set = fuse_value ;
```

Purpose

The **fuses** section of the source file provides a means for explicitly declaring the state of any fuse in the associated device.

Usage



085-0713-001

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)

The **fuses** statement provides device-specific information, and precludes changing devices without editing the statement in the source file.

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.

Caution: *When fuse states are specified using the FUSES-section syntax, the resulting fuse values supersede the fuse values obtained through the use of equations, truth tables and state diagrams, and will affect device simulation accordingly.*

The PC versions have a limit of 32 fuses per statement.

Examples

```
FUSES
3552 = 1 ;
[3478...3491] = ^Hff;
```

See Also

cnt10rom.abl

Goto

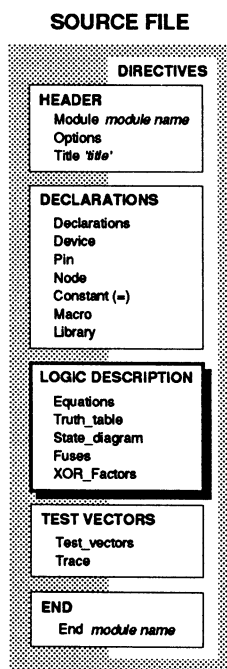
Syntax

```
GOTO state_exp ;
```

Purpose

The **GOTO** statement is used under the **State_diagram** section to cause an unconditional transition to the state indicated by *state_exp*.

Usage



085-0713-001

state_exp an expression identifying the next state, optionally followed by **WITH_ENDWITH** transition equations.

GOTO statements can be nested with If-Then-Else and CASE statements.

Examples

```
GOTO 0 ; "goto state 0
GOTO x+y ; "goto the state x + y
```

See Also

State_diagram
Case
If-then-else
With-endwith

If-Then-Else

Syntax

IF-THEN-ELSE

```
IF exp THEN state_exp
[ ELSE state_exp ] ;
```

Chained IF-THEN-ELSE

```
IF expr THEN state_exp
ELSE IF exp THEN state_exp
ELSE state_exp ;
```

Nested IF-THEN-ELSE

```
IF exp THEN state_exp
ELSE IF exp THEN
    IF exp THEN state_exp
    ELSE state_exp
ELSE state_exp ;
```

Purpose

The IF-THEN-ELSE statements are used under the **State_diagram** section to describe the progression from one state to another in a state machine.

Usage

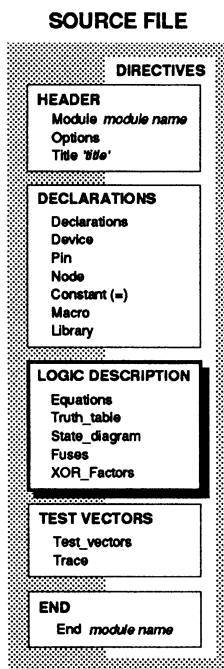
exp any valid expression

state_exp an expression identifying the next state, optionally followed by WITH_ENDWITH transition equations.

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.

If-Then-Else statements can be nested with GOTO, CASE and With-endwith statements.



085-0713-001

Chained IF-THEN-ELSE Statements

Any number of IF-THEN-ELSE statements can be chained, but the final statement must end with a semicolon. 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 where the conditions are not mutually exclusive.

Chained IF-THEN-ELSE statements can provide multiway branching transition logic. For example, multiple IF-THEN-ELSE statements can be chained to describe a three-way branch as follows:

```
STATE S0:
  IF (address < ^h0400)
    THEN S0
  ELSE
    IF (address <= ^hE100)
      THEN S2
    ELSE
      S1;
```

Not that the indenting and formatting of this statement is not significant to its operation: breaking a complex transition statement across many lines and indenting simply improves the readability of the design description.

Nested IF-THEN-ELSE Statements

IF-THEN-ELSE and CASE statements can also be combined and nested.

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

```
if a then 1
  else
if b then 2
  else
if c then 3
  else 0 ;
```

Nested IF-THEN-ELSE Statements

A complex state transition could be written with nested transitions as follows:

```
STATE S0:
  CASE (select == 1): IF (address == ^h0100)
    THEN S16
  ELSE
    IF (address > ^hE100)
      THEN S17
    ELSE
      S0;

  (select == 2): S2;

  (select == 3): IF (address <= ^hE100)
    THEN IF (reset)
      THEN S3;
    ELSE S0;
  ELSE S17;

  (select == 0): S0;
ENDCASE;
```

See Also

State_diagram
Case
Goto
With-endwith

Istype — Attribute Declarations

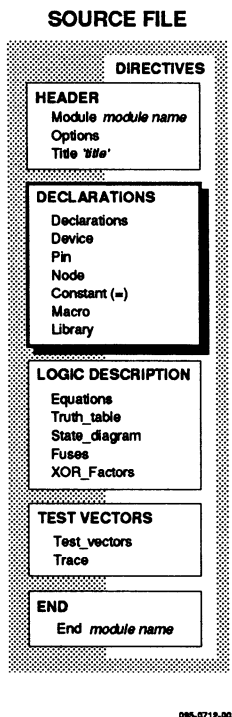
Syntax

```
signal [,signal...] [PIN|NODE [##s ] ]   ISTYPE 'attr
[,attr ]...';
```

Purpose

The ISTYPE statement defines attributes or characteristics of signals (pins and nodes).

Usage



signal

a pin or node identifier

attr

a string that specifies attributes for the signal(s). Valid strings are listed below and described under 'attr' earlier in this chapter.

Signal attributes are specified through the use of the ISTYPE statement. The syntax for signal declarations allow pin or node declarations to be combined with ISTYPE statements in a single declaration. The attributes defined with ISTYPE are used to specify the architectural constraints for signals that have not been assigned to a specific device, and pin or node number, or when a device (and/or pin number) is specified that has programmable characteristics.

When more than one signal is listed on the left side of the ISTYPE statements, all attributes listed on the right side of the ISTYPE statement are applied to each of the signals.

Declarations of the pin and node names used in the ISTYPE statement must be made before the ISTYPE statement.

Valid attributes are

'buffer'	'com'	'invert'
'neg'	'pos'	'reg'
'reg_d'	'reg_g'	'reg_jk'
'reg_sr'	'reg_t'	'xor'

Examples

```
F0, A istype 'neg, reg' ;
```

This declaration statement defines **F0** and **A** as negative polarity latches. Both **F0** and **A** must be defined previously in the module. The following signal declarations are all valid

```
q3,q2,q1,q0  NODE ISTYPE 'reg_SR';
Clk,a,b,c    PIN 1,2,3,4;
reset       PIN;
reset       ISTYPE 'com';
Output      PIN 15 ISTYPE 'reg,invert';
```

See Also

.ext
'attr'

Pin
Node

"Dot Extensions" and "Attribute Assignment" in the chapter
"ABEL-HDL Language Structure"

Library

Syntax

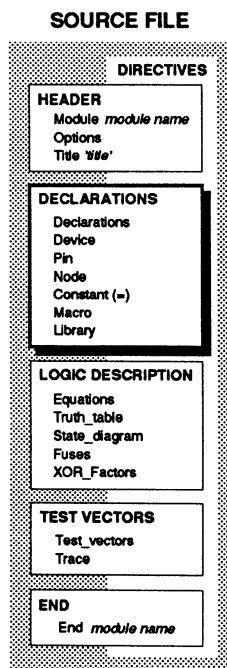
```
LIBRARY 'name'
```

Purpose

The LIBRARY statement causes the contents of the indicated file to be inserted in the ABEL-HDL source file. The insertion begins at the point where the LIBRARY statement is located.

Usage

name a string that specifies the name of the library file, excluding the file extension



085-0712-001

See Also

Module
@Include

Macro

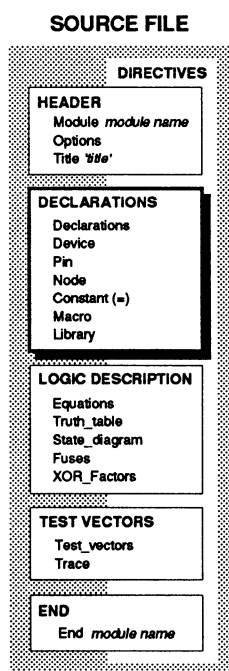
Syntax

```
macro_id MACRO [ (dummy_arg [, dummy_arg ]... ) ] {block} ;
```

Purpose

The macro declaration statement defines a macro. Macros are used to include ABEL-HDL code in a source file without typing or copying the code everywhere it is needed.

Usage



macro_id an identifier naming the macro

dummy_arg a dummy argument

block a block

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 and end-of-lines) is substituted exactly as it appears in the block.

When debugging your source file, you can use the **-list expand** option to examine macro statements. The **-list expand** option causes the parsed and expanded source code and the macros and directives that caused code to be added to the source to be written to the listing file.

Macros and Declared Equations

Declared equations can be used instead of macros for constant expressions, and result in faster processing. The file, **mac.abl**, in Figure 4-9 demonstrates the difference:

Figure 4-9
Differences Between MACRO
and Declared Equations

```

module mac
title 'Demonstrates difference between MACRO and declared equations'
m1 device 'P16H8';
A,B,C,X1,X2,X3 pin 1,2,3,14,15,16;
Y1 macro {B # C};
Y2 = B # C;

equations
X1 = A & Y1;
X2 = A & (Y1);
X3 = A & Y2;

" Note: Because Y1 is a text replacement macro the equation
" for X1 will expand to A & B # C. If the desired function
" was A & (B # C) use parentheses around the macro or use
" a sub expression in the declarations.

" The macro could also be written      Y1 macro {(B # C)};

test_vectors ([A,B,C] -> [X1,X2,X3])
[0,0,0] -> [ 0, 0, 0];
[0,0,1] -> [ 1, 0, 0];
[0,1,0] -> [ 0, 0, 0];
[0,1,1] -> [ 1, 0, 0];
[1,0,0] -> [ 0, 0, 0];
[1,0,1] -> [ 1, 1, 1];
[1,1,0] -> [ 1, 1, 1];
[1,1,1] -> [ 1, 1, 1];

end

```

Examples

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 compiler, AHDL2PLA. 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 a Boolean equation, but it could have been specified using another ABEL-HDL 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 Boolean equation, above. The truth table format is discussed under Truth_table 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 AHDL2PLA program to terminate abnormally with errors. This error can often be detected by examining the 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.

See Also

= (Constant Declarations)
 "Arguments and Argument Substitution" in the chapter "ABEL-HDL Language Structure"

Module

Syntax

```
MODULE modname [ ( dummy_arg [, dummy_arg ] ... ) ]
```

Purpose

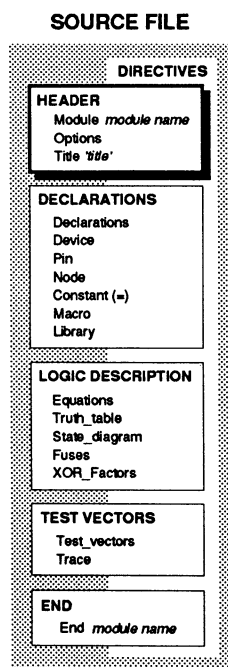
The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.

Usage

modname a valid identifier naming the module

dummy_arg a dummy argument

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.



095-0711-001

Examples

```
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.

See Also

Options	Truth_table
Title	State_diagram
Declarations	Fuses
Equations	Test_vectors
XOR_Factor	End

"Arguments and Argument Substitution" in the chapter "ABEL-HDL Language Structure"

Node

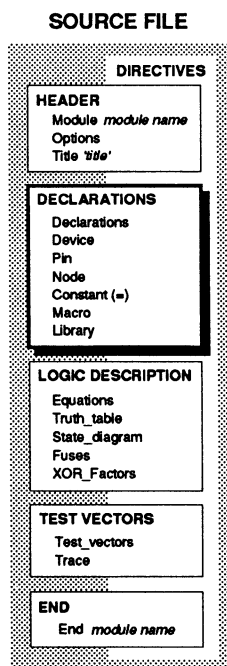
Syntax

```
[!]node_id [, [!]node_id...] NODE [node# [, node# ]] [ISTYPE
'attributes' ];
```

Purpose

The NODE keyword is used to declare those signals that may be assigned to buried nodes within a device.

Usage



085-0712-001

node_id an identifier used for reference to a node in a logic design

node# the node number on the real device

attributes a string that specifies node attributes for devices with programmable nodes. Any number of valid attributes can be listed, separated by commas. Attributes are listed in Table 4-9 under "Attributes" in "ABEL-HDL Language Structure."

Note: The use of the NODE keyword does not restrict a signal to a buried node. It is quite possible that a signal declared with the NODE keyword will be assigned to a device I/O pin by a device fitter.

The ending semicolon is required after each declaration.

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.

The ending semicolon is required. The following example declares three nodes A, B, and C.

```
A, B, C NODE ;
```

The node attribute string, *attributes*, specifies node attributes. Attributes can be defined in this way with the ISTYPE statement. The ISTYPE statement and attributes are discussed later in this chapter.

The node declaration,

```
B NODE istype 'reg' ;
```

specifies that node B is a buried flip-flop.

See Also

'attr'
Istype
Pin
Module
"Attribute Assignment" in the chapter "ABEL-HDL Language Structure"
"Architecture-independent Designs" in the chapter "Design Considerations"

Options

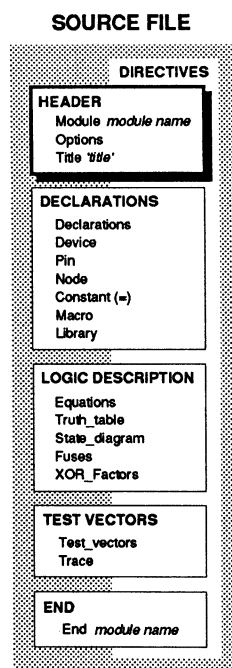
Syntax

```
OPTIONS 'option [, option ]...' [;]
```

Purpose

The **options** statement provides an alternate method of defining processing options that affect the way in which the source file is processed by the language processor.

Usage



option a string containing valid command line options (see below).

Options are normally passed from the command line or from the menus when the language processor is invoked. **Options** allows the explicit statement of processing options directly in the source file. Options specified on the command line or in the ABEL Design Environment override options specified with the **options** statement. The following commands are NOT valid with the **options** statement:

```
-i filename
-o filename
-vectors
-syntax
-silent
-list
-errlog
```

Complete information on command line options is in the chapter "Using ABEL Processing Modules."

Options is useful when a source file requires a specific type or level of processing for that source file to be successfully processed.

The **options** statement is similar to the FLAG statement used in earlier versions of ABEL.

Examples

In many cases the output polarity is important to preserve proper register preset and powerup behavior. In these situations, the following statement is useful:

```
Options '-reduce fixed';
```

See Also

Module

Pin

Syntax

```
[!]pin_id [, [!]pin_id...] PIN [pin# [, pin# ]] [ISTYPE
'attr' ];
```

Purpose

The PIN keyword is used to declare those input and outputs signals that must be available on a device I/O pin.

Usage

pin_id an identifier used to refer to a pin in a module

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 'attr' earlier in this chapter.

The declaration can also specify pin attributes.

When lists of pin_ids and pin#s are used in a 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.

The ending semicolon is required after each declaration.

Pin numbers do not have to be assigned for AHDL2PLA, PLAOpt or PLASim. Pin assignments must be made before processing with Fuseasm and JEDSim. You can either do pin assignments manually or use an optional fitter program to automatically assign pins.

Note: Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later in the process by a fitter (see the SmartPart User Manual if you have this option).

The ! operator in pin declarations indicates that the pin is active-low, and will be automatically negated by the language processor.

Attributes specifies pin attributes. Attributes can be defined in this way with the ISTYPE statement and are recommended for all pins that are not declared with a device/pin number and will be used as outputs.

Examples

```
Clock, !Reset, S1 PIN 1,15,3;
```

Clock is assigned to pin 1, Reset to pin 15, and S1 to pin 3.

See Also

Istype

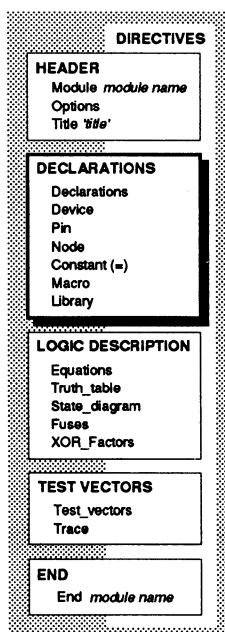
Node

'attr'

Module

"Architecture-independent Designs" in the chapter "Design Considerations"

SOURCE FILE



095-0712-001

Property

Syntax

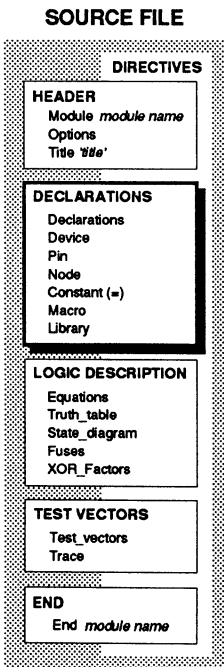
```
property_id PROPERTY 'string' ;
```

Purpose

The **property** declaration statement allows you to specify additional design information associated with an external processing module (such as a specialized device fitter).

Usage

- property_id**Identifiers properties relevant to specific external modules
- string**Argument containing the actual property data.



085-0712-001

See Also

amd_cm8.abl

State_diagram

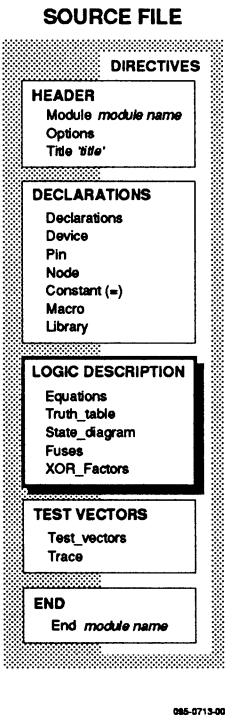
Syntax

```
State_diagram state_reg
    [-> state_out ]
[STATE state_exp : [equation ]
    [equation ]
    :
    :
    :
trans_stmt ; ...]
```

Purpose

The state description describes the operation of a sequential state machine implemented with programmable logic.

Usage



- 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** IF-THEN-ELSE, CASE or GOTO statements, optionally followed by WITH-ENDWITH transition equations.

The specification of a state description requires the use of the **State_diagram** syntax, which defines the state machine, and the **If-Then-Else**, **Case**, **Goto** and **With-endwith** statements which determine the operation of the state machine.

An ending semicolon is required after each transition statement.

The STATE DIAGRAM construct is discussed here, and the syntaxes of the IF-THEN-ELSE, CASE, and GOTO statements are presented briefly, and are discussed further in their respective sections. Syntax of the WITH-ENDWITH statement, which permits the specification of equations in terms of transitions, is also presented.

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.

Transition Statements

Transition statements indicate the conditions that cause transitions from one state to the next. Each state in an ABEL-HDL state diagram must contain at least one transition statement. Transition statements can consist of GOTO statements, IF-THEN-ELSE conditional statements, CASE statements, or combinations of these different statements.

GOTO Syntax

```
GOTO state_exp ;
```

The GOTO statement is used to unconditionally jump to a different state. When GOTO is used, it is the only transition for the current state. Example:

```
STATE S0:
    GOTO S1;    "unconditional branch to state S1
```

CASE Syntax

```
CASE expression : state_exp ;
[ expression : state_exp ; ] ...
ENDCASE ;
```

The CASE statement is used to list a sequence of mutually-exclusive transition conditions and corresponding next states. Example:

```
STATE S0:
    CASE (sel == 0): S0 ;
        (sel == 1): S1 ;
    ENDCASE
```

When using the CASE statement, the conditions must be mutually exclusive; that is, no two transition conditions can be true at the same time or the resulting next state is unpredictable.

IF-THEN-ELSE Syntax

```
IF expression THEN state_exp
[ ELSE state_exp ] ;
```

IF-THEN-ELSE statements are used to indicate mutually exclusive transition conditions. Example:

```
STATE S0:
    IF (address > ^hE100) THEN S1 ELSE S2;
```

The ELSE clause is optional. A sequence of IF-THEN statements written with no ELSE clauses is equivalent to a sequence of CASE conditional statements. IF-THEN-ELSE statements can be chained and nested. See IF-THEN-ELSE for more information.

WITH-ENDWITH Syntax

```
state_exp WITH equation ;
[equation ; ] ...ENDWITH
```

The WITH-ENDWITH statement can be used in any of the above transition statements (the GOTO, IF-THEN-ELSE or CASE statements) in place of a simple state expression. For example, to specify that a set of registered outputs are to contain a specific value after one particular transition, you would specify the equation for it with a WITH statement similar to the following:

```
STATE S0:
    IF (reset)
        THEN S9 WITH
            ErrorFlag := 1;
            ErrorAddress := address;
        ENDWITH
    ELSE
        IF (address <= ^hE100)
            THEN S2
        ELSE
            S0;
```

The WITH-ENDWITH statement is also useful when you are describing output behavior for registered outputs since registered outputs written only for a current state would lag by one clock cycle.

Examples

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 [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 ;
```

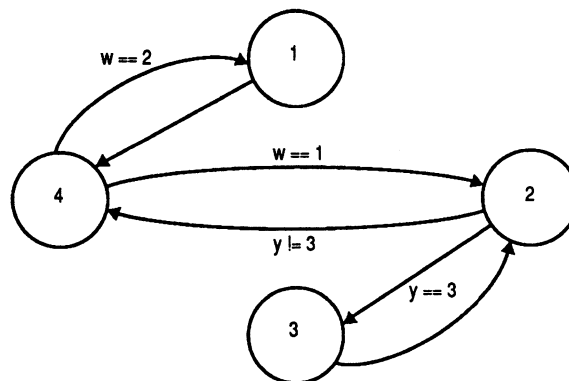
The next state diagram specifies a more complex state machine where the *state_reg* is specified with a constant set containing the signals a and b. Assuming that the state machine starts in state 1 (a = 0, b = 1), the sequence of states will be

```
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 4-10 shows the pictorial state diagram for this state machine.

Figure 4-10
Pictorial State Diagram



**State Descriptions and
Pin-to-pin Descriptions**

Sequential circuits described with ABEL-HDL's state diagram language are normally written with a pin-to-pin behavior in mind, regardless of the flip-flop type specified. Consider the following simple state machine (Figure 4-11):

Figure 4-11
*Architecture-independent State
Machine*

```
module statema
title 'State machine example. Dave Pellerin';

  clock,hold,reset    pin;
  P1,P0              pin  istype 'reg';
  P1,P0              istype 'buffer';
  X = .x.;

equations

  [P3,P2,P1,P0].clk    = clock;
  [P3,P2,P1,P0].ar     = reset;

" state declarations...
declarations
  stateA = [0,0];
  stateB = [1,0];
  stateC = [1,1];
  stateD = [0,1];

state_diagram [P1,P0]

  state stateA:
    goto stateB;

  state stateB:
    goto stateC;

  state stateC:
    goto stateD;

  state stateD:
    goto stateA;

test_vectors([clock,reset]->[P1,P0])
  [ .c. , 1 ]-> stateA;
  [ .c. , 0 ]-> stateB;
  [ .c. , 0 ]-> stateC;
  [ .c. , 0 ]-> stateD;
  [ .c. , 0 ]-> stateA;
  [ .c. , 1 ]-> stateA;

end
```

This state machine will operate the same (in terms of the behavior seen on its outputs) no matter what type of register is substituted for 'reg' in the signal declarations. To allow this flexibility, the specification of 'buffer' or 'invert' is required whenever a state diagram is written for a register type other than 'reg'.

See Also

Case
@Dcset
If-then-else
Goto
With-endwith
Equations
Truth_tables
Module
The chapter "Design Considerations"

Test_vectors

Syntax

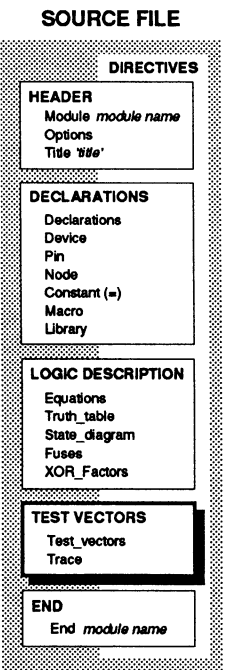
```
Test_vectors [note ]
(input [,input ]... -> output[,output ]...)

[invalues -> outvalues ; ]
:
```

Purpose

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs.

Usage



085-0714-001

- 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** pin-to-pin output value or set of output values resulting from the given inputs

Test vectors are used for simulation of an internal model of the device and functional testing of the design and device. The number of test vectors is unlimited.

The format 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 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 a special constant (for example, .X. and .C.). Each line of the table (each input/output listing) must end with a semicolon. Test vector output values always represent the pin-to-pin value for the output signals.

Test vectors must be sequential for state machines. That is, the test vectors must go through valid state transitions.

Two simulation programs, PLASim and JEDSim, are provided as part of the ABEL software package. PLASim simulates the operator of the design independent of any device. JEDSim 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. PLASim and JEDSim are discussed further in the chapters "Understanding ABEL," "Using ABEL Processing Modules," and "Using Advanced Features."

The **Trace** keyword can be used to control simulator output from within the source file.

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.

Examples

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 (that is, interpreted pin-to-pin).

See Also

Module

Trace

"Simulating a Design" in the chapter "Using ABEL Processing Modules"
"Test Vectors and Simulation" in the chapter "Using Advanced Features"

Title

Syntax

```
title 'string'
```

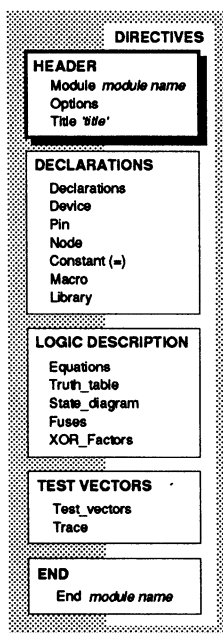
Purpose

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.

Usage

The title is specified in the string following the keyword, **title**. The string is opened and closed by an apostrophe and is limited to 324 characters.

SOURCE FILE



085-0711-001

Use of the title statement is optional.

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.

Examples

An example of a title statement that spans three lines and describes the logic design follows:

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

See Also

Module

Trace

Syntax

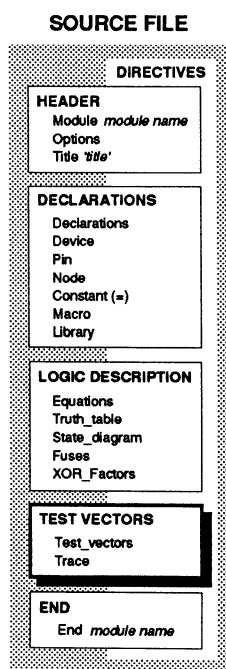
```
TRACE (inputs -> outputs) ;
```

Purpose

The TRACE statement is used to control the display features of the PLASim and JEDSim simulation modules.

Usage

TRACE statements can be placed before a test vector section, or imbedded within a sequence of test vectors.



085-0714-001

Examples

```
TRACE      ( [A,B] -> [C ] );
TEST_VECTORS ( [A,B] -> [C,D] )
           0 -> 3 ;
           1 -> 2 ;
TRACE      ( [A,B] -> [ D] );
           2 -> 1 ;
           3 -> 0 ;
```

See Also

Test_vectors

Truth_table

Syntax

```

TRUTH_TABLE ( in_ids -> out_ids )
                inputs -> outputs ;

    or

TRUTH_TABLE ( in_ids :> reg_ids )
                inputs :> reg_outs ;

    or

TRUTH_TABLE
( in_ids :> reg_ids -> out_ids )
  inputs :> reg_outs -> outputs ;

```

Purpose

Truth tables specify outputs as functions of different input combinations in a tabular form.

Usage

inputs the inputs to the logic function

outputs the outputs from the logic function

reg_outs the registered (clocked) outputs

-> indicates the input to output function for combinational outputs.

:> indicates the input to output function for registered outputs.

Truth tables are another way to describe logic designs with ABEL-HDL and may be used in lieu of or in addition to equations and state diagrams. A truth table is specified with a header describing the format of the table and with the table itself.

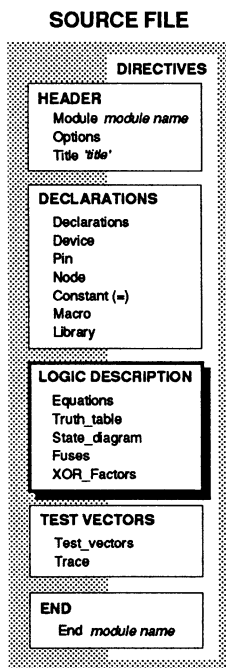
An ending semicolon is required after each line in the truth table.

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.

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 "ABEL-HDL Language Structure").

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.



085-0713-001

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.

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.

Truth Table Logic and @DCSET

The logic produced from a truth table description can differ from that produced in previous versions of ABEL. In earlier versions of ABEL, a truth table description of a partially-specified function produced logic that was no more efficient than if equations had been written describing the function. Truth tables written in ABEL-HDL, however, can produce dramatically smaller circuits if the @DCSET directive is specified.

Examples

This 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 ( [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 ;
```

When writing truth tables in ABEL-HDL, particularly when describing registered circuits, follow the same rules for dot extensions, attributes, and pin-to-pin/detailed descriptions described earlier for writing equations. The only difference between equations and truth tables is the ordering of the inputs and outputs.

The following two fragments of source code, for example, are functionally equivalent:

Fragment 1:

```
equations
```

```
    q := a & load # !q.fb & !load;
```

Fragment 2:

```
truth_table    (  [a ,q.fb,load] :> q)
                [0 , 0 , 0 ] :> 1;
                [0 , 1 , 0 ] :> 0;
                [1 , 0 , 0 ] :> 1;
                [1 , 1 , 0 ] :> 0;
                [0 , 0 , 1 ] :> 0;
                [1 , 0 , 1 ] :> 1;
                [0 , 1 , 1 ] :> 0;
                [1 , 1 , 1 ] :> 1;
```

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 ( [en, A , B ] -> C )
[ 0,.X.,.X.] -> .X.;" don't care w/enab off
[ 1, 0 , 0 ] -> 0 ;
[ 1, 0 , 1 ] -> 1 ;
[ 1, 1 , 0 ] -> 1 ;
[ 1, 1 , 1 ] -> 0 ;
```

See Also

Module
Equations
State_diagram
@Dcset
led1.abl
led7.abl

When-Then-Else

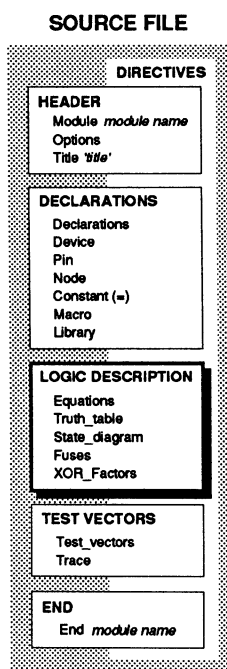
Syntax

```
[ WHEN condition      THEN ] [ ! ] element=expression;
[ ELSE equation ];
      or
[ WHEN condition      THEN ] equation;  [ ELSE equation];
```

Purpose

The **When-then-else** statement is used in equations to describe a logic function.

Usage



085-0713-001

condition any valid expression

element an identifier naming a signal or set of signals, or an actual set, to which the value of the expression will be assigned

expression any valid expression

= and := combinatorial and registered assignment operators

Equations use the two assignment operators = (combinatorial) and := (registered) described in the chapter "ABEL-HDL Language Structure."

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

```
WHEN B THEN A=B; ELSE A=C;
```

See Also

"Equations" in "ABEL-HDL Language Structure"

With-endwith

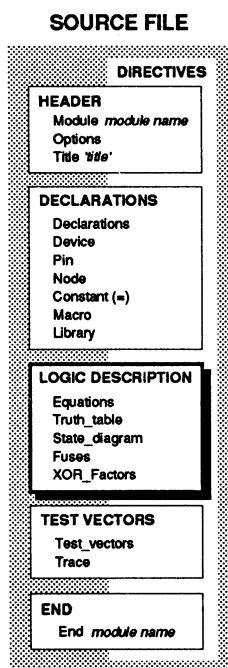
Syntax

```
trans_stmt state_exp WITH equation
[equation ]..
ENDWITH ;
```

Purpose

The WITH-ENDWITH statement is used under the **State_diagram** section and, when used in conjunction with the IF-THEN or CASE statement, allows output equations to be written in terms of transitions.

Usage



086-0713-001

trans_stmt IF-THEN-ELSE, GOTO or CASE statement

state_exp the next state

equation an equation for state machine outputs

The WITH-ENDWITH statement can be used in any transition statements in place of a simple state expression.

The WITH-ENDWITH statement is also useful when you are describing output behavior for registered outputs since registered outputs written only for a current state would lag by one clock cycle.

To specify that a set of registered outputs are to contain a specific value after one particular transition, you would specify the equation for it with a WITH statement similar to the following:

```
STATE S0:
  IF (reset) THEN S9 WITH   ErrorFlag := 1;
                           ErrorAddress := address;
                           ENDWITH
  ELSE IF (address <= ^hE100)
    THEN S2
    ELSE S0;
```

Examples

```
State 5 :
  IF a == 1 then 1   WITH x := 1 ;
                    y := 0 ;
                    ENDWITH;
  ELSE 2   WITH x := 0 ;
            y := 1 ;
            ENDWITH ;
```

See Also

State_diagram
Case
Goto
If-then-else

XOR_Factors

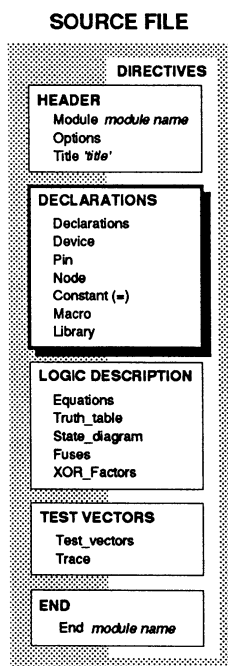
Syntax

```
XOR_Factors
signal name = xor_factors ;
```

Purpose

The XOR_factors section allows you to specify a Boolean expression that is to be factored out of, and XORed with, the sum-of-products reduced equations. This can result in dramatic reductions in the size of the reduced equations if a device featuring XOR gates is used.

Usage



086-0712-001

The XOR_factors feature 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_Factors 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.

When trying the factors, use reduce choose (auto polarity) to determine if complementing both sides of the XOR equation will fit.

After deciding the best XOR_Factors, remember to revise the source file to use an XOR device for the final design.

Note: The assignment operator used in XOR_Factors equations must match that used in the Equations section.

Examples

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 indicates that (A & B) and (C & D) are good candidate factors as shown below.

```
!Q16 =  A &  B & (!C # !D)
        # (!A # !B) &  C &  D;
```

If we run the following source file through ABEL software, the program reduces the equations according to the XOR_Factors, 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_Factors
    Q16 := A & B;

  equations
    !Q16 := A & B & !D
          # !B & C & D
          # !A & C & D
          # A & B & !C;
  end

```

Using A & B as the XOR_Factors, the reduced equations will be

```
!Q16 := ((A & B) $ (C & D));
```

Example 2

The example octalf.abl (Figure 4-12) uses a more complex high level equation:

Figure 4-12
Source File Using XOR_Factor

```

module OCTALF
  title' Octal counter with xor factoring
  Adam Zilinskas      Data I/O Corp.   14 Aug 1990'

  octalf device 'P20X8';          "Also works with P22V10

  D0,D1,D2,D3,D4,D5,D6,D7      pin 3,4,5,6,7,8,9,10;
  Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0      pin 16,15,17,18,19,20,21,22;
  CLK,I0,I1,OC,CarryOut,CarryIn pin 1,2,11,13,14,23;

  H,L,X,Z,C = 1, 0, .X., .Z., .C.;
  Data = [D7,D6,D5,D4,D3,D2,D1,D0];
  Count = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];

  Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0      istype 'invert';

  Mode = [I1,I0];
  Clear = [ 0, 0];
  Hold = [ 0, 1];
  Load = [ 1, 0];
  Inc = [ 1, 1];

  xor_factor
    Count := Count & I0;

  " ...Comments edited...
  equations
    Count := (Count + 1) & (Mode == Inc) & !CarryIn
            # (Count ) & (Mode == Inc) & CarryIn
            # (Count ) & (Mode == Hold)
            # (Data ) & (Mode == Load)
            # (0 ) & (Mode == Clear);

    !CarryOut = !CarryIn & (Count == ^hFF);

    Count.C = CLK;
    Count.OE = !OC;

```

```

test_vectors 'test load and increment'
  ([CLK,OC,Mode ,Data,CarryIn] -> [CarryOut,Count])
  [ C ,L ,Load , 1 , X ] -> [ H , 1 ];
  [ C ,L ,Inc , X , L ] -> [ H , 2 ];
  [ C ,L ,Inc , X , L ] -> [ H , 3 ];
  [ C ,L ,Inc , X , L ] -> [ H , 4 ];
  [ C ,L ,Inc , X , L ] -> [ H , 5 ];
  [ C ,L ,Load , 3 , X ] -> [ H , 3 ];
  [ C ,L ,Inc , X , L ] -> [ H , 4 ];
  [ C ,L ,Load , 7 , X ] -> [ H , 7 ];
  [ C ,L ,Inc , X , L ] -> [ H , 8 ];
  [ C ,L ,Load , ^h0F, X ] -> [ H , ^h0F ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h10 ];
  [ C ,L ,Load , ^h1F, X ] -> [ H , ^h1F ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h20 ];
  [ C ,L ,Load , ^h3F, X ] -> [ H , ^h3F ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h40 ];
  [ C ,L ,Load , ^h7F, X ] -> [ H , ^h7F ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h80 ];
  [ C ,L ,Load , ^hFF, L ] -> [ L , ^hFF ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h00 ];

test_vectors 'test load'
  ([CLK,OC,Mode ,Data,CarryIn] -> [CarryOut,Count])
  [ C ,L ,Load , ^hFF, L ] -> [ L , ^hFF ];
  [ C ,L ,Load , ^hFE, X ] -> [ H , ^hFE ];
  [ C ,L ,Load , ^hFD, X ] -> [ H , ^hFD ];
  [ C ,L ,Load , ^hFB, X ] -> [ H , ^hFB ];
  [ C ,L ,Load , ^hF7, X ] -> [ H , ^hF7 ];
  [ C ,L ,Load , ^hEF, X ] -> [ H , ^hEF ];
  [ C ,L ,Load , ^hDF, X ] -> [ H , ^hDF ];
  [ C ,L ,Load , ^hBF, X ] -> [ H , ^hBF ];
  [ C ,L ,Load , ^h7F, X ] -> [ H , ^h7F ];
  [ C ,L ,Load , ^hFF, L ] -> [ L , ^hFF ];

test_vectors 'test count'
  ([CLK,OC,Mode ,Data,CarryIn] -> [CarryOut,Count])
  [ C ,L ,Clear, X , X ] -> [ H , 0 ];
  [ C ,L ,Inc , X , L ] -> [ H , 1 ];
  [ C ,L ,Inc , X , L ] -> [ H , 2 ];
  [ C ,L ,Inc , X , L ] -> [ H , 3 ];
  [ C ,L ,Inc , X , L ] -> [ H , 4 ];
  [ C ,L ,Inc , X , L ] -> [ H , 5 ];
  [ C ,L ,Inc , X , L ] -> [ H , 6 ];
  [ C ,L ,Inc , X , L ] -> [ H , 7 ];
  [ C ,L ,Inc , X , L ] -> [ H , 8 ];
  [ C ,L ,Inc , X , L ] -> [ H , 9 ];
  [ C ,L ,Inc , X , L ] -> [ H , ^hA ];
  [ C ,L ,Inc , X , L ] -> [ H , ^hB ];
  [ C ,L ,Inc , X , L ] -> [ H , ^hC ];

test_vectors 'test hold and High Z'
  ([CLK,OC,Mode ,Data,CarryIn] -> [CarryOut,Count])
  [ C ,L ,Load , ^hFE, X ] -> [ H , ^hFE ];
  [ C ,L ,Inc , X , L ] -> [ L , ^hFF ];
  [ C ,L ,Inc , X , H ] -> [ H , ^hFF ];
  [ C ,L ,Hold , X , L ] -> [ L , ^hFF ];
  [ C ,L ,Inc , X , L ] -> [ H , ^h00 ];
  [ X ,H , X , X , X ] -> [ X , Z ];

"...Comments deleted..."

end OCTALF;

```

5 *Design Considerations*

This chapter discusses issues that need to be considered when creating a design with ABEL-HDL. The topics covered are listed below:

- Architecture-independent Language Features
- Pin-to-pin Vs. Detailed Descriptions for Registered Designs
- Using Active-low Declarations
- Flip-flop Equations
- Feedback Considerations — Using Dot Extensions
- @DCSET Considerations and Precautions
- Polarity Control
- Exclusive OR Equations
- State Machines
- Using Complement Arrays

Architecture-independent Language Features

The **Device** keyword is an optional feature in ABEL-HDL. There is no need to specify a particular PLD architecture in an ABEL-HDL source file. You can also omit pin numbers from signal declarations. The absence of device declarations and/or pin numbers implies the later use of the device selector and/or device fitters, respectively.

If no device is specified, or a device is specified but no pin numbers are specified, you may need to specify certain attributes about declared signals that are used in your design. This is because the ABEL-HDL compiler will not be able to imply ("default") these signal attributes from predetermined device attributes. In the absence of any signal attributes or other detailed information (such as the dot extensions described later) your design may not operate consistently when transferred to different target devices.

Device Independence Vs. Architecture Independence

The requirement for signal attributes does not mean that a complex design must always be specified with a particular device in mind. While you may still have to understand some of the differences between, for example, a P22V10 PAL and an EP600 EPLD, you don't have to specify a particular device when describing your design.

The attributes and dot extensions provided in ABEL-HDL help you refine your design to work consistently when moving from one class of device architecture to another; for example from devices having inverted outputs to those with a particular kind of reset/preset circuitry. The more you refine your design using these language features, the more restrictive your design becomes in terms of the number of device architectures it is appropriate for. However, by using attributes and dot extensions carefully, you can avoid specifying a particular device type, and instead target your design to a more general class of device architectures.

Signal Attributes

Signal attributes remove ambiguities that arise when no specific device architecture is declared. If no device-related attributes are used (either implied by a `DEVICE` statement or expressed in an `ISTYPE` statement), the design may not operate the same when targeted to different device architectures. See "Attributes," "Pin Declaration," "Node Declaration" and "Istype" in the "Language Reference."

Signal Dot Extensions

Signal dot extensions, like attributes, are a way to more precisely describe the behavior of a circuit that may be targeted to a variety of different architectures. Dot extensions are applied to signals and are used to remove the ambiguities in equations.

Refer to "Dot Extensions" later in this chapter and in "Language Structure" or *.ext* in the "Language Reference" for more information.

Pin-to-pin Vs. Detailed Descriptions for Registered Designs

ABEL-HDL supports two assignment operators that can be used when writing high-level equations. The `=` operator is used to specify a combinational assignment, while the `:=` operator is used to specify a registered assignment. It is important to understand the exact semantics of these two assignment operators.

Strictly combinational designs (whether composed of equations or truth tables) can usually be written with only the circuits inputs and outputs in mind. When registers are required, however, consideration must be given to internal circuit elements (such as output inverters, presets and resets) related to the memory elements (typically flip-flops) that are associated with the design outputs. Signal extensions, signal attributes, and the `:=` assignment operator are all considered when describing such circuits.

Using `:=` for Pin-to-pin Designs

When the `:=` operator is used in an equation, it implies a memory element is going to be associated with the output defined by the equation. For example, the equation

```
foo := !foo # Preset;
```

implies that `foo` will hold its current value until the memory element associated with that signal is clocked (or unlatched, depending on the register type).

This equation is a pin-to-pin description of the output signal `foo`. The equation describes the signal's behavior in terms of desired output pin values for various input conditions. Pin-to-pin descriptions are useful when describing a circuit that is completely architecture-independent.

Language elements that are useful for pin-to-pin descriptions are the `:=` operator, and the `.CLK`, `.OE` and `.FB` dot extensions described in the Language Reference. The three dot extensions help to resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities in Pin-to-pin Descriptions

In the equation above (`foo := !foo # Preset;`), there is an ambiguous feedback condition. The signal `foo` appears on the right side of the equation, but there is no indication of whether that fed-back signal should originate at the register, should come directly from the combinational logic that forms the input to the register, or should come from the I/O pin associated with `foo`. There is also no indication of what type of register should be used (although register synthesis algorithms could, theoretically, map this equation into virtually any register type). The equation could be more completely specified by writing:

```
foo.CLK = Clock;           "Register clocked from input
foo     := !foo.FB # Preset; "Register feedback
```

This set of equations describes the circuit completely, and specifies enough information that the circuit will operate identically in virtually any PLD in which it can be fit. The feedback path is specified to be from the register itself, and the .CLK equation specifies that the memory element is clocked, rather than latched.

Detailed Circuit Descriptions

In contrast, the same circuit can be specified in a detailed form of design description, as follows:

```
foo.CLK = Clock;           "Register clocked from input
foo.D   = !foo.Q # Preset;  "D-type f/f used for register
```

In this form of the design, specifying the D input to a D-type flip-flop and specifying feedback directly from the register restricts the device architectures the design can be implemented in. Furthermore, the equations only describe the inputs to and feedback from the flip-flop, and do not provide any information regarding the configuration of the actual output pin. This means the design will operate quite differently when implemented in a device with inverted outputs (a simple P16R4 PAL device, for example), versus in a device with non-inverting outputs (such as an EP600).

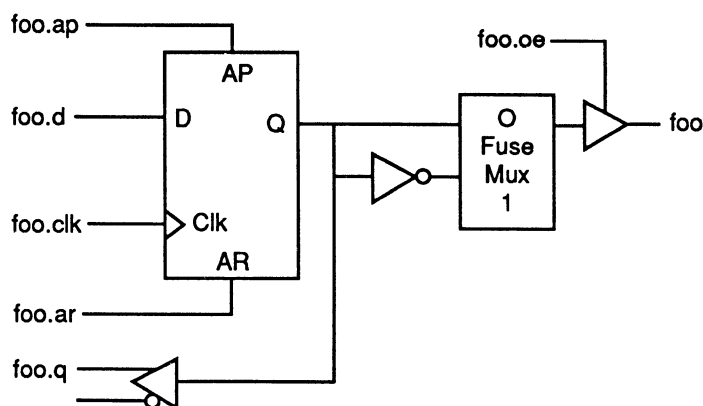
To maintain the correct behavior using detailed equations, one additional language element, a 'buffer' (or its complement, 'invert') attribute, is required. The 'buffer' attribute ensures that the final implementation in a device has no inversion between the specified D-type flip-flop and the output pin associated with *foo*. For example, add the following to the declarations section:

```
foo pin istype 'buffer';
```

Detailed Descriptions: Designing for Macrocells

One way to understand the difference between pin-to-pin and detailed description methods is to think of detailed descriptions as macrocell specifications. A macrocell is a block of circuitry normally (but not always) associated with a PLD's I/O pin. The following diagram (Figure 5-1) illustrates a typical (although fictional) macrocell associated with signal *foo*:

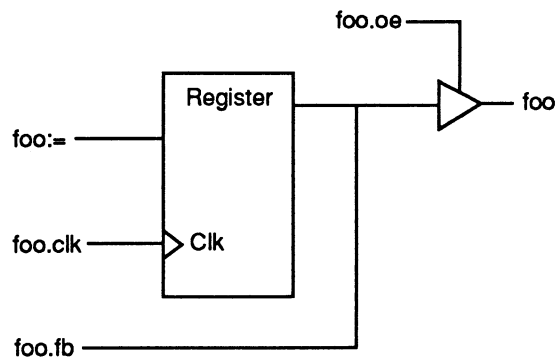
Figure 5-1
Detail Macrocell



Detailed descriptions are written for the various inputs ports (shown in Figure 5-1 with dot extension labels) of the macrocell. Note that the macrocell shown features a configurable inversion between the Q output of the flip-flop and the output pin labeled **foo**. If this inverter is used (or if a device is selected that features a fixed inversion), then the behavior seen on the **foo** output pin will be inverted from the logic applied to or observed on the various macrocell ports, including the feedback port **foo.q**.

Pin-to-pin descriptions, on the other hand, allow you to describe your circuit in terms of the behavior expected on an actual output pin, regardless of the architecture of the underlying macrocell. The following diagram (Figure 5-2) illustrates the pin-to-pin concept:

Figure 5-2
Pin-to-pin Macrocell



When pin-to-pin descriptions are written in ABEL-HDL, the "generic macrocell" shown above is synthesized from whatever type of macrocell actually exists in the target device. Note that there is currently no support for the pin-to-pin specification of register reset and preset behavior.

Examples of Pin-to-pin and Detailed Descriptions

Two equivalent design descriptions, one pin-to-pin and one detailed, are shown below for comparison:

Pin-to-pin Design

```
module foo_1
    foo      pin      istype 'reg';
    Clock,Preset pin;

    equations
        foo.clk = Clock;
        foo     := !foo.fb # Preset;

    test_vectors ([Clock,Preset] -> foo)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;

end
```

Detail Design

```

module foo_2
    foo      pin      istype 'reg_D,buffer';
    Clock,Preset pin;

    equations
        foo.CLK = Clock;
        foo.D    = !foo.Q # Preset;

    test_vectors ([Clock,Preset] -> foo)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;

end

```

The first description can be targeted into virtually any PLD (if register synthesis and device fitting features are available) while the second one can be targeted only to devices featuring D-type flip-flops and non-inverting outputs.

To implement the second (detailed) design into a device with inverting outputs, the source file would need to be modified as follows:

Detail Design with Inverted Outputs

```

module foo_3
    foo      pin      istype 'reg_D,invert';
    Clock,Preset pin;

    equations
        foo.CLK = Clock;
        !foo.D   = foo.Q # Preset;

    test_vectors ([Clock,Preset] -> foo)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;

end

```

In this version of the design, the existence of an inverter between the output of the D-type flip-flop and the output pin (specified with the 'invert' attribute) has necessitated a change in the equation for `foo.D`.

As this example shows, device independence and pin-to-pin description methods are clearly desirable, since the first version of the above design describes the circuit completely for any implementation. Although writing completely architecture-independent design descriptions is often impossible, the careful use of pin-to-pin descriptions and generalized dot extensions (such as `.FB`, `.CLK` and `.OE`) will allow you to implement your design into any one of a particular class of devices. (For example, any device that features enough flip-flops and appropriately configured I/O resources.) Note that the need for particular types of device features (such as register preset or reset) will limit your ability to describe your design in a completely architecture-independent way.

If, for example, a built-in register preset feature is used in a simple design, the devices possible for the design are limited to a certain set of target architectures. Consider this version of the design:


```

module foo_5
    foo      pin      istype 'reg,buffer';
    Clock,Preset pin;

    equations
        foo.CLK = Clock;
        foo.AP  = Preset;
        foo     := !foo.fb ;

    test_vectors ([Clock,Preset] -> foo)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;

end

```

The equation for `foo` still uses the `:=` assignment operator and `.FB` for a pin-to-pin description of `foo`'s behavior, but the use of `.AP` to describe the reset function requires consideration of different device architectures. The `.AP` extension, like the `.D` and `.Q` extensions, is associated with a flip-flop input, not with a device output pin. If the target device has inverted outputs, the design will not reset properly, so this ambiguous reset behavior is removed by using the 'buffer' attribute, with the result of reducing the number of devices for the design to those that feature non-inverted outputs.

The versions 5 and 7 of the design above and below are unambiguous, but each is restricted to certain classes of devices:

```

module foo_7
    foo      pin      istype 'reg,invert';
    Clock,Preset pin;

    equations
        foo.CLK = Clock;
        foo.AR  = Preset;
        foo     := !foo.fb ;

    test_vectors ([Clock,Preset] -> foo)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;

end

```

When to Use Detailed Descriptions

Although the pin-to-pin description is preferable, there are frequently situations in which more detailed descriptions must be used. If you are unsure about which method to use for various parts of your design, examine the design's requirements. If the design uses specific features of a device (such as register preset or unusual flip-flop configurations) then detailed descriptions are probably necessary. If your design is a simple combinational function, or matches the "generic" macrocell in its requirements, then you can probably use simple pin-to-pin descriptions.

Using := for Alternative Flip-flop Types

In ABEL-HDL you can specify a variety of flip-flop types using attributes such as 'reg_D' and 'reg_JK'. However, the use of these attributes does not enforce the use of a specific flip-flop type when a device is selected later, and does not affect the meaning of the := assignment operator. The := assignment can be thought of as a delay operator. The type of register that most closely matches the := operator's behavior is the D-type flip-flop. The primary use for attributes such as 'reg_D', 'reg_JK' and 'reg_SR' is to control the generation of logic from state diagrams.

Using := for flip-flop types other than D-type is only possible if register synthesis features are available to convert the generated equations into equations appropriate for the alternative flip-flop type specified. Since the use of register synthesis to convert D-type flip-flop stimulus into JK or SR-type stimulus usually results in inefficient circuitry, the use of := for these flip-flop types is discouraged. Instead, you should use the .J and .K extensions (for JK-type flip-flops) or the .S and .R extensions (for SR-type flip-flops) and use a detailed description method (including 'invert' or 'buffer' attributes) to describe designs for these register types.

There is no provision in the language for directly writing pin-to-pin equations for registers other than D-type. State diagrams, however, may be used to describe pin-to-pin behavior for any register type.

Using Active-low Declarations

In ABEL-HDL you can write design descriptions using active-low signals. Active-low signals are those that are declared with a '!' operator, as shown below:

```
!foobar    pin    istype 'reg';
```

When a signal is declared as active-low, it is complemented automatically whenever used in the subsequent design description. This complementing is done for any use of the signal itself, including uses as an input, output, and in test vectors. Complementing is also performed for any use of the dot extensions .fb and .q when they are applied to an active-low signal.

The following two designs, for example, are identical in operation:

Design 1

```

module act_low1
    q0,q1    pin istype 'reg';
    clock    pin;
    reset    pin;

    equations
        [q1,q0].clk = clock;
        ![q1,q0] := (![q1,q0] + 1) & !reset;

    test_vectors ([clock,reset] -> [!q1,!q0])
        [ .c. , 1 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 0 ] -> [ 1 , 0 ];
        [ .c. , 0 ] -> [ 1 , 1 ];
        [ .c. , 0 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 1 ] -> [ 0 , 0 ];

end

```

Design 2

```

module act_low2
    !q0,!q1  pin istype 'reg';
    clock    pin;
    reset    pin;

    equations
        [q1,q0].clk = clock;
        [q1,q0] := ([q1,q0] + 1) & !reset;

    test_vectors ([clock,reset] -> [ q1, q0])
        [ .c. , 1 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 0 ] -> [ 1 , 0 ];
        [ .c. , 0 ] -> [ 1 , 1 ];
        [ .c. , 0 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 1 ] -> [ 0 , 0 ];

end

```

Both of these designs describe an up counter with active-low outputs. The first example inverts the signals explicitly (in the equations and in the test vector header) while the second example uses an active-low declaration to accomplish the same thing.

Flip-flop Equations

Pin-to-pin equations (using the `:=` operator) are only supported for D-type flip-flops. ABEL-HDL does not allow the `:=` operator for T-type, SR-type or JK-type flip-flops and has no provision for specifying a desired output pin value for these types.

If you write an equation of the form:

```
foo := 1;
```

and the output, `foo`, has been declared as a T-type flip-flop, the ABEL-HDL compiler will give a warning and convert the equation to

```
foo.T = 1;
```

Since the T input to a T-type flip-flop does not directly correspond to the value observed on the associated output pin, this equation will not result in the pin-to-pin behavior desired.

To produce the desired pin-to-pin behavior for alternate flip-flop types, you must consider the specific behavior of the flip-flop used and write detailed equations that stimulate the inputs of that flip-flop. The correct detailed equation to set and hold a T-type flip flop is

```
foo. T = !foo.Q;
```

Feedback Considerations — Using Dot Extensions

When feedback is required in your design, the source of that feedback is normally dictated by the architecture of the target PLD. When no specific feedback path is specified, the design may operate differently when moved to different device types. Specifying feedback paths (with the .FB, .Q or .PIN dot extensions) can help to eliminate architectural ambiguities. Specifying feedback paths will also allow you to use the architecture-independent PLASim simulation module.

The following rules should be kept in mind when you are using feedback:

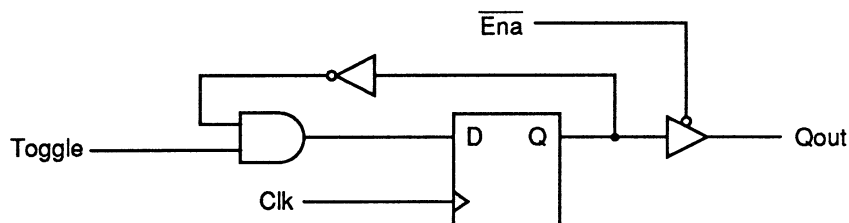
- **No Dot Extension** — A fed back signal with no dot extension (for example, `count := count+1;`) will result in pin feedback if it exists in the target device. If no pin feedback exists, the register feedback will be used, with the value of the register contents complemented (normalized) if needed to match the value observed on the pin.
- **.FB Extension** — A signal specified with the .FB extension (for example, `count := count.fb+1;`) will result in register feedback normalized to the pin value if a register feedback path exists. If no register feedback is available, pin feedback will be used. In this case, the Fuseasm module will check to make sure that the output enable does not conflict with the pin feedback path; an error will be generated if the output enable is not constantly enabled.
- **.PIN Extension** — If a signal is specified with the .PIN extension (for example, `count := count.pin+1;`), the pin feedback path will be used. If the specified device does not feature pin feedback, an error will be generated. Note that output enables will frequently affect the operation of fed back signals that originate at a pin.
- **.Q Extension** — Signals specified with the .Q extension (for example, `count.d = count.q + 1;`) will originate at the Q output of the associated flip-flop. The value fed-back may or may not correspond to the value observed on the associated output pin; if an inverter is located between the Q output of the flip-flop and the output pin (as is the case in most registered PAL-type devices), the value of the fed back signal will be the complement of the value observed on the pin.
- **.D Extension** — Some devices, such as the E0310 and P18CV8, allow feedback of the input to the register. To select this feedback, use the .D extension.

Dot Extensions and Architecture-Independence

To be architecture-independent, you must think of your design in terms of its pin-to-pin behavior rather than in terms of specific device features (such as flip-flop configurations or output inversions).

For example, consider the simple circuit shown in Figure 5-3. The circuit shown will toggle whenever the Toggle input is asserted with a high value, and will relax to a low value when Toggle is low. The circuit also contains a three-state output enable controlled by the active-low Enable input.

Figure 5-3
Dot Extensions and Architecture-Independence: Circuit 1



The following simple ABEL-HDL design (Figure 5-4) describes this simple one-bit synchronous circuit. The design description uses architecture-independent dot extensions to describe the circuit in terms of its behavior as observed on the output pin of the indicated device. Since this design is architecture-independent, it will operate the same (disregarding initial powerup state) regardless of the type of device specified.

Figure 5-4
Pin to Pin One-bit Synchronous Circuit

```

module pin2pin

    Clk      pin 1;
    Toggle   pin 2;
    Ena      pin 11;
    Qout     pin 19 istype 'reg';

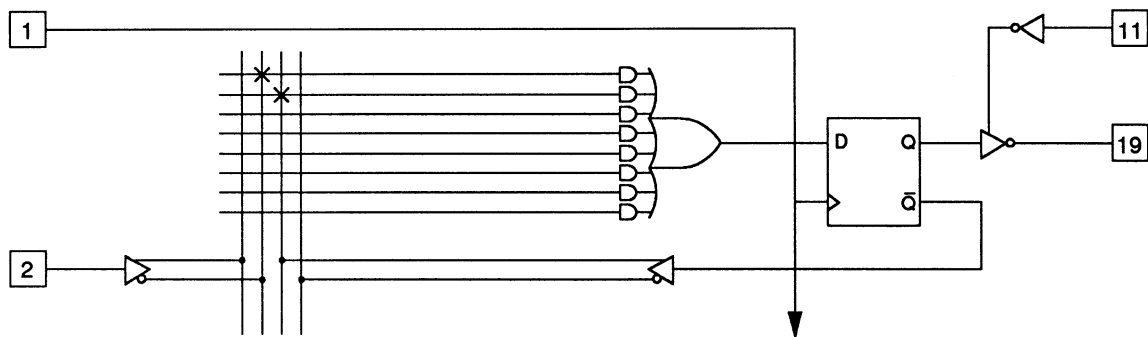
    equations
        Qout      := !Qout.FB & Toggle;
        Qout.CLK  = Clk;
        Qout.OE   = !Ena;

    test_vectors([Clk,Ena,Toggle] -> [Qout])
        [.c., 0 , 0 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 1;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 0;

end
  
```

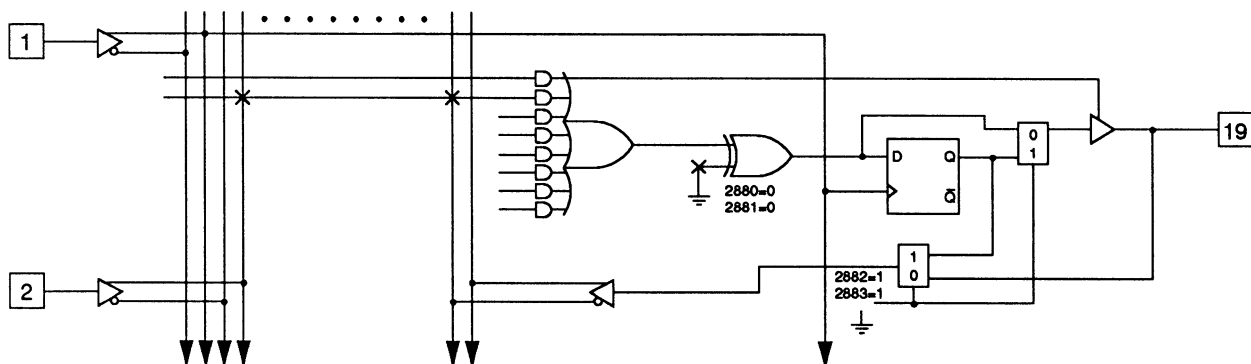
If this circuit is implemented in a simple P16R8 PAL device (either by adding a device declaration statement or by specifying the P16R8 in the Fuseasm process), the result would be a circuit like the one illustrated in Figure 5-5. Note that this circuit is somewhat different from the specified circuit; since the P16R8 features inverted outputs, the design equation has been automatically modified by Fuseasm to fit the P16R8's architecture.

Figure 5-5
Dot Extensions and Architecture-
Independence: Circuit 2



If this design is implemented into a device with a different architecture, such as an E0320, the resultant circuit may be quite different. But, since this is an architecture-independent pin-to-pin design description, the circuit behavior in the programmed device will be the same. Figure 5-6 illustrates the circuit that results when an E0320 is specified instead of a P16R8.

Figure 5-6
Dot Extensions and Architecture-
Independence: Circuit 3



Dot Extensions and Detail Design Descriptions

In some cases, you may need to be more specific about how a circuit is implemented in a target device. Many of the more complex device architectures feature many configurable features, and you may require that these features be used in a particular way. These reasons can include the need for precise powerup and preset operation or, in some cases, the need to control internal elements specific to a particular device.

The circuit previously described using architecture-independent dot extensions, for example, could be described using detailed dot extensions in the following ABEL-HDL source file (Figure 5-7):

Figure 5-7
*Detail One-bit Synchronous
Circuit with Inverted Qout*

```
module detail1
    d1      device 'P16R8';
    Clk     pin 1;
    Toggle  pin 2;
    Ena     pin 11;
    Qout     pin 19 istype 'reg_D';

    equations
        !Qout.D    = Qout.Q & Toggle;
        Qout.CLK   = Clk;
        Qout.OE    = !Ena;

    test_vectors([Clk,Ena,Toggle] -> [Qout])
        [.c., 0 , 0 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 1;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 0;
end
```

This version of the design will result in exactly the same fuse pattern as indicated in Figure 5-5. As written, this design assumes the existence of an inverted output for the signal Qout. This is why the Qout.D and Qout.Q signals are reversed from the architecture-independent version of the design presented earlier. (Note that the inversion operator applied to Qout.D does not correspond directly to the inversion found on each output of a P16R8. The equation for Qout.D actually refers to the D input of one of the P16R8's flip-flops; the output inversion found in a P16R8 is located after the register and is assumed rather than specified).

To implement this design in a device that does not feature inverted outputs, the design description must be modified. The following example (Figure 5-8) shows how to write this detailed design for the E0320 device:

Figure 5-8**Detail One-bit Synchronous
Circuit with Non-inverted Qout**

```

module detail2
    d2      device  'E0320';
    Clk     pin 1;
    Toggle  pin 2;
    Ena     pin 11;
    Qout    pin 19 istype 'reg_D';

    equations
        Qout.D    = !Qout.Q & Toggle;
        Qout.CLK  = Clk;
        Qout.OE   = !Ena;

    test_vectors([Clk,Ena,Toggle] -> [Qout])
        [.c., 0 , 0 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 1;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 0;
end

```

This design would result in the same circuit and E0320 fuse pattern previously illustrated in Figure 5-6.

@DCSET Considerations and Precautions

The @DCSET directive (Don't Care Set) can be used to reduce the amount of logic required for an incompletely specified function. Consider the following ABEL-HDL truth table:

```

truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
    [ 0, 0, 0, 0]->[ 0, 0, 0, 1];
    [ 0, 0, 0, 1]->[ 0, 0, 1, 1];
    [ 0, 0, 1, 1]->[ 0, 1, 1, 1];
    [ 0, 1, 1, 1]->[ 1, 1, 1, 1];
    [ 1, 1, 1, 1]->[ 1, 1, 1, 0];
    [ 1, 1, 1, 0]->[ 1, 1, 0, 0];
    [ 1, 1, 0, 0]->[ 1, 0, 0, 0];
    [ 1, 0, 0, 0]->[ 0, 0, 0, 0];

```

This truth table has four inputs, and therefore sixteen (2^4) possible input combinations. The function specified, however, only indicates eight significant input combinations. For each of the design outputs (f3 through f0) the truth table specifies whether the resulting value should be 1 or 0. For each output, then, each of the eight individual truth table entries can be either a member of a set of true functions called the on-set, or a set of false functions called the off-set. Using output f3 for an example, the eight input conditions can be listed as on-sets and off-sets as follows (maintaining the ordering of inputs as specified in the truth table above):

on-set of f3	off-set of f3
0 1 1 1	0 0 0 0
1 1 1 1	0 0 0 1
1 1 1 0	0 0 1 1
1 1 0 0	1 0 0 0

The remaining eight input conditions that do not appear in either the on-set or off-set are said to be members of the dc-set, as follows for f3:

dc-set of f3

```

0 0 1 0
0 1 0 0
0 1 0 1
0 1 1 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 1

```

Expressed as a Karnaugh map, the on-set, off-set and dc-set would appear as follows (with ones indicating the on-set, zeroes indicating the off-set, and dashes indicating the dc-set):

		\ i1 i0			
i3	i2\				
		00	01	11	10
00	00	0	0	0	-
	01	-	-	1	-
	11	1	-	1	1
	10	0	-	-	-

If the don't-care entries in the Karnaugh map are used for optimization, the function for f3 can be reduced to a single product term ($f3 = i2$) instead of the two ($f3 = i3 \& i2 \# i1 \& i0$) otherwise required.

In previous versions of ABEL, the set of don't-care input conditions was not used to reduce the amount of logic required for the final circuit. Instead, they were either assigned to the off-set of the function (in the case of devices with non-inverted outputs) or assigned to the on-set (in the case of devices with inverted outputs).

ABEL-HDL uses this level of optimization if the @DCSET directive is included in the ABEL-HDL source file as shown in Figure 5-9:

Figure 5-9
Source File Showing @DCSET
Optimization

```

module dc
    i3,i2,i1,i0    pin;
    f3,f2,f1,f0    pin;

    @dcset

    truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
        [ 0, 0, 0, 0]->[ 0, 0, 0, 1];
        [ 0, 0, 0, 1]->[ 0, 0, 1, 1];
        [ 0, 0, 1, 1]->[ 0, 1, 1, 1];
        [ 0, 1, 1, 1]->[ 1, 1, 1, 1];
        [ 1, 1, 1, 1]->[ 1, 1, 1, 0];
        [ 1, 1, 1, 0]->[ 1, 1, 0, 0];
        [ 1, 1, 0, 0]->[ 1, 0, 0, 0];
        [ 1, 0, 0, 0]->[ 0, 0, 0, 0];
    end

```

This example results in a total of four single-literal product terms, one for each output. The same example with no @DCSET directive results in a total of twelve product terms.

For truth tables, this level of optimization is almost always preferable. For state machines, however, you may not want undefined transition conditions to result in unknown states, or you may want to use a default state (determined by the type of flip-flops used for the state register) for state diagram simplification.

When using @DCSET, you must be careful not to specify any overlapping conditions in your truth tables and state diagrams. Overlapping conditions will result in the message

onset and offset not orthogonal

when PLAOpt is invoked.

Polarity Control

Automatic polarity control is a powerful feature in ABEL-HDL where ABEL converts a logic function for both non-inverting and inverting 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.

$$(1) F1 = (A \& B);$$

$$(2) !F1 = !(A \& B);$$

$$(3) !F1 = !A \# !B;$$

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 a non-inverting device such as the P10H8 than in an inverting 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 positive polarity. For example, the inverted form of F2, equation (3), uses fewer product terms than equation (2).

$$(1) F2 = (A \# B) \& (C \# D);$$

$$(2) F2 = (A \& C) \# (A \& D) \# (B \& C) \# (B \& D);$$

$$(3) !F2 = (!A \& !B) \# (!C \& !D);$$

Programmable polarity devices are popular because they can provide a mix of non-inverting and inverting outputs to achieve the best fit.

Targeted Device	Istype	plaopt -reduce	Active Level
Non-inverting	'pos' none	fixed choose	high high*
Inverting	'neg' none	fixed choose	low low*
Programmable inversions	'pos' 'neg' none	fixed fixed choose	high low high/low*

* See Automatic Polarity Selection

Automatic Polarity Selection

ABEL can perform automatic polarity selection of the design's equations dependent upon the device architecture selected. This feature allows you to map a given design into different devices without having to rewrite the logic of the design.

For PAL devices with fixed inversion, Fuseasm will map active low equations to each output; and for non-inverting devices, it will map active high equations. Devices with programmable inversions allow a higher degree of flexibility in mapping equations. Fuseasm takes advantage of that feature by selecting the most optimized equations for each output and programming the inversions accordingly.

FPLA devices require different approach to perform polarity selection. The best way is to reduce your equations by calling PLAOpt with **-reduce group fixed**.

Polarity Control in ABEL

In ABEL-HDL, the polarity of the design equations and target device (in the case of programmable polarity devices) is controlled in either of two ways:

- Using 'pos' and 'neg'
- Using 'invert' and 'buffer'

Using 'pos' and 'neg' to Control Equation and Device Polarity

The 'pos' and 'neg' attributes can be used to specify the equation polarity produced by the AHDL2PLA compiler. If 'neg' is specified for a signal, that signal will be double-complemented before being written to the ABEL-PLA file. This double complement has no effect on the logic of the circuit, and should not be confused with active-low logic. The double complement affects only the form and size of the circuit as expressed in sum-of-products form. Unlike previous versions of ABEL, specifying 'pos' and 'neg' will not guarantee that the respective polarity will be used in a programmable polarity device. This is because the PLAOpt program, by default, optimizes the design for both polarities, and the Fuseasm module will choose the polarity that results in the fewest product terms.

To use 'pos' and 'neg' to force a programmable polarity device to have a specific output polarity, you must specify the **-reduce fixed** option to PLAOpt. This will tell PLAOpt to minimize the design for only the polarity used in the ABEL-PLA file, specified with the 'pos' or 'neg' attribute.

Using 'invert' and 'buffer' to Control Device Polarity

An alternative method for specifying the desired state of a programmable polarity output is to use the 'invert' or 'buffer' attributes. These attributes will cause the Fuseasm module to ensure that an inverter gate either does or does not exist between the output of a flip-flop and its corresponding output pin.

These attributes are particularly useful for devices such as the P22V10, where the reset and preset behavior is affected by the programmable inverter. Note that the 'invert' and 'buffer' attributes do not actually control device polarity — they only enforce the existence or nonexistence of an inverter between a flip-flop and its output pin. The polarity of devices that feature a fixed inverter in this location and a programmable inverter before the register may not be specified using 'invert' and 'buffer'.

Exclusive OR Equations

Designs written for XOR devices should contain the 'xor' attribute for architecture-independence.

Optimizing XOR Devices

Exclusive-OR (XOR) gates can be used to advantage when designing with ABEL. XOR gates existing in devices can be used directly (by writing design equations that include XOR operators) or the XOR gates can be used automatically, either to minimize the total number of product terms required for an output, or to emulate alternate flip-flop types.

Using XOR Operators in Equations

If you want to write design equations that include XOR operators, you must either specify a device that features XOR gates in your ABEL-HDL source file, or specify the 'xor' attribute for all output signals that will be implemented with XOR gates. Once the ABEL-HDL compiler (AHDL2PLA) has been informed of the existence of XOR gates in this way, it will preserve one top-level XOR operator for each design output. For example,

```
module X1
    foo      pin      istype 'com,xor';
    a,b,c    pin;
equations
    foo = a $ b & c;
end
```

Also, 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. See example OCTALF.ABL.

Using Implied XORs in Equations

High-level operators used in equations will often result in XOR operators being generated. If you specify the 'XOR' attribute, these implied XORs will be preserved, resulting in a decrease in the number of product terms required for the function. For example,

```

module X2
    q3,q2,q1,q0      pin istype 'reg,xor';
    clock             pin;
    count = [q3..q0];
equations
    count.clk = clock;
    count := count + 1;
end

```

This design describes a simple four-bit counter. Since the addition operator results in XOR operators for the four outputs, the 'xor' attribute can be used to reduce the amount of circuitry generated.

Note: The high level operator that generates the XOR operators must be the lowest priority operation in the equation. An equation such as

$$\text{count} := (\text{count} + 1) \& !\text{reset};$$
will not result in top-level XOR operators being preserved, since the & operator is the top-level operator.

Using XORs for Flip-flop Emulation

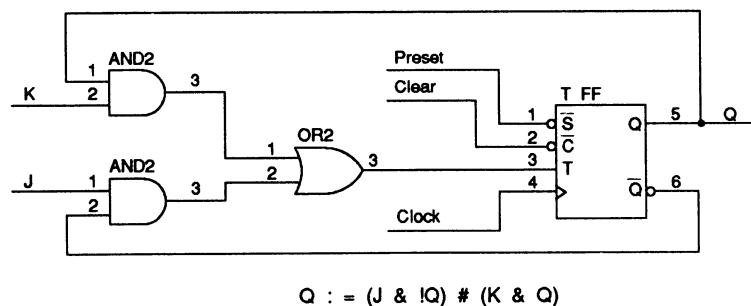
Another way to use XOR gates is for flip-flop emulation. If you are using an XOR device that has outputs featuring an XOR gate and D-type flip-flops, you can write your design as if you were going to be implementing it in a device with T-type flip-flops. The PartMap module will automatically use the XOR gates and D-type flip-flops to emulate the T-type flip-flops specified. When using XORs in this way, you should not use the 'xor' attribute for output signals.

For the most optimal use of flip-flop emulation, you can specify **-reduce dt** in PLAOpt when optimizing the circuit. This will result in a mixture of D-type and T-type flip-flops being used to obtain the smallest possible implementation.

JK Flip-Flop Emulation

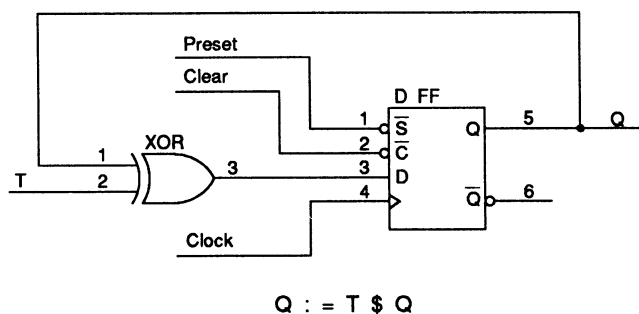
JK flip-flops may be emulated using a variety of circuitry found in programmable devices. When a T-type flip-flop is available, JK flip-flops can be emulated by ANDing the Q output of the flip-flop with the K input. The !Q output is then ANDed with the J input. This specific approach is useful in devices such as the Intel/Altera E0600 and E0900. Figure 5-10 illustrates the circuitry and the boolean expression:

Figure 5-10
 JK Flip-flop Emulation Using T
 Flip-flop



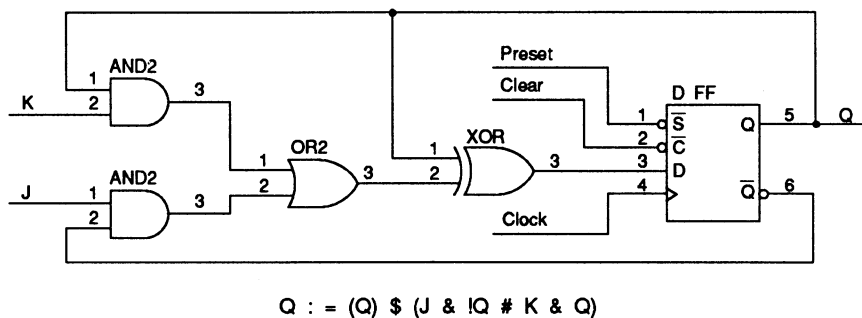
A D flip-flop can be used with an XOR gate in the absence of a T flip-flop to emulate a JK flip-flop. This technique is useful in devices such as the P20X8. The circuitry and boolean expression is as shown below in Figure 5-11:

Figure 5-11
T Flip-flop Emulation Using D Flip-flop



Finally, a JK flip-flop can also be emulated by combining the D flip-flop emulation of a T flip-flop in Figure 5-11 with the circuitry of Figure 5-10. Figure 5-12 illustrates this concept:

Figure 5-12
JK Flip-flop Emulation,
D Flip-flop with XOR



Integrated Circuit in Digital Electronics
Arpad Barna and Dan Porat
John Wiley & Sons 1973

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 the ALU of 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 PLDs output registers and buried registers are used to store the history of the states it has traversed through.

State diagrams are powerful features of ABEL-HDL, 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.

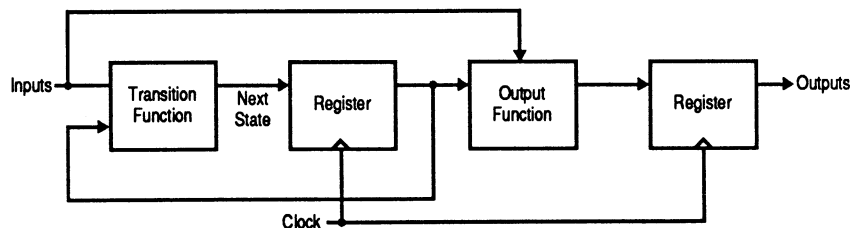
The following subsections provide state machine considerations:

- Mealy and Moore State Machines
- Debugging State Machines with Simulate
- Use Identifiers Rather Than Numbers for States
- "Power On" Register States
- Unsatisfied Transition Conditions, D-Type Flip-Flops
- Unsatisfied Transition Conditions, Other Flip-Flops
- Number Adjacent States for One Bit Change
- 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 5-13), 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.

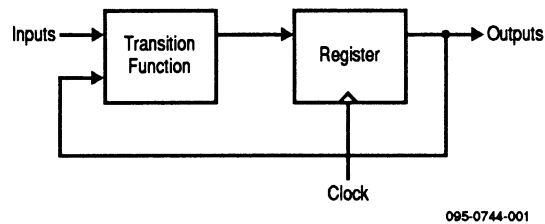
Figure 5-13
Flow Diagram for a Mealy State Machine



095-0743-001

In a Moore machine (Figure 5-14), 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-HDL designs implement a sequence detector in state machine syntax. The first design shown in Figure 5-17 implements the sequence detector using a Mealy machine. The second design shown in Figure 5-18 is the equivalent Moore machine.

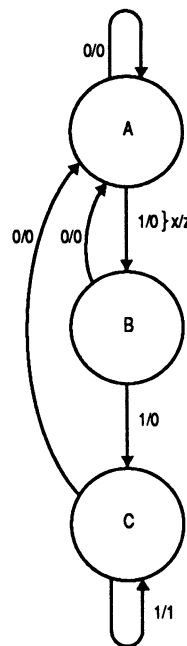
Figure 5-14
Flow Diagram for a Moore State Machine



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).

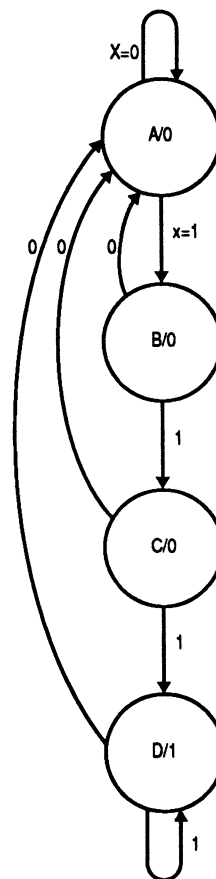
The state diagram for the Mealy sequence detector is given in Figure 5-15 which is to be compared with the equivalent Moore sequence detector in Figure 5-16. In both cases, state A is a state that has been arrived at after $X = 0$, so that any sequence of input 1s has been interrupted and Z cannot be equal to 1.

Figure 5-15
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 $X = 1$, then 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 5-16
Moore State Diagram of a
Sequence Detector



005-0747-001

Figure 5-17
Mealy Source File

```

module mealy
title 'Mealy machine description of a Sequence detector
      Data I/O Corp. by Jeffrey Davis '

      mealy device 'f167';

"Inputs
      clk    pin 1;
      PR     pin 16;
      X      pin 8;

"Output
      Z       pin 15 istype 'buffer,reg_RS';
      Q1, Q0  node   istype 'buffer,reg_RS';

"State Register assignment
      sreg    = [Q1,Q0];
      A      = [ 0, 0]; "use one bit changes
      B      = [ 0, 1];
      C      = [ 1, 1];

Equations
      sreg.pr = PR;
      sreg.clk = clk;
      Z.clk   = clk;

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.s = X ;
      if X then C
      else A      with Z.r = 1 ; endwith ;

test_vectors
      ([clk,PR, X] -> [ sreg ,  Z ])
      [.c.,1 , 0] -> [  C , .x. ];
      [.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 , 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 ];

end

```

Figure 5-18
Moore Source File

```

module moore
title 'Moore machine description of a Sequence detector
      Data I/O Corp. by Jeffrey Davis '

      moore device 'P22V10';

"Inputs
      clk    pin 1;
      PR     pin 2;
      X      pin 3;

"Output
      Q1,Q0,Z  pin 21,22,23 istype 'buffer,reg_D';

"State Register assignment
      sreg     = [Q1,Q0, Z];
      A       = [ 0, 0, 0];"use one bit changes
      B       = [ 0, 1, 0];
      C       = [ 1, 1, 0];
      D       = [ 1, 1, 1];

Equations
      sreg.ar  = PR;
      sreg.clk = clk;

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 ])
      [.c.,1 , 0] -> [  A  ];
      [.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 , 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

```

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-HDL, 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 5-19, 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. Their declared values define the value of the state register (sreg) for each state. The declared values are 0, 1, and 2.

Figure 5-19
Using Identifiers for States

```

module Sequence
title 'State machine example D. B. Pellerin Data I/O Corp';

sequence device 'p16r4';

q1,q0 pin 14,15 istype 'reg,invert';
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;

equations
[q1,q0,abort].clk = clock;
[q1,q0,abort].oe = !enab;

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.fb;

State B: " Advance to state C unless reset is active
in_B = 1; " or hold is active. Turn on abort indicator
in_C = 0; " 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,abort,in_B,in_C])
[ .p. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 1 , 0 ]->[ A , 1 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 1 , 0 , 0 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

end

```

Powerup Register States

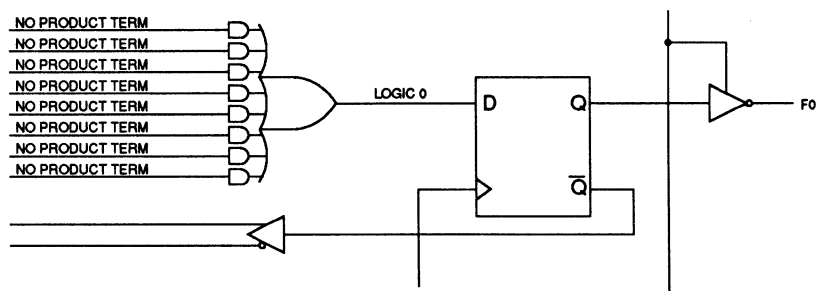
If a state machine design depends on a specific starting state, you must define the register powerup state in the state diagram description of the state machine or make sure that your design goes to a known state at powerup, or the next state is undefined.

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 5-20, 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 5-20
D-type Register with False Inputs



This action can be used to eliminate some conditions in your state diagram and some product terms in the converted design by leaving the cleared-register state transition implicit. If no specified transition condition 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 stay there.

In general, this means that you should always have a state assigned to the cleared-register state. If you don't assign a state to the cleared register state, you must define every possible condition so that some condition is always met for each state. The automatic transition to the cleared-register state can also be used to your advantage by eliminating product terms and explicit definitions of transitions. Illegal conditions can also be satisfied with the cleared state.

Other Flip-Flops

If none of the state conditions is met in a state machine that employs JK, RS, and T-type flip-flops, the state machine does not advance to the next state, but holds its present state 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. However, this holding action can be used to advantage in some designs.

If you want to prevent the hold, 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.

Precautions for Using @DCSET

When the @DCSET directive is used, there are certain design practices that must be avoided. The most common design technique that conflicts with the use of this optimization option is the use of mixed equations and state diagrams to describe default transitions. For example, consider the following design (Figure 5-21):

Figure 5-21
@DCSET-Incompatible State
Machine Description

```

module TRAFFIC
title 'Traffic Signal Controller
Kim-Fu Lim Data I/O Corp 5 June 1990'

    traffic device 'F167';

    Clk, SenA, SenB    pin    1, 8, 7;
    PR                pin    16;          "Preset control
    GA, YA, RA        pin    15, 14, 13;
    GB, YB, RB        pin    11, 10, 9;

    "Node numbers are not required if fitter is used
    S3, S2, S1, S0    node    31, 32, 33, 34;
    COMP              node    43;
    S3, S2, S1, S0    istype 'buffer, reg_rs';

    H, L, Ck, X       = 1, 0, .C., .X.;
    Count              = [S3, S2, S1, S0];

    "Define Set and Reset inputs to traffic light flip flops
    GreenA = [GA.S, GA.R];
    YellowA = [YA.S, YA.R];
    RedA    = [RA.S, RA.R];
    GreenB  = [GB.S, GB.R];
    YellowB = [YB.S, YB.R];
    RedB    = [RB.S, RB.R];
    On      = [ 1 , 0 ];
    Off     = [ 0 , 1 ];

    test_vectors ([Clk, PR, SenA, SenB] -> [Count, GA, YA, RA, GB, YB, RB])
    [ 0 , 0, 0 , 0 ] -> [ X , X, X, X, X, X, X];
    [ 1 , 1, 1 , 1 ] -> [ 15 , X, X, X, X, X, X];
    [ 1 , 0, 1 , 1 ] -> [ 15 , X, X, X, X, X, X];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];

    test_vectors ([Clk, PR, SenA, SenB] -> [Count, GA, YA, RA, GB, YB, RB])
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 1 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 2 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 3 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 4 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 5 , L, H, L, L, L, H];
    [ Ck, 0, 1 , 0 ] -> [ 8 , L, L, H, H, L, L];
    [ Ck, 0, 1 , 0 ] -> [ 12 , L, L, H, H, L, L];
    [ Ck, 0, 1 , 0 ] -> [ 13 , L, L, H, L, H, L];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 1 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 2 , H, L, L, L, L, H];
    [ Ck, 0, 1 , 1 ] -> [ 3 , H, L, L, L, L, H];
    [ 1 , 1, 1 , 1 ] -> [ 15 , X, X, X, X, X, X];
    [ 1 , 0, 1 , 1 ] -> [ 15 , X, X, X, X, X, X];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H];

```

```

@page
equations
    [GB,YB,RB].AP = PR;
    [GA,YA,RA].AP = PR;

    [GB,YB,RB].CLK = Clk;
    [GA,YA,RA].CLK = Clk;

    [S3,S2,S1,S0].AP = PR;
    [S3,S2,S1,S0].CLK = Clk;

    "Use Complement Array to initialize or restart
    [S3,S2,S1,S0].R      = (!COMP & [1,1,1,1]);
    [GreenA,YellowA,RedA] = (!COMP & [On,Off,Off]);
    [GreenB,YellowB,RedB] = (!COMP & [Off,Off,On]);

state_diagram Count
    State 0:      if ( SenA & !SenB ) then 0 with COMP = 1;
                  if (!SenA & SenB ) then 4 with COMP = 1;
                  if ( SenA == SenB ) then 1 with COMP = 1;

    State 1:      goto 2 with COMP = 1;
    State 2:      goto 3 with COMP = 1;
    State 3:      goto 4 with COMP = 1;

    State 4:      GreenA = Off;
                  YellowA = On ;
                  goto 5 with COMP = 1;

    State 5:      YellowA = Off;
                  RedA    = On ;
                  RedB    = Off;
                  GreenB   = On ;
                  goto 8 with COMP = 1;

    State 8:      if (!SenA & SenB ) then 8 with COMP = 1;
                  if ( SenA & !SenB ) then 12 with COMP = 1;
                  if ( SenA == SenB ) then 9 with COMP = 1;

    State 9:      goto 10 with COMP = 1;
    State 10:     goto 11 with COMP = 1;
    State 11:     goto 12 with COMP = 1;

    State 12:     GreenB = Off;
                  YellowB = On ;
                  goto 13 with COMP = 1;

    State 13:     YellowB = Off;
                  RedB    = On ;
                  RedA    = Off;
                  GreenA   = On ;
                  goto 0 with COMP = 1;

end

```

This design uses the complement array feature of the Signetics FPLA devices to perform an unconditional jump to state [0,0,0,0]. The equation that specifies this transition ([P7,P6,P5,P4].R = (!COMP & [1,1,1,1])); will conflict with the dc-set generated for P7.R, P6.R, P5.R, and P4.R as a result of the @DCSET directive. This will result in an error and failure when the logic for this design is optimized by PLAOpt.

To correct the problem, the @DCSET directive must be removed, so that the implied dc-set equations will be folded into the off-set for the resulting logic function. Another option is to rewrite the module as shown in Figure 5-22.

Figure 5-22
@DCSET-Compatible State
Machine Description

```

module TRAFFIC1
title 'Traffic Signal Controller
Mike McClure   Data I/O Corp   5 June 1990'

    traffic1      device 'F167';

    Clk,SenA,SenB  pin   1, 8, 7;
    PR            pin   16;      "Preset control
    GA,YA,RA      pin   15,14,13;
    GB,YB,RB      pin   11,10,9;

    S3,S2,S1,S0   node 31,32,33,34;
    S3,S2,S1,S0   istype 'buffer,reg_rs';

    H,L,Ck,X      = 1, 0, .C., .X.;

    Count         = [S3,S2,S1,S0];

    "Define Set and Reset inputs to traffic light flip flops
    GreenA = [GA.S,GA.R];
    YellowA = [YA.S,YA.R];
    RedA    = [RA.S,RA.R];
    GreenB  = [GB.S,GB.R];
    YellowB = [YB.S,YB.R];
    RedB    = [RB.S,RB.R];
    On      = [ 1 , 0 ];
    Off     = [ 0 , 1 ];

test_vectors
    ([Clk,PR,SenA,SenB] -> [Count,GA,YA,RA,GB,YB,RB])
    [ 0 , 0, 0 , 0 ] -> [ X , X, X, X, X, X, X ];
    [ 1 , 1, 1 , 1 ] -> [ 15 , X, X, X, X, X, X ];
    [ 1 , 0, 1 , 1 ] -> [ 15 , X, X, X, X, X, X ];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];

test_vectors
    ([Clk,PR,SenA,SenB] -> [Count,GA,YA,RA,GB,YB,RB])
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 1 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 2 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 3 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 4 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 5 , L, H, L, L, L, H ];
    [ Ck, 0, 1 , 0 ] -> [ 8 , L, L, H, H, L, L ];
    [ Ck, 0, 1 , 0 ] -> [ 12 , L, L, H, H, L, L ];
    [ Ck, 0, 1 , 0 ] -> [ 13 , L, L, H, L, H, L ];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 1 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 2 , H, L, L, L, L, H ];
    [ Ck, 0, 1 , 1 ] -> [ 3 , H, L, L, L, L, H ];
    [ 1 , 1, 1 , 1 ] -> [ 15 , X, X, X, X, X, X ];
    [ 1 , 0, 1 , 1 ] -> [ 15 , X, X, X, X, X, X ];
    [ Ck, 0, 1 , 0 ] -> [ 0 , H, L, L, L, L, H ];

```

```
@page
equations
    [GB,YB,RB].AP = PR;
    [GA,YA,RA].AP = PR;

    [GB,YB,RB].CLK = Clk;
    [GA,YA,RA].CLK = Clk;

    [S3,S2,S1,S0].AP = PR;
    [S3,S2,S1,S0].CLK = Clk;

@DCSET
state_diagram Count

    State 0:      if ( SenA & !SenB ) then 0;
                  if (!SenA &  SenB ) then 4;
                  if ( SenA == SenB ) then 1;

    State 1:      goto 2;
    State 2:      goto 3;
    State 3:      goto 4;

    State 4:      GreenA = Off;
                  YellowA = On ;
                  goto 5;

    State 5:      YellowA = Off;
                  RedA    = On ;
                  RedB    = Off;
                  GreenB  = On ;
                  goto 8;

    State 6:      goto 0;
    State 7:      goto 0;

    State 8:      if (!SenA &  SenB ) then 8;
                  if ( SenA & !SenB ) then 12;
                  if ( SenA == SenB ) then 9;

    State 9:      goto 10;
    State 10:     goto 11;
    State 11:     goto 12;

    State 12:     GreenB = Off;
                  YellowB = On ;
                  goto 13;

    State 13:     YellowB = Off;
                  RedB    = On ;
                  RedA    = Off;
                  GreenA  = On ;
                  goto 0;

    State 14:     goto 0;

    State 15:     "Power up and preset state
                  RedA    = Off;
                  YellowA = Off;
                  GreenA  = On ;
                  RedB    = On ;
                  YellowB = Off;
                  GreenB  = Off;
                  goto 0;

end
```

Number Adjacent States for One-Bit Change

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 Fuseasm module 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 Cn states are identified:

State Register Bit Values

State Name	Q3	Q2	Q1
A	0	0	0
B	0	0	1
C1	1	0	1
C2	1	1	1
C3	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 Cn states. Whenever Q3 is high, the machine is in one of the Cn 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 the section above.

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 JK and/or RS 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 5-23, 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. These nodes must be declared, and can be given any names.

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 5-24 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 5-23, 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 5-24 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 5-23
Transition Equations for a
Decade Counter

```

module DECADE
title 'Decade Counter    Uses Complement Array
Michael Holley    Data I/O Corp    3 May 1990'

    decade            device 'F105';

    Clk,Clr,F0,PR     pin  1,8,18,19;
    P3,P2,P1,P0      node 40,39,38,37;
    COMP             node 49;

    F0,P3,P2,P1,P0   istype 'reg_rs';

    _State            = [P3,P2,P1,P0];
    H,L,Ck,X         = 1, 0, .C., .X.;

equations
[P3,P2,P1,P0,F0].ap = PR;
[F0,P3,P2,P1,P0].clk = Clk;

"Output      Next State      Present State      Input
[F0.S, COMP,      P0.S] = !P3 & !P2 & !P1 & !P0 & !Clr; "0 to 1
[      COMP,      P1.S,P0.R] = !P3 & !P2 & !P1 & P0 & !Clr; "1 to 2
[      COMP,      P0.S] = !P3 & !P2 & P1 & !P0 & !Clr; "2 to 3
[      COMP,      P2.S,P1.R,P0.R] = !P3 & !P2 & P1 & P0 & !Clr; "3 to 4
[      COMP,      P0.S] = !P3 & P2 & !P1 & !P0 & !Clr; "4 to 5
[F0.R, COMP,      P1.S,P0.R] = !P3 & P2 & !P1 & P0 & !Clr; "5 to 6
[      COMP,      P0.S] = !P3 & P2 & P1 & !P0 & !Clr; "6 to 7
[      COMP,P3.S,P2.R,P1.R,P0.R] = !P3 & P2 & P1 & P0 & !Clr; "7 to 8
[      COMP,      P0.S] = P3 & !P2 & !P1 & !P0 & !Clr; "8 to 9
[      COMP,      P3.R,P2.R,P1.R,P0.R] = !COMP; "Clear

"After Preset, clocking is inhibited until High-to-Low clock transition.
test_vectors ([Clk,PR,Clr] -> [_State,F0 ])
[ 0 , 0 , 0 ] -> [ _X , X];
[ 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

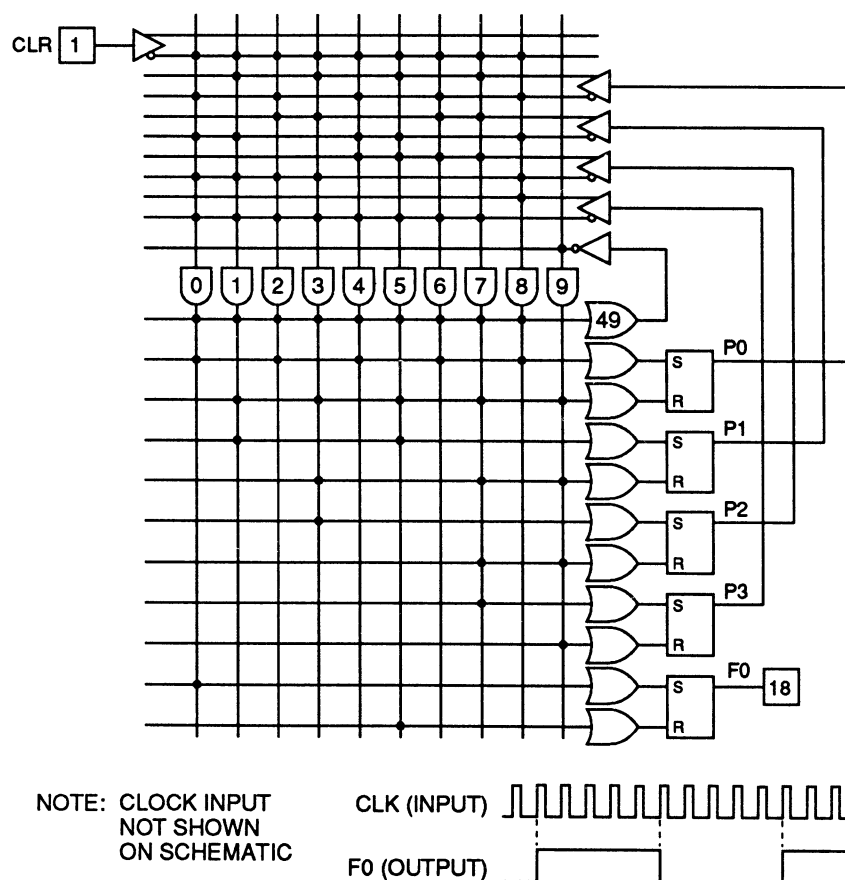
end

```

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, the fuses are blown as indicated in Figure 5-24. Figure 5-24 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.

Figure 5-24
Abbreviated F105 Schematic



6 *Source File Examples*

The following logic design examples are representative of programmable logic applications and serve to illustrate significant ABEL features. These examples will also help you get started creating your own source files. For complete information on creating a source file, see the chapters "ABEL-HDL Language Structure" and "Language Reference."

All the examples presented in this section are included on your ABEL distribution disk or tape. A list and brief description of all examples distributed with ABEL can be found in the file named **examples.txt** on your distribution disk or tape.

You can process these design examples with ABEL software, either as they stand, or with your own modifications, to create programmer load files.

The examples are divided into three sections — one for equations, one for state diagrams and one for truth tables — plus a blackjack machine implemented with two source files that use equations and one that uses a state diagram.

Equations

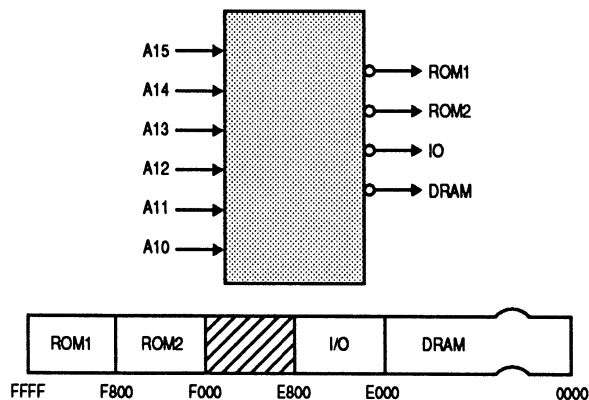
6809 Memory Address Decoder

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL implementation of such a design.

Design Specification

Figure 6-1 shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (DRAM), I/O (IO), 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.

Figure 6-1
*Block Diagram: 6809 Memory
Address Decoder*



The address ranges associated with each section of memory are shown below. These address ranges can also be seen in the source file in Figure 6-3.

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

Design Method

Figure 6-2 shows a simplified block diagram for the address decoder. The decoder is implemented with equations employing relational and logical operators as shown in Figure 6-3.

Significant simplification is achieved by grouping the address bits into a set named Address. The ten address bits that are not used for the address decode are given "don't care" values in the set, indicating that the address in the overall design (that beyond the decoder) contains 16 bits, but that bits 0 to 9 do not affect the decode of that address. In contrast, defining the set as

```
Address = [A15,A14,A13,A12,A11,A10]
```

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 6-2
Simplified Block Diagram: 6809
Memory Address Decoder

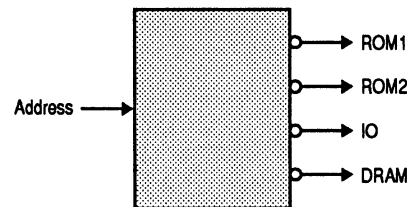


Figure 6-3
6809 Memory Address Decoder
Source File

```
module M6809A
title '6809 memory decode   Jean Designer   Data I/O Corp Redmond WA'

        m6809a 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
```

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.

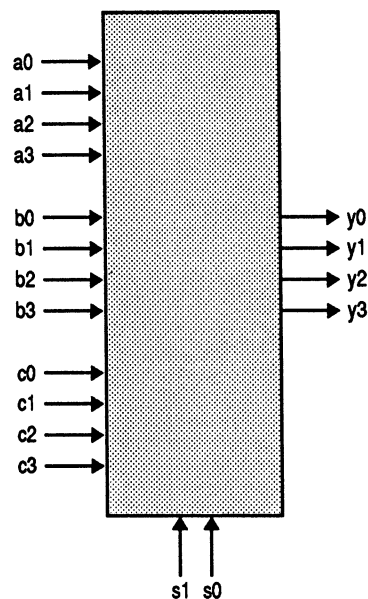
12 to 4 Multiplexer

The following describes the implementation of a 12-input to 4-output multiplexer using high level equations.

Design Specification

Figure 6-4 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 a0-a3, b0-b3, and c0-c3. The outputs are y0-y3. The routing of inputs to outputs is straightforward: a0 or b0 or c0 is routed to the output y0, a1 or b1 or c1 is routed to the output y1, 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 S0 and S1.

Figure 6-4
Block Diagram: 12 to 4
Multiplexer



Design Method

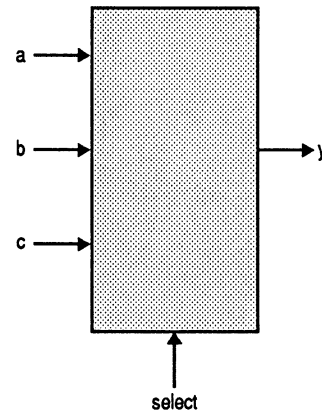
Figure 6-5 shows a block diagram for the same multiplexer after sets have been used to group the signals. All of the inputs have been grouped into the sets a, b, and c; the outputs and select lines are grouped into the sets, *y* and *select*, respectively. This grouping of signals into sets takes place in the declaration section of the source file listed in Figure 6-6.

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 = c;
```

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 6-5
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 6-6
Source File: 12 to 4 Multiplexer

```
module Mux12T4
title '12 to 4 multiplexer    5 June 1990
Dave Pellerin    Data I/O Corp.  Redmond WA'

mux12t4          device 'P16V8S';

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
```

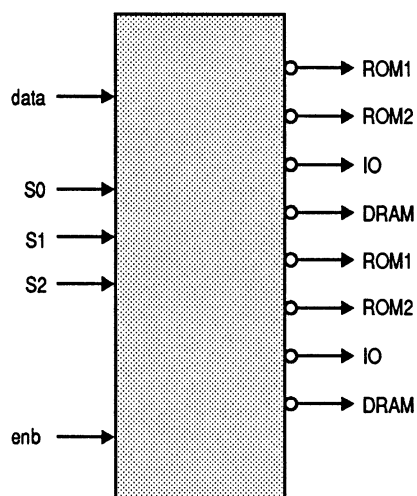
1 to 8 Demultiplexer

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.

Design Specification

Figure 6-7 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, y0-y7, according to the value present on the select lines, s0-s2. 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.

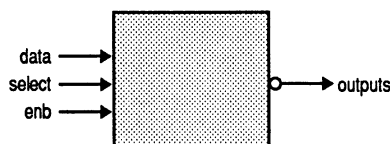
Figure 6-7
Block Diagram: 1 to 8
Demultiplexer



Design Method

Figure 6-8 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 description of this logic is shown in Figure 6-9. 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.

Figure 6-8
Simplified Block Diagram:
Demultiplexer



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 6-9

Source File: 1 to 8
Demultiplexer

```
module DMUX1T8
title '1 to 8 line demultiplexer
Susan Todd and Mark Kuenster   Data I/O Corp.   5 Aug 1990'

    dmux1t8 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, H, H]; "Select y0
[ H ,   1 , H ] -> [H, H, H, H, H, H, H, 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];

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, H]; "Select y0
[ H ,   1 , L ] -> [H, H, H, H, H, H, H, H]; "Select y1
[ H ,   2 , L ] -> [H, H, H, H, H, H, H, H]; "Select y2
[ H ,   3 , L ] -> [H, H, H, H, H, H, H, H]; "Select y3
[ H ,   4 , L ] -> [H, H, H, H, H, H, H, H]; "Select y4
[ H ,   5 , L ] -> [H, H, H, H, H, H, H, H]; "Select y5
[ H ,   6 , L ] -> [H, H, H, H, H, H, H, H]; "Select y6
[ H ,   7 , L ] -> [H, H, H, H, H, H, H, H]; "Select y7
[ L ,   0 , L ] -> [Z, Z, Z, Z, Z, Z, Z, Z];

end
```

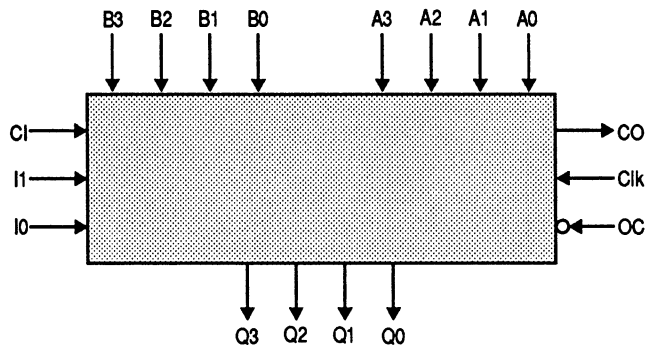
4-Bit Counter/- Multiplexer

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.

Design Specification

Figure 6-10 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).

Figure 6-10
Block Diagram: 4-Bit Counter
With 2 Input Multiplexer



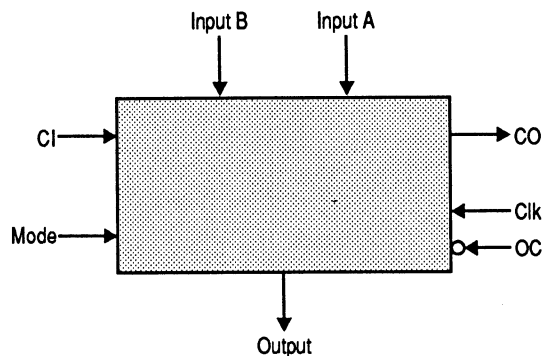
Mode	I1	I0	Description
Hold	0	0	count remains unchanged
Load A	0	1	load A0-A3 into the count registers on the clock pulse
Load B	1	0	load B0-B3 into the count registers on the clock pulse
Increment	1	1	increment the count by 1 on clock pulse

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.

Design Method

The counter is described with high level equations. Figure 6-11 shows the simplified block diagram corresponding to the ABEL implementation of the counter design. Figure 6-12 shows the source file.

Figure 6-11
Simplified Block Diagram: 4-Bit
Counter With 2 Input Multiplexer



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 6-12 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 6-12 include the output enable pin (OC).

Figure 6-12
Source file: 4-bit Counter with 2
Input Mux

```

module COUNT4
title '4-bit counter with 2 input mux' 12 Oct 1989
based on an example by Birkner/Coli in the MMI PAL Handbook
Dan Burrier & Mike McGee Data I/O Corp.'

count4 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
[Q3..Q0].clk = Clk;
[Q3..Q0].oe = !OC;

!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;         "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 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 6-13 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 6-13
Multiple Equations Sections, 4-Bit Counter

```

module COUNT4A
title '4-bit counter with 2 input mux' 25 Apr 1990
based on an example by Birkner/Coli in the MMI PAL Handbook
Lisa Matheson Data I/O Corp.'

count4a 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 " global
[Q3..Q0].clk = Clk;
[Q3..Q0].oe = !OC;

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 ;

@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;         "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

```

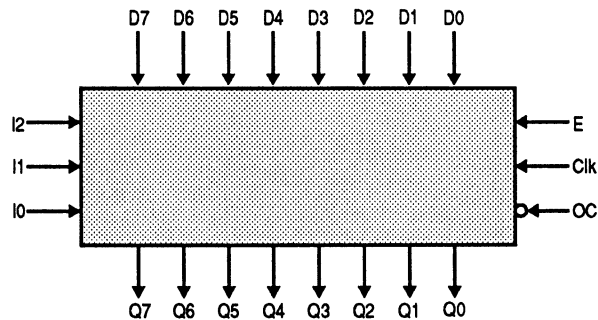
8-Bit Barrel Shifter

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.

Design Specification

Figure 6-14 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.

Figure 6-14
Block Diagram: 8-Bit Barrel Shifter



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 6-15 and Figure 6-16 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.

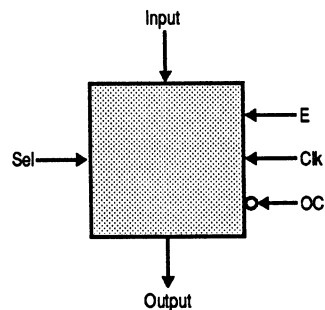
One high level equation is used to describe the entire function of the barrel shifter. The equation is expressed in sum of products form and assigns a value to the output set. Each product in the equation corresponds to one of the possible shifts and defines the outputs for that shift. Thus, the product 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]
```

Figure 6-15
Simplified Block Diagram: 8-Bit
Barrel Shifter



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,Z] to cover all eight outputs.

Figure 6-16
Source File: 8-Bit Barrel Shifter

```

module BARREL
title '8-bit barrel shifter
Don Flaherty    Data I/O Corp  9 Aug 1990'

    barrel 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.clk = Clk;
    Output.oe  = !OC;

    !Output := E & ( (Sel == 0) & ![D7,D6,D5,D4,D3,D2,D1,D0]
                    # (Sel == 1) & ![D0,D7,D6,D5,D4,D3,D2,D1]
                    # (Sel == 2) & ![D1,D0,D7,D6,D5,D4,D3,D2]
                    # (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

```

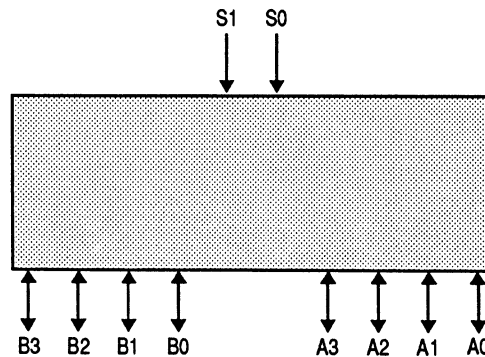
Bidirectional Three-State Buffer

A four-bit bidirectional buffer with tristate outputs is presented here. The design is implemented in an F153 FPLA with bidirectional inputs/outputs and programmable output polarity. Simple Boolean equations are used to describe the function.

Design Specification

Figure 6-17 shows a block diagram for this four-bit buffer. Signals A0-A3 and B0-B3 function both as inputs and outputs depending on the value on the select lines, S0-S1. When the select value (the value on the select lines) is 1, A0-A3 are enabled as outputs. When the select value is 2, B0-B3 are enabled as outputs. (The choice of 1 and 2 for select values is arbitrary.) For any other values of the select lines, both the A and B outputs are at high impedance. Output polarity for this design is positive.

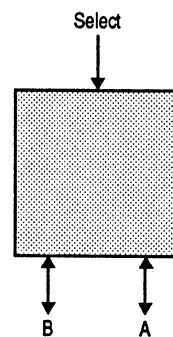
Figure 6-17
*Block Diagram: Bidirectional
Three-State Buffer*



Design Method

A simplified block diagram for the buffer is shown in Figure 6-18. The A and B inputs/outputs are grouped into two sets, A and B. The select lines are grouped into the select set. Figure 6-19 shows the source file that describes the design.

Figure 6-18
*Simplified Block Diagram:
Bidirectional Three-State Buffer*



High-impedance and don't-care values are declared to simplify notation in the source file. The equations section describes the full function of the design. What appear to be unresolvable equations are written for A and B, with both sets appearing as inputs and outputs. The enable equations, however, enable only one set at a time as outputs; the other set functions as inputs to the buffer.

Test vectors are written to test the buffer when either set is selected as the output set and for the case when neither is selected. The test vectors are written in terms of the previously declared sets so that the element values do not need to be listed separately.

Figure 6-19
Source File: Bidirectional
Three-State Buffer

```

module tsbuffer
title 'bidirectional three state buffer      9 Aug 1990
Brenda French & Mary Bailey  Data I/O Corp'

    TSB1      device 'F153';

    S1,S0      Pin 1,2;          Select = [S1,S0];
    A3,A2,A1,A0 Pin 12,13,14,15;  A      = [A3,A2,A1,A0];
    B3,B2,B1,B0 Pin 16,17,18,19;  B      = [B3,B2,B1,B0];

    X,Z        = .X., .Z.;

equations
    A = B;
    B = A;

    A.oe = (Select == 1);
    B.oe = (Select == 2);

test_vectors
    ([Select,  A,  B]-> [  A,  B])
    [ 0      , 0, 0]-> [  Z,  Z];
    [ 0      ,15,15]-> [  Z,  Z];

    [ 1      , X, 5]-> [ 5,  X];
    [ 1      , X,10]-> [10,  X];

    [ 2      , 5, X]-> [ X,  5];
    [ 2      ,10, X]-> [ X,10];

    [ 3      , 0, 0]-> [  Z,  Z];
    [ 3      ,15,15]-> [  Z,  Z];

end

```

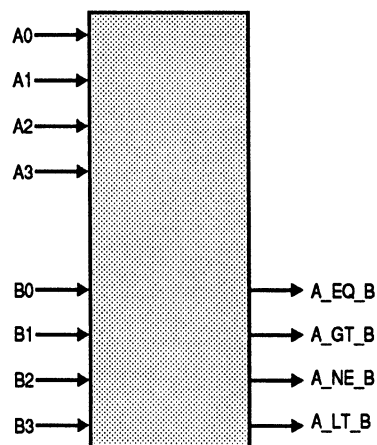
4-Bit Comparator

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.

Design Specification

The comparator, as shown in Figure 6-20, 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.

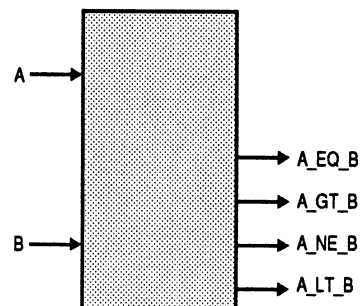
Figure 6-20
Block Diagram: 4-Bit Comparator



Design Method

Figure 6-21 and Figure 6-22 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.

Figure 6-21
Simplified Block Diagram: 4-bit Comparator



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 6-22
Source File: 4-Bit Comparator

```

module COMP4A
title '4-bit look-ahead comparator'
Steve Weil & Gary Thomas Data I/O Corp.'
5 June 1990

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 16,17,18,19;

A_EQ_B istype 'neg';

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

```

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 report output from the compiler (AHDL2PLA) and the simulators (PLASim and JEDSim) 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.

Truth Table Examples

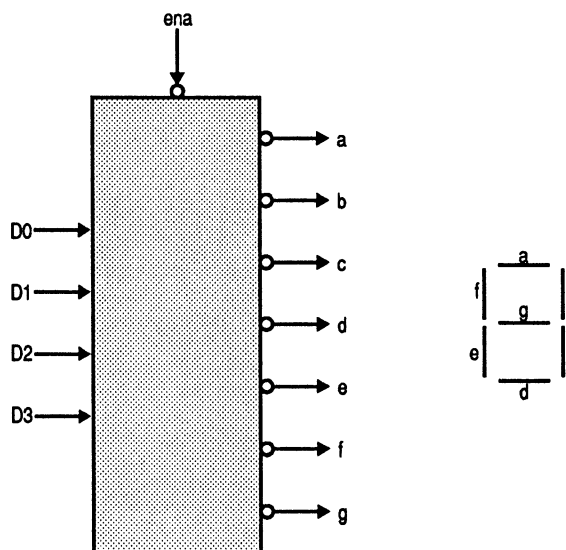
7-Segment Display Decoder

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 6-23 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.

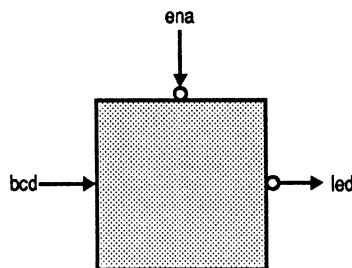
Figure 6-23
Block Diagram: Seven-Segment Display Decoder



Design Method

Figure 6-24 and Figure 6-25 show the simplified block diagram and the source file for the ABEL implementation of the display decoder. 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 6-24
*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.

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.

Figure 6-25
Source file: 4-bit Counter with 2
Input Mux

```

module BCD7
title 'seven segment display decoder    1 Aug 1990
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
"      ---
bcd7 device 'P16P8';

D3,D2,D1,D0,Ena pin 2,3,4,5,6;
a,b,c,d,e,f,g  pin 13,14,15,16,17,18,19 istype 'com';

bcd      = [D3,D2,D1,D0];
led      = [a,b,c,d,e,f,g];

ON,OFF   = 0,1;
L,H,X,Z  = 0,1,.X.,.Z.;

equations
    led.oe = !Ena;

@dcset
truth_table (bcd -> [ 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] -> [ a , b , c , d , e , f , g ])
    [ 0 , 0 ] -> [ ON, ON, ON, ON, ON, ON, OFF];
    [ 0 , 1 ] -> [ OFF, ON, ON, OFF, OFF, OFF, OFF];
    [ 0 , 2 ] -> [ ON, ON, OFF, ON, ON, OFF, ON];
    [ 0 , 3 ] -> [ ON, ON, ON, ON, OFF, OFF, ON];
    [ 0 , 4 ] -> [ OFF, ON, ON, OFF, OFF, ON, ON];
    [ 0 , 5 ] -> [ ON, OFF, ON, ON, OFF, ON, ON];
    [ 0 , 6 ] -> [ ON, OFF, ON, ON, ON, ON, ON];
    [ 0 , 7 ] -> [ ON, ON, ON, OFF, OFF, OFF, OFF];
    [ 0 , 8 ] -> [ ON, ON, ON, ON, ON, ON, ON];
    [ 0 , 9 ] -> [ ON, ON, ON, ON, OFF, ON, ON];
    [ 0 ,10 ] -> [ X , X , X , X , X , X , X ];
    [ 0 ,11 ] -> [ X , X , X , X , X , X , X ];
    [ 0 ,12 ] -> [ X , X , X , X , X , X , X ];
    [ 0 ,13 ] -> [ X , X , X , X , X , X , X ];
    [ 0 ,14 ] -> [ X , X , X , X , X , X , X ];
    [ 0 ,15 ] -> [ X , X , X , X , X , X , X ];
    [ 1 ,15 ] -> [ Z , Z , Z , Z , Z , Z , Z ];

end

```

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.

State Description Examples

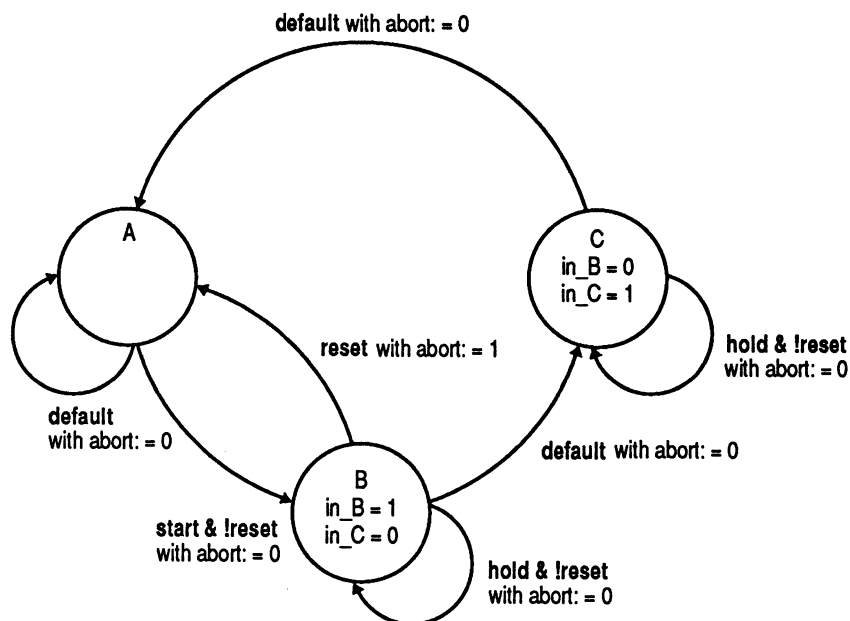
Three-State Sequencer

The following design is a simple sequencer that demonstrates the use of ABEL state diagrams. The design is implemented in a P16R4 device. There is no limit on number of State Diagram states that can be processed by ABEL; the number allowed depends on the number of transitions and the path of the transitions. For example, a 64 state counter uses fewer terms (and smaller equations) than a 63 state counter. For larger counter designs, use the syntax `CountA:= CountA + 1` to create a counter rather than using a state machine. See also example `COUNT116.abl` for further information on counter implementation.

Design Specification

Figure 6-26 shows the sequencer design, with a state diagram showing the transitions and desired outputs. The state machine starts in state A and remains in that state until the 'start' input becomes high. It then sequences from state A to state B, from state B to state C, and back to state A. It remains in state A until the 'start' input is high again. If the 'reset' input is high, the state machine returns to state A at the next clock cycle. If this reset to state A occurs during state B, an 'abort' synchronous output goes high, and remains high until the machine is again started.

Figure 6-26
State Diagram: Three-state Sequencer



During states B and C, asynchronous outputs 'in_B' and 'in_C' become high to indicate the current state. Activation of the 'hold' input will cause the machine to hold in state B or C until 'hold' is no longer high, or 'reset' becomes high.

Design Method

The sequencer is described by using a STATE_DIAGRAM section in the ABEL source file. Figure 6-27 shows the ABEL source file for the sequencer. In the source file, the design is given a title, the device type is specified, and pin declarations are made. Constants are declared to simplify the state diagram notation. The two state registers are grouped into a set called 'sreg'. The three states A, B, and C are declared with appropriate values specified for each.

State values have been chosen for this design that allow the use of register preload to ensure that the machine starts in state A. For larger state machines with more state bits, careful numbering of states can dramatically reduce the logic required to implement the design. Using constant declarations to specify state values saves time when later changes to these values are made.

The state diagram begins with the STATE_DIAGRAM statement that names the set of signals to be used for the state register. The set to be used is 'sreg'.

Within the STATE_DIAGRAM, IF-THEN-ELSE statements are used to indicate the transitions between states, and the input conditions that cause each transition. In addition, equations are written in each state that indicate the outputs required for each state or transition.

For example, state A reads:

```
State A:
  in_B = 0;
  in_C = 0;
  if (start & !reset) then B with
  abort := 0;
  else A with abort := abort;
```

This means that if the machine is in state A and 'start' is high, but 'reset' is low, then the machine will advance to state B, but in any other input condition the machine will remain in state A.

The equations for in_B and in_C indicate the those outputs should remain low while the machine is in state A, while the equations for 'abort', specified with the 'with' keyword, indicate that 'abort' should go low if the machine transitions to state B, but should remain at its previous value if the machine stays in state A.

Test Vectors

The specification of the test vectors for this design is similar to other synchronous designs. The first vector is a preload vector, to put the machine into a known state (state A), and the following vectors exercise the functions of the machine. The A, B, and C constants are used in the vectors to indicate the value of the current state, improving the readability of the vectors.

Figure 6-27
Source File: Three-state
Sequencer

```

module Sequence
title 'State machine example D. B. Pellerin Data I/O Corp';

sequence          device 'pl6r4';

q1,q0             pin    14,15 istype 'reg,invert';
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;

equations
[q1,q0,abort].clk = clock;
[q1,q0,abort].oe  = !enab;

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.fb;

  State B:           " Advance to state C unless reset is active
    in_B = 1;         " or hold is active. Turn on abort indicator
    in_C = 0;         " 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,abort,in_B,in_C])
[ .p. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 1 , 0 ]->[ A , 1 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 1 , 0 , 0 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

end

```

Blackjack Machine

This section contains an advanced logic design described with ABEL 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.

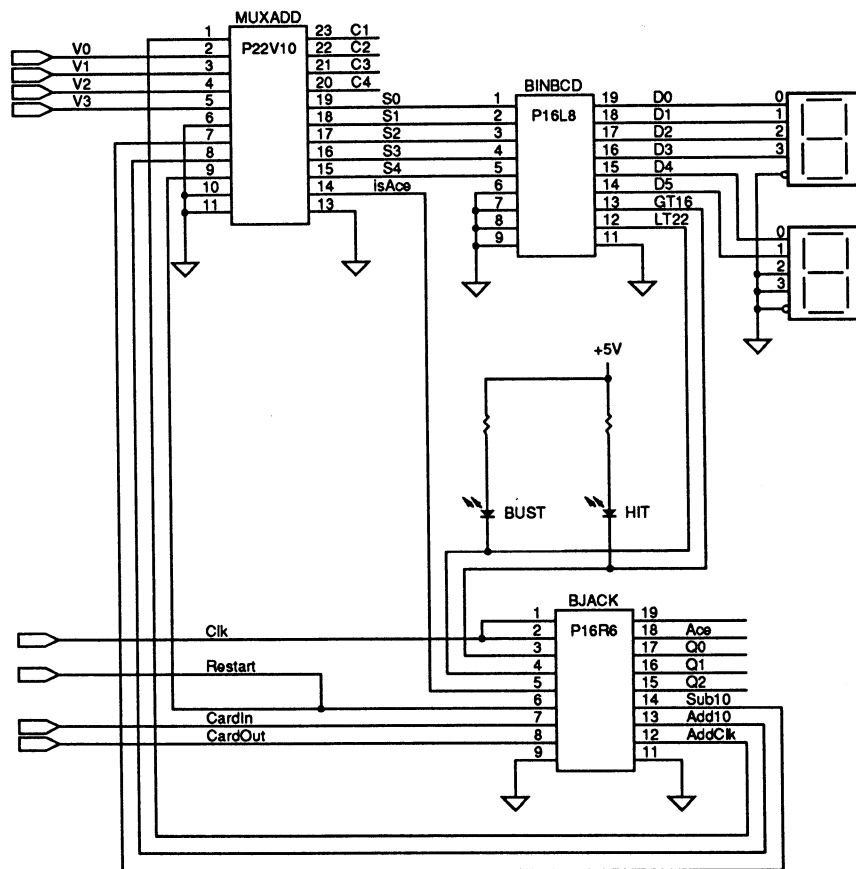
The blackjack machine consists of these functions: a card reader that reads each card as it is drawn, control logic that tells it how to play each hand (based on the total point value of the cards currently held), and display logic that displays scores and status on the machine's four LEDs. (For the 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 6-28 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 6-28
*Schematic of a Blackjack Machine
 Implemented in Three PLDs*



096-0775-001

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 6-28) 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 6-28), 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
P16R6	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 6-29. 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 6-29 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. The trace statement lets you observe the carry signals in simulation.

Figure 6-29
Source File:
Multiplexer/Adder/Comparator

```

module MuxAdd
title '5-bit ripple adder with input multiplex
Michael Holley Data I/O Corp. 26 Mar 1990'

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;

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];

S4,S3,S2,S1,S0 istype 'reg' ;

" Input Multiplexer
Data = Add10 & Sub10 & Card
# !Add10 & Sub10 & ten
# Add10 & !Sub10 & minus_ten;

equations
Score := Data $ Score $ CarryIn;

CarryOut = Data & Score # (Data # Score) & CarryIn;

Score.ar = !(Clr # Clr);

Score.c := AddClk;

is_Ace = Card == 1;

trace
([AddClk,Clr,Add10,Sub10,Card] -> [Score,is_Ace,CarryOut])
test_vectors
([AddClk,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 6-30. 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 6-30 and described in "Test Vectors — BINBCD", 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 and the -e (expand) option. The generated *.lst file shows 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 6-30 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}; (in the binbcd.abl source file shown in Figure 6-30), followed by 31 vectors provided by the @repeat directive. The latter 31 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 6-30 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).

The use of macros and directives to create test vectors is described in more detail in the chapter "Advanced Features."

Figure 6-30
Source file: 4-bit Counter with 2
Input Mux

```
module BINBCD
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;
    LT22,GT16      istype 'neg';

    D5,D4          pin 14,15;
    D5,D4          istype 'neg';
    bcd2           = [D5,D4];
    D3,D2,D1,D0    pin 16,17,18,19;
    D3,D2,D1,D0    istype 'neg';
    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
```

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.

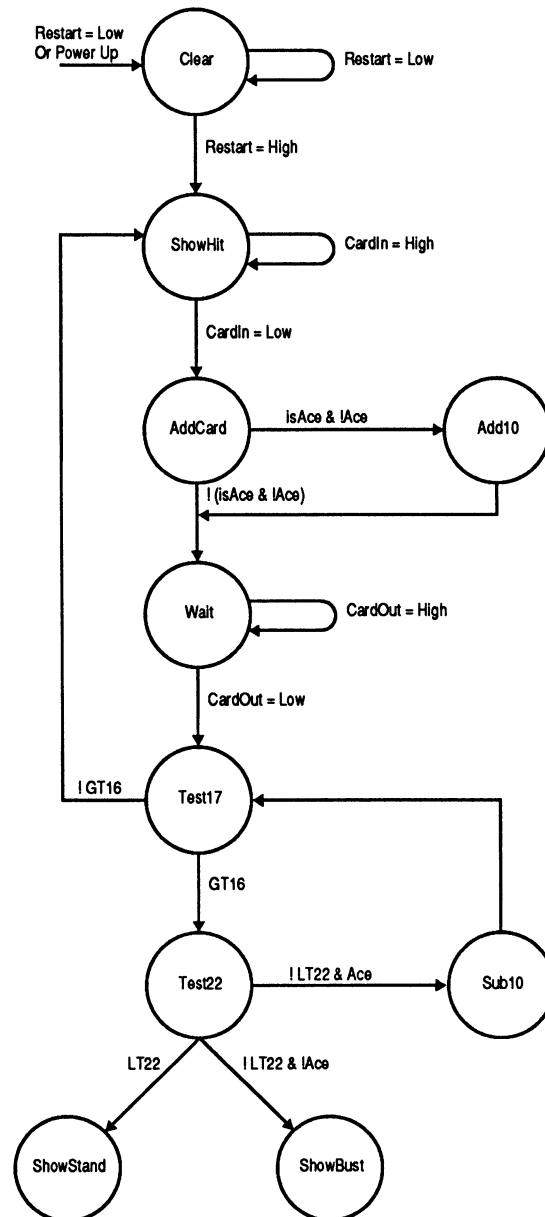
On the basis of these stored states, plus input from each newly drawn card, the blackjack controller decides whether or not a +10 or -10 value is to be sent to the adder.

Design Method — BJACK

The ability of ABEL to accept design input in a variety of forms is especially helpful in the case of state machines, which are easiest to describe with a state diagram.

In describing a state machine, the first step is to develop a bubble diagram. Figure 6-31 shows a bubble diagram (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 6-31
Pictorial State Diagram: Blackjack Machine



You must then express the bubble diagram in the form shown in state_diagram portion of Figure 6-32. There is a one-to-one correlation between the bubble diagram and the state diagram described in the source file (Figure 6-32). The table below provides a more detailed description of the state identifiers (state machine states) illustrated in the bubble 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 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.
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 6-32, each of the state identifiers (For example, ShowHit) is 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. 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 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 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 6-32 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 6-32
Source File: State Machine
(Controller)

```

module BJACK
title 'BlackJack state machine controller
Michael Holley Data I/O Corp. 9 Aug 1990'

    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

    AddClk         istype 'com';
    Ace            istype 'invert,reg_D';
    Add10,Sub10,Q2,Q1,Q0 istype 'invert,reg_D';

    Qstate         = [Add10,Sub10,Q2,Q1,Q0];
    Clear          = [ 1 , 1 , 1, 1, 1]; "31
    ShowHit        = [ 1 , 1 , 1, 1, 0]; "30
    AddCard        = [ 1 , 1 , 0, 0, 0]; "24
    Add_10         = [ 0 , 1 , 0, 0, 0]; "16
    Wait_          = [ 1 , 1 , 0, 0, 1]; "25
    Test_17        = [ 1 , 1 , 0, 1, 0]; "26
    Test_22        = [ 1 , 1 , 0, 1, 1]; "27
    ShowStand      = [ 1 , 1 , 1, 0, 0]; "28
    ShowBust       = [ 1 , 1 , 1, 0, 1]; "29
    Sub_10         = [ 1 , 0 , 0, 0, 1]; "17
    Zero_          = [ 0 , 0 , 0, 0, 0]; "0

    equations
    [Qstate,Ace].c = Clk;
    [Qstate,Ace].oe = !Ena;

```

```
@page
@dcset
state_diagram Qstate
```

```
State Clear:  AddClk    = !ClkIN;
               Ace      := Low;
               if (Restart==Low) then Clear else 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;
               if (Restart==Low) then Clear else ShowBust;

State ShowStand: AddClk    = Low;
               Ace      := Ace;
               if (Restart==Low) then Clear else ShowStand;

State Zero:    goto Clear;
```

@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 ];" 1
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];" 2
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];" 3

[ L , C , L , L , H , L , H , InOut ] -> [ L ,ShowHit , L ];" 4
[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];" 5
[ L , C , L , L , H , L , H , _In ] -> [ L ,Wait , L ];" 6
[ L , C , L , L , H , L , H , InOut ] -> [ L ,Wait , L ];" 7
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_17 , L ];" 8
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];" 9
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];" 10

[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];" 11
[ L , C , L , L , H , L , H , _In ] -> [ L ,Wait , L ];" 12
[ L , C , L , L , H , L , H , InOut ] -> [ L ,Wait , L ];" 13
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_17 , L ];" 14
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_22 , L ];" 15
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowStand , L ];" 16
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowStand , L ];" 17
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];" 18

```

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 ];" 19
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];" 20

[ 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 , 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 ,Test_22 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , L , H , L , L , 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
```


7 Using ABEL Processing Modules

Prerequisites

Before ABEL can work with your design, you need to create a *source file* that contains a description of your design. The ABEL 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 behind the tab "Creating a Design."

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

Source files can be entered in any ASCII editor. The ABEL Design Environment has an internal editor for simple editing which is described here.

If you have source files from earlier ABEL versions, refer to the chapter "Backward Compatibility" for information on program changes and conversion of old files.

Design Flow

Before you need to select a device or assign pins, you can

- Create and compile your design
- Check for syntax errors
- Reduce equations
- Simulate your design.

These functions do not require any device information and are referred to as *architecture-independent* functions.

You must select a device and assign pins before you can

- Create a Programmer Load File
- Simulate the device and design

These functions require a device be specified and pins be assigned, and are referred to as *device-specific* functions.

Input

ABEL software uses a source file, user-specified options and several libraries to process a file. Library locations are specified in the startup files specific to your operating system. Options can be specified from within the ABEL Design Environment or from the command line.

Program Summary

Table 7-1 gives an overview of the main ABEL modules. Detailed information on available options is provided in the following pages.

Menu options are given where they apply below in boxed text in the margins, or with buttons [X] (•) or entry fields *option: 1* .

Command line options are given where they apply below in bold courier font (**bold courier**), and a summary is provided at the end of this chapter and in the Reference Card.

Table 7-1
Program Summary

Function	Program	Input Files	Output Files	Description
ABEL				
Design	abel4	ABEL-HDL .abl	ABEL-HDL .opt ABEL-HDL .sav Internal .dmc	The ABEL Design Environment incorporates an integrated editor and menu system to drive the other ABEL programs (below).
Compile	ahdl2pla	ABEL-HDL .abl Text .inc	PLA .tt1 Vector .tmv List .lst Fuse .fus	Checks the design for syntax errors and then translates it into PLA format.
Simulate Equations	plasim	PLA .ttn Vector .tmv	Results .smn	Simulates the design independent of the device.
Optimize	plaopt	PLA .tt1	PLA .tt2	Optimizes the design using Espresso.
Create Programmer Load File	fuseasm	PLA .ttn Device Library	Load .jed or .pxx Results .doc	Maps a design into the specified part and creates a fuse map in various formats for downloading to programmers.
Simulate JEDEC File	jedsim	JEDEC .jed Vector .tmv Device Library	Results .sim	Simulates the design and the device together to assure compatibility.
SmartPart Option				
Select Devices	devsel	PLA .ttn Device database	Dev. List .sel Chip report .chp	Selects devices that meet the basic requirements of the design.
Fit Devices	fit	PLA .tt2 Dev. List .sel	PLA .tt3 Results .fit	Fits the design into the specified part(s) and assigns pins.
PLDgrade Option				
Fault Grade	afsim	JEDEC .jed Vector .tmv Device Library	Results .fts	Fault grades the design and device and gives the fault coverage of supplied test vectors

Using Help

Help in the ABEL Design Environment

Context-sensitive

Context-sensitive help is available at any point in the ABEL Design Environment by pressing the **Help** key.

The **Help** key is **F1** on a PC, or **L2** on a SUN. For other systems, consult your *Installation Guide* for your help key. The help is specific to your location in the source file or menus.

Help in Edit or View Window

Pressing the **Help** key in the edit window calls up help for the text at the cursor position. Pressing the **Help** key while in Help will give help on key strokes.

Help in a Menu

Pressing the **Help** key in a pull-down menu will give a synopsis of the menu and short descriptions of each item on the menu. Pressing **Help** again provides key stroke help.

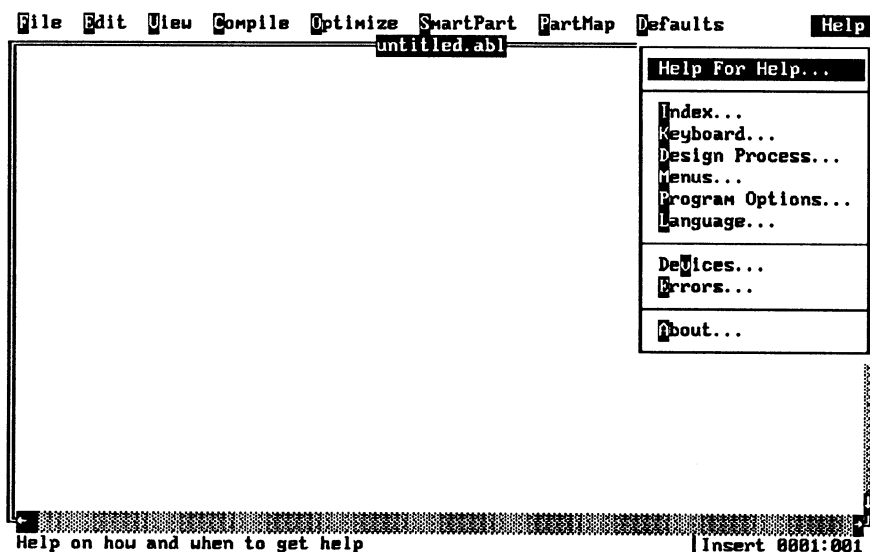
Help in a Dialog Box

Pressing the **Help** key in a dialog box will first give help for the field at the cursor position. Pressing **Help** a second time will give an overview of the dialog box.

Help Menu

The **Help** menu shown in Figure 7-1 contains help on keyboard commands, the design process, program options, language elements, devices and errors.

Figure 7-1
Help Menu



Help for Help ...

The **Help for Help ...** selection contains instructions on using Help.

Index ...

Index ... contains a list of every help topic to select from for online help.

Keyboard ...

Keyboard ... gives help on key strokes in the menus.

Design Process ...

Design Process ... gives help on selected topics in the design process.

Menus ...

Menus ... gives help on each main menu selection.

Program Options ...

Program Options ... gives help on setting program options and summarizes the effect of options.

Language ...

Language ... gives help on ABEL-HDL keywords, syntax and design considerations.

Devices ...

Devices ... contains a list of supported devices and gives the chip diagram with pin numbers for the selected device, plus other useful device-specific information.

Errors ...

Errors ... contains help on program messages received during processing. Note the number of the message, then select the number from the error list for more information on the message.

About ...

About ... gives telephone numbers to call for assistance and ordering, and information on Data I/O's BBS, user registration, warranty service and update service.

Help for the Command Line

Help for command line options and program usage is available for most programs by entering

***program_name* -help**

or

***program_name* -usage**

Using the ABEL Design Environment

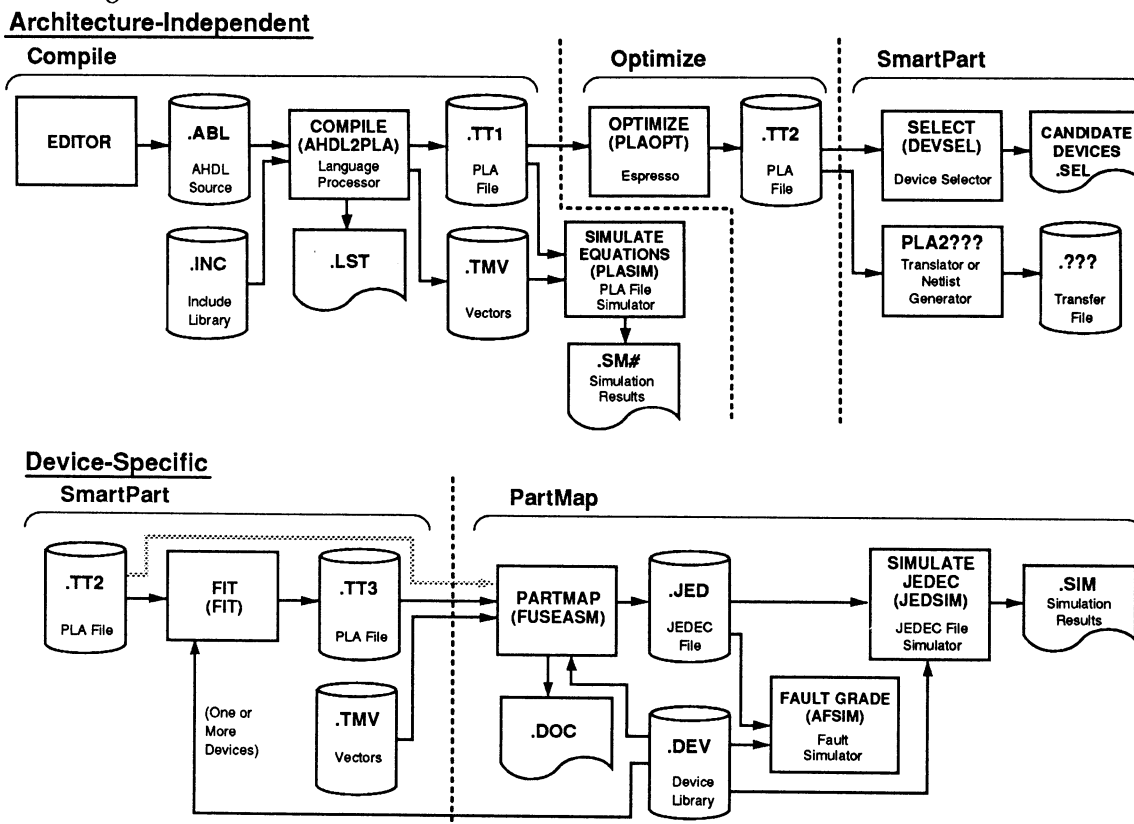
The ABEL Design Environment provides an easy way to specify options for the ABEL processing modules.

The following features of the ABEL Design Environment facilitate the edit-process-error loop:

- Context-sensitive help at every point in the design process
- Auto-update feature that automatically processes the design if any files required for a selected action are missing or out of date.
- Program options are specified interactively, so you do not need to remember command line options.
- Integrated editor allows you to enter a source file, or make simple corrections to existing files.
- The last options chosen for each source file are stored (in *.opt file) and automatically recalled when that source file is opened.

Figure 7-2 shows the functions provided by the ABEL Design Environment main menu.

Figure 7-2
ABEL Design Environment
Menu and Program Flow



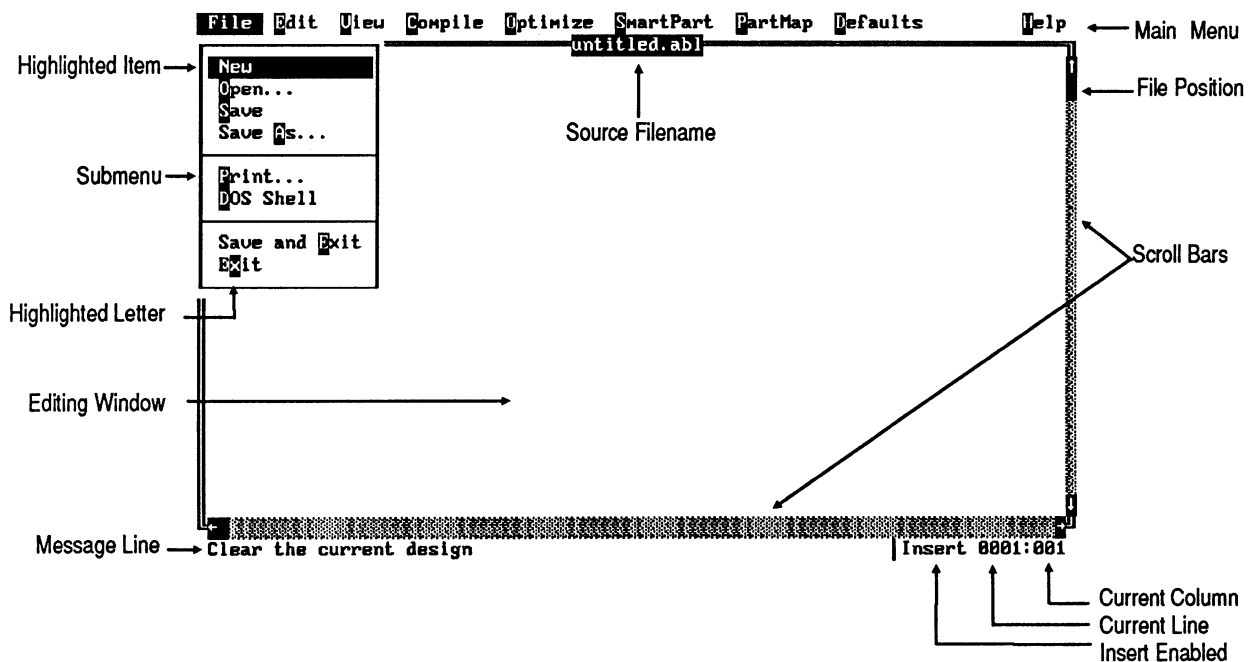
095-0664-001A

Starting the ABEL Design Environment

The ABEL Design Environment is started with the `abel4` executable optionally followed by an initial design filename. `Abel4` will invoke the editor and open the file, if specified.

Figure 7-3 shows the main screen of the ABEL Design Environment. Note that screens will vary slightly from that shown on different systems and with new software releases.

Figure 7-3
Main Menu



Basic Menu Operation

To access a top-level menu selection in the ABEL Design Environment, press **[Alt]** and the highlighted letter. A submenu selection can then be made by pressing the highlighted letter in the desired selection, or by using the arrow keys to highlight the desired selection and pressing **[Enter]**. **[ESC]** toggles between the last menu used and the editor.

Menu selections followed by ellipses (. . .) call up dialog boxes for further information. Selections without ellipses run an ABEL command or perform an action.

Dialog Boxes A dialog box is a screen that allows you to select one to a number of different program options. They can contain command buttons, check boxes, mode buttons and entry fields. Use **Tab** (forward) and **Shift – Tab** (backward) to move between selections in a dialog box. Press **Space** to toggle check boxes on and off and select mode buttons. Select command buttons by pressing **Enter**. See **Keyboard Help** for system-specific commands.

Command Buttons Dialog boxes contain one or more of the following command buttons, which can be selected by highlighting the button and pressing **Enter**.

< OK > The **<OK>** button (or **F5**) saves the entries made to the dialog box and returns you to the previous menu.

< Cancel > The **<Cancel>** button (or **Esc**) cancels the entries made to the dialog box and returns you to the previous menu.

< action > Buttons that contain an action name perform that action when selected.

Check Boxes
[X] Check boxes toggle a selection on or off with **Space**, for example, application program options.

Mode Buttons
(•) Mode buttons select a particular option from a list of mutually exclusive options. Mode buttons are selected with **Space**.

Entry Fields
option: When highlighted, text entered at the keyboard goes to the entry field.

Automatic Design Updating

The ABEL Design Environment has an auto-updating feature that allows the user to specify a desired end result without having to perform the intermediate steps.

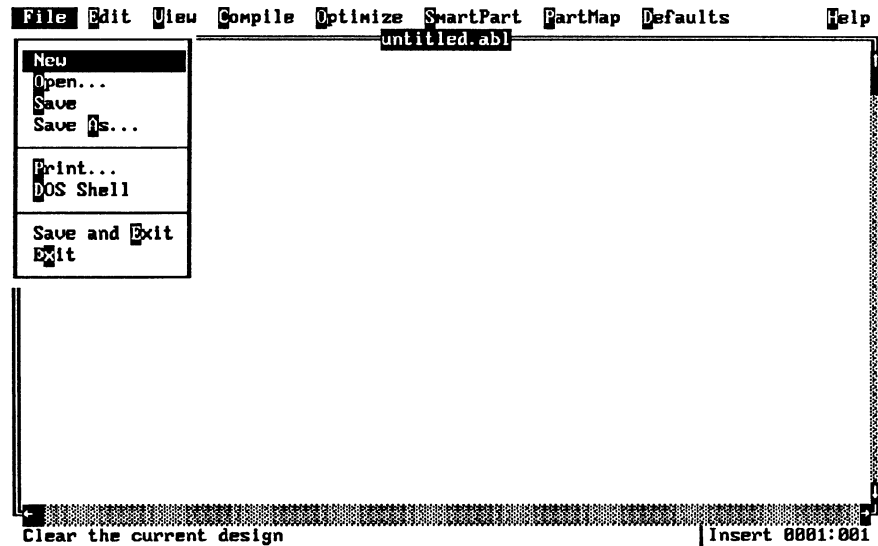
Auto-updating occurs whenever a file is out of date; that is, a preceding file is newer or missing. Auto-updating can be turned off on the **Defaults** menu.

For a demonstration of auto-updating, load any example file into the ABEL Design Environment and select **PartMap**, then **Simulate JEDEC**. The ABEL Design Environment will run all the programs needed to produce a JEDEC file and then simulate it. This is equivalent to using the **abel** batch file in previous ABEL releases, and can also be selected from the menus by pressing **F4**.

File Management

The File menu shown in Figure 7-4 contains basic file and system functions.

Figure 7-4
File Menu



New

Open an empty file.

Open ...

Open a design source file. A dialog box will prompt for the filename.

Save

Save a design source file under the current filename. The current file is also saved automatically whenever the file is compiled.

Save as ...

Save a design source file under a new filename. A dialog box will prompt you for the new filename.

Print ...

Print a file. A dialog box will prompt you for the filename. You can specify any file, including ABEL-HDL Source files, Compiler Listing files and Simulation Results files.

System Commands

Shell

Exits to an operating system shell. To return to the ABEL Design Environment from the shell, enter

exit

Save and Exit

Saves the current file and exits the ABEL Design Environment.

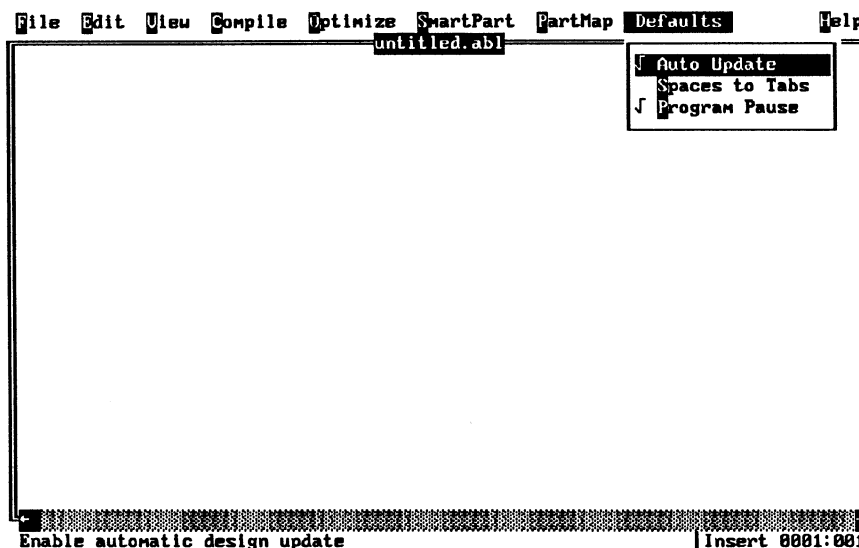
Exit

Exits the ABEL Design Environment without saving the current file.

Setting Menu Defaults

The Defaults menu shown in Figure 7-5 contains global processing options.

Figure 7-5
Defaults Menu



☒ Auto update

Enable or disable automatic design updating. Default is enabled.

☐ Spaces to tabs

If this option is checked, spaces in source file will be replaced with tabs when the file is saved. Default is disabled.

☒ Program Pause

If this option is checked, the ABEL Design Environment will pause after running any program. Default is disabled.

Options

The ABEL Design Environment automatically saves all options set for each source file in *source_filename.opt*. This way, whenever a source file is opened, all options previously set will be restored.

Optional Programs

The menus contain selections that can only be used if you have the ABEL options to which they refer. These optional features are all selections under the **SmartPart** menu, and the **PLDgrade** and **PLDgrade Options . . .** functions under the **PartMap** menu. If you do not have these options installed, their menu selections will not have a highlighted letter and cannot be selected. Contact your Data I/O representative for ordering information.

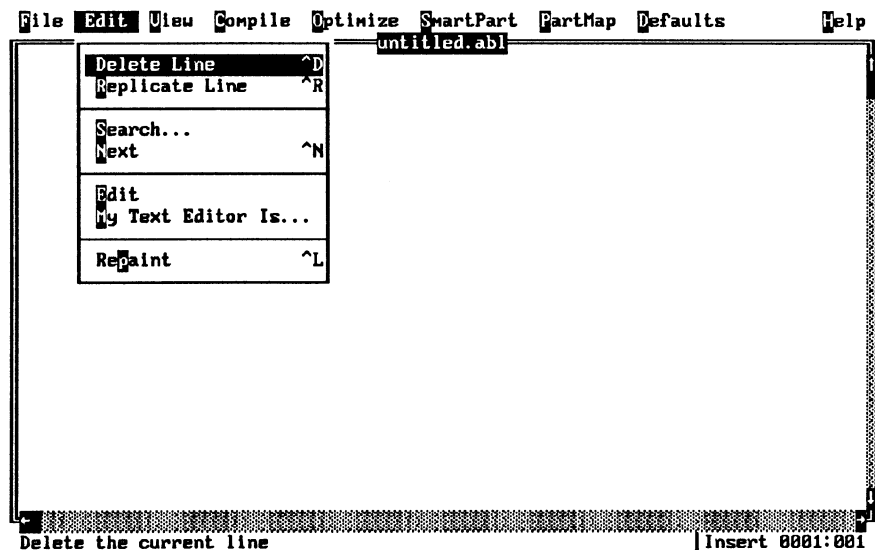
Correcting a Source File

The ABEL Design Environment has a simple integrated editor that is useful for correcting source files. Although you can enter an entire source file, the ABEL Design Environment editor has limited editing functions intended for small corrections. The editing functions are detailed below.

You can use your own editor from the ABEL Design Environment by specifying the executable name under **My Text Editor Is ...** and then selecting **Edit**.

If you use the integrated editor, the **Edit** menu shown in Figure 7-6 contains basic editing functions. Also see **Keyboard Help** for system-specific commands.

Figure 7-6
Edit Menu



Delete Line ^D

Deletes the line the cursor is on. A line can also be deleted by pressing **Ctrl** - **D**.

Replicate Line ^R

Replicates (copies) the line the cursor is on and places it below the current line. A line can also be replicated by pressing **Ctrl** - **R**.

Search ...

Searches for a word in the file.

Next ^N

Finds the next occurrence of the **Search ...** text. The next occurrence may also be found by pressing **Ctrl** - **N**.

Edit

Runs the editor specified with **My Text Editor Is ...**

My Text Editor Is ...

Specifies an editor to use when the **Edit** selection is made.

Repaint	^L
---------	----

Redraws the screen. The screen can also be redrawn by pressing **Ctrl** - **L**.

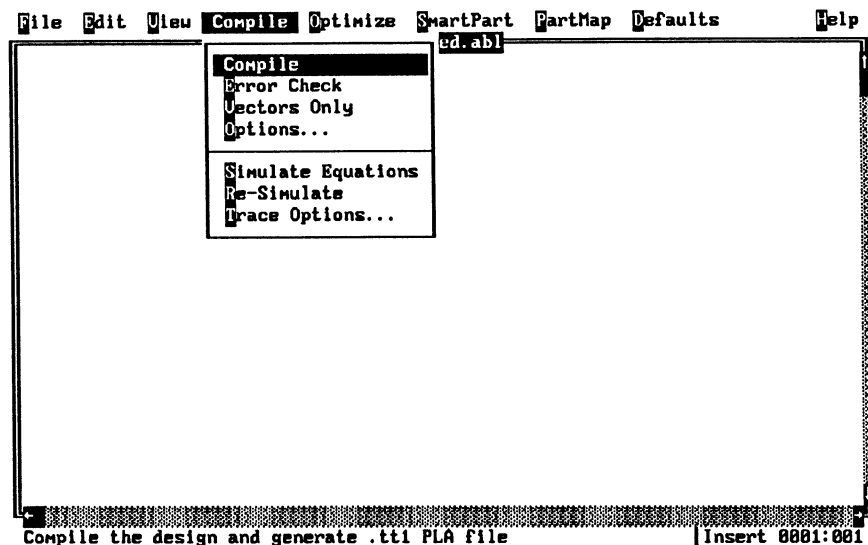
Specifying a Source File from the Command line

An input file is specified on the command line by typing the program name followed by the input filename. The file extension only needs to be specified if it is NOT .abl.

Compiling a Source File

Compilation is performed by the AHDL2PLA program invoked from the Compile menu shown in Figure 7-7.

Figure 7-7
Compile Menu



Compile

```
ahdl2pla ahdlfile[.abl] [ options ]
```

The AHDL2PLA program

- checks for and flags syntax errors in the *module_name.lst* file.
- converts state diagrams and truth tables into equations
- translates test vectors and creates a test vector file, *module_name.tmv*
- expands macros
- converts equations with sets to equivalent equations without sets
- replaces 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 rules in Table 7-2 :

Table 7-2
AHDL2PLA Reduction Rules

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

- transfers the equations into a standard PLA format file (*.ttn)

Setting Compile Options

Error Check

ahdl2pla ahd1file -syntax

Checks for and flags syntax errors. No compilation is performed.

Vectors Only

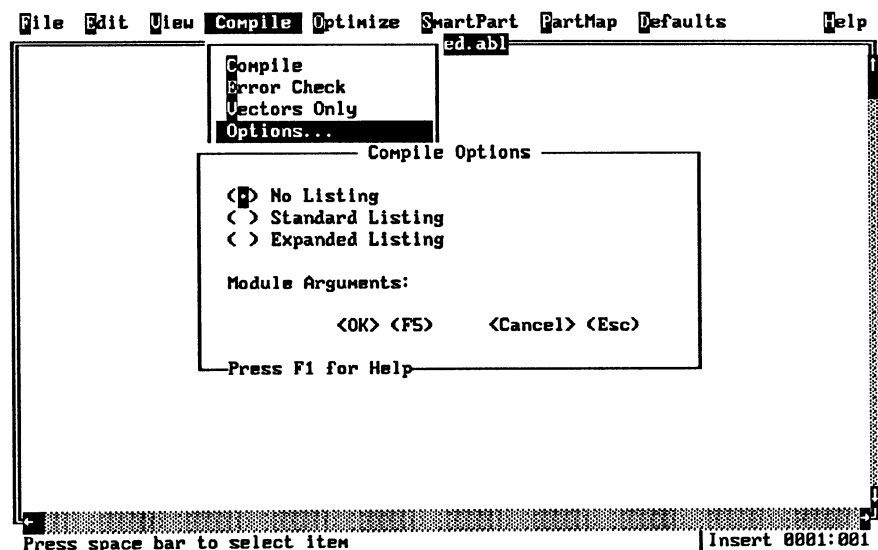
ahdl2pla ahd1file -vectors

Compiles test vectors only and writes a test vector file (*.tmv). The PLA file will not be created. This option is useful if you have edited the test vectors in your design file and need the corrected vectors available in the test vector file. This file will overwrite the previous *.tmv file.

Options ...

The Options ... selection brings up a dialog box with Compile options as shown in Figure 7-8.

Figure 7-8
Compile Options Dialog Box



(•) No Listing

ahdl ahdlfile

The No Listing option means no list file will be created.

(•) Standard Listing

ahdl2pla ahdlfile -list

The Standard Listing option causes the parsed source code to be written to a list file. The listing contains the source file as it was before processing plus error messages.

(•) Expanded Listing

ahdl2pla ahdlfile -list expand

The Expanded Listing option causes the parsed and expanded source code to be written to a list file. Text included by macros and directives is shown with the directives that caused code to be added to the source.

Module Arguments:

ahdl2pla ahdlfile -args args

The Module Arguments 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 separated by spaces. Argument substitution is discussed further in the "Language Reference."

Output Vector Filename

ahdl2pla ahdlfile -ovectors filename

The -ovectors option specifies a filename for the test vector file output by AHDL. If a filename is not specified with -ovectors, the test vectors are written to *module_name.tmv*.

Batch File Output

ahdl2pla ahdlfile -batch

The -batch option creates a batch file that can be used by the abelbat2 file.

Simulating a Source File

Two Kinds of
Simulation

The simulation facilities of ABEL allow you to find and correct design errors at two stages in the design process. PLASim simulates equations and allows you to debug your design before selecting a device and assigning pins. JEDSim simulates the function of the design within the selected device so problems between the device and design can be caught before programmable logic devices are programmed.

Simulation of the logic description in the source file is useful in the early design stage before a device has been selected, but does not take the device that is to be programmed into account. To simulate whether the programmed device would function as desired, device specific information must be used in conjunction with the programming information. This is how the JEDSim module performs simulation. Simulation is covered in detail in the chapter "Advanced Features."

Response files are useful for specifying custom sets of simulation options when running simulation from the command line.

Simulating Equations

Simulate Equations

```
plasim plfile.ttn [options]
```

The ABEL processor PLASim simulates equations in a PLA file. Any PLA file can be used for equation simulation, so you can simulate your design after compilation, after optimization and after device fitting. The **Simulate Equations** and related menu selections are found in the **Compile**, **Optimize**, and **SmartPart** menus.

Selecting **Simulate Equations** from the **Compile** menu, or using PLASim from the command line by specifying the PLA output file produced by AHD2PLA (usually .tt1) simulates the equations in the unoptimized PLA file.

Re-Simulate

```
plasim plfile.ttn [options]
```

Re-Simulate from the **Compile** menu causes AHD2PLA to parse only the test vectors section of the source file, then uses PLASim to simulate equations with the updated test vectors. The **-ivectors** option on the command line is used to specify a filename for a set of test vectors to be used in simulation. If **-ivectors** is used, the test vectors specified in *filename* are used and the test vectors in the input file are ignored.

Optimized and Fitted Equations

Simulating the compiled equations before optimization and fitting is most efficient; however, you can simulate optimized or fitted equations by selecting **Simulate** from the **Optimize** or **SmartPart** menus, or on the command line by specifying the PLA output files produced by PLAOpt (usually .tt2) or Fit (usually .tt3). Fitted equations are produced by an optional device fitter program called SmartPart.

Simulating Design and Device

The **Simulate JEDEC** menu selection (JEDSim program) from the **PartMap** menu does not execute equations or apply inputs to ABEL 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. A device must be selected or a device list specified. JEDEC files produced by the **PartMap** menu selection (Fuseasm program) already contain the device type.

Simulate JEDEC

```
jedsim jedecfile -device device
```

The **Simulate JEDEC** selection from the **PartMap** menu simulates the design using the options set in **Trace Options ...**. JEDSim uses the device specified in **Trace Options ...**, or in the PLA input file.

Re-Simulate JEDEC

```
jedsim jedecfile -ivectors vector_file
```

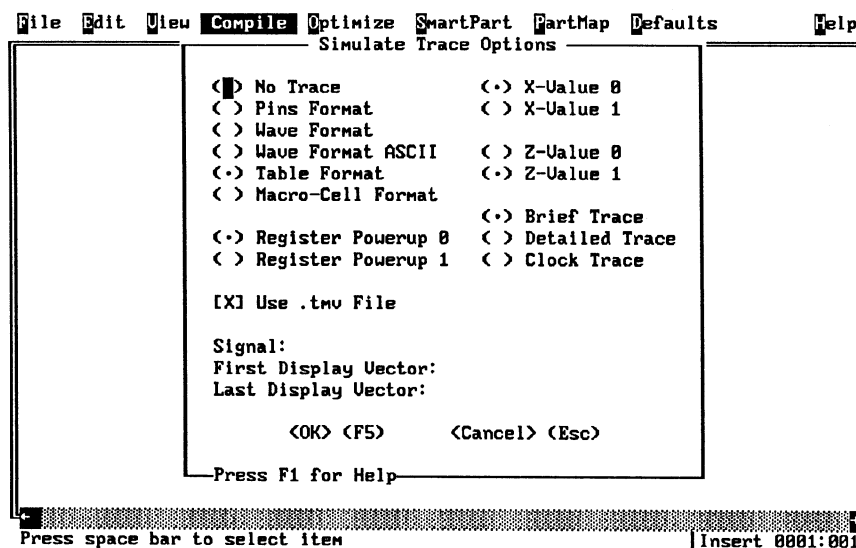
Re-Simulate JEDEC option from the **PartMap** menu simulates equations in the JEDEC file with updated test vectors. Auto-updating runs AHDL2PLA on the test vector section of the source file, runs Fuseasm with the new vector file, then runs JEDSim on the new JEDEC file.

Setting Simulation Trace Options

Trace Options ...

The **Trace Options ...** selection calls up the **Simulate Trace Options** dialog box shown in Figure 7-9 for setting equation simulation trace options. Though the **Trace Options ...** selection appears on several menus, they all call up the same dialog box. Changing the trace options under **Compile** changes the trace options under **Optimize**, **SmartPart** and **PartMap**. The trace options determine the level of information that is provided by PLASim and JEDSim. 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. The trace level is used for all vectors unless specific vectors are specified with **-break**. When **-break** is used, only errors are shown for vectors not specified.

Figure 7-9
Simulate Trace Options Dialog Box



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

(•) No Trace

```

plasim plfile -trace none
jedsim jedecfile -trace none

```

Only errors are shown.

(•) Pins Format

```

plasim plfile -trace pins
jedsim jedecfile -trace pins

```

The pins option shows test vectors for each pin for the device.

(•) Wave Format

```

plasim plfile -trace wave
jedsim jedecfile -trace wave

```

The wave option 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. The wave option output will be printed in character graphics on the PC; ASCII characters can be specified by specifying **-trace wave ascii**.

When JEDEC vectors are used, the number of signals listed for wave format is limited to 25. If no signals are specified with **Signal** (or **-signal**), and the design contains more than 25 signals, inputs with the largest pin number are omitted first, then outputs are omitted.

**(•) Wave Format
ASCII**

```

plasim plfile -trace wave ascii
jedsim jedecfile -trace wave ascii

```

This option is the same as the wave option, except that the waveform will be printed in ASCII characters instead of IBM-PC character graphics.

(•) Table Format

```

plasim plfile [-trace table]
jedsim jedecfile [-trace table]

```

The table option is similar to trace wave except that the waveform is replaced by H, L, and Z for logic high, logic low, and high-impedance state. This is the default trace format.

When JEDEC vectors are used, the number of signals listed for table format is limited to 60. If no signals are specified with **Signal** (or **-signal**), and the design contains more than 60 signals, inputs with the largest pin number are omitted first, then outputs are omitted.

**(•) Macro-Cell
Format**

```

plasim plfile -trace macro
jedsim jedecfile -trace macro

```

The Macro-Cell option shows the internal nodes and the device outputs plus the test vectors. Use Macro-Cell with **detail** for the most help with determining where and why simulation errors occur.

If no signals are specified with **Signal** (or **-signal**), the first output macrocell will be shown. This format produces large files; especially when used with **detail**. Use break points for desired vectors (First/Last Display Vector or **-break**) to limit the size of the file.

(•) Reg. Powerup *n*

```
plasim plaffile -initial (0|1)
jedsim jedecfile -initial (0|1)
```

The powerup state of all registers for simulation can be set with the Register Powerup or -initial option. -initial 1 or -initial h sets all registers to 1, while -initial 0 or -initial l sets all registers to 0. If no -initial option is specified, registers are set to the default state specified in the device file.

(•) X-value *n*

```
plasim plaffile -x (0|1|h|l)
jedsim jedecfile -x (0|1|h|l)
```

(•) Z-value *n*

```
plasim plaffile -z (0|1|h|l)
jedsim jedecfile -z (0|1|h|l)
```

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 an "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 JEDSim operation with the don't-cares set to 0 (option -x 0), and then again with them set to 1 (option -x 1). Refer also to "Don't Cares in Simulation" in the chapter "Using Advanced Features."

(•) Brief Trace

```
plasim plaffile -trace brief
jedsim jedecfile -trace brief
```

The brief option shows the final output (after the outputs have stabilized) and test vectors for errors only. Brief is the default for all format options.

(•) Detailed Trace

```
plasim plaffile -trace detail
jedsim jedecfile -trace detail
```

The detail option shows all iterations for each vector before the part stabilized.

(•) Clock Trace

```
plasim plaffile -trace clock
jedsim jedecfile -trace clock
```

The clock option shows three iterations for clocked vectors, and two iterations for up/downs.

[X] Use .tmv File

```
plasim plaffile -ivectors filename
jedsim jedecfile -ivectors filename
```

Use the .tmv file in place of test vectors in the file. The -ivectors option on the command line is used to specify a filename for a set of test vectors to be used in simulation. If -ivectors is used, the test vectors in *vector_file* are used and the test vectors in the input file are ignored.

The default operation of PLASim is to use the vectors in the .tmv file with the same base name as the PLA input file. JEDSim by default uses the vectors in the JEDEC file.

Note that the JEDEC files used during device simulation contain test vectors only for input and output pins. This option allows you to include test vectors for internal nodes in the JEDEC simulation.

Signal: 1

```
plasim plafile -signal signal_names #s
jedsim jedecfile -signal signal_names #s
```

When a trace method is specified, the Signal option may be used to specify the signal names and/or pin or node numbers to be watched during the simulation process. If no entries are given for Signal, all signals used in the test vectors will be watched. Pin or node numbers can be specified by looking in the PLA file for the column number of the desired signal; or if you have already assigned pins. You can also run Simulate Equations with the macrocell format trace method and use any of the identifiers used in the output file.

Each specified signal name is separated by a space. For example,

```
Signal: sig1 sig2 20
```

The order the signal names are entered on the command line determines the order of the data in the output file.

You can insert a blank column in the Table and Macrocell formats by entering 999 as a Signal option. For example, to insert a blank column between sig1 and sig2, enter

```
Signal: sig1 999 sig2
```

Note: Dot extensions must be included when specifying signal names.

First Display Vector: 1
Last Display Vector: 1

```
plasim plafile -break first_vector [last_vector ]
jedsim jedecfile -break first_vector [last_vector ]
```

The First/Last Display Vector options allow you to view simulation output for only specific test vectors. This selective tracing can be useful in large designs to pinpoint simulation errors. A break point is specified with the number of the test vector at which the break is to occur.

When break points are specified, trace None is used until the first vector is reached, then the trace level specified with -trace is used for the vectors specified. After the last vector specified with -break, the trace level returns to None. If a last vector is not specified, all vectors following the first vector will be traced. Only one -break option is allowed. For example,

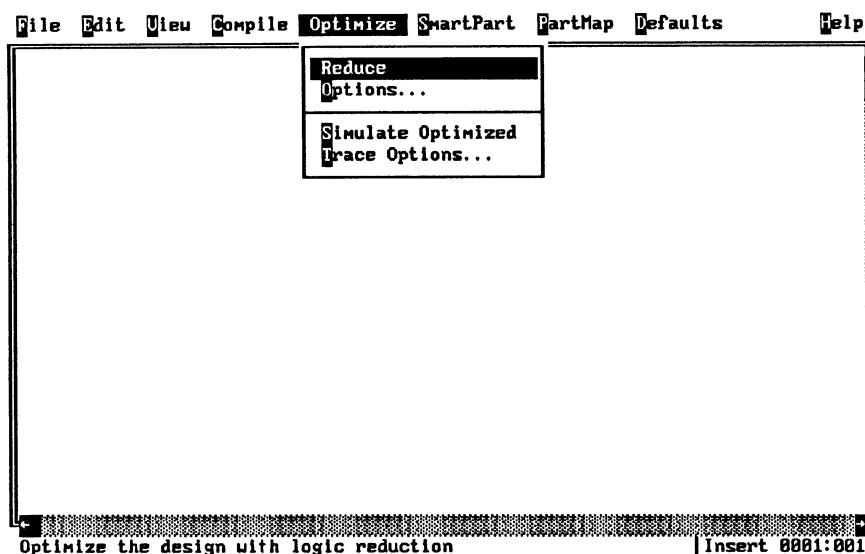
```
jedsim u09a.jed -o u09a.sim -trace macro -break 5 8
```

This starts the simulator with the default trace none. At the fifth test vector, the trace is set to macro and remains there until after vector 8, when the trace is reset to none.

Optimizing a Source File

The **Optimize** menu shown in Figure 7-10 contains optimization and equation simulation functions.

Figure 7-10
Optimize Menu



Reduce

plaopt plafile

Optimizes the design with the options specified. **Optimize** (PLAOpt) produces a temporary file with the module name and a .tt2 extension.

Setting Optimization Options

Options...

plaopt plafile -reduce [options]

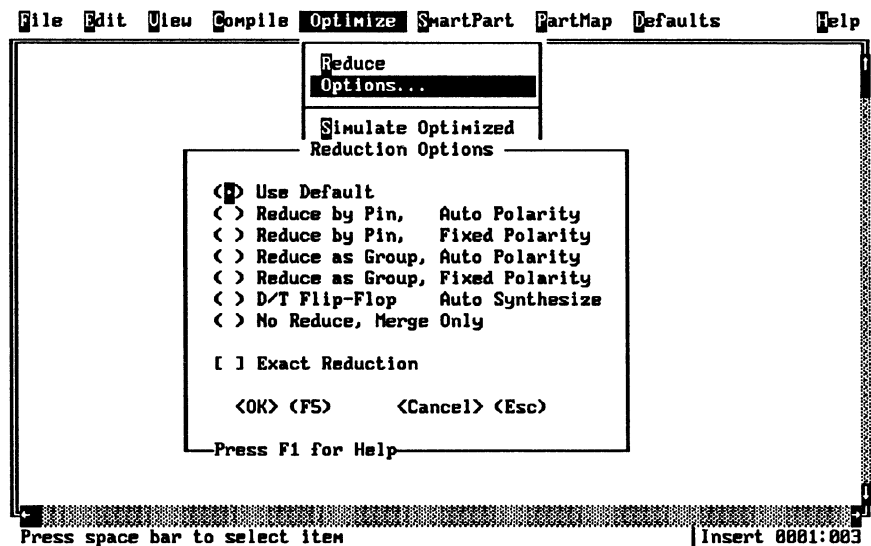
The **Options ...** selection calls up a dialog box with optimization options.

Certain reduction methods are mutually exclusive: you can choose either **bypin** or **group** reduction, and either **auto** or **fixed** polarity. The exact reduction method can be selected with any valid combination of the other options.

The **Optimize Options** dialog box is shown in Figure 7-11:

The auto-polarity selections allow PLAOpt to produce equations for both positive and negative polarity. If both sets of equations are available, the fitter programs and Fuseasm will choose the best set for the selected device. The fixed polarity selections limit PLAOpt to the polarity specified with 'pos' or 'neg' in the file ('pos' is implied if no attribute is specified), and subsequently limit device selection to devices with the specified polarity.

Figure 7-11
Optimize Options Dialog Box



(•) Use default

plaopt plafile

Uses the in-file setting or the program default to reduce the file. If you do not explicitly choose a reduction level, Espresso pin-by-pin, auto-polarity reduction is performed unless you have already declared a device name that starts with an F. For a declared device with an initial F (FPLA architecture), PLAOpt performs Espresso group, fixed polarity reduction.

(•) Reduce by Pin,
Auto Polarity

plaopt plafile -reduce bypin choose

The **bypin** option produces the best fit for PAL architecture devices and a satisfactory fit for FPLAs. Espresso by the pin performs logic reduction for each output, independent of the other outputs.

The **Auto-polarity** reduction option tells PLAOpt to optimize each output for both polarities so the Fitter and Fuseasm can choose the polarity that results in the smallest number of product terms.

(•) Reduce by Pin,
Fixed Polarity

plaopt plafile -reduce bypin fixed

The **bypin** option produces the best fit for PAL architecture devices and a satisfactory fit for FPLAs. Espresso by the pin performs logic reduction for each output, independent of the other outputs.

The **Fixed polarity** reduction option tells PLAOpt that polarity is fixed and cannot be changed to enhance optimization. In this case, polarity is controlled by the 'pos' and 'neg' attributes.

**(•) Reduce as Group,
Auto Polarity****plaopt *plafile* -reduce group choose**

The **group** reduction option will minimize equations for an FPLA architecture where a single product term can be used by multiple outputs.

The **Auto-polarity** option tells PLAOpt to choose the polarity that results in the fewest total number of product terms. This option should only be used for FPLA devices that feature programmable polarity, since PLAOpt will not produce both the positive and negative forms of equations reduced as a group.

**(•) Reduce as Group,
Fixed Polarity****plaopt *infile* -reduce group fixed**

The **group** reduction option will minimize equations for an FPLA architecture where a single product term can be used by multiple outputs.

The **Fixed polarity** reduction option tells PLAOpt that polarity is fixed and cannot be changed to enhance optimization. In this case, polarity is controlled by the 'pos' and 'neg' attributes.

**(•) D/T Flip-flop
Auto-Synthesize****plaopt *infile* -reduce dt**

The **-reduce dt** option makes the most optimal use of D/T flip-flop emulation. PLAOpt will use a mixture of D-type and T-type flip-flops to obtain the smallest possible implementation. This option can save many product terms in configurable D/T registers or parts with XOR terms.

See also "Using XORs for Flip-flop Emulation" in the chapter "Design Considerations."

**(•) No Reduce,
Merge Only**

Merges all compiled equations into a single ABEL-PLA file. No logic reduction is performed.

[X] Exact Reduction**plaopt *infile* -reduce exact**

The **Exact reduction** option determines the minimum solution by removing the order dependence of the input data. This process can be time consuming, and should only be selected if you can't fit your design into a device using the other options, and you suspect that additional minimization is possible. (There are rare cases when the exact option produces less optimal solutions.)

Selecting a Device

SmartPart

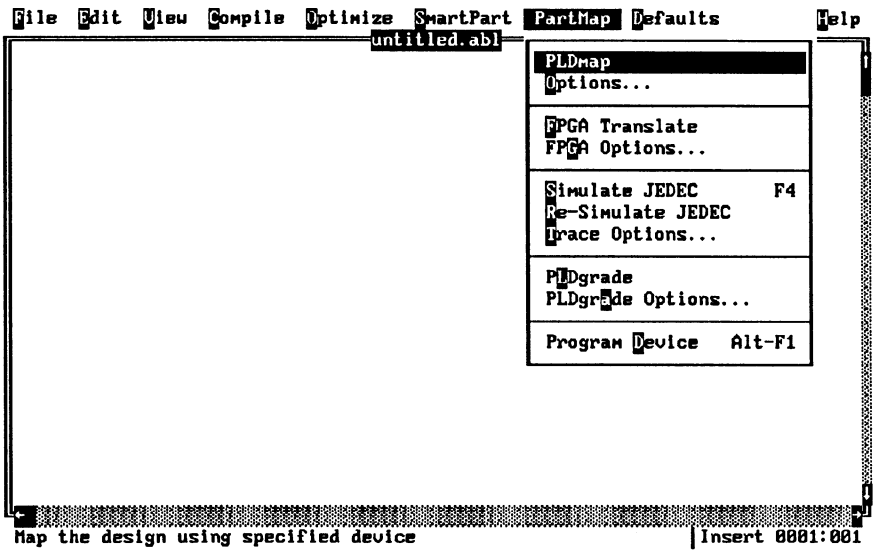
The **SmartPart** menu contains automatic device selection, device fitting, and equation simulation functions. The SmartPart options are available only if you have the optional SmartPart™ Device Selection and Fitting software. See the *SmartPart User Manual* if you have this option.

Creating a Fusemap from a Design

Mapping from a PLA File

The **PartMap** menu shown in Figure 7-12 contains fusemap creation and JEDEC simulation functions. JEDEC simulation is covered above in "Simulating a Design."

Figure 7-12
PartMap Menu



PLDmap

fuseasm filename

The programmer load file created by **PLDmap** (Fuseasm) contains fuse states for programming the device and test vectors to test it once it has been programmed. **PLDmap** creates a JEDEC format load file unless you specify a different format under **PartMap: Options** . . .

The filename of each load file is the name of the device identifier for the module. The file extension is **.jed** for JEDEC format programmer load files, **.pof** for Altera POF format and **.pxx** for all other load file formats, where **xx** corresponds to the number specified with the format option.

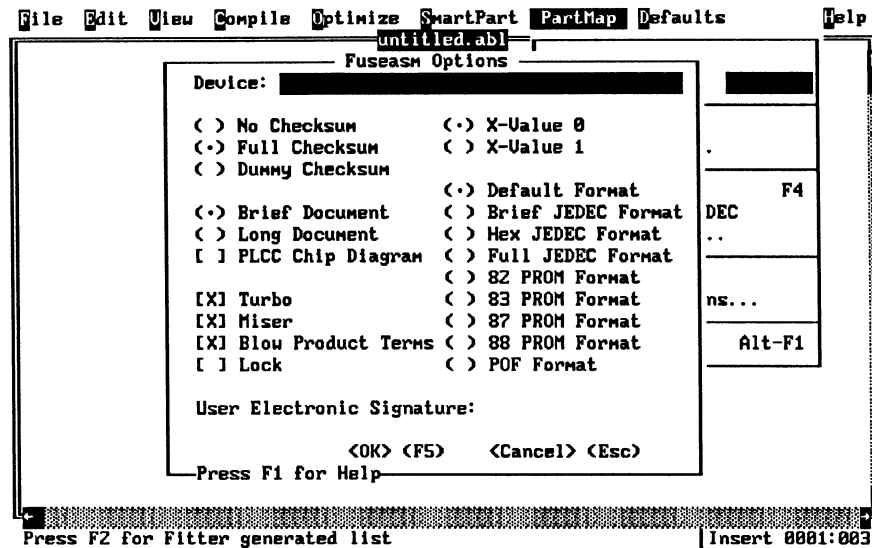
The JEDEC files describe the logic and test functions to be programmed into the device, and contain the test vectors included in the design file.

Setting Map Options

Options...

Calls up a dialog box for setting Fuseasm Options shown in Figure 7-13.

Figure 7-13
PartMap Options Dialog Box



Device: l

`fuseasm platile -device device`

Specifies the device to map the design into. The device must be supported by ABEL, and is specified by a device filename.

If you have SmartPart, press **[F2]** to select from the list of devices that SmartPart selected and fit. If you do not have SmartPart, or have not run Fit or Fit from List, you must manually specify the device unless a device was specified in the ABEL-HDL source file.

() No Checksum

`fuseasm platile -checksum none`

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

() Full Checksum

`fuseasm platile -checksum full`

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

() Dummy Checksum

`fuseasm platile -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 calculations.

() Brief Document

`fuseasm platile -document brief`

The brief document file contains Programmed Logic, Chip Diagram, and Resource Allocation report.

(•) Long Document**`fuseasm plfile -document long`**

The long document file contains all of the sections of the brief document, plus a fusemap.

**[X] PLCC Chip
Diagram****`fuseasm plfile -document plcc`**

Specifies PLCC format chip diagram for .doc file.

[X] Turbo**`fuseasm plfile -config noturbo`**

The E0310, E0320, E0600, E0900, and E1800 have 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. The **Turbo** option for Fuseasm controls the correct fuses for each device.

To put the device in the low power mode, turn off the **Turbo** check box, or on the command line, use

`fuseasm plfile -config noturbo`**[X] Miser****`fuseasm plfile -config nomiser`**

The E0320 and E1800 also have MISER bits which can be programmed. To turn off this option, turn off the **Miser** check box, or on the command line, use

`fuseasm plfile -config nomiser`**[X] Blow Product
Terms****`fuseasm plfile -config ptintact`**

The **Blow product terms** option tells Fuseasm to disconnect unused fuses in the OR plane of an FPLA. To leave unused fuses connected, turn off the **Blow product terms** check box, or use the command line:

`fuseasm plfile -config ptintact`

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 in non-erasable parts prevents future modifications.

[X] Lock**`fuseasm plfile -config lock`**

Program the security fuse.

(•) X-value *n***`fuseasm plfile -x (0|1)`**

Gives the value for don't cares in the fusemap. **X-value** (-x (0|1)) overrides the default values. As a default, ".X." is given the logical value 0.

(•) type Format**`fuseasm plafile -format type`**

Default selects the download file format appropriate for the part selected. JEDEC brief is used for most PLDs. Supported microprocessor formats, their codes, and the file extension given to the programmer load files are given in Table 7-3.

Table 7-3
*Data Translation Format Codes
and File Extensions*

Format	Format Code	File Ext.
JEDEC — Brief fuse list (default)	brief	.jed
JEDEC — Normal fuse map	full	.jed
JEDEC — Hex fuse list*	hex	.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
Altera Programmer Object File	pof	.pof

* JEDEC hex works with UniSite Version 2.7 or later.

Note: PROM programmer load file formats (Motorola and Intel) do not contain test vectors, even if test vectors were specified in the source file.

**User Electronic
Signature: I__**
`fuseasm plafile -ues signature_word`

The P16V8, P16Z8, P18V10, P20V8, P22V10G, P26CV12, F6001 and other devices all have bits reserved for a User Electronic Signature Word. Use the **-ues** option to set the electronic ID. Only one word can be specified for *signature_word*.

FPGA Translation

FPGA Translate

Translates a PLA file into the selected FPGA format.

FPGA Options ...

Calls up a dialog box for setting FPGA Translate Options.

(•) PDS**`pla2eqn plafile -language pds`**

Translates the PLA file into PDS (PALASM-2) format.

(•) Xilinx PDS**`pla2eqn plafile -language lca`**

Translates the PLA file into Xilinx PDS format.

(•) Signetics Snap**`pla2eqn plafile -language snap`**

Translates the PLA file into Signetics Snap format.

(•) ABEL-PLA

Uses ABEL-PLA format for FPGAs. Copies the .tt2 file to .pla so that cleanup4 will not delete the PLA file.

Fault Grading

PLDgrade

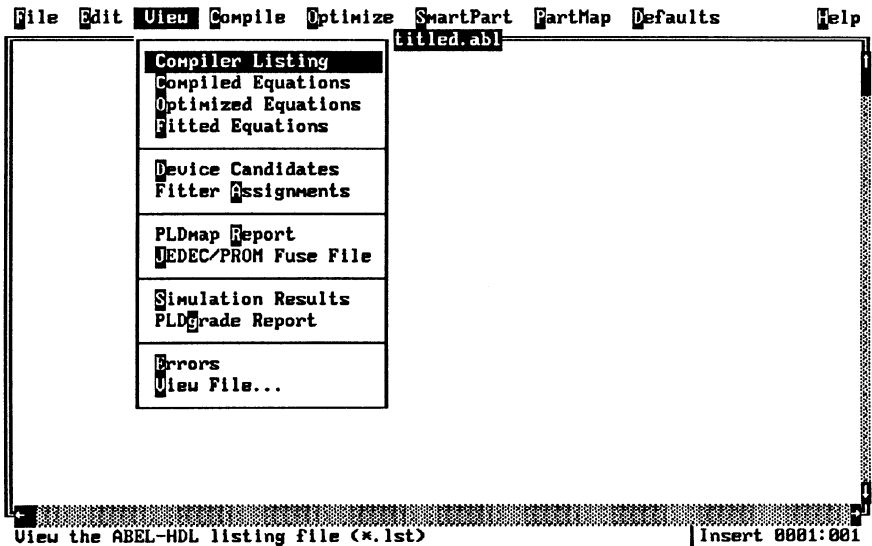
PLDgrade

The **PartMap: PLDgrade** menu selection performs fault simulation and testability analysis functions. The PLDgrade selections are only available if you have the PLDgrade option. See the *PLDgrade User Manual* if you have this option.

Viewing Processing Results

The View menu shown in Figure 7-14 contains several options for viewing processing results. From the command line, use your operating system commands to view the file (for example, type *filename* for DOS).

Figure 7-14
View Menu



The PLA files (.ttn) mentioned below contain logic descriptions in a standard PLA format. The PLA format file is used for optimization, simulation of equations, selection and fitting of a device, and finally generation of a programmer load file. PLA format files can also be transferred to other vendor tools, and PLA files created by other tools can be used with the ABEL language processors.

Compiler Listing

Displays the listing file created by the AHDL2PLA compiler. These files are named *module_name.lst*. Contains the source code, error messages, and the effect of @INCLUDE directives.

Compiled Equations

Displays the generated equations created by the AHDL2PLA compiler. These files are created by PLA2EQN from PLA files.

Optimized Equations

Displays the optimized equations created by the PLAOpt optimizer. These files are PLA files named *module_name.tt2*.

Fitted Equations	Displays the fitted equations created by the fitter. These files are PLA files named <i>module_name.tt3</i> .
Device Candidates	Displays the list of device candidates created by Devsel, the Device Selector. These files are named <i>module_name.sel</i> . The Device Selector is part of the optional SmartPart program.
Fitter Assignments	Displays the pin assignments made by the Fit program in the file named <i>module_name.fit</i> . The Device Fitter is part of the optional SmartPart program.
PLDmap Report	<p>Displays the map document files created by Fuseasm. These files are named <i>device_id.doc</i>, and contain a chip diagram, reduced logic equations and Device Utilization report.</p> <p>In the fuse map, intact connections are shown as Xs, and blown fuses (no connection) are shown as dashes.</p>
JEDEC/PROM Fuse File	<p>Displays the fuse file (also called the programmer load file) created by Fuseasm. These files are named after the module name, with an extension that indicates the format (for example, *.jed). The programmer load files contain the data necessary for a logic programmer to program a logic device with your design. The default JEDEC format gives a programmer load file that contains the fuse states, test vectors and other design information. 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 Plus program.)</p> <p>One programmer load file is created for each device specified in the source file. Complete specification of the JEDEC standard format is given in an appendix.</p>
Simulation Results	Displays the simulation results file created by the latest run of PLASim or JEDSim. These files are named <i>module_name.smn</i> (PLASim) or <i>module_name.sim</i> (JEDSim). Output from the simulation step indicating whether simulation was completed successfully, and if not, where predicted and actual results differed. See "Advanced Simulation" in the chapter "Advanced Features" for more information.
PLDgrade Report	Displays JEDEC fault grading results. Fault grading is provided by the optional PLDgrade program.
Errors	Displays error file created during processing.
View File . . .	Displays any file. If you get an out-of-memory message, the file is too large to be displayed in the ABEL Design Environment.

Downloading to Programmers

Downloading the programmer load file differs for the model of logic programmer used. In most cases, you must set up your system to transmit the *.jed file via a serial I/O port to the programmer. The programmer must be set up to receive the load file before the system begins transmission of the load file. The mechanics of establishing the communications link between the system and the programmer depend upon your hardware.

Program Device . . .

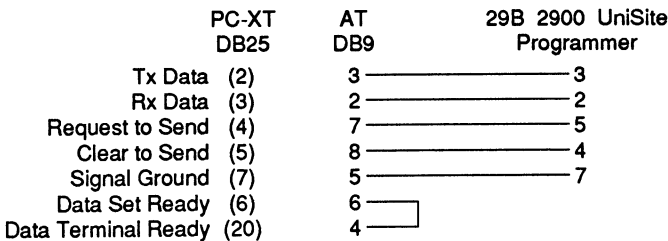
abelprog

Program Device from the **PartMap** menu or **abelprog** from the command line invokes the terminal emulator with the configuration necessary for ABEL. The ABEL configuration is in a file called **abelprog.cfg**, which is by default located in the same directory as the device libraries (for example, c:\dataio\lib4). You can edit this file to make changes to the baud rate, stop bits, or other options. There are also a number of other .cfg files, which can be used by copying them to abelprog.cfg. See the *HiTerm User Manual* for a complete description of each .cfg file.

Cable Configuration for PCs

The cable configuration shown in Figure 7-15 is for hardware handshaking between PC XT/AT and Data I/O Model 29B, 2900 and UniSite programmers.

Figure 7-15
PC to Programmer Cable Configuration



Note: The LogicPak and Model 29 will not work reliably at over 4800 baud. Using the Full JEDEC format option on Fuseasm may given satisfactory operation at 9600 baud.

Using the Command Line

The ABEL software can be run as a batch process, or each step can be individually executed. Abel4bat is a batch file that contains the commands needed to run all steps of the language processor automatically. The command

abel4bat

can be followed by any of the options for the main program modules (AHDL2PLA, PLAOpt, Fuseasm, and JEDSim).

Command line options consist of a dash (-) followed by a keyword, followed by a space and modifiers separated by spaces. All keywords can be abbreviated to the first three characters.

Response File

The `abel4bat` file and any of the processing modules will accept a response file that contains the desired options. Response files allow you to specify more command line options than are allowed on the command line. The syntax for a response file is

sp

For example,

```
ahdl2pla infile @options.rsp
```

Turning Off Options

Any option can be cancelled or turned off by using **-nooption** or **+option**. For example, the option **-errlog filename** specifies that errors should be written to *filename*. If this option was specified in the source file, and you wanted to turn it off from the command line, use either of the following two options:

-noerrlog

+errlog

Command Line Summary

Table 7-4 summarizes the available command line options. Note that where more than one selection can follow an option, the selections are separated with spaces. For example,

```
plasim -i buffer.abl -trace table clock
```

Table 7-4
Command Line Option Summary

Option	AHDL2PLA	PLAOpt	Fuseasm	PLA/JEDSim	Batch*	Options Keyword*	Menu
-i <i>filename</i> — Input Filename	◆	◆	◆	◆	◆		File
-o <i>filename</i> — Output Filename	◆	◆	◆	◆	◆		(automatic)
-ivector <i>filename</i> — Input Vector File			◆	◆	◆		Trace Opt.
-help — Help	◆	◆	◆	◆	◆		Help
-usage — Usage	◆	◆	◆	◆	◆		Help
-device — Device Type			◆		◆		PartMap
-silent — Turn Off Output	◆	◆	◆	◆	◆		N/A
-errlog <i>filename</i> — Error Log File	◆	◆	◆	◆	◆		N/A
-llst [expand] — List File	◆				◆	◆	Compile
-syntax — Check Errors Only	◆				◆		Compile
-args <i>arguments</i> — Pass Arguments To Source File	◆				◆		Compile
-vectors — Test Vector File Only	◆				◆		Compile
-reduce [bypin group] [choose fixed] [dt] [exact] — Reduction Method		◆			◆		Optimize
-checksum [none full dummy] — Checksum			◆		◆	◆	PartMap
-format [82 83 87 88 pof hex brief full] — Programmer Load File Output Format			◆		◆	◆	PartMap
-config ptintact — Connect Unused OR Fuses			◆		◆	◆	PartMap
-config noturbo — Turbo Bits			◆		◆	◆	PartMap
-config nomlser — Miser Bits			◆		◆	◆	PartMap

Option	AHDL2PLA	PLAOpt	Fuseasm	PLA/JEDSim	Batch*	Options Keyword*	Menu
-config lock — Program Security Fuse			◆		◆	◆	PartMap
-ues — Electronic ID			◆		◆	◆	PartMap
-document [brief long] [plcc] — Document Format			◆		◆	◆	PartMap
-trace [pins wave table macro none] [brief clock detail] — Trace Level				◆	◆	◆	Trace Opt.
-break <i>first_vector</i> [<i>last_vector</i>] — Break Points				◆	◆		Trace Opt.
-x (0 1 h l) — Value for Don't Cares (.X.)			◆	◆	◆	◆	Trace Opt./ PartMap
-z (0 1 h l) — Value for High Z (.Z.)				◆	◆	◆	Trace Opt.
-signal [<i>pin#s</i>] [<i>signal_names</i>] — Watch Signals				◆	◆		Trace Opt.
-initial (0 1 h l) — Powerup State				◆	◆	◆	Trace Opt./ PartMap
-43 — EGA Expanded Text Mode -50 — VGA Text Mode -bw — Black and White for EGA/VGA							abel4
<p>*Batch denotes commands valid for the abel4bat.bat batch file. Options denotes commands valid for the Options statement in the source file. Menu gives the menu selection where the equivalent option is found..</p> <p>● denotes the option is valid; ◆ denotes the option is valid and is documented with that program.</p>							

Batch File Operation

One batch file is provided with the ABEL package. **Abel4bat** processes a file through AHDL2PLA, PLAOpt, Fuseasm and JEDSim. You can specify up to three options following **abel4bat**, or you can modify the subordinate batch file, **abelbat2**, to include the required options. Options specified on the command line override source file options, and source file options override the **abelbat2** options. Menu options override all other options.

Utilities

The ABEL package comes with several utilities for translating files. All the utilities can be accessed from the command line. These utilities are

- **PLAsplit** — PLA Design File Splitter
- **PLAmerge** — PLA Design File Merger
- **PLA2DASH** — PLA Schematic Generator
- **JED2AHDL** — JEDEC to ABEL-HDL translator
- **PLA2EQN** — PLA format translator
- **IFLDOC** — JEDEC to Signetics Table
- **Abellib** — Library manager
- **Cleanup4** — Cleans up temporary files generated by ABEL.
- **Finddev4** — Finds a specific device. For more information, see "Device Support" in the *User Notes*.

Partitioning and Merging PLA Files

The PLAsplit and PLAmerge utilities help you to use devices efficiently by allowing you to split a large design into two separate PLA files to be programmed into two devices, or merge two small designs to share a single device.

PLAsplit

```
plasplit plfile -signal output_name [output_name ...]  
-o plfile
```

PLAsplit will extract from the input file all the outputs specified with **-signal** and their required inputs, and create an independent PLA file that contains only those functions. You can use PLAsplit to divide a design into independent PLA design files that can be programmed into a number of devices.

A large design can be 'interactively' partitioned into multiple devices, or the individual PLA files can be processed using different optimization methods and merged later.

PLAmerge

```
plamerge -i plfile -a plfile -o filename
```

PLAmerge will merge two PLA files into one file that can be programmed into a single device. The PLA file specified with **-a** will be merged with the PLA file specified with **-i**. If there is any conflicting declarations (for example, two different device declarations), the information in the file specified with **-i** will be retained.

The PLA files must be the same file type, containing on-sets only or both on-sets and off-sets.

Creating a FutureNet Schematic

PLA2DASH

```
pla2dash plafile -o outfile
```

PLA2DASH takes a PLA file and creates a FutureNet command file that can be processed by the FutureNet schematic capture program to create a schematic representing the indicated PLA file.

Translating to Other Formats

JED2AHDL

```
jed2ahdl infile -o outfile -report mapfile
```

JED2AHDL translates JEDEC files into ABEL-HDL source files.

FPGA Translate

```
pla2eqn infile -o outfile  
[-language abel|pds|lca|snap|none]
```

PLA2EQN translates PLA format files to equations and terms used (none), ABEL-HDL (abel), Palasm-2 source file (pds), Signetics Snap source file (snap) or Palasm-2 source file for Xilinx XACT (lca) format. Most of these options are available under **FPGA Translate** in the **PartMap** menu in the ABEL Design Environment.

Creating Signetics Tables

IFLDOC

```
ifldoc -i jedecfile -o outfile [-device device]
```

IFLDOC converts JEDEC format files to listing files in a format similar to Signetic Program Logic tables. The listing file is for documentation only; it will not drive any device programmer.

The device type must be specified unless the JEDEC file was produced by ABEL or GATES.

Library Management

Library Manager

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

Abellib is the ABEL library manager used to manage the ABEL device and include libraries, abel4lib.dev and abel4lib.inc. Note that the device library (abel4lib.dev) used by Fuseasm is different from the device database used by the optional SmartPart package.

The device library (abel4lib.dev) is a single file that contains all the device files for devices currently supported by the ABEL software package. The "include" library (abel4lib.inc) is a single file that contains text files that may be included in a source file via the **library** statement.

Usage

The abellib library manager 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`, 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 `abel4lib.dev`. If you do not specify a library name on the command line, this file is referenced by `abellib`.

Use the following 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 example,

```
cd c:\dataio\lib4
```

2. Create a device file. For example,

```
abellib abel4lib.dev -e p16r8.dev
```

where `abel4lib.dev` is the library and `p16r8.dev` is the new device file.

3. Repeat step 2 as necessary to extract the desired devices from the library. Newly created *.dev files will be 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 in the current directory.
4. Since ABEL looks for "`abel4lib.dev`" for device specifications, you must rename any existing "`abel4lib.dev`" library file, then rename your new library "`abel4lib.dev`." For example in DOS,

```
ren abel4lib.dev oldlib.dev
ren newlib.dev abel4lib.dev
```

to ensure that ABEL accesses the newly created library.

Library File Usage

When an external file is required to process your ABEL design, such as a device or "include" file, ABEL 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 `ABEL4DEV` environment variable

3. Directories indicated in the PATH environment variable

Cleaning Up Extra Files

Cleanup4

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

```
cleanup4
```

This command will delete files with the extensions .bak, .dmc, .eq?, .err, .fit, .fus, .lst, .sav, sel, .sm?, .sim, .tmv and .tt?.

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

```
cleanup4 all
```

Cleanup4 all will delete all the temporary files, and all the *.doc, *.jed, *.pof and *.p8?. The *.abl files will not be affected.

*Note: Running an ABEL 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 software from the command line without running cleanup4, the *.tmv file from the original run will be read by Fuseasm, and the test vectors will be added to the JEDEC file.*

Finding a Device

The program finddev4 finds a specific device in abel4lib.dev. For more information, see "Device Support" in the *User Notes*.

8 *Using Advanced Features*

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

- Timing Problems and PLAOpt
- Advanced Simulation

Timing Problems and PLAOpt

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. The optimization performed by the language processor is designed to eliminate redundancy, whether it was intentional or not. Thus, if you run PLAOpt, you will counteract any attempts at solving timing problems by the introduction of redundancy. If you have intentionally redundant logic, use **-reduce none** or do not run PLAOpt.

Figure 8-1 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.

Thus the output F defined by the equation:

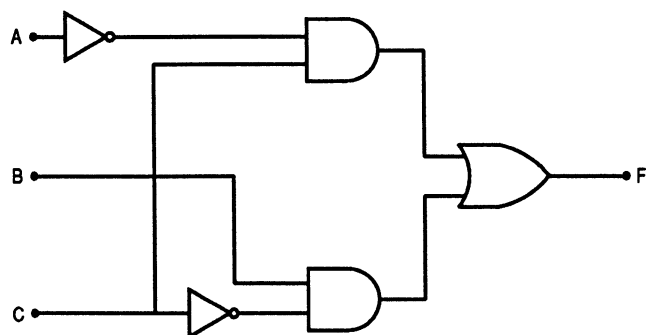
$$F = B \ \& \ !C \ \# \ !A \ \& \ C$$

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 8-2.

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

$$F = B \ \& \ !C \ \# \ !A \ \& \ C \ \# \ !A \ \& \ B$$

Figure 8-1
Circuit Using an Input and Its Complement

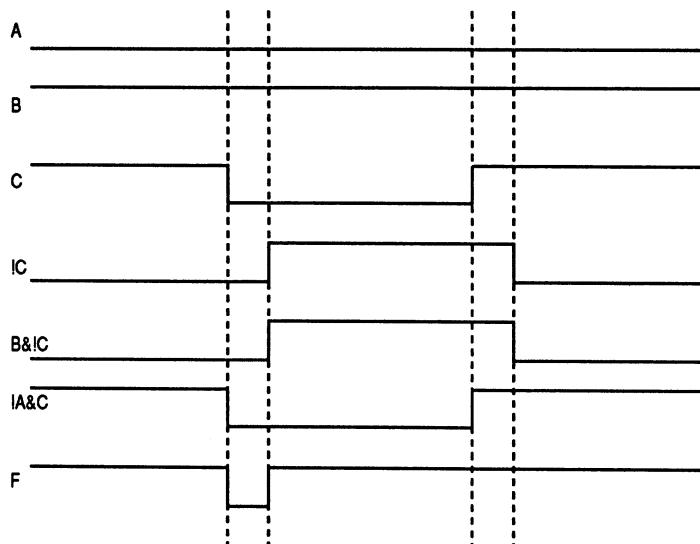


095-0745-001

This new equation does, in fact, remove the timing problem. But, if PLAOpt is used to minimize the PLA file, the redundant term will be eliminated, resulting in the original equation with the hazard present.

Information on PLAOpt, optimization, and the different reduction methods is presented in "Using ABEL Processing Modules."

Figure 8-2
Timing Diagram for $F = B \& !C$
!A & C.



095-0746-001



Advanced Simulation

This section explains the PLASim and JEDSim simulation programs and gives some suggestions concerning their use. The following topics are discussed:

- How ABEL simulates your design
- Trace levels and break points
- Simulation and designs with buffered outputs
- Simulation and unspecified inputs
- Simulation and designs with feedback
- Register preloads in the simulator
- Test vectors and simulation
- Debugging state machines
- Multiple test vector sections
- Using macros and directives to create test vectors
- Don't cares in simulation
- Preset, reset and preload registers
- Asynchronous circuits




How ABEL Simulates Your Design

The ABEL simulators simulate the operation of a design by building a model of the circuit, applying test vectors and calculating the results.

The Simulator Model

Using the PLA file (PLASim), or the JEDEC and device files (JEDSim), the simulator builds a model of the design that includes macrocells, sum-terms and product terms. Use **-trace macro** to display the model.

JEDEC Test Vectors



JEDEC and .tmv Vectors

JEDEC vectors only include test conditions for pins; the .tmv file vectors allow testing of internal nodes. Also, the vectors in the .tmv file can have different values for input than for output. For example, the .tmv file allows you to apply a 0 to a bidirectional pin that is an input before the clock and test for an H after the clock. A JEDEC vector could only have the H. Use the **-ivectors filename** option for JEDSim to use the .tmv file for simulation in place of the JEDEC vectors.

**JEDEC Vector
Conversion**

Internally, the JEDEC simulator uses the same test_vector format as the vectors from AHDL2PLA containing input vectors and output vectors. JEDSim must convert the JEDEC vectors to the same format as the vectors from AHDL2PLA. When reading JEDEC test vectors, the simulator copies the H, L and Z into the output vector, and all test conditions into the input vector. It also makes the following conversions:

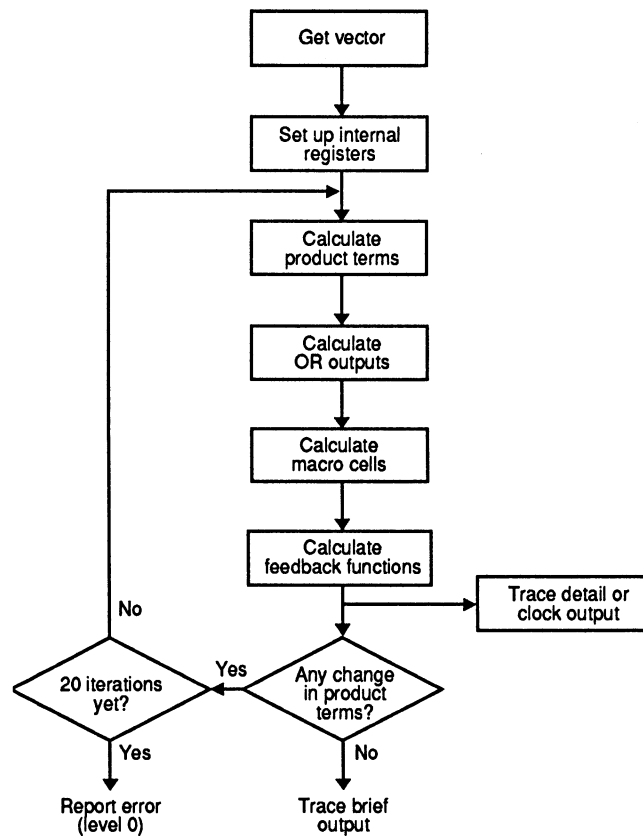
H	Converted to 1.
L	Converted to 0.
.Z.	Converted to 1 or the user-specified value.
.X.	Converted to 0 or the user-specified value.
.C.	Expanded to three vectors with .C. taking on the values 0, 1, and then 0. Use -trace clock to observe the clock conversions.
.U.	Expanded to two vectors taking on the values 0 and then 1. Use -trace clock to observe the clock conversions.
.K.	Expanded to three vectors with .K. taking on the values 1, 0, and then 1. Use -trace clock to observe the clock conversions.
.D.	Expanded to two vectors taking on the values 1 and then 0. Use -trace clock to observe the clock conversions.

**PLASim and JEDSim
Operation**

Figure 8-3 shows a flow diagram that depicts PLASim and JEDSim operation when evaluating the inputs to the output. The simulators get the first test vector and perform any setup of internal registers (within the PLD) that results from the vector applied to the inputs. The simulators then calculate 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. The results of the simulators' calculations are written to the .smn or .sim file.

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. The simulators use an iterative method to compute the outputs. After the feedback paths have been calculated, the simulators check 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 later in this chapter.

Figure 8-3
Simulation Processing Flow
Diagram



Trace Levels and Break Points

Trace levels and break points allow you to control the amount of information that PLASim/JEDSim provides and make the analysis of a faulty design much easier. PLASim and JEDSim can provide simple error messages (indicating that the actual outputs differ from the outputs you predicted in your test vectors), or detailed information about the states of internal registers and gates of a device during simulation. If you simulate a design at trace method none, you can determine whether any errors exist. If there are errors, then you can use the more detailed trace levels 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 formats and levels are given below.

With small designs, you can rerun the simulators 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 break points 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, but if you rerun a simulator with a more detailed 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 a simulator with break points set for test vector 20, the detailed information is provided only for that vector.

In summary:

1. Simulate designs at `-trace none` to determine the existence of errors.
2. Once an error is found, increase the trace level until you have enough information to correct the error.
3. Use break points to limit simulation data collection to the vectors of concern.

No Trace

Figure 8-5 shows the output of PLASim created during processing of `regfb.abl` (Figure 8-4) with `-trace none`. After compiling `regfb.abl`, the following command line was used to produce Figure 8-5:

```
plasim regfb -trace none
```

Figure 8-4
Regfb.abl Source File

```
module regfb
  title 'Operation of the simulator on devices with feedback
        Data I/O Corp.      31 July 1990'

  FB2      device 'P16R4';

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

  F1 istype 'reg_D,invert';

equations
  F1.D      = D1 & INIT;
  F2        = D2 & F1.Q;

  F2.OE     = D3;
  F1.C      = Clk;
  F1.OE     = !OE;

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

end regfb
```

Figure 8-5
No Trace Simulation Output

```
Simulate ABEL 4.00  Date Tue Aug 14 10:54:16 1990
Fuse file: 'regfb.ttl'  Vector file: 'regfb.tmv'  Part: 'PLA'

Operation of the simulator on devices with feedback
Data I/O Corp.          31 July 1990

4 out of 4 vectors passed.
```

In Figure 8-6, one of the test vectors was changed to produce an error, and simulation was run through JEDSim at trace none (only errors are shown). When an error occurs, the simulation output for trace none lists the number of the vector that failed, the name and number of the failed output, and the nature of the failure.

Figure 8-6
No Trace Simulation Output
Showing Error

```
Simulate ABEL 4.00  Date Mon Aug  6 14:48:31 1990
Fuse file: 'fb2.jed'  Vector file: 'fb2.jed'  Part: 'P16R4'

ABEL 4.00 Data I/O Corp. JEDEC file for: P16R4 V8.0
Created on: Mon Aug  6 14:45:51 1990

Operation of the simulator on devices with feedback
DATA I/O Corp.          31 July 1990

Vector 4
F2 13, 'L' found 'H' expected

3 out of 4 vectors passed.
```

Brief Trace

Figure 8-7 shows a brief table trace simulation output for the same source file that produced Figure 8-5. Brief trace is the default for all formats.

Figure 8-7
Brief Trace Simulation Output

```
Simulate ABEL 4.00  Date Tue Aug 14 10:56:35 1990
Fuse file: 'regfb.ttl'  Vector file: 'regfb.tmv'  Part: 'PLA'

Operation of the simulator on devices with feedback
Data I/O Corp.          31 July 1990

      I
      C  N
      l O I D D D  F F
      k E T 1 2 3  1 2

V0001  C 0 0 1 1 1  H L
V0002  C 0 0 0 0 1  H L
V0003  C 0 1 1 1 1  L H
V0004  0 0 0 0 0 1  L L
4 out of 4 vectors passed.
```

Clock Trace

Trace Clock provides information similar to that described for **Brief**, except that the clock output shows the device inputs and outputs for each clock pulse. Figure 8-8 shows the clock output for the same source file used to generate Figure 8-7. The command used to produce Figure 8-8 is

```
plasim regfb -trace clock
```

Note that the output is expanded over that shown in Figure 8-7 to show each clock pulse of the simulate operation.

Figure 8-8
Clock Trace Simulation Output

```
Simulate ABEL 4.00   Date Tue Aug 14 10:58:41 1990
Fuse file: 'regfb.ttl'  Vector file: 'regfb.tmv'  Part: 'PLA'

Operation of the simulator on devices with feedback
      Data I/O Corp.           31 July 1990

      C      I
      l O I D D D      F F
      k E T 1 2 3      1 2

V0001  0 0 0 1 1 1      H L
        1 0 0 1 1 1      H L
        0 0 0 1 1 1      H L
V0002  0 0 0 0 0 1      H L
        1 0 0 0 0 1      H L
        0 0 0 0 0 1      H L
V0003  0 0 1 1 1 1      H L
        1 0 1 1 1 1      L H
        0 0 1 1 1 1      L H
V0004  0 0 0 0 0 1      L L
4 out of 4 vectors passed.
```

Detail Trace

Trace Detail provides information similar to that described for clock, except that the detail output shows the device inputs and outputs for each iteration of the simulator. Figure 8-9 shows the detail output for the same source file used to generate Figure 8-7. The command used to produce Figure 8-9 is

```
plasim fb2 -trace detail
```

Note that the output is expanded over that shown in Figure 8-8 to show each iteration of the simulate operation that takes place to stabilize the device output. In this case, Vector 3 takes an extra iteration of the high clock pulse to stabilize.

Figure 8-9
Detail Table Format Simulation Output

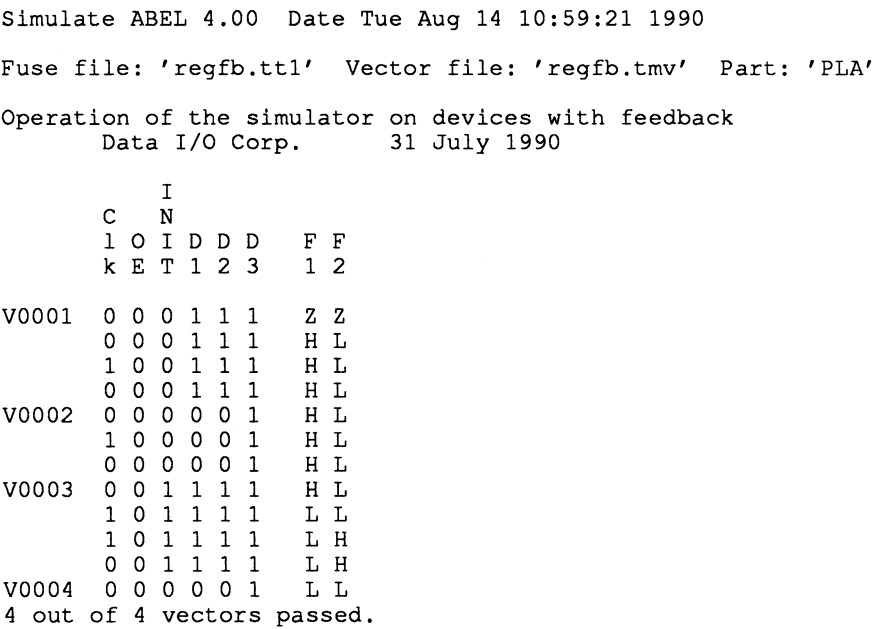


Table Trace

Table Trace is the default format. Table gives a table with signal levels represented by H, L, and Z for logic high, logic low, and high-impedance state.

Figures 8-7, 8-8 and 8-9 show the table output for all trace levels.

Pins Trace

The trace pins 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; that is, the expected output of the device appears on the Vector Out line below the actual output. After compiling, reducing and mapping `regfb.abl`, the following command line was used to produce the file in Figure 8-10.

```
jedsim fb2 -trace pins
```

Figure 8-10
Pins Format Simulation Output

```
Simulate ABEL 4.00X   Date Wed Aug 29 10:21:15 1990
Fuse file: 'fb2.jed' Vector file: 'regfb.tmv' Part: 'P16R4'

ABEL 4.00X Data I/O Corp. JEDEC file for: P16R4 V8.0
Created on: Wed Aug 29 10:21:07 1990

Operation of the simulator on devices with feedback
Data I/O Corp.           31 July 1990

Vector 1
Vector In  [C0111.....0.....]

Device In  [001110000000011110001111]
Device Out [.....ZLHHHHZZ.....]

Vector 2
Vector In  [C0001.....0.....]

Device In  [000010000000011110001111]
Device Out [.....ZLHHHHZZ.....]

Vector 3
Vector In  [C1111.....0.....]

Device In  [011110000000101110000111]
Device Out [.....ZHLHHHZZ.....]

Vector 4
Vector In  [00001.....0.....]

Device In  [000010000000001110000111]
Device Out [.....ZLLHHHZZ.....]

4 out of 4 vectors passed.
```

Macrocell Trace

Macrocell simulation provides the same information as Table, plus internal device information such as OR-gate outputs, register outputs, and the final outputs. Figure 8-11 shows one portion of a macro listing with pointers to its various parts and a portion of the corresponding logic diagram. Only a portion of the macro simulation output is shown since macro output files can be quite large. After compiling, reducing and mapping `regfb.abl`, the following command line was used to produce the file.

```
jedsim fb2 -trace macro
```

Figure 8-11
Macro Format Simulation
Output

```

Simulate ABEL 4.00X   Date Wed Aug 29 10:22:32 1990

Fuse file: 'fb2.jed' Vector file: 'regfb.tmv' Part: 'P16R4'

ABEL 4.00X Data I/O Corp. JEDEC file for: P16R4 V8.0
Created on: Wed Aug 29 10:21:07 1990

Operation of the simulator on devices with feedback
      Data I/O Corp.      31 July 1990

Vector 1
Vector In  [C0111.....0.....]

Pin  11 [0] ]-----
                |
                F1 Pin 14 | \
                |  |  >O--- H   Vec=H
                |  |  /
                |  |  /
                |  Q = L |
                |  OR = L |
                |  CK = L |
                |-----

PT 1280 [FFFFFFF] ]---
Pin   1 [0] ]---

PT 1536 [T] ]-----
                |
                F2 Pin 13 | \
                |  |  >O--- L   Vec=L
                |  |  /
                |  |  /
                |  OR = H |
                |-----

PT 1568 [TFFFFFFF] ]---

Vector Out [.....LH.....]
:
:
:

```

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 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.

Trace macro will produce very large files if all pins and nodes are traced for all vectors. To get a reasonably sized file, use the **Signal** (-signal) and **First/Last Display Vector** (-break) options to specify the pins or nodes for only the desired vectors. If a **Signal** option is not specified, the first I/O pin in the device will be traced.

Table 8-1 defines the notation used in the simulation macro output files to identify product terms and nodes.

Table 8-1
*Notation Used in Simulation
Macro Output Files*

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

Wave Trace

Wave Trace provides a waveform representation of the inputs and/or outputs of the device for each of the specified test vectors. The **First/Last Display Vector** option (break points) specifies the vectors to be used and the **Signal** option specifies which inputs and outputs are to appear in the output file. Up to 14 pins can be specified (blank columns inserted with 999 count as a pin in the output). If no **Signal** is used, PLASim/JEDSim automatically generates the signals appearing at the first 14 output pins.

Figure 8-12 shows the waveform generated by running the same source file used in Figure 8-5. After compiling **regfb.abl**, the following command line was used to produce the file.

```
plasim regfb -trace wave
```

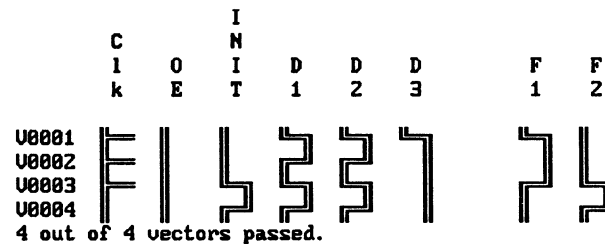
Notice that the ABEL simulators automatically watch only those signals specified in the test vectors.

Figure 8-12
Wave Format Simulation
Output

Simulate ABEL 4.00X Date Wed Aug 15 15:41:52 1990

Fuse file: 'regfb.ttl' Vector file: 'regfb.twv' Part: 'PLA'

Operation of the simulator on devices with feedback
Data I/O Corp. 31 July 1990



Simulation and Designs with Buffered Outputs

When a design with 3-state buffered outputs is simulated with trace wave and/or table, 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 **Table Trace**, 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 option. Using don't cares (Xs) in the input test vector can cause the input(s) to be unspecified. The ABEL simulators do not propagate unknown values.

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.

PLASim/JEDSim 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 method you simulate under. Figure 8-13 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 table and table detail are shown in Figures 8-4 through 8-9.

Figure 8-13
Synchronous Feedback Circuit

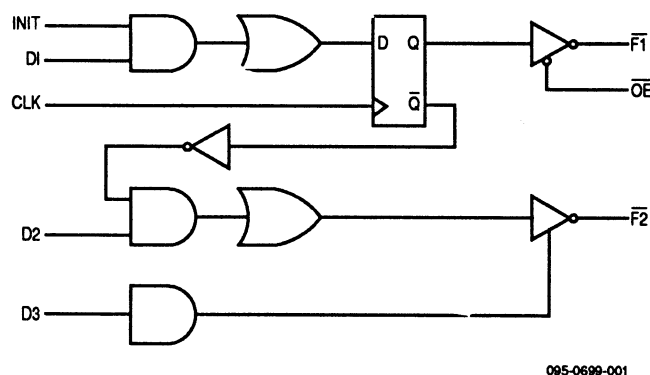


Table trace output shows the test vectors and the final outputs after the clock pulse. **Table clock** shows the test vectors and the value of the outputs before and after the clock. **Macro trace** results in a large simulation output file. If you wish to examine the trace output for this circuit, you can run ABEL on `regfb.abl` with trace macro and examine the simulation results.

The second feedback example in Figure 8-14 shows an asynchronous circuit that requires more than one simulation iteration before the outputs stabilize. Figure 8-15 shows the source file describing the circuit and Figures 8-16 and 8-17 show the simulation output for brief trace and detail trace. Trace none output is not shown since there are no simulation errors in this design and **none** only reports errors.

Brief trace shows the final outputs after they have stabilized, and also the test vectors. Detail trace 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. Macro trace output is not shown but can be generated by running ABEL with "feedback.abl" shown in Figure 8-15.

Figure 8-14
Asynchronous Feedback Circuit

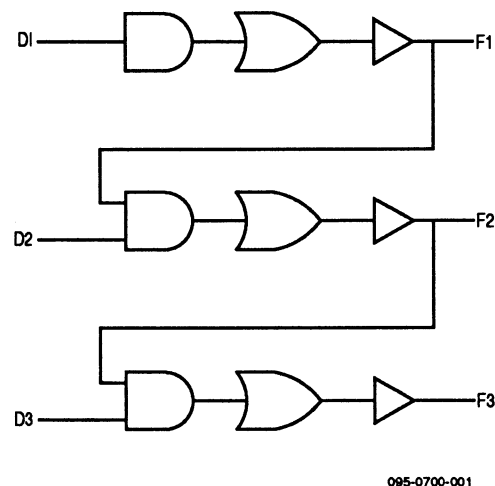


Figure 8-15
Source File: Asynchronous
Feedback Circuit

```

module feedback
  title 'Operation of the simulator on devices with feedback
        DATA I/O Corp.      24 Feb 1983'

      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

```

The command line used to product Figure 8-16 is

```
plasim feedback
```

Figure 8-16
Brief Trace Simulation Output:
Asynchronous Feedback Circuit

```

Simulate ABEL 4.00  Date Mon Aug  6 14:55:57 1990
Fuse file: 'feedback.ttl' Vector file: 'feedback.tmv'

Operation of the simulator on devices with feedback
        DATA I/O Corp.      24 Feb 1983

      D D D      F F F
      1 2 3      1 2 3

V0001  0 0 0      L L L
V0002  1 1 1      H H H
2 out of 2 vectors passed.

```

The command line used to product Figure 8-17 is

```
plasim feedback -trace detail
```

Figure 8-17
Detail Trace Simulation
Output: Asynchronous
Feedback Circuit

```

Simulate ABEL 4.00  Date Mon Aug  6 14:57:01 1990
Fuse file: 'feedback.ttl' Vector file: 'feedback.tmv'

Operation of the simulator on devices with feedback
        DATA I/O Corp.      24 Feb 1983

      D D D      F F F
      1 2 3      1 2 3

V0001  0 0 0      L L L
V0002  1 1 1      H L L
        1 1 1      H H L
        1 1 1      H H H
2 out of 2 vectors passed.

```

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 (that is, 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.

Test Vectors and Simulation

PLASim and JEDSim simulate equations and the programmed device with user-supplied design test vectors. The more comprehensive and detailed your test vectors are, the more comprehensive and useful your simulation results will be.

With test vectors, you specify the required input pattern and expected outputs at the device pins. PLASim and JEDSim will apply the inputs from the test vectors to the simulated circuit and compare the simulated output with the output specified in the test vectors. If there is any difference, an error is indicated.

Note that the simulators cannot test a mode in your design if you do not write a test vector to force that mode of operation. It is to your advantage to create complete sets of test vectors that test all functions of your logic design, and to use PLASim/JEDSim regularly as design changes are made.

Note: The nodes shown on ABEL 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 the implemented state machine to follow a different sequence from the test vectors. The simulated design becoming unsynchronized with the test vectors can cause so many errors during simulation that the original error is difficult to isolate.

To test and debug larger state machines

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

Periodically forcing (with your test vectors) the state machine to a known state and then letting the state transitions take place limits the cascading of errors to a smaller number of states and makes it much easier to find initial errors.

Starting with a small set of test vectors that tests only part of the state machine's function again helps isolate any errors. When operation of one 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 makes errors easier to pin point and simplifies the testing of large state machines.

Multiple Test Vector Sections

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 8-18 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.

Figure 8-18
Source File with Multiple Test Vector Sections

```
module simple
  title 'Simple ABEL example
  Dan Poole   Data I/O Corp   15 Oct 1987'

      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
```

Automatic Signal Selection

Figure 8-19
Simulation Results Showing Automatic Signal Selection

The ABEL simulators automatically show only signals used in test vectors for all trace formats except for pins, unless specific signals have been specified with `-signal`. (Pins format shows all inputs and outputs.) If you run the module shown above (module simple), with the default table format, the simulation results are given for only the signals used in the test vectors (Figure 8-19):

```
Simulate ABEL 4.00   Date Thu Aug  9 14:45:02 1990
Fuse file: 'simple.ttl'  Vector file: 'simple.tmv'

Simple ABEL example

***** Test And Gate *****

      A A A      A
      1 2 3      D

V0001  0 0 0      L
V0002  1 0 0      L
V0003  0 1 0      L
V0004  0 0 1      L
V0005  1 1 1      H

***** Test Nand Gate *****

      N N N      N
      1 2 3      D

V0006  0 0 0      H
V0007  1 0 0      H
V0008  0 1 0      H
V0009  0 0 1      H
V0010  1 1 1      L
10 out of 10 vectors passed.
```

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 8-20 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 8-20
*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 5 Aug 1990'

  m6809 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

```


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 test vectors are shown in simulation. Figure 8-21 shows the test vectors produced by the directive and macro. The listing was produced by PLASim.

In the second source file, shown in Figure 8-22, 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 8-23 shows the test vectors created.

Figure 8-21
*Simulation Results Showing
Test Vectors Created with a
Macro and @IF/@IRP Directives*

```

Simulate ABEL 4.00X   Date Tue Aug 28 15:00:28 1990
Fuse file: 'm6809b.ttl' Vector file: 'm6809b.tmv' Part: 'PLA'
6809 memory decode
Jean Designer  Data I/O Corp Redmond WA   5 Aug 1990

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

V0001  0 0 0 0 0      L H H H
V0002  1 0 0 0 0      L H H H
V0003  1 1 1 0 0      H L H H
V0004  1 1 1 0 1      H H H H
V0005  1 1 1 1 0      H H L H
V0006  1 1 1 1 1      H H H L
6 out of 6 vectors passed.

```

Figure 8-22
Test Vectors Described with a
Macro, @CONST and
@REPEAT Directives

```

module M6809C
  title '6809 memory decode
  Jean Designer      Data I/O Corp Redmond WA    5 Aug 1990'

  m6809c              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

```

Figure 8-23
Simulation Results Showing
Vectors Created with a Macro,
@CONST and @REPEAT

```

Simulate ABEL 4.00X   Date Tue Aug 28 15:01:49 1990
Fuse file: 'm6809c.ttl' Vector file: 'm6809c.tmv' Part: 'PLA'

6809 memory decode
Jean Designer      Data I/O Corp Redmond WA    5 Aug 1990

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

V0001 1 1 0 1 0   L H H H
V0002 1 1 0 1 1   L H H H
V0003 1 1 1 0 0   H L H H
V0004 1 1 1 0 1   H H H H
V0005 1 1 1 1 0   H H L H
V0006 1 1 1 1 1   H H H L
6 out of 6 vectors passed.

```

Don't Cares in Simulation

In ABEL 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 simulator to choose a value for the input designated by the `.X.` in the test vector(s), or to disregard an output signal's state. The default value used in the simulator for the don't care inputs is zero; however, the don't care option (`-x`) can be used in the PLASim/JEDSim command line to specify zero or one (0 or 1) for the don't care value. (Refer also to the PLASim/JEDSim section of "Using ABEL Processing Modules.")

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 option is set to `-x 1`. 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 1s and 0s in a test vector that needs to be checked by the ABEL simulator.

Note: Logic programmers use JEDEC vectors to test the device. Occasionally vectors that pass in ABEL simulation will fail on the programmer. Data I/O logic programmers such as the UniSite or 2900 show on the operator's terminal which pin on which vector failed. The LogicPak and Model 60 used in the front mode will give only "Error 75" for a vector failure. To determine the pin(s) and vector(s) that failed, the programmer must be used in terminal mode (Select E1). See the LogicPak or Model 60 manuals, and the Data I/O application note on test vectors and error 75.

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 PLASim/JEDSim operation with the don't cares set to 0 (option `-x 0`), and then again with them set to 1 (option `-x 1`).

The simulators ignore output pins that are not specified in the test vectors and will not indicate an error due to conflict between a specified value and the value determined by the simulator. The `.X.` constant at an output pin tells the simulators 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 8-24 shows how the `.X.` value can be assigned to the outputs prior to the PLASim/JEDSim step of the language processor.

Figure 8-24
*Assignment of Don't Care
Value (.x.) to Design Outputs*

```
module findout
options '-ivector findout.tmv'
title 'The ABEL simulator will find the output levels
Ngoc Nicholas      Data I/O Corp      9 Aug 1990'

      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 format pins, wave and table, you can observe the actual output values determined by the simulator. In Figure 8-25, 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.

Figure 8-25
*PLASim Results with Outputs
Specified as Don't Care*

```
Simulate ABEL 4.00X   Date Tue Aug 28 15:07:54 1990
Fuse file: 'findout.ttl' Vector file: 'findout.tmv'

The ABEL simulator will find the output levels
Ngoc Nicholas      Data I/O Corp      9 Aug 1990

      A      B      Y      Y
      A      B      1      2

V0001  0      0      H      H
V0002  0      1      L      L
V0003  1      0      L      L
V0004  1      1      L      H
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 PLASim/JEDSim steps of the language processor to verify operation of the design by performing the required operations of these registers.

Note: For preload, ABEL 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, reset 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 8-26, 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. 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 8-26
Test Vectors for Special Preset
Conditions

```

module preset
  title '2-bit counter to demonstrate power on preset    9 Aug 1990
  Bob Hamilton    Data I/O Corp'

  preset device 'F167';

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

  Ck,X = .C.,.X.;

equations
  [P1,P0].PR = PR;
  [P1,P0].C  = Clk;

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

test_vectors
  ([Clk,PR,Hold] -> [P1,P0])
  [ 1 , 1, 0 ] -> 3;
  [ 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;

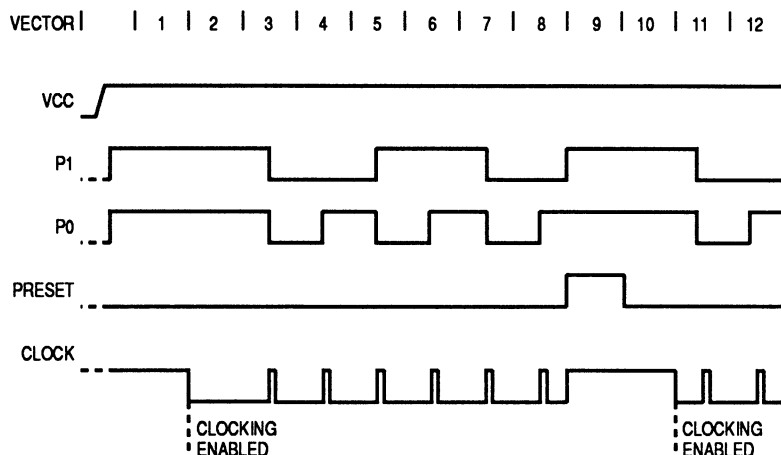
  " Notes on preset from the Signetics Data Sheet
  "
  " The PR input provides an asynchronous preset to logic '1' of all
  " State and Output Register bits. Preset overrides the Clock, and
  " when held High, clocking is inhibited and all outputs are High.
  " Normal clocking resumes with the first clock 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 8-27 is a timing diagram that shows the action of the test vectors in Figure 8-26. 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 8-27
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.

095-0748-001

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.

TTL Preload

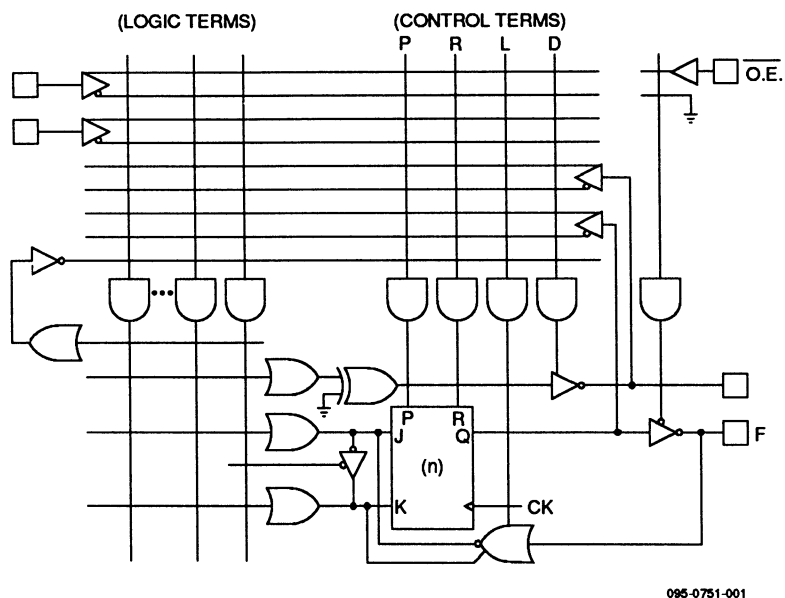
Certain programmable logic devices, such as the F159 FPLS (field programmable logic sequencer) allow internal registers to be forced (preloaded) to a known state by means of the TTL preload function. Figure 8-28 shows a typical FPLS layout. To preload the output register in such a device, four conditions must be present:

- The output is placed in the high-impedance state
- The desired register state is placed on the output pin
- The load control term is activated
- A clock pulse is applied to the clock input

The fifth test vector in Figure 8-29 shows how each of the above conditions are met. In the fifth test vector

- OE (Ena) pin is held at 1

Figure 8-28
Internal Register of the F159



- The output (F0) is pulled low by inserting a 0 in the input side of the test vector
- A 1 is applied to the LOAD input, which activates the load control term (LA)
- A clock pulse is provided by the C (.C.) applied to the clock input

Figure 8-29
Invoking the TTL Preload
Function

```

module TTLload
  title 'TTL load example
  Dave Kohlmeier  Data I/O Corp  9 Aug 1990'

  TTL59  device  'F159';

  C,L,H,X,Z      = .C.,0,1,.X...Z.;

  Clk,J_IN,K_IN,LOAD,Ena,F0      pin 1,2,3,4,11,12;

  F0 istype 'reg_JK,invert';

equations
  F0.OE = !Ena;
  F0.J  = J_IN;
  F0.K  = K_IN;
  F0.L  = LOAD;
  F0.C  = Clk;

test_vectors
  ([Clk,Ena,J_IN,K_IN,LOAD,F0] -> F0)
  [ C , L , 1 , 0 , 0 , X] -> 0 ; "Set
  [ C , L , 0 , 1 , 0 , X] -> 1 ; "Reset
  [ C , L , 1 , 1 , 0 , X] -> 0 ; "Toggle
  [ C , L , 1 , 1 , 0 , X] -> 1 ; "Toggle
  [ C , H , 0 , 0 , 1 , 0] -> X ; "Load
  [ 0 , L , 0 , 0 , 0 , X] -> 0 ; "Test
  [ C , L , 1 , 1 , 0 , X] -> 1 ; "Toggle
end

```


The sixth test vector tests to make sure the register was loaded with the 0 applied to the output by the fifth test vector. The sixth test vector enables the output (Ena at 0) and allows the output to be tested (F0 = X on the input side and 0 on the output side of the vector) while holding the clock input at 0.

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 (that is, 10 through 15). Figure 8-30 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;
```

To verify that the design will recover from each of the illegal states, appropriate test vectors are included. This group of test vectors preloads the device to each possible illegal state and then verifies that the device clocks 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 8-30
*Defining Illegal States and Test
Vectors for Illegal States*

```

module CNT10P
  title 'decimal counter
Note: preload the data on pins into the registers
Denny Siu  Data I/O Corp  9 Aug 1990'

  cnt10p          device 'P16R4';

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

  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;

  equations
    [Q3,Q2,Q1,Q0].c = Clk;
    [Q3,Q2,Q1,Q0].oe = !OE;

  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;

  "Ensure return from illegal state
    State S10:     GOTO S0;
    State S11:     GOTO S0;
    State S12:     GOTO S0;
    State S13:     GOTO S0;
    State S14:     GOTO S0;
    State S15:     GOTO S0;

```

```

@page
test_vectors 'Test Counter'
( [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

```

An example of a state machine that contains branch conditions is given in the blackjack machine in the chapter "Source File 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 8-31 show how this state machine can be preloaded to test these three branches of the design.

Figure 8-31
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 , O , 1 X , X ] -> [ H,Test22 ]; "Verify preload
[ 0 , C , 1 X , X ] -> [ H,ShowStand];

[ 1 ,.P., X 0 ,Test22] -> [ X, X ];
[ 0 , O , 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,Sub_10 ];

```

**Preset/Reset Controlled
by Product Term**

In programmable logic devices such as the P22V10, preset and reset functions are controlled by product terms. An example of controlling the preset and reset functions is given in Figure 8-32.

Figure 8-32
*Controlling Reset/Preset by
Product Term*

```

module reset22a
title 'Demonstrates Asynchronous Reset and Synchronous Preset

Tim Charnley      Data I/O Corp    9 Aug 1990'

    reset22a      device 'P22V10';

    Clk,I1,I2,R,S,T Pin 1,2,3,4,5,6;
    Q1,Q2         Pin 14,15 istype 'buffer';

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

equations

    Output      := Input;    "Registered buffer

    Output.AR   = R & !T;

    Output.SP   = S & !T;

    Output.Clk  = Clk;

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

```

Note: In devices like the P22V10, there is a common reset for all registers. While a reset equation for one output will reset all of the registers in the P22V10, you should write individual reset equations for architecture-independence.

Note: For devices that have programmable polarity at the output of the registers, preset and reset functions may complement the output pins.

In the P22V10, the reset is asynchronous while preset is synchronous (to the clock input). The test vectors in Figure 8-32 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 or Preload 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.

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 PLASim/JEDSim 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 PLASim/JEDSim with trace clock or detail, simulation data will be written 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 PLASim/JEDSim output for a device with a clock input is shown in Figure 8-8 under "Clock Trace" earlier in this chapter. 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.

9 *Error Messages*

Message Types

Messages fall into the following categories:

Command Error	Command errors indicate an unrecognized use of the command line.
Device Error	Device errors indicate a problem with the device selected.
Fatal Error	Fatal errors indicate a condition that does not allow processing to continue.
Internal Error	Internal errors indicate a program functional error.
Logical Error	Logical errors indicate a problem with the design logic.
Note	Notes are messages provided for your information.
Syntax Error	Syntax errors indicate an unrecognized use of ABEL-HDL.
Warning	Warnings call attention to changes made by one of the processing modules that may affect the functionality of the design, but do not interrupt processing.

Note: Help for ABEL error messages can also be found in the ABEL Design Environment under **Errors** in the **Help** menu. Note the number of the error, then select that number from the **Errors** list for information on correcting the error.

AHDL2PLA Messages

Command Error 1000: You must specify an ABEL-HDL source file.	An ABEL-HDL source file is expected as the input to AHDL2PLA. The file should be specified as the first argument to the AHDL2PLA program.
Syntax Error 1001: String expected	A string argument was expected in the indicated location in the source file. A string is a sequence of one or more characters delimited by ' (single-quote) or ' (back-quote).
Logical Error 1002: Source line length exceeds nn chars.	Individual lines of an ABEL-HDL source file can not exceed the indicated number of characters.
Fatal Error 1003: Unable to open input file 'xxxx'.	The indicated file could not be opened by AHDL2PLA. If the indicated file exists, check to make sure that it has the appropriate file protection settings to enable you to access it.
Warning 1004: FLAGS statement obsolete - mapping old flags to new options.	The FLAGS statement, used in previous versions of ABEL, has been replaced with the OPTIONS statement in ABEL-HDL. In most cases the old-style flags specified in a FLAGS statement will be automatically mapped to the proper options.
Syntax Error 1005: Identifier length exceeds nn chars.	Identifiers (module, device, signal and constant names) must not exceed the indicated length.
Syntax Error 1006: Undefined token	An unrecognized sequence of non-alphabetic characters was encountered in the source file.
Syntax Error 1007: String length exceeds nn chars.	A string was encountered that exceeds the maximum allowable number of characters. This is usually an indication that the terminating string delimiter was unintentionally omitted.
Syntax Error 1008: Only ^b, ^o, ^d or ^h radix allowed.	The ^ (circumflex) character was encountered in the source file with an unknown radix character immediately following it.
Syntax Error 1009: Can't have letters imbedded in a number.	An incorrect sequence of numeric and alphabetic characters was encountered in the source file.
Logical Error 1010: Numeric overflow	An operation performed on constant values resulted in a number larger than allowable.
Syntax Error 1011: Actual argument length exceeds 35 chars.	An actual argument (the text passed as an argument to a macro or module) exceeds the indicated maximum number of characters.
Syntax Error 1012: No more than nn arguments are declared.	The macro or module being processed has only the indicated number of dummy arguments declared, and the macro or module was called with too many actual arguments.

Warning 1013: Attribute 'FEED_OR' is obsolete - substituting '.D' instead.

The 'FEED_OR' signal attribute was used in previous versions of ABEL to specify the feedback configuration for the indicated signal. In ABEL-HDL, the feedback path is specified as a part of the related design description by using dot extensions. The appropriate dot extension to indicated the combinatorial feedback path from a registered output is typically '.D', so this dot extension will be appended to all affected signals when they are referenced as inputs.

Warning 1014: Attribute 'FEED_PIN' is obsolete - substituting '.PIN' instead.

The 'FEED_PIN' signal attribute was used in previous versions of ABEL to specify the feedback configuration for the indicated signal. In ABEL-HDL, the feedback path is specified as a part of the related design description by using dot extensions. The appropriate dot extension to indicated the feedback path from an output pin is '.PIN', so this dot extension will be appended to all affected signals when they are referenced as inputs.

Warning 1015: Attribute 'FEED_REG' is obsolete - substituting '.FB' instead.

The 'FEED_REG' signal attribute was used in previous versions of ABEL to specify the feedback configuration for the indicated signal. In ABEL-HDL, the feedback path is specified as a part of the related design description by using dot extensions. The appropriate dot extension to indicated the registered feedback path is typically '.FB', so this dot extension will be appended to all affected signals when they are referenced as inputs.

Fatal Error 1016: Unable to open output file 'xxxx'.

The indicated file could not be opened. This can indicated a full disk drive, invalid path, or file protection problems.

Syntax Error 1017: ENDCASE expected

The CASE transition statement must be terminated with an ENDCASE statement.

Syntax Error 1018: No register type specified for 'xxxx' - please use ISTYPE.

The indicated signal has not been declared with a register type. If no device and pin number have been declared, you should use the ISTYPE statement to specify that the signal is either combinatorial (ISTYPE 'COM') or registered (ISTYPE 'REG' or ISTYPE 'REG_X' where X is either D, T, SR, JK, JKD or G).

Syntax Error 1019: Keyword 'xxxx' expected

The indicated keyword was expected in the source file.

Syntax Error 1020: Identifier expected

An identifier (module, device, signal or constant name) was expected in the source file.

Syntax Error 1021: Undefined compiler directive '@xxxx'.

An '@' sign was encountered that was followed by an unknown directive name.

Syntax Error 1022: Special constant must end with a '.'.

Special constants are delimited by '.' (period) characters. This Error can also indicated incorrect usage of dot extensions, which are prefixed by '.' characters.

Syntax Error 1023: Undefined special constant 'xxxx'.	An unknown special constant was found delimited by '.' (period) characters.
Syntax Error 1024: Label expected	A label was expected in the source file. A label is a sequence of alphanumeric characters that does not begin with a number, and is not an ABEL-HDL reserved word (keyword).
Syntax Error 1025: Can't use ':=' operator with dot extension signal 'xxxx'.	The ':=' (registered assignment) operator is used to indicated a pin-to-pin relationship between an input expression and a corresponding registered output pin. Dot extensions signals are inherently combinatorial, and should therefore be described with the '=' (unlocked assignment) operator.
Syntax Error 1026: ';' expected	A semicolon was expected in the source file.
Syntax Error 1027: Module label doesn't match 'xxxx'.	The END statement was followed by a module name that does not match the module name specified in the previous MODULE statement.
Syntax Error 1028: EQUATIONS, STATE_DIAGRAM, TRUTH_TABLE or END expected.	A keyword was encountered out of place in the source file. This can indicate any number of syntax problems, including misplaced or missing semicolons or parenthesis, unbalanced IF-THEN, WHEN-THEN or CASE statements, or earlier syntax problems.
Syntax Error 1029: PIN, NODE, DEVICE, ISTYPE, MACRO or '=' expected.	A label, keyword or operator was encountered out of place in the declarations section of the source file. This Error will often result if a comma is omitted from a list of identifiers.
Warning 1030: Inconsistency in number of parameters - declaration ignored.	A constant declaration statement had an unbalanced number of items on the right and left side of the '=' (constant assignment) operator.
Syntax Error 1031: Undefined label 'xxxx'.	A label was referenced that was not declared in a declarations section of the source file.
Syntax Error 1032: Signal not allowed.	A signal was not allowed as an operand or argument in the indicated expression.
Syntax Error 1033: Max of nn elements in set was exceeded.	Set widths are restricted to 32 elements.
Syntax Error 1034: ']' expected	A set was not terminated with a ']' (right bracket) delimiter character.
Syntax Error 1035: Illegal operation on special constant.	Special constants are not allowed as operands in the indicated expression.
Syntax Error 1036: Can't compare set with nn members to one with nn members.	Set widths must match when performing operations on two sets.

Syntax Error 1037: ')' expected

The ')' (right parenthesis) character was expected as a delimiter.

Syntax Error 1038: Pin number, ISTYPE or semicolon expected.

A signal (PIN or NODE) declaration was not properly terminated.

Warning 1039: Label 'xxxx' is already defined - declaration ignored.

The indicated label was already used in a previous declaration. The new declaration will be ignored. This can indicate a missing or misplaced EQUATIONS keyword.

Syntax Error 1040: Number expected

A number was expected in the source file. Numbers can be prefixed by a radix specification, or can be specified as a set of binary values.

Syntax Error 1041: Keyword 'xxxx' is out of context.

The indicated keyword was used in an incorrect manner.

Logical Error 1042: Premature end of source file.

The end of the source file was encountered before an END statement was found.

Warning 1043: ':=' should not be used with output type 'xxxx'.

The ':=' (clocked assignment) operator is intended for use with D-type flip-flops only. For alternate flip-flop types, you should use the '=' (combinatorial assignment) operator in conjunction with either a '.T', '.J', '.K', '.S', '.R', or other register input port dot extension.

Warning 1045: 'INVERT' or 'BUFFER' not specified for 'xxxx' - output uncertain.

An output signal has been described using a register input port dot extension ('.D', '.T', '.J', '.K', '.S', or '.R') but the associated output signal has not been declared with an 'INVERT' or 'BUFFER' attribute. Since there is no information about the existence of an inverter between the register and the associated output pin, the actual value observed on that pin is uncertain.

Syntax Error 1046: Cannot operate on signal 'xxxx'.

The indicated signal was used in an expression in which only numbers or constant values are allowed.

Syntax Error 1047: Signal or .X. expected

A declared signal (pin or node) or the special constant '.X.' was expected in the source file.

Syntax Error 1048: '=' or ':=' expected

An assignment operator was expected in the source file.

Syntax Error 1049: Expression element expected

An expression was expected in the source file. Expressions are composed of one or more signals, sets, numbers or special constants operated on by valid ABEL-HDL operators. Expressions may or may not be enclosed in a pair of parenthesis.

Warning 1050: 'INVERT' or 'BUFFER' not specified for 'xxxx' - feedback uncertain.	Feedback from a registered signal has been described using the '.Q' dot extension, but the associated output signal has not been declared with an 'INVERT' or 'BUFFER' attribute. Since there is no information about the existence of an inverter between the register and the associated output pin, the relationship between the value observed on that pin and the value fed back from the register is uncertain.
Syntax Error 1051: GOTO, IF or CASE expected	A state transition (GOTO, IF-THEN-ELSE, or CASE) statement was expected.
Syntax Error 1052: THEN expected	The THEN portion of an IF-THEN-ELSE statement was omitted.
Syntax Error 1053: Signal 'xxxx' is combinatorial.	The ':=' (clocked assignment) operator was used to describe a signal that was declared as combinatorial.
Syntax Error 1054: '(' expected	A '(' (left parenthesis) character was expected as a delimiter.
Syntax Error 1055: '->' expected	The '->' operator was expected to delimit the input and output portions of a truth table or test vector header or entry.
Logical Error 1056: Extension 'xxxx' is not legal for node 'xxxx'.	The indicated dot extension is not appropriate for the indicated node.
Syntax Error 1057: Can't map set onto a non-set element.	A set was specified in a test vector where a single-bit value was expected.
Syntax Error 1058: Special constant not allowed.	Special constants are not allowed in the current context.
Syntax Error 1059: Undefined operation on sets.	Sets may not be used as operands of '>>' (shift right), '<<' (shift left), '*' (multiply), '/' (divide) or '%' (modulo) operators.
Syntax Error 1060: Can't map set onto a different sized set.	Set widths must match when performing operations on two sets, or when using nested sets in truth tables or test vectors.
Warning 1061: No output type specified for 'xxxx' - please use ISTYPE.	You have used the indicated signal as an output, but have not declared whether it is combinatorial or registered.
Warning 1062: Extension 'xxxx' may not be legal for signal 'xxxx'.	You have used a dot extension that does not match the known dot extensions for the output type declared for the indicated signal.
Syntax Error 1063: Dummy argument 'xxxx' not recognized.	You have indicated a dummy argument (by prefixing a label with the '?' character) that does not match the dummy arguments declared in the macro or module argument list.
Syntax Error 1064: Signal number for 'xxxx' is too large.	You have specified a very large pin or node number.

Syntax Error 1065: Pin number nn is not defined for this device.	You have specified a pin number that is undefined for the declared device.
Syntax Error 1066: Node number nn is not defined for this device.	You have specified a node number that is undefined for the declared device.
Syntax Error 1067: Only one label allowed.	You have specified a list of labels in a macro definition. Only one label is allowed for each macro.
Syntax Error 1068: Block expected	A block was expected following a macro keyword or directive. A block is a section of ABEL-HDL source statements enclosed by braces.
Syntax Error 1069: Closing '}' of block not found.	A terminating '}' (right brace) delimiter was not found in the source file.
Warning 1070: Pin number nn is already declared as 'xxx'.	You have declared the same pin number for two different named signals.
Syntax Error 1071: Closing ' of string not found.	The terminating ' (single quote) character was omitted from a string. This Error can occur if a single quote (or back quote) character is imbedded in a string without being prefixed by a '\' (backslash) character.
Syntax Error 1072: ':+' or ':*' expected	You have specified @ALTERNATE (use alternate operators) and have incorrectly specified an XOR or XNOR operation.
Syntax Error 1073: Constant label expected	The @CONST directive requires a constant label. This label must not have been previously declared (as a signal, macro or other identifier) except that it can be used in previous @CONST directive statement.
Syntax Error 1074: '=' expected	The '=' (assignment operator) was omitted from an @CONST directive or from a fuse assignment statement.
Syntax Error 1075: Undefined operation on signal.	A signal identifier is not legal in the current context.
Syntax Error 1076: Number is too large.	Numbers (either literal values or values produced by complex expressions) must not exceed the maximum size of 32 bits (unsigned).
Syntax Error 1077: ':>' or '->' expected	The ':>' or '->' operator was expected to delimit the input and output portions of a test vector header or entry line.
Syntax Error 1078: ':>' expected	The ':>' operator was expected in a test vector entry line to match the ':>' operator found in the test vector header.
Syntax Error 1079: Radix nn is not one of 2, 8, 10 or 16.	The radix value specified in a @RADIX statement must be either 2 (binary), 8 (octal), 10 (decimal) or 16 (hexadecimal).
Syntax Error 1080: nn actual arguments expected	A macro or module was invoked with an incorrect number of actual arguments.

Syntax Error 1081: Dummy argument expected	The '?' character indicates that the following sequence of characters is a dummy argument. The Error indicates that the dummy argument label does not match any of the dummy arguments listed in the macro or module declaration.
Syntax Error 1082: Digit not in radix nn.	You have specified a radix value less than 10 (decimal) and have used a digit that is not legal for that number system.
Warning 1084: Unrecognized attribute 'xxxx' - ignored.	The attribute string specified in an ISTYPE statement was not recognized as a valid signal attribute.
Syntax Error 1085: 'xxxx' attribute not allowed on this pin.	The specified attribute is not legal for the device and pin or node number indicated, or the attribute conflicts with a previous attribute specified for the same signal.
Warning 1086: Signal 'xxxx' has not been declared as combinatorial.	You have written a combinatorial function for the indicated signal, but have not specified that the signal is combinatorial (ISTYPE 'COM').
Warning 1087: Signal 'xxxx' has not been declared as registered.	You have written a registered (pin-to-pin) function for the indicated signal, but have not declared the signal as registered (ISTYPE 'REG' or ISTYPE 'REG_X' where X is a register type).
Logical Error 1091: nn actual arguments specified on command line.	The number of actual arguments specified on the AHDL2PLA command line exceeds the number of dummy arguments specified in the MODULE statement.
Syntax Error 1092: Use '#' for OR instead of 'l'.	The 'l' operator (used in some programming languages for a bitwise OR) is not used in ABEL-HDL. Instead you should use the '#' operator.
Syntax Error 1093: 'xxxx' undefined, maybe 'xxxx' was meant.	An identifier was encountered that is undefined, but a similar identifier (one that matches if upper and lower case are ignored) exists in the design.
Syntax Error 1094: No operators allowed in this context.	You have attempted to use a complex expression (one containing operators) when only signals, numbers or special constants are allowed. This Error most frequently occurs when dot extensions are applied to complex expressions.
Logical Error 1096: Arithmetic error	A divide by zero or similar situation was encountered when evaluating an expression.
Syntax Error 1098: Negative declaration not allowed.	Negative (active-low) declarations are only allowed in PIN and NODE statements.
Syntax Error 1099: Invalid set range.	A range (specified with the '..' operator) is bounded by incompatible signals or numbers. Ranges must be specified that will result in 32 or fewer set elements.

Syntax Error 1100: Bad element 'xxxx' in range.

The expansion of a range expression resulted in the creation of a previously unknown (undeclared) element.

Syntax Error 1101: Fuse number expected

A number was expected that corresponds to a fuse location in the previously declared device.

Syntax Error 1102: Fuse value 1 or 0 expected

A fuse value was expected. A value of 1 indicates an intact fuse, and a value of 0 indicates a programmed (blown) fuse.

Syntax Error 1103: Illegal nested set in state diagram header.

A state diagram state register must be specified as a set of signals, and the set must not be nested (contain other subsets).

Syntax Error 1104: Dot extension 'xxxx' not legal in this context.

The indicated dot extension is either unrecognized, or was used in an incorrect manner. Dot extensions can only be applied to signals or sets if signals.

Syntax Error 1105: Signal label expected

An identifier was used that is not a previously declared signal, and a signal was expected.

Syntax Error 1109: Register type of 'xxxx' does not match other set elements.

State diagrams require matching register types. The state register set specified contains signals of differing register types.

Syntax Error 1110: Pin nn (VCC) can't be used as input.

You have specified a device and pin number combination that cannot be used as an input in your design.

Syntax Error 1111: Pin nn (GND) can't be used as input.

You have specified a device and pin number combination that cannot be used as an input in your design.

Warning 1113: ENABLE keyword obsolete - use .OE extension instead.

The ENABLE keyword is no longer supported in ABEL-HDL. Instead, you should use the '.OE' dot extension to indicate a three-state output.

Syntax Error 1114: 'xxxx' attribute not allowed for state registers.

The indicated attribute is illegal for describing state register types. You must specify either 'COM', 'REG', or 'REG_X' where X is a valid register type.

Syntax Error 1116: Multiple attribute list not allowed in this context.

You have used the '=' operator to specify signal attributes for signals that have not been assigned pins. You should use the 'ISTYPE' attribute to assign attributes instead of the '=' operator.

Syntax Error 1120: Only one device allowed.

Multiple device declarations are no longer supported in ABEL-HDL. Instead, you should use the PLA splitter utility to split a large design into multiple devices.

Syntax Error 1121: Device declaration must be made before signal declarations.

If a device declaration is made in the source file, it must be made before any signal declarations.

Logical Error 1122: File name 'xxxx.xxx' too long.

The indicated file name is too long for the operating system.

Warning 1124: 'INVERT' or 'BUFFER' not specified for 'xxx' - assuming 'buffer'.	You have not specified whether the indicated output signal will have an inversion between the state register and the associated output pin. No inversion will be assumed, resulting in possibly incorrect operation if the design is implemented in a device with inverting outputs.
Syntax Error 1126: Can't use ':=' operator with dot extensions.	The ':=' (clocked assignment) operator is used to describe the operation of an registered output pin. If dot extensions are used, you should use the '=' assignment operator, since dot extensions (including register input ports) are inherently combinatorial.
Syntax Error 1128: Can't have operators in state register.	The state register set must consist of a set of signals, with no complex operators.
Syntax Error 1129: Can't expand expression - possibly recursive definition.	The AHDL2PLA compiler has detected a recursive constant declaration. Check to make sure you have not referenced the same identifier on both sides of the '=' operator.
Syntax Error 1130: Missing set element.	Set elements can consist of signals, numbers, special constants, sets or expressions and are separated by commas. One of these items was expected in the set.
Syntax Error 1131: Missing set range element.	The '..' (set range) operator was used with no second boundary element.
Syntax Error 1132: Keyword IN no longer supported.	Multiple device declarations and corresponding device-specific sections are no longer supported in ABEL-HDL. Instead, you should use the PLA splitter utility to split a large design into multiple devices.
Syntax Error 1133: Signal 'xxx' already defined as type 'INVERT'.	You have specified the 'BUFFER' attribute for a signal that was previously declared as 'INVERT'.
Syntax Error 1134: Signal 'xxx'	You have specified the 'INVERT' attribute for a signal that was already defined as type 'BUFFER'.
Fatal Error 1135: Unknown Error	

PLAOpt Messages

Fatal Error 2001: Requires an input file with a .TTn extension.	Plaopt requires input files with names such as m6809a.tt1 or foo.tt4. The default output file would be m6809a.tt2 or foo.tt5.
Fatal Error 2002: Input and output file names are the same 'file_name'.	The input and output files must have different names.
Fatal Error 2003: Can't open input file 'file_name'.	Input file not found.

Fatal Error 2004: AUXPLA and Output file names have the same 'file_name'.

The AUXPLA file from the input PLA file has the same name as the output file. The AUXPLA is created when the @DCSET is used in an PLAOpt source file.

Fatal Error 2005: Latte failed on 'file_name'.

While producing the @DCSET offsets, the espresso program (latte) was not found or produced an internal error.

Fatal Error 2006: Can't open input file 'file_name'.

While processing the @DCSET, PLAOpt could not open an working file.

Fatal Error 2007: Can't open AUXPLA file 'file_name'.

While processing the @DCSET, PLAOpt could not open the AUXPLA file.

Fatal Error 2008: Can't open output file 'file_name'.

While processing the @DCSET, PLAOpt could not open the output file.

Fatal Error 2009: Can't open input file 'file_name'.

While running '-reduce none', PLAOpt could not open the file.

Fatal Error 2010: Can't open output file 'file_name'.

While running '-reduce none', PLAOpt could not open the file.

Fatal Error 2011: Latte failed on 'file_name'.

While running '-reduce group fixed', the espresso program (latte) was not found or produced an internal error.

Fatal Error 2012: Latte failed on 'file_name'.

While running '-reduce group choose', the espresso program (latte) was not found or produced an internal error.

Fatal Error 2013: Can't open input file 'file_name'.

While running '-reduce group choose', PLAOpt could not open the file.

Fatal Error 2014: Can't open output file 'file_name'.

While running '-reduce group choose', PLAOpt could not open the file.

Fatal Error 2015: Latte (on-set) failed on 'file_name'.

While running '-reduce bypin fixed', the espresso program (latte) was not found or produced an internal error.

Fatal Error 2016: Can't open input file 'file_name'.

While running '-reduce bypin fixed', PLAOpt could not open the file.

Fatal Error 2017: Can't open output file 'file_name'.

While running '-reduce bypin fixed', PLAOpt could not open the file.

Fatal Error 2019: Latte (on-set) failed on 'file_name'.

While running '-reduce bypin choose', the espresso program (latte) was not found or produced an internal error.

Fatal Error 2020: Latte (off-set) failed on 'file_name'.

While running '-reduce bypin choose', the espresso program (latte) was not found or produced an internal error.

Fatal Error 2021: Can't open input file 'file_name'.	While running '-reduce bypin choose', PLAOpt could not open the file.
Fatal Error 2022: Can't open input file 'file_name'.	While running '-reduce bypin choose', PLAOpt could not open the file.
Fatal Error 2023: Can't open output file 'file_name'.	While running '-reduce bypin choose', PLAOpt could not open the file.
Fatal Error 2024: Latte (on-set) failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2025: Latte (off-set) failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2026: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2027: Design must use D or T flip flops.	The PLAOpt design must use the .D or .T detailed equations.
Fatal Error 2028: Can't open output file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2029: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2030: Can't open output file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2031: Latte failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2032: Latte failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2033: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2034: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2035: Can't open output file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2036: Latte failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2037: Latte failed on 'file_name'.	While running '-reduce dt', the espresso program (latte) was not found or produced an internal error.
Fatal Error 2038: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.
Fatal Error 2039: Can't open input file 'file_name'.	While running '-reduce dt', PLAOpt could not open the file.

Fatal Error 2040: Can't open output file 'file_name'.

While running '-reduce dt', PLAOpt could not open the file.

Fatal Error 2041: Could not rename 'file_name' to 'file_name'.

The file could not be renamed. Check that the file does not exist.

Fuseasm Messages

Logical Error 5000: xxx.C and xxx.C have inconsistent modes.

Use either asynchronous or synchronous clock through out the design.

Logical Error 5001: xxx pin ## can't be used as output.

Reassign pin for the equation with a valid output pin.

Logical Error 5003: xxx pin ## can't be used as input nor feedback.

Reassign pin for the input with a valid input/bidir pin.

Fatal Error 5004: Pin ## is an illegal assignment for xxx.

Pin is out of range, try some other device or pin number.

Logical Error 5005: xxx pin ## can't be used as registered output.

Reassign pin for the output with a registered pin.

Logical Error 5006: Register of xxx pin ## has no preset.

Reassign pin for the output with a registered pin with preset.

Logical Error 5007: Register of xxx pin ## has no reset.

Reassign pin for the output with a registered pin with reset.

Logical Error 5008: Register of xxx pin ## has no preload.

Reassign pin for the output with a registered pin with preload.

Warning 5009: Register of xxx pin ## has no asynchronous reset.

Part has Synchronous Reset; Reset may not work as expected.

Warning 5010: Register of xxx pin ## has no synchronous reset.

Part has Asynchronous Reset; Reset may not work as expected.

Warning 5011: Register of xxx pin ## has no asynchronous preset.

Part has Synchronous Preset; Preset may not work as expected.

Warning 5012: Register of xxx pin ## has no synchronous preset.

Part has Asynchronous Preset; Preset may not work as expected.

Logical Error 5013: xxx can't be programmed to one-input flip flop of pin ##.

Select a device with the specified flip-flop type.

Logical Error 5014: Register of xxx pin ## has no FC line.

Select a device with Dynamic Register Control feature.

Logical Error 5016: xxx can't map to pin ##'s [OE | CK].

Check the equation for invalid setting of the macrocell.

Logical Error 5017: xxx.OE is not allowed on pin ##.

Select a part with the specified output enable.

Logical Error 5018: xxx can't map to OE of pin ##; only VCC or GND allowed.	Select a part with the specified output enable.
Logical Error 5019: xxx can't map to OE of pin ## only pin ## allowed.	Modify the OE equation to use the external OE pin.
Logical Error 5020: xxx can't map to OE of pin ##; output is always enabled.	Select a part with the specified output enable.
Logical Error 5022: Polarity of pin ##'s OE doesn't match with xxx.	Invert the OE equation.
Internal Error 5023: drive_pin(): invalid enable type.	Contact the Data I/O Customer Resource Center.
Logical Error 5024: xxx pin ## is not a registered pin.	Reassign pin for the output with a registered pin.
Logical Error 5025: Status of xxx register has already been set for this group.	Reassign pin for the output with a registered pin.
Logical Error 5027: Register of xxx pin ## can't be bypassed.	Reassign pin for the output with a bypassable registered pin.
Internal Error 5028: progTerm(): bad node.	Contact the Data I/O Customer Resource Center.
Logical Error 5029: Register type of xxx pin ## is invalid.	Try other device with the specified flip flop type.
Logical Error 5030: Dynamic register type of xxx pin ## requires FC equation.	Write FC equation to use the dynamic register control.
Logical Error 5031: FC equation of xxx pin ## must be low to use D flip flop.	Invert the FC equation.
Logical Error 5032: Dynamic register type of xxx pin ## requires FC equation.	Write FC equation to use the dynamic register control.
Internal Error 5033: pregtype(): invalid register type.	Contact the Data I/O Customer Resource Center.
Internal Error 5034: Process_IC pSIG for ## is NULL.	Contact the Data I/O Customer Resource Center.
Logical Error 5035: xxx pin ## can't be used as feedback.	Reassign pin for the feedback with bidir pin.
Logical Error 5036: OR feedback is not allowed on xxx pin ##.	Reassign pin for the feedback with pin that has OR feedback.
Logical Error 5037: Registered feedback is not allowed on xxx pin ##.	Reassign pin for the feedback with pin that has Q feedback.
Logical Error 5038: Pin feedback is not allowed on xxx pin ##.	Reassign pin for the feedback with pin that has pin feedback.

Logical Error 5039: Too many feedbacks are used on xxx pin ##.	Check the use of feedbacks in equation with the physical feedback paths.
Logical Error 5040: Only one feedback is allowed on xxx pin ##.	Check the use of feedbacks in equation with the physical feedback path.
Internal Error 5041: term_alloc(): PT is null; signum = ##.	Contact the Data I/O Customer Resource Center.
Logical Error 5042: xxx can't use XOR on pin ##.	Reassign pin for the output with pin that has 2-input XOR gate.
Internal Error 5043: term_alloc(): Pin nn's TT should have been modified.	Contact the Data I/O Customer Resource Center.
Logical Error 5044: ## too many terms for output xxx pin ##.	Read the Design Consideration chapter in the ABEL manual on Polarity Control.
Fatal Error 5045: Device file not specified.	Run fuseasm with -dev <i>device_name</i> ; <i>device_name</i> is the targeted device.
Fatal Error 5046: Invalid pin number assigned to signal xxx.	Pin is out of range; try some other device.
Fatal Error 5047: Can't open file xxx.	Run cleanup4 to make rooms in the directory.
Fatal Error 5048: Unable to close file yyy.	Run cleanup4 to make rooms in the directory.
Logical Error 5050: Too many terms used by.	Read the Design Consideration chapter in the ABEL manual on Polarity Control.
Internal Error 5051: Internal error: Pin2MC.	Contact the Data I/O Customer Resource Center.
Internal Error 5052: Pin2MC324 Pin ##.	Contact the Data I/O Customer Resource Center.
Internal Error 5053: EP_fixup(): invalid polarity type.	Contact the Data I/O Customer Resource Center.
Logical Error 5054: Polarity of 'xxx.OE' pin ## does not match with that of the enable.	Invert the OE equation.
Internal Error 5055: E0600_fixup(): OE has to be a select mux.	Contact the Data I/O Customer Resource Center.
Internal Error 5056: E1800_fixup OE has to be a select mux.	Contact the Data I/O Customer Resource Center.
Warning 5057: Only the first ## signature chars will be programmed.	Use only one word with -ues <i>signature_word</i> .
Internal Error 5058: Too many product terms on pin ###.	Contact the Data I/O Customer Resource Center.
Fatal Error 5059: Too many terms used. ## max; try group reduction.	Read the Design Consideration chapter in the ABEL manual on Polarity Control.

Internal Error 5060: Too many bits in PROM word.	Contact the Data I/O Customer Resource Center.
Command Error 5061: Invalid output type for PROM ##.	Use -format with one of the valid type for PROM.
Warning 5062: Input/feedback xxxxxx has been reduced out.	PLAopt has reduced out the input.
Logical Error 5063: Input xxx has no 'uncomplemented' array connection.	Invert the input.
Logical Error 5064: Input xxx has no 'complement' array connection.	Invert the input.
Logical Error 5065: Local feedback xxxxxx can't be used in quadrant ##.	Use global feedback; that is, feedback that can feed to any quadrant.
Internal Error 5066: Startfuse #### is not start of term.	Contact the Data I/O Customer Resource Center.
Internal Error 5067: Startfuse #### is not in any array.	Contact the Data I/O Customer Resource Center.
Internal Error 5068: fuseinc of array nn is zero.	Contact the Data I/O Customer Resource Center.
Logical Error 5069: Select mux xxx can't be re-configured.	Check the equation for conflicting setting of the macrocell.
Logical Error 5070: xxx requires pin ## as its input.	Modify the equation to use the pin as input.
Logical Error 5071: Pin ## can't be assigned to xxx.	Reassign pin for the equation with a valid output pin.
Internal Error 5072: rpt_resrc(): Illegal feedback.	Contact the Data I/O Customer Resource Center.
Logical Error 5073: ? not an odd # of pins on side of device for PLCC.	Strange PLCC; ignore the message.
Warning 5074: Signal xxx was declared but not used.	Make sure this is not a mistake.
Internal Error 5076: Get_reg_str bad reg type.	Contact the Data I/O Customer Resource Center.
Fatal Error 5078: Writing to document file xxx.	Run cleanup4 to make rooms in the directory.
Internal Error 5079: Missing keyword %s in xxx.fus.	Contact the Data I/O Customer Resource Center.
Internal Error 5080: ## fuses in a set; exceed ## limit in xxx.fus.	Contact the Data I/O Customer Resource Center.
Internal Error 5081: 'c' expected, 'c' found while reading xxx.fus	Contact the Data I/O Customer Resource Center.
Logical Error 5082: Fuse #### in xxx.fus is out of range.	Check the fuses statement for invalid fuse numbers.

Fatal Error 5083: EOF found when writing X to JEDEC Hex file.

Run cleanup4 to make rooms in the directory.

Fatal Error 5084: EOF found when writing X to JEDEC file.

Run cleanup4 to make rooms in the directory.

Internal Error 5085: negative jedec number.

Contact the Data I/O Customer Resource Center.

Internal Error 5086: No symbol found in xxx.tmv.

Contact the Data I/O Customer Resource Center.

Warning 5087: Only one feedback is allowed on pin ##; mapping xxx to xxx.FB

Check the feedback for incorrect use by the software.

Command Error 5088: Programmer load file not specified. Use -o <filename>

Run fuseasm *infile* -o *filename.jed*.

Fatal Error 5089: Format code ## is not allowed. Use -format default.

Run fuseasm *infile* -format default.

Internal Error 5090: Illegal character xx found while reading xxx.tmv.

Contact the Data I/O Customer Resource Center.

Internal Error 5091: xx expected. xx found while reading xxx.tmv.

Contact the Data I/O Customer Resource Center.

Logical Error 5092: xxx node ## can't use shared feedback.

Move one of the two equations that use shared feedback to another free pin.

Logical Error 5095: PR/RE is not allowed for latched output xxx.

Remove PR/RE equation for latched output.

Internal Error 5096: Cluster has been clobbered.

Contact the Data I/O Customer Resource Center.

Fatal Error 5097: Pin ## is an illegal pin assignment for xxx.

Pin is out of range, try some other device.

Fatal Error 5098: PLA file type is not supported.

Run plaopt *infile* -reduce default.

Internal Error 5099: Invalid output pin ##.

Contact the Data I/O Customer Resource Center.

Internal Error 5100: Pal_choose bad pin ## index.

Contact the Data I/O Customer Resource Center.

Internal Error 5101: get_node(): illegal extension.

Contact the Data I/O Customer Resource Center.

Warning 5102: Polarity of yyy is not specified. Assume pos polarity.

Output has programmable inversion both before and after register.

Warning 5103: Polarity of yyy is not specified. Pin has neg polarity.

Output has fixed inversion after register; it may invert the PR/RE output.

Internal Error 5104: oe_choose(): bad enable type.

Contact the Data I/O Customer Resource Center.

Internal Error 5105: comb_choose pSIG ## is null.	Contact the Data I/O Customer Resource Center.
Logical Error 5106: xxx can't map to node ##; wrong polarity.	Output has fixed polarity type which is different from the equation; Run PLAOpt with -red default.
Logical Error 5107: xxx can't map to pin ##; wrong polarity.	Output has fixed polarity type which is different from the equation; Run PLAOpt with -red default.
Internal Error 5108: invalid_output_pin(): bad piny for pin ##.	Contact the Data I/O Customer Resource Center.
Logical Error 5109: xxx istype 'BUFFER' is not allowed on pin ##.	Select a device with no or programmable inversions.
Logical Error 5110: xxx istype 'INVERT' is not allowed on pin ##.	Select a device with fixed or programmable inversions.
Logical Error 5111: PR/RE is not allowed for combinatorial output xxx.	Modify to the primary equation to be registered.
Logical Error 5112: xxx.XR can't be used with xxx.T.	XOR is required to perform D-T emulation.
Command Error 5113: No input file specified.	Run fuseasm with a valid PLA file.
Logical Error 5114: Input to register xxx can only be from pin ##.	Check the equation for improper input register assignment.
Command Error 5115: Illegal use of options.	Check the command line options for illegal flags.

PLASim and JEDSim Messages

Warning 6001: Device not supported by simulator.	The current version of JEDSim doesn't support this device. Simulate the design with PLASim.
Fatal Error 6002: Error writing to output.	The disk does not have enough free space to contain the file being written.
Fatal Error 6003: Input file not specified.	A JEDEC file is expected as the input to JEDSim.
Fatal Error 6004: Could not open input file xxx.	The indicated file could not be opened by JEDSim. If the indicated file exists, check to make sure that it has the appropriate file protection settings to enable you to access it.
Fatal Error 6005: Could not open output file xxx.	The file could not be opened by JEDSim because there was not free space on the disk or a protected file of the same name already exists.

Fatal Error 6006: Device type must be specified.

The ABEL device type must be specified on the command line if the JEDEC file does not have the device type in the header. For example, to specify a P16R8 device type add "-device P16R8" to the command line. The JEDEC file header could be modified to include the following text string: "JEDEC file for: P16R8".

Fatal Error 6007: Could not open vector file xxx.

The ".tmv" vector file produced by AHDL2PLA could not be opened. If it was deleted, rerun AHDL2PLA to recreate it.

Fatal Error 6029: Single number expected in '-signal' xxx

A valid decimal number was not found after the "-signal" option. The ABEL 3.x range (14..17) is not supported.

Fatal Error 6030: No breakpoint vector number(s) found.

A valid decimal number was not found after the "-break" option.

Fatal Error 6031: No signals found in '-signal' list.

No signal names or numbers were found after the "-signal" option.

Warning 6032: '-signal' signal xxx not found in device.

The signal xxx was not found in the device. The names are case sensitive.

Warning 6033: No matching signals found in '-signal' list, using defaults.

No matching signals were found.

Fatal Error 6034: File name too long xxx.

The indicated file name is too long.

Fatal Error 6035: Reading input part of test vector.

The .tmv test vector file is defective.

Fatal Error 6036: Reading test vector '-'.

The .tmv test vector file is defective.

Fatal Error 6037: Reading output part of test vector.

The .tmv test vector file is defective.

Fatal Error 6038: Reading test vector ';;'.

The .tmv test vector file is defective.

Fatal Error 6051: Multiple .C, .LE or .LH specified for xxx.

Only one clock input is allowed for a register. Remove the unnecessary equation and reprocess the design.

Warning 6052: Second part of xxx XOR equation not found in PLA.

Two parts of a XOR equation must be present, indicated with a .X1 and .X2 suffix. This defective PLA file was missing one part of the XOR equation.

Warning 6054: No clock equation found for xxx.

PLASim requires a clock equation for each registered equation.

Fatal Error 6055: Incomplete design, can not simulate.

Information required by PLASim was missing from the PLA file.

Fatal Error 6056: Input file not specified.

A PLA file is expected as the input to PLASim.

Fatal Error 6057: Could not open input file xxx.	Check to make sure the input file exists and that you have read/write access.
Fatal Error 6058: Default output name requires input file with a .TTn extension.	To use the default output filename, the input file must have a .ttn extension.
Fatal Error 6059: Could not open output file xxx.	The disk does not have enough free space to contain the file being written.
Fatal Error 6060: Vector file not specified.	A ".tmv" test vector file must be specified on the command line with the "-ivector" option or in the PLA file with a "VECTORFILE" comment.
Fatal Error 6061: Could not open vector file xxx.	The ".tmv" vector file produced by AHDL2PLA could not be opened by PLASim. If it was deleted, rerun AHDL2PLA to recreate it.
Fatal Error 6062: Could not open input file xxx.	The ".tt0" PLA file produced by could AHDL2PLA not be opened by PLASim. If it was deleted, rerun AHDL2PLA to recreate it.
Fatal Error 6063: Could not open output file xxx.	The indicated working file could not be opened by PLASim.
Fatal Error 6064: Could not open input file xxx.	The indicated working file could not be opened by PLASim.
Fatal Error 6065: PLA has ## signals, the limit is ##.	The number of input and output signals in the PLA file exceed the maximum allowed in PLASim.
Internal Error 6066: Unknown extension xxx on pla output.	The PLA file is defective.
Warning 6100: Skipping unknown JEDEC field xx.	A reserved field identifier was used.
Fatal Error 6101: L field address exceeds fuse limit ##.	The fuses specified in the L field exceeded the maximum allowable fuses address for the device. This error could be caused by selecting the wrong device.
Fatal Error 6102: Fuse address ## exceeds fuse limit ##.	While processing a string of fuses in a L field, the calculated fuse address exceeded the maximum allowable fuses address for the device. This error could be caused by selecting the wrong device.
Fatal Error 6103: Illegal fuse state xx at fuse address ##.	The allowable fuse states for a L field are "1" and "0".
Fatal Error 6104: H field address exceeds fuse limit ##.	The fuses specified in the H field exceeded the maximum allowable fuses address for the device. This error could be caused by selecting the wrong device.

Fatal Error 6105: Fuse address ## exceeds fuse limit ##.

While processing a string of fuses in a H field, the calculated fuse address exceeded the maximum allowable fuses address for the device. This error could be caused by selecting the wrong device.

Fatal Error 6106: Illegal fuse state xx at fuse address ##.

The allowable fuse states for a H field are "0" through "9" and "A" through "F".

Fatal Error 6107: Illegal default fuse state xx.

The allowable default fuse states are "0" and "1".

Warning 6108: Illegal default test condition xx.

The allowable default test conditions are "0" and "1".

Fatal Error 6109: QF value ## incorrect, ## fuses expected.

The number of fuses specified in the JEDEC QF field doesn't agree with the device file.

Fatal Error 6110: QP value ## incorrect, '##' pins expected.

The number of pins specified in the JEDEC QP field doesn't agree with the device file.

Fatal Error 6111: QV value ## is too large, '##' vectors allowed.

The PC version is limited to 65000 bytes of memory for test vector storage. This means a 24 pin device can have 2708 vectors and a 68 pin device can have 955 vectors.

Warning 6112: File Fuse Checksum = #### RAM Fuse Checksum = ####.

The calculated fuse checksum didn't match the value in the C field.

Warning 6114: Hex number expected.

Allowed hexadecimal digits are "0" through "1" and "A" through "F".

Warning 6115: Hex number too large.

Hexadecimal number was too large, normal limit is 4 digits.

Fatal Error 6116: Expecting end of field character '*'.

The field had extra characters.

Warning 6117: Vector ## is out of sequence.

A test vector was missing or out of sequence.

Warning 6118: Wrong number of test conditions in vector ##.

The test vectors should have the same number of test conditions as the QP field.

Warning 6119: PIN or NODE xxx has invalid or missing number.

The PIN and NODE note fields require a number for each signal.

Warning 6120: Pin or node ## has multiple names xxx and yyy.

The PIN and NODE note fields require a unique number for each signal.

Fatal Error 6500: Input file not specified.

No input JEDEC file was specified on the command line.

Fatal Error 6501: Could not open input file 'xxxx'.

The indicated file could not be opened by JED2AHD.L. If the file exists, check to make sure that it has the appropriate file protection settings to enable you to access it.

Fatal Error 6502: Could not open output file 'xxxx'.	The indicated file could not be opened for write. Check that you have available disk space, the specified path is valid, and the file directory protection allows you write access.
Fatal Error 6503: Device type must be specified.	There is no device name specified in the JEDEC file. You must specify a device name on the command line in this case.
Fatal Error 6504: Could not open report file 'xxxx'.	The indicated file could not be opened for write. Check that you have available disk space, the specified path is valid, and the file directory protection allows you write access.
Logical Error 6505: PROMs are not supported by JED2AHDL.	JED2AHDL cannot translate PROM devices.
Logical Error 6506: Device not supported by JED2AHDL.	This device cannot be translated by JED2AHDL. This usually means that the device contains some complex feature that is not supported in the JED2AHDL translator.
Internal Error 6507:	An internal program error has occurred. Contact the Data I/O Customer Resource Center.
Fatal Error 6508: Error writing to disk.	Check that disk space is available, and that the directory is not write protected.
Logical Error 6509: unknown f/f type 'nn' on pin 'nn'.	A flip-flop type was encountered that is not known to JED2AHDL. Contact the Data I/O Customer Resource Center.
Fatal Error 6510: Could not rename 'xxxx' to 'yyyy'.	If an ABEL source file exists that has the same name as the source file that JED2AHDL will create, it renames the existing file to "name.bak". This error indicates the rename failed. This is usually indicative of a file protection problem.

10 *Backward Compatibility*

This chapter contains the most recent changes to ABEL, and is not meant to be a complete list of all changes made since the first version of ABEL.

Syntax Changes

The table below gives the new language options for the ABEL 3.2 language elements that have changed. Complete information on new language options can be found in the chapter "Language Reference." Complete information on new processing options can be found in the chapter "Using ABEL Processing Modules."

ABEL 3.2 Syntax	New Syntax	Description
Signature Macro	-ues (Fuseasm)	User Electronic Signature Word
*FLAG	Options keyword	Processing Options
Turbo/Miser macros	-config noturbo/-nomiser (Fuseasm)	Turbo/Miser bits

*Old FLAG statements are converted to the Options statement and options are updated when compiled.

Multiple Devices in a Module

Multiple devices in a module are no longer supported. Designs that previously used more than one device statement and 'IN' statements will need to be modified. One possible modification is to create separate modules for each device specified. The 'IN' keyword is obsolete.

Automatic Node Names

Versions 3.0 through 3.2 of ABEL supported automatic node declarations. For certain devices, nodes could be referred to by name (such as P0 through P5 in an F105) without a node declaration. This feature is no longer supported. Designs that use automatic node names will need to be modified and the appropriate node declarations added.

Assignment Operators

It is important to understand the exact semantics of the two assignment operators that can be used when writing high-level equations. The `=` operator is used to specify a combinational equation, while the `:=` operator is used to specify a registered equation. The only correct use of the `:=` operator is in the pin-to-pin description of a register (D-type flip-flop, attribute `'reg'` or `'reg_d'`) output. Use of the `:=` in other contexts (such as other register types or for dot extensions) will result in a warning error. AHDL2PLA will attempt to resolve previous uses of `:=` by generating either the appropriate dot extensions or by substituting an unclocked assignment (`=`), but you should verify that the resulting circuit works as intended. Refer to the section on "Pin-to-pin Vs. Detailed Descriptions" in "Design Considerations" and to "Assignment Operators" in "Language Structure."

State Machines

The equation generation algorithm used in AHDL2PLA is different from earlier ABEL versions. The logic generated for complex state machines can be considerably different than in previous versions, possibly resulting in a different number of product terms. If you find a design no longer fits in a device because more product terms were produced, try using the `@DCSET` directive to reduce the terms required. (Refer to `@DCSET` in the chapter "Language Reference" and the cautions on using `@DCSET` in the chapter "Design Considerations.") Experimenting with the `'pos'` and `'neg'` attributes and renumbering state values may also reduce product terms.

Attribute Changes

`'Pin'`, `'Eqn'` and `'Fuse'` Attributes

The `'pin'`, `'eqn'` and `'fuse'` attributes were used in versions 3.0 through 3.2 to specify the configuration of select multiplexers in devices such as the E0600, E0900 and E1800. These attributes are no longer necessary, and will be ignored if specified in a design. Designs for the E1800 may need to be modified if the `'pin'` attribute was used to configure a clock signal. Use the `.CLK` dot extension to describe a clock from a pin.

`'Feed_Pin'`, `'Feed_Or'` and `'Feed_Reg'` Attributes

The `'feed_pin'`, `'feed_or'` and `'feed_reg'` attributes were used to specify the precise configuration of a feedback path for an output signal. These attributes have been made obsolete by the dot extensions `.FB` and `.PIN` (for `'feed_reg'` and `'feed_pin'`), and by the extensions `.D`, `.T`, `.J` and `.S` (for `'feed_or'`). AHDL2PLA will attempt to convert the feedback attributes to dot extensions during processing, but designs should be checked to make sure the converted feedback path is the expected result.

'Pos' and 'Neg' Attributes and Polarity Control

The automatic polarity control features of ABEL have been improved, but the changes may cause confusion when certain classes of designs are processed. In previous versions of ABEL, designs for devices featuring programmable polarity have used the 'pos' and 'neg' attributes to specify a particular polarity configuration. These files will have to be processed with the "fixed" option to PLAOpt to keep PLAOpt's automatic polarity selection features from overriding the specified output polarity. If this option is not specified, PLAOpt will choose the polarity that results in the fewest number of product terms regardless of the 'pos' and 'neg' attributes.

This is particularly important when using devices such as the P22V10, in which the powerup state and register preset operation are affected by the polarity used. To avoid having to specify the "fixed" option every time you used PLAOpt, you can modify the design to add the command line option, **-reduce fixed**, in an Options statement.

Note: The 'invert' and 'buffer' attributes can also be used to enforce the existence or non-existence of an output inverter to solve powerup and preset problems. Refer to the chapter "Design Considerations" in the ABEL User Manual.

Dot Extension Changes

Valid dot extensions and attributes have been revised. Please refer to the chapters "ABEL-HDL Language Structure" and "Language Reference."

On devices with selectable pin/register feedback (such as the E0310, P18CV8, P22CV102, P20CG10 and P18G8 devices) where the inverter between the register and pin was used, change any .Q dot extensions to .FB. The .Q will not function the same as earlier versions on these devices. The .FB extension will allow you to move from a device such as the P22CV102 to a device such as the P22V10 with no changes in the source file.

New Language Features

The next table gives new language features that were not available in ABEL 3.2. Refer to the main manual for complete information.

New Syntax	Description
@DCSET	Use Don't Care Set for Optimization
@ONSET	Don't Use Don't Care Set for Optimization
TRACE	Watch signals
PROPERTY	Custom fitter properties

In addition, the following changes have been made:

- The module name and device name can now have the same name.
- Dot extensions have been revised (see above).
- Attributes have been revised (see above).

Conceptual Changes

Unlike ABEL, which was intended as a design language with which to describe the circuitry of a specific PLD, ABEL-HDL allows designs to be entered and verified with little or no concern for the target device architecture.

In order to allow such "architecture-independent" design descriptions, certain language changes have been introduced in ABEL-HDL. In addition, certain design assumptions that could previously be made based on the known architecture of a specified device are now required to be specified in the language.

Architecture Independent Language Features

The "DEVICE" keyword is now optional. There is no need to specify a particular PLD architecture in an ABEL-HDL source file. It is also possible to omit pin numbers from signal declarations. For designs intended for PLD implementations, the absence of device declarations and/or pin numbers implies the later use of the device selector and/or device fitters, respectively.

If no device is specified, or a device is specified but no pin numbers are specified, you may need to specify certain attributes about declared signals that are used in your design. This is because the ABEL-HDL compiler will not be able to imply ("default") these signal attributes from the device attributes.

Refer to the discussion on Architecture Independence in the chapter "Design Considerations."

PIN and NODE Declarations

The PIN and NODE keywords have a more clearly defined, architecture independent purpose in ABEL-HDL. The PIN keyword is used to declare those input and outputs signals that must eventually be available on a device I/O pin. The NODE keyword is used to declare those signals that may be assigned to buried nodes within a device. (The use of the NODE keyword does not, however, restrict a signal to a buried node. It is quite possible that a signal declared with the NODE keyword will be assigned to a device I/O pin by a device fitter.) See "Pin" and "Node" in the "Language Reference" for more information.

Signal Attributes

Signal attributes are specified through the use of the ISTYPE statement. The syntax for signal declarations has been simplified somewhat so that pin or node declarations can be combined with ISTYPE statements in a single declaration.

The following signal declarations are all valid (use of the "=" operator for attribute assignment is still supported, but is discouraged):

```
q3,q2,q1,q0      NODE ISTYPE 'reg_SR';
Clk,a,b,c         PIN  1,2,3,4;
reset            PIN;
reset            ISTYPE 'com';
Output           PIN  15      ISTYPE 'reg,invert';
```


Attributes remove ambiguities that arise when no specific device architecture is declared. If no device-related attributes are expressed in an ISTYPE statement, the design may not operate the same when targeted to different device architectures. Refer to the 'attr' section in the "Language Reference" for more information and valid attributes.

Program Changes

The program modules have been rewritten to support architecture-independent design and Open ABEL. The program names have also been changed, so you can have an older version of ABEL and the new ABEL residing on the same system. Refer to the chapter "Using ABEL Processing Modules" for complete information.

Old Module	New Module
Parse & Transfor	AHDL2PLA
Reduce	PLAOpt
Fusemap	Fuseasm
Simulate	PLASim & JEDSim
Document	Not available. Some options transferred to Fuseasm.

Option Changes

The ABEL command line options now have a standard syntax. All options consist of a dash (-) followed by a keyword. Some options can have one or more modifiers, which are separated from the keyword and each other with spaces.

The table below gives the new command line options equivalent to the ABEL 3.2 Command Line Options. **This table is NOT a complete list of the new options**, but is provided to assist experienced ABEL 3.2 users to become acquainted with the new options.

ABEL 3.2 Options	New Option
-a — Pass Arguments	-args <i>arguments</i>
-b — Break Points	-break <i>first_vector last_vector</i> *
-c — Checksum	-checksum [none dummy full]
-d — Load File Format	-format [default brief full hex pof 82 83 87 88]
-e — Expanded Output	-list expand
-f — Fusemap Terms	-document long (Fuseasm)
-g — Chip Diagram	-document plcc (Fuseasm)
-h — Include File Path	No longer necessary
-i — Input File Name	-i <i>filename</i> (must have a space)
-j — JEDEC File Path	No longer necessary

-k — Unused OR Fuses	-config ptintact
-l — List File	-list *
-n — Device Type	-device <i>device</i> , -idevice <i>filename</i> **
-o — Output File Name	-o <i>filename</i> (must have a space)
-p — -e & Directives	-list expand
-q — List Equations	pla2eqn program
-r — Reduction	-reduce [bypin group] [choose fixed] [exact]*
-s — List Symbol Table	Not supported
-t — Trace Level	-trace [none pins pins macro wave table] [brief clock detail]*
-u — Power Up State	-initial (0 1 h l)
-w — Watch Points	-signal <i>signal_names pin#s</i>
-x — Don't Cares (.X.)	-x (0 1 h l)
-y — Device File Path	No longer necessary
-z — High Z (.Z.)	-z (0 1 h l)

*Option usage differs from ABEL 3.2 conventions. Please refer to the chapter "Using ABEL Processing Modules."

** -device usage is the same as -n, -idevice usage differs.

The next table gives a short overview of the new features available from the command line with the respective programs in parentheses. Refer to the chapter "Using ABEL Processing Modules" for complete information.

New Command Line Option (Program)	Description
-ues (Fuseasm)	Electronic ID. Replaces the directive @ID.
-config lock (Fuseasm)	Program security fuse.
-errlog <i>filename</i> (All programs)	Error log file.
-help (All programs)	Gives command line options.
-usage (All programs)	Gives command line options.
-ivector <i>filename</i> (PLASim and JEDSim)	Input vector filename.
-ovector <i>filename</i> (AHDL2PLA)	Output vector filename.
-vectors (AHDL2PLA)	Create vector file only.
-silent	Turn off output to screen.

Simultaneous Operation with Earlier Versions

The program names have also been changed, so you can have an older version of ABEL and the new ABEL residing on the same system. A predefined constant, `_ABELHDL_`, can be used with `@IFDEF` and `@IFNDEF` to allow you to use both versions of ABEL on the same source file.

For ABEL-HDL code specific to ABEL 4.0, use

```
@IFDEF _ABELHDL_ {block}
```

For code specific to ABEL 3.2 and earlier, use

```
@IFNDEF _ABELHDL_ {block}
```

For example, a source file can contain an ABEL 4.0 attribute that is ignored by ABEL 3.2 by putting the ABEL 4.0-specific code in the `@IFDEF _ABELHDL_`:

```
@IFDEF _ABELHDL_
{
q0,q1,q2,q3 istype 'invert';
}
```

Conversely, ABEL 3.2 code can be added that ABEL 4.0 will ignore using `@IFNDEF _ABELHDL_`:

```
@IFNDEF _ABELHDL_
{
    L0.OE istype 'eqn';
}
```

Changes from Versions Prior to ABEL 3.2

The `ENABLE` keyword is obsolete. Use the `.oe` dot extension instead.

A `Declarations` keyword has been added to allow declarations anywhere in a source file. See "Declarations" in the chapter "Language Reference."

A 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			

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

```
<STX>File for PLD 12S8 Created on 8-Feb-85 3:05PM
6809 memory decode 123-0017-001
Joe Engineer Advanced Logic Corp *
QP20* QF448* QV8*
F0* X0*
L00111110111111111111111111111111*
L28101111111111111111111111111111*
L56111011111111111111111111111111*
L12010101111011110111111111111111*
L24010101111011101111111111111111*
L36010101110111011111111111111111*
V0001000000XXXNXXXHHHLXXN*
V0002010000XXXNXXXHHHLXXN*
V0003100000XXXNXXXHHHLXXN*
V0004110000XXXNXXXHHHLXXN*
V0005111000XXXNXXXHLHHXXN*
V0006111010XXXNXXXHHHHXXN*
V0007111100XXXNXXXHHHLHXXN*
V0008111110XXXNXXXLHHHXXN*
C124E*<ETX>8A76
```

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

Notation 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>] <f.name> {<m.name>} <l.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' | 'E' | '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>} '*'
```

For 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> '*'
```

For 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> '*'
```

For 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:

```
<fuse 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 '*'
```

For 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. For 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.

Following are three examples:

```
L0000
11111011111111111111111111
10111111111111111111111111
11101111111111111111111111
00000000000000000000000000*

L0000
111110111111111111111111111011111111
111111111111111111
111011111111
1111111111111111
00000000000000000000000000*

L00      1111101111111111111111111111*
L28 1011111111111111111111111111*
L56 1110111111111111111111111111*
L84 00000000000000000000000000000*
```

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								
Fuse No	7	6	5	4	3	2	1	0	
word 01	msb								
Fuse No.	15	14	13	12	11	10	9	8	

word 62	msb								lsb
Fuse No.	-	-	-	-	499	498	497	496	

Figure B-4
Computing the Fuse Checksum

QF500*
F0* L0000 01001110 00001000 11110000 11111111 01010001*
C021A*

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
	- - - - - - - -	
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' | 'I' | '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

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. For 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. For 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 B-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. For example,

```
V0001 000000XXXXXXXHHHLXXN*
V0002 010000XXXXXXXHHHLXXN*
V0003 100000XXXXXXXHHHLXXN*
V0004 110000XXXXXXXHHHLXXN*
```

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. For example,

```
P 1 2 3 4 5 6 14 15 16 17 7 8 9 10 11 12 13 18 19 20*
```

```
V0001 111000HLHHNNNNNNNNNN*
```

```
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 XX00XXXXXXNXXXXXXXXXXN*
```

```
V02 XX01XXXXXXNXXXXXXXXXXN*
```

```
V03 XX11XXXXXXNXXXXXXXXXXN*
```

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.

For example, (16R4 type programmable array logic device)

```
V1 PXXXXXXXXN1XX1101XXN*   Preload
V2 OXXXXXXXXN0XXHHLHXXN*   Test (don't clock)
V3 CXXXXXXXXN0XXHHLHXXN*   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 B0110100XXXXXXXXXXXXX* (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 B0110100101010101010* (preload first 18)
V06 B111010010001XXXXXX* (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> '*'
```

For 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
```

For example,

```
S0100010000111100011110110*
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> '*'
```

For example,

```
A25*
APD25*
```


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
01010111011110111111111111111111*
L0224
01010111101110111111111111111111*
L0336
01010111011101111111111111111111*
N Test Vectors*
V0001 000000XXXNXXXHHHLXXN*
V0002 010000XXXNXXXHHHLXXN*
V0003 100000XXXNXXXHHHLXXN*
V0004 110000XXXNXXXHHHLXXN*
V0005 111000XXXNXXXHLHXXN*
V0006 111010XXXNXXXHHHXXN*
V0007 111100XXXNXXXHLHXXN*
V0008 111110XXXNXXXLHHHXXN*
N Fuse RAM checksum*
C124E*
N Signature Analysis test information*
T01*
S000000000000000000000000*
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*
QF448* QV8* F0*
V1 000000000N000HHHL00N*
V2 010000000N000HHHL00N*
V3 100000000N000HHHL00N*
V4 110000000N000HHHL00N*
L0 11111011111111111111111111111111*
L28 10111111111111111111111111111111*
L56 11101111111111111111111111111111*
L84 00000000000000000000000000000000*
L112 01010111011110111111111111111111*
L224 01010111101110111111111111111111*
L336 01010111011101111111111111111111*
L140 11111111111111111111111111111111*
L140 00000000000000000000000000000000*
C124E*
V8 111111111N111HHHL11N*
V6 111010000N000HHHH00N*
V7 111100000N000HHLH00N*
V5 111000000N000HLHH00N*
V8 111110000N000LHHH00N*

```


C Operator Rules

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

```
!Ak :: = [!ak, !ak-1, ..., !a1]
-Ak :: = !Ak + 1
Ak.OE :: = [ak.OE, !ak-1.OE, ..., a1.OE]
ak Ak ak-1 a1
Ak & Bk :: = [ak & bk, ak-1 & bk-1, ..., a1 & b1]
Ak # Bk :: = [ak # bk, ak-1 # bk-1, ..., a1 # b1]
Ak $ Bk :: = [ak $ bk, ak-1 $ bk-1, ..., a1 $ b1]
Ak !$ Bk :: = [ak !$ bk, ak-1 !$ bk-1, ..., a1 !$ b1]
Ak == Bk :: = (ak == bk) & (ak-1 == bk-1 & ... & (a1 == b1))
Ak != Bk :: = (ak != bk) # (ak-1 != bk-1 # ... # (a1 != b1))
Ak + Bk :: = Dk
    where:
        dn :: = an $ bn $ cn-1
        cn :: = (an $ bn) # (an & cn-1) # bn & cn-1
        c0 :: = 0
Ak - Bk :: = Ak + (-Bk)
Ak < Bk :: = ck
    where
        cn :: = (!an & (bn # cn-1) # an & bn & cn-1)! = 0
        c0 :: = 0
Ak = Bk :: = !(Bk Ak)
```


Glossary

ABEL-HDL	ABEL Hardware Description Language. A behavioral design language used to describe logic circuits at a high level.
ABEL	A product consisting of a high-level hardware description language (ABEL-HDL) and processing modules that compile, reduce and simulate a design, and generate a programmer load file.
ABEL-PLA	ABEL-Programmable Logic Array. The ABEL-PLA format is used as the design transfer format between the various modules in the ABEL system, and as a design transfer vehicle for other design tools. ABEL-PLA is based on the Espresso format developed at the University of California at Berkeley. See also <i>PLA</i> .
ABEL Design Environment	The integrated editor and menu program for running the ABEL processing modules.
active level	The active level of a circuit is the electrical level associated with a true logic condition. A circuit that produces a zero voltage (assuming standard TTL levels) for a true condition is said to be active-low. A circuit that produces higher than zero electrical levels for these conditions is said to be active-high. Note that the active level of a circuit is not directly related to the existence of (or lack of) output inverters on device outputs.
architecture	The configuration of I/O pins, internal signals and circuit elements in a programmable logic device. A particular device architecture may be represented by many different manufactures' devices and be available with many different package and technology options.
architecture-dependent	A method of design description that is specific to a particular device architecture (or a limited set of architectures).

architecture-independent	A method of design description that allows a design to be implemented in a wide variety of programmable device architectures.
asynchronous	An asynchronous circuit is one in which the outputs change at any time, depending on the order in which the inputs change. See also <i>synchronous</i> .
auto-update	Allows the user to specify a desired end result without having to perform the intermediate steps. Auto-updating occurs whenever a file is out of date; that is, a preceding file is newer or missing. Auto-updating can be turned off on the Defaults menu.
batch file	A file containing operating system commands to perform a series of operations; also referred to as shell script.
candidate device	A device architecture that has been determined (by the device selector) to be a potential target for a specified logic design. Candidate devices are also selected based on device-specific constraints (such as available package types and technologies).
combinational/ combinatorial	A type of circuit that is a collection of logic gates with a set of inputs and outputs. The outputs are a direct function of the inputs, and no storage elements are used.
constant	In ABEL-HDL, a declared identifier and associated value that can be used in subsequent language statements. Constant declarations may be as simple as a numeric value, or as complex as whole ABEL-HDL logic expressions.
dc-set	The set of input conditions (known as don't care conditions) for which the output value of a logic circuit is unknown or irrelevant. In a type 'fr' PLA format, the dc-set is indicated by dashes found in the output array. See also <i>PLS</i> , <i>on-set</i> , <i>off-set</i> .
declaration	An ABEL-HDL statement that assigns a unique name to a design element, or modifies certain attributes of a design element. Design elements can include modules, devices, signals (pin and nodes), macros and constants.
design environment	The user interface and underlying software modules that provide the required features and the look-and-feel of ABEL4.
detail model	<p>The method (or way of thinking) used when designing for specific programmable device types - typically those devices featuring complex configurable macrocells.</p> <p>ABEL-HDL language elements associated with the detail model are the dot extensions described in the "Language Reference" chapter.</p>

device-independent	See <i>architecture-independent</i> .
device-specific	A method of design description that is specific to a particular device.
device selector	A software module that accepts an ABEL-PLA file and various device selection constraints to produce a list of candidate architectures. The SmartPart device selector also produces a detailed report of the specific manufacturer's devices that correspond to each of the device architectures selected.
equation	An ABEL-HDL design entry method that uses an extended form of Boolean equations to describe the operation of a circuit output in terms of its inputs.
expression	A syntactically correct sequence of ABEL-HDL operators, identifiers, and numeric or set values that can be evaluated to obtain a result, or converted to a Boolean expression.
Espresso	A logic minimization algorithm developed at the University of California at Berkeley. Espresso supports a variety of minimization options.
Fit	The generic fitter program that assigns the pins and performs the device-specific mapping of the design into the part.
Fitter	A software module that assigns device resources (such as I/O pins) of a specified device in order to implement a specified logic design. A fitter may or may not perform design conversion steps in order to fit a design to the constraints of the target device architecture.
Fuseasm	A software module that creates an architecture-specific fuse pattern reflecting the specified logic circuit. The fuse pattern (typically a JEDEC or PROM format file) can be used by device programming hardware to program actual devices.
fuse map	A representation of the fuses in a programmable logic device that shows which fuses are blown (disconnected) or left intact (connected).
identifier	A name or label used in a source file to reference input pins, nodes, sets, constants, keywords, or output pins.
JEDEC	(Joint Electronic Device Engineering Council) Used in the documentation to signify a standard data format for transfer of programming data to logic programmers.
keyword	A reserved identifier that is part of the ABEL-HDL language.

list file	The report file produced by the ABEL-HDL compiler. This file contains numbered source file lines and shows where errors (if any) occurred. The list file can also be used to examine how complex source file elements (such as macros and directives) were expanded.
normalized to the pin	Register feedback normalized to the pin means that register feedback is used, but the feedback will be the same polarity as on the output pin. This is architecture-independent feedback that allows you to write equations with pin-to-pin descriptions without consideration of whether or not the value of the register is inverted from the pin value.
off-set	The complete set of input conditions (normally expressed as one or more product terms) that will cause a circuit output to be false. Off-sets are indicated in a type 'fr' PLA by zeroes appearing in the output array. In a type 'f' PLA, off-sets are implied by omission. See also <i>PLA</i> , <i>on-set</i> , <i>dc-set</i> .
on-set	The complete set of input conditions (normally expressed as one or more product terms) that will cause a circuit output to be true. On-sets are indicated in an PLA representation by ones appearing in the output array. See also <i>PLA</i> , <i>off-set</i> , <i>dc-set</i> .
pin-to-pin model	The method (or way of thinking) used when designing a circuit that can be implemented in a wide variety of device architectures. As the name implies, pin-to-pin descriptions refer only to the values expected on device output pins for various input combinations and previous output pin states.
PLA	Programmable Logic Array. A form of logic description that represents sum-of-product (2-level) logic as a table of input product terms and associated output values. See also <i>ABEL-PLA</i> , <i>product term</i> .
PLAOpt	A software module that optimizes a design (in the form of an ABEL-PLA file) for specified criteria. PLAOpt is based on the Espresso logic minimization algorithm. See also <i>Espresso</i> .
polarity	<p>In a programmable logic device, polarity refers to the number of inversions located between the sum-of-products matrix and an associated output pin. Zero inversions (or any even number of inversions) indicates positive polarity, while 1 inversion (or any odd number of inversions) indicates negative polarity.</p> <p>In a logic design, polarity refers to the form of logic representation as expressed in a PLA. A PLA with inverted outputs is negative polarity, while a PLA with non-inverted outputs is positive polarity.</p>

	Device and design polarities are simply gate-level configurations, and are not directly related to circuit active levels.
product term	A unique combination of input values, where each input value is expressed as either a true, false, or don't-care. A product term can be expressed as a Boolean AND operation.
programmer load file	A file that can be used by a programmer to burn parts. It contains fuse states and possibly test vectors. A JEDEC file is an example of a programmer load file.
reduction	The process of reducing user-supplied logic descriptions to a near-minimal form so that the number of product terms required to implement a function is reduced.
registered	A registered output is one that is the output of a clocked storage device.
Selector	A software module that selects candidate devices based on a design and user-specified criteria such as number of pins and manufacturer.
sequential	A type of circuit that contains logic gates and storage elements that retain values resulting from earlier input values.
simulation	The testing of the function of a logic design. Simulation applies user-supplied inputs to the logic function and compares the outputs obtained with the user-supplied outputs.
source file	A file containing one or more complete modules that describe logic designs. The source file is used as input to the language processor (AHDL2PLA).
state machine	A digital circuit that utilizes some form of memory and feedback to implement sequential operations. State machines have the concept of a current state and calculated next state.
synchronous	A circuit in which the outputs change only upon application of a clock pulse regardless of when the state of the inputs changes.
twos complement	A form of binary complementation in which all the bits of a binary number are complemented (inverted) and then 1 is added to the number. Adding the twos complement of a first number to a second number is the same as subtracting the first from the second.

Index

'attributes'

- 'attr', 4-10
 - and polarity control, 5-17
 - istype, 4-33
 - See also* Attributes
- 'buffer', 4-11
- 'com', 4-11
- 'eqn', obsolete, 9-2
- 'feed_or', obsolete, 9-2
 - Dot extensions, 9-2
- 'feed_pin', obsolete, 9-2
 - See also* .PIN
- 'feed_reg', obsolete, 9-2
 - .FB, 9-2
- 'fuse', obsolete, 9-2
- 'invert', 4-11
- 'neg', 4-11
 - changes, 9-3
- 'pin', obsolete, 9-2
- 'pos', 4-11
 - changes, 9-3
- 'reg', 4-11
- 'reg_d', 4-11
- 'reg_g', 4-11
- 'reg_jk', 4-11
- 'reg_sr', 4-12
- 'reg_t', 4-12
- 'xor', 4-12

-options

- 43, 7-31
- 50, 7-31
- a, obsolete, 9-5

See also -args
-args, 7-14
-b, obsolete, 9-5
See also -break
-batch, 7-14
-break, 7-19
-bw, 7-31
-c, obsolete, 9-5
See also -checksum
-checksum, 7-24
-config lock, 7-25
-config nomiser, 7-25
-config noturbo, 7-25
-config ptintact, 7-25
-d, obsolete, 9-5
See also -format
-device, 7-24
-document, 7-24
-e, obsolete, 9-5
See also -list
-errlog, 7-31
-f, obsolete, 9-5
-format, 7-26
-g, obsolete, 9-5
-h, obsolete, 9-5
-help, 7-31
-i, 7-31
 changes, 9-5
-initial, 7-18
-ivector, 7-31
-j, obsolete, 9-6
-k, obsolete, 9-6
See also -blow
-l, obsolete, 9-6
See also -list
-list, 7-14
-n, obsolete, 9-6
See also -device
-o, 7-31
 changes, 9-6
-ovectors, 7-14
-p, obsolete, 9-6
See also -list
-q, obsolete, 9-6
-r, obsolete, 9-6
See also -reduce
-reduce, 7-20
-reduce bypin, 7-21
-reduce choose, 7-21
-reduce dt, 7-22
-reduce exact, 7-22
-reduce fixed, 7-21
-reduce group, 7-22
-signal, 7-19
-silent, 7-31

- t, obsolete, 9-6
 - See also* -trace
- trace, 7-16
- u, obsolete, 9-6
 - See also* -initial
- ues, 7-26
- usage, 7-31
- vectors, 7-31
- w, obsolete, 9-6
 - See also* -signal
- x, 7-18, 7-25
 - changes, 9-6
- y, obsolete, 9-6
- z, 7-18
 - changes, 9-6

.constants.

- .C., 3-5
- .D., 3-5
- .F., 3-5
- .K., 3-5
- .P., 3-5
- .SVn., 3-5
- .U., 3-5
- .X., 3-5

.dot extensions

- .AP, 4-3
- .AR, 4-3
- .CE, 4-3
- .CLK, 4-3
- .D, 4-3
- .ext, 4-2
 - See also* Dot extensions
- .FB, 4-3
- .FC, 4-3
- .J, 4-3
- .K, 4-3
- .LD, 4-3
- .LE, 4-3
- .LH, 4-3
- .OE, 4-3
- .PIN, 4-3
- .PR, 4-3
- .Q, 4-3
- .R, 4-3
- .RE, 4-3

.S, 4-3
.SP, 4-3
.SR, 4-3
.T, 4-3
.X, 8-23
 See also Don't cares

= and :=

:=
 alternate flip-flop types, 5-8
 changes, 9-2
=
 4-8
 changes, 9-2

@directives

@Alternate, 4-14
@Const, 4-14
 example, 8-21
@Dcset, 4-15
 example, 5-15
@Exit, 4-15
@Expr, 4-15
@If, 4-16
@Ifb, 4-16
@Ifdef, 4-16
@Ifiden, 4-17
@Ifnb, 4-17
@Ifndef, 4-17
@Ifniden, 4-18
@Include, 4-18
@Irp, 4-19
 example, 8-20
@Irpc, 4-20
@Message, 4-20
@Onset, 4-20
@Page, 4-21
@Radix, 4-21
@Repeat, 4-21
 example, 8-21
@Standard, 4-22

^base

^b, 3-7

^d, 3-7

^h, 3-7

^o, 3-7

A

ABEL Design Environment

help, 7-4

overview, 7-6

program messages, A-1

starting, 7-7

tutorial, 2-7

viewing files, 7-27

ABEL-HDL

introduction to, 3-2

structure, 3-19

syntax, 3-3

ABEL-PLA, 1-2

abel4lib.dev, 7-34

ABELHDL, 9-7

abellib, 7-34

Active-low declarations, 5-8

actlow1.abl, 5-9

actlow2.abl, 5-9

Addition, 3-9

Afsim

See PLDgrade User Manual

AHDL2PLA, 7-12

-args, 7-14

-batch, 7-14

-list, 7-14

-ovectors, 7-14

summary, 7-3

See also Compile

AHDL2PLA logic reduction rules, 7-13

Altera Programmer Object File format, 7-26

@Alternate, 4-14

disabling, 4-22

AND, 3-8

alternate operator for, 4-14

.AP, 4-3

.AR, 4-3

Architecture

See Device

Architecture independence, 5-1

attributes, 5-2

dot extensions, 5-2, 5-11

dot extensions, example, 5-11

and reset, 8-32

resolving ambiguities, 5-3

- tutorial, 2-11
- Arguments, 3-18, 7-14
 - defining in Module statement, 4-38
 - dummy, 8-20
 - See also* Options
- Arithmetic operators, 3-9
- Arrays, complement, 5-34
- ASCII
 - table, D-1
 - valid characters, 3-3
- Assignment operators, 3-10
 - changes, 9-2
- Assignments
 - device, 4-26
 - multiple, 6-12
 - multiple, to same identifier, 3-12
 - node, 4-39
 - pin, 4-41
- Asynchronous preset
 - See* .AP
- Asynchronous reset
 - See* .AR
- Attributes, 4-10
 - 'buffer', 4-11
 - 'com', 4-11
 - 'eqn', obsolete, 9-2
 - 'feed_or', obsolete, 9-2
 - 'feed_pin', obsolete, 9-2
 - 'feed_reg', obsolete, 9-2
 - 'fuse', obsolete, 9-2
 - 'invert', 4-11
 - 'neg', 4-11
 - 'pin', obsolete, 9-2
 - 'pos', 4-11
 - 'reg', 4-11
 - 'reg_d', 4-11
 - 'reg_g', 4-11
 - 'reg_jk', 4-11
 - 'reg_sr', 4-12
 - 'reg_t', 4-12
 - 'xor', 4-12
 - and architecture independence, 5-2
 - changes, 9-2, 9-4
 - istype, 4-33
 - obsolete, 9-2
- Auto polarity, 7-21
- Auto-synthesize D/T flip flop, 7-22
- Automatic design updating, 7-8
 - disabling, 7-10
 - tutorial, 2-10
- Automatic node names, 9-2
- Automatic signal selection, 8-19

B

- $\wedge b$, 3-7
- Backward compatibility, 9-1
- barrel.abl, 6-17
- Base numbers, 3-7
 - changing, 4-21
- Batch operation, 7-32
- bcd7.abl, 6-25
- Bidirectional 3-state buffer, example, 6-18
- Bidirectional outputs
 - See* Feedback
- Binary, 3-7
- binbcd.abl, 6-35
- bjack.abl, 6-39
- Blackjack machine, 6-29
- Blocks, 3-6
- Boolean equations
 - See* Equations
- Boolean set operations, C-1
- Branch conditions
 - testing, 8-31
- Break points
 - advanced use of, 8-5
- Brief trace, 7-18, 8-7
- 'buffer', 4-11
 - example, 5-6
 - and polarity control, 5-18
- Buffered outputs
 - and simulation, 8-13
- Buried nodes
 - declaring, 4-39
- By pin reduction, 7-21

C

- .C., 3-5
- Cable configuration for downloading, 7-29
- Cancel button, 7-8
- Cascading
 - See* Feedback
- Case keyword, 4-23
- .CE, 4-3
- Chained if-then-else, 4-31
- Changes
 - See* "Backward Compatibility"
- Changing devices
 - See* Architecture independence and SmartPart User Manual
- Characteristics
 - See* Attributes
- Check boxes, 7-8
- Checksum, 7-24
- Chip diagram, 7-24
 - PLCC, 7-25
- cleanup4, 7-36

- .CLK, 4-3
- Clock inputs, 8-33
 - See also* feedback
 - and synchronous functions, 8-24
- Clock trace, 7-18, 8-8
 - and test vectors, 8-33
- Clocked memory element, 4-11
- Closing a module, 4-27
- cnt10p.abl, 8-30
- Code
 - See* ABEL-HDL
- 'com', 4-11
- Combinatorial device, attribute for, 4-11
- Command buttons, 7-8
- Command line, 7-29
 - help, 7-5
 - See also* Options
- Comment fields, B-6
- Comments, 3-6
- comp4a.abl, 6-22
- Compatibility with earlier versions, 9-1
- Compile menu, 7-12
 - tutorial, 2-8
- Compile options
 - arguments, 7-14
 - dialog box, 7-13
 - error check, 7-13
 - listing file, 7-14
 - vectors only, 7-13
- Complement arrays, 5-34
 - example, 5-35
- Complement operator, 3-12
- Complement signal attribute, 4-11
- Complementing
 - See* active-low
- @Const, 4-14
 - example, 8-21
- Constants, 3-5
 - declarations, 4-8
 - declared in macros, 4-14
 - intermediate expressions, 4-9
- Conversion
 - See* Backward Compatibility
- Converting old source files, 9-1
- Copy line, 7-11
- count4.abl, 6-11
- count4a.abl, 6-13
- Custom fitters
 - passing information to, 4-42
 - See also* SmartPart User Manual

D

- .D., 3-5, 3-7
- .D, 4-3, 5-10
 - example, 4-7
- D flip-flop
 - clocked memory element, 4-11
 - dot extensions, 4-4
 - gated clocked memory element, 4-11
 - unsatisfied transition conditions, 5-28
- D/T flip-flop auto-synthesize, 7-22
- Data fields, B-5
- Dc-set, 5-15
 - and optimization, 5-15
- dc.abl, 5-15
- @Dcset, 4-15
 - disabling, 4-20
 - precautions, 5-14
 - with state machines, 5-29
- Debugging state machines, 8-17
- decade.abl, 5-35
- Decimal, 3-7
- Declarations
 - active-low, 5-8
 - constants, 4-8
 - device, 4-26
 - fuses, 4-29
 - macro, 4-35
 - node, 4-39
 - pin, 4-41
 - signal, 3-23
 - structure, 3-22
- Declarations keyword, 4-25
- Declared equations vs. macros, 4-35
- Defaults menu, 7-10
- Definition fields, B-6
- Delete line, 7-11
- Design considerations, 5-1
- Design environment
 - See* ABEL Design Environment
- Design examples
 - See* Examples
- Design file
 - See* Source files
- Design process, 1-3
- Design updating
 - See* Automatic design updating
- Designs, partitioning, 7-33
- Detail descriptions, 5-4
 - and dot extensions, 5-13
 - example, dot extensions, 5-13 - 5-14
 - example, inverting, 5-6
 - example, non-inverting, 5-6
 - and macrocells, 5-4
 - when to use, 5-7
- Detail trace, 7-18, 8-9

- detail1.abl, 5-13
- detail2.abl, 5-14
- Device dependence
 - See* Architecture independence
- Device independence
 - See* Architecture independence
- Device keyword, 4-26
 - and architecture independence, 5-1
 - changes, 9-4
- Device programming fields, B-7
- Device selection, 7-22
- Device simulation, 7-15
 - tutorial, 2-9
- Device support
 - See* User Notes
- Device testing fields, B-9
- Device type, 7-24
- Device-specific
 - See* Architecture independence
- Devices
 - with clock inputs, 8-33
 - declaring fuse states, 4-29
 - finding, 7-33
 - more than one in module, 9-1
 - programmable polarity, 5-16
 - programming, 7-29
 - See also* User Notes
- Devsel
 - See* SmartPart User Manual
- Dialog boxes, 7-8
- DIP Chip diagram, 7-24
- Directives, 4-13
 - @Alternate, 4-14
 - @Const, 4-14
 - @Dcset, 4-15
 - @Exit, 4-15
 - @Expr, 4-15
 - @If, 4-16
 - @Ifb, 4-16
 - @Ifdef, 4-16
 - @Ifiden, 4-17
 - @Ifnb, 4-17
 - @Ifndef, 4-17
 - @Ifniden, 4-18
 - @Include, 4-18
 - @Irp, 4-19
 - @Irpc, 4-20
 - @Message, 4-20
 - @Onset, 4-20
 - @Page, 4-21
 - @Radix, 4-21
 - @Repeat, 4-21
 - @Standard, 4-22
 - changing base numbering system, 4-21
 - creating test vectors with, 8-19

- examining result of, 4-13
- if blank, 4-16
- if defined, 4-16
- if identical, 4-17
- if not blank, 4-17
- if not defined, 4-17
- if not identical, 4-18
- Display vector, 7-19
- Division, 3-9
- dmux1t8.abl, 6-8
- DOCUMENT, 9-5
- Don't cares
 - x, 7-25
 - @Dcset, 4-15
 - and simulation, 8-23
 - and simulation, 7-18
- Dot extensions, 4-2
 - .AP, 4-3
 - .AR, 4-3
 - .CE, 4-3
 - .CLK, 4-3
 - .D, 4-3, 5-10
 - .FB, 4-3, 5-10
 - .FC, 4-3
 - .J, 4-3
 - .K, 4-3
 - .LD, 4-3
 - .LE, 4-3
 - .LH, 4-3
 - .OE, 4-3
 - .PIN, 4-3, 5-10
 - .PR, 4-3
 - .Q, 4-3, 5-10
 - .R, 4-3
 - .RE, 4-3
 - .S, 4-3
 - .SP, 4-3
 - .SR, 4-3
 - .T, 4-3
 - and architecture independence, example, 5-11
 - and architecture independence, 5-2, 5-11
 - and detail descriptions, 5-13
 - example, detail, 5-13 - 5-14
 - and feedback, 5-10
 - no, 5-10
- Downloading, 7-29
- Dummy arguments, 3-18, 7-14
 - defining in Module statement, 4-38
 - example, 8-20

E

- Earlier ABEL versions, 9-1
- Editing a file, 7-11
- Editor, using your own, 7-11
- 8-bit barrel shifter, example, 6-15
- Emulation of flip-flops, 5-19
- Enable, obsolete, 9-7
 - See also* .OE
- Enables
 - See* .OE
- End keyword, 4-27
- Endcase, 4-23
- Endwith, 4-55
- Entry fields, 7-8
- 'eqn', obsolete, 9-2
- Equal, 3-9
- Equation simulation, 7-15
 - tutorial, 2-8
- Equations
 - for flip-flops, 5-9
 - multiple to same signal, 6-12
 - overview, 3-12
 - tutorial, 2-3
 - when-then-else, 4-28, 4-54
 - XOR, 5-18
- Equations keyword, 4-28
- Error 75, 8-23
- Error check, 7-13
- Errors, A-1
 - pinpointing in simulation, 8-6
- Espresso by pin, 7-21
- ETX
 - See* checksum
- Exact reduction, 7-22
- Examples
 - 1 to 8 demultiplexer, equations, 6-7
 - 12 to 4 multiplexer, equations, 6-4
 - 4-bit comparator, equations, 6-20
 - 4-bit counter/multiplexer, equations, 6-9
 - 8-bit barrel shifter, equations, 6-15
 - adder, 6-29
 - bidirectional 3-state buffer, equations, 6-18
 - binary to BCD converter, 6-29
 - blackjack machine, 6-29
 - blackjack machine, state machine, 6-29
 - memory address decoder, equations, 6-2
 - multiplexer, 6-29
 - 7-segment display decoder, truth tables, 6-23
 - three-state sequencer, state machine, 6-26
- Exclusive ORs
 - See* XORs
- Execution
 - See* Operation
- @Exit, 4-15
- Exiting the menus, 7-9

Expanded listing, 7-14
 source code from directives, 4-13
@Expr, 4-15
Expressions, 3-11
 directive for, 4-15
Extensions
 See Dot extensions

F

.F., 3-5
Factors, XOR, 4-56
Fatal error
 See Errors
Fault grading, 7-27
 See PLDgrade User Manual
.FB, 4-3, 5-10
.FC, 4-3
'feed_or', obsolete, 9-2
'feed_pin', obsolete, 9-2
'feed_reg', obsolete, 9-2
Feedback
 and asynchronous circuits, 8-15
 and dot extensions, 5-10
 See also Dot extensions
 and simulation, 8-14
 and synchronous circuits, 8-14
feedback.abl, 8-16
Fields
 comment, B-6
 data, B-5
 definition, B-6
 device programming, B-7
 device testing, B-9
 programming, B-2
 testing, B-2
File management, 7-9
Files
 including in source file, 4-18, 4-34
 See also Source files, Output files, Input files
 temporary, 7-36
Find, 7-11
finddev4
 See User Notes
findout.abl, 8-24
Fit
 See SmartPart User Manual
Fitters
 passing information to, 4-42
 See also SmartPart User Manual
Fixed polarity reduction, 7-21
FLAG statement
 See Options
FLAG statement, obsolete, 9-1

- Flags
 - See* Options
- Flip-flop mode control
 - See* .FC
- Flip-flops, 5-28
 - auto-synthesize D/T, 7-22
 - D-type, 5-28
 - detail descriptions, 5-7
 - and dot extensions, 5-9
 - emulation with XORs, 5-19
 - state diagrams, 5-8
 - using := with, 5-8
- foo1.abl, 5-5
- foo2.abl, 5-6
- foo3.abl, 5-6
- foo5.abl, 5-7
- foo7.abl, 5-7
- Form feed, 4-21
- Format of programmer load file, 7-26
- 4-bit counter/multiplexer, example, 6-9, 6-20
- FPGA translation, 7-26
 - command line, 7-34
 - dialog box, 7-26
- FPLAs and polarity, 5-17
- FPLS, preloading internal registers, 8-27
- Fuseasm, 7-23
 - checksum, 7-24
 - config lock, 7-25
 - config nomiser, 7-25
 - config noturbo, 7-25
 - config ptintact, 7-25
 - device, 7-24
 - document, 7-24
 - format, 7-26
 - ues, 7-26
 - x, 7-25
 - summary, 7-3
- 'fuse', obsolete, 9-2
- Fusemap, 7-23
- Fuses
 - security, 7-25
 - unused, 7-25
- Fuses keyword, 4-29
 - caution on, 4-29
- FutureNet schematic, 7-34

G

- G latch dot extensions, 4-4
- Gates
 - See* XORs
- Generic designs
 - See* Architecture independence
- Getting started

See also "Tutorials"
See Installation Guide
Glitch
 See Timing problems
Goto keyword, 4-30
Greater than, 3-9
Group reduction, 7-22
Guided session
 See "Tutorials"

H

^h, 3-7
Hazard
 See Redundancy
Header, 3-22, 4-49
Help, 7-4
Help menu, 7-4
Hexadecimal, 3-7
High impedance values and simulation, 7-18

I

Identifiers, 3-3
 choosing, 3-5
 in state machines, 5-26
 multiple assignments to, 3-12
 reserved, 3-4
@If, 4-16
If blank, 4-16
If defined, 4-16
If identical, 4-17
If not blank, 4-17
If not defined, 4-17
If not identical, 4-18
If-then-else keywords, 4-31
@Ifb, 4-16
@Ifdef, 4-16
 backward compatibility, 9-7
@Ifiden, 4-17
IFLDOC, 7-34
@Ifnb, 4-17
@Ifndef, 4-17
 backward compatibility, 9-7
@Ifniden, 4-18
Illegal states
 defining, 8-29
 example, 8-29
 and supervoltage preload, 8-29
@Include, 4-18
Indefinite repeat, 4-19
 character, 4-20
Independence

- See* Architecture independence
- Initialization
 - and preload vectors, 8-17
 - See also* Powerup state
- Input files
 - ABEL-HDL structure, 3-1
 - See also* Source files
 - summary, 7-3
- Input pin, 4-41
- Inputs, unspecified in simulation, 8-14
- Installation
 - See* Installation Guide
- Intel format, 7-26
- Interface
 - See* ABEL Design Environment
- Intermediate expressions, 4-9
- Internal error
 - See* Errors
- Internal registers, TTL preload, 8-27
- Internal signals in simulation output, 8-11
- 'invert', 4-11
 - example, 5-6
 - and polarity control, 5-18
- Inverting outputs
 - and preload, 8-25
 - attribute for, 4-11
- @Irp, 4-19
- @Irpc, 4-20
- Istype keyword, 4-33

J

- J, 4-3
- Jamload
 - See* Preload
- JED2AHDL, 7-34
- JEDEC file translation, 7-34
- JEDEC format, 7-26
- JEDEC notations, B-3
- JEDEC simulation, 7-15
- JEDEC Standard Number 3-A, B-1
- JEDSim
 - break, 7-19
 - initial, 7-18
 - signal, 7-19
 - trace, 7-16
 - x, 7-18
 - z, 7-18
 - break points, 7-19
 - summary, 7-3
 - See also* Simulation
- JK flip-flop

and :=, 5-8
clocked memory element, 4-11
dot extensions, 4-4
emulation of, 5-19

K

.K., 3-5, 4-3
Keystroke scripts
 See Macros
Keywords, 3-4
 case, 4-23
 declarations, 4-25
 device, 4-26
 end, 4-27
 equations, 4-28
 fuses, 4-29
 goto, 4-30
 if-then-else, 4-31
 istype, 4-33
 library, 4-34
 macro, 4-35
 module, 4-38
 node, 4-39
 options, 4-40
 pin, 4-41
 property, 4-42
 state_diagram, 4-43
 test_vectors, 4-47
 title, 4-49
 trace, 4-50
 truth_table, 4-51
 when-then-else, 4-54
 with-endwith, 4-55
 XOR_factors, 4-56

L

Language
 See ABEL-HDL
Latch enable
 See Dot extensions
.LD, 4-3
.LE, 4-3
Less than, 3-9
.LH, 4-3
Library keyword, 4-34
Library manager
 See abellib
Limiting simulation output, 8-6, 8-11
Listing file, 7-14
Logic descriptions, 3-25
Logic operators, 3-8

M

- m6809a.abl, 6-3
- m6809b.abl, 8-20
- m6809c.abl, 8-22
- mac.abl, 4-36
- Macro keyword, 4-35
 - example, 4-36
- Macrocell trace, 7-17, 8-11
 - notations, 8-12
- Macros
 - creating test vectors with, 8-19
 - vs. declared equations, 4-35
- Mapping
 - design to programmer load file, 7-23
 - See* PartMap or Downloading
- Mealy, 5-21
- Mealy vs. Moore, 5-21
- mealy.abl, 5-24
- Memory address decoder, example, 6-2
- Menu operation, 7-7
- Menus, 7-6
 - tutorial, 2-7
 - See also* ABEL Design Environment
- Merge only, 7-22
- Merging PLA files, 7-33
- @Message, 4-20
- Messages, A-1
- Migrating designs
 - See* Architecture independence
- Minimization
 - See* Optimization
- Minus, 3-9
- Miser bits, 7-25
- Miser macro, 9-1
- Mode buttons, 7-8
- Module
 - beginning, 4-38
 - defined, 3-22
 - ending, 4-27
- Module keyword, 4-38
- Modulus, 3-9
- Moore, 5-21
- moore.abl, 5-25
- Motorola format, 7-26
- Multiple devices, 9-1
- Multiplication, 3-9
- mux12t4.abl, 6-6
- muxadd.abl, 6-33

N

- 'neg', 4-11
 - changes, 9-3
 - and polarity control, 5-17
- Nested if-then-else, 4-32
- No trace, 7-17, 8-6
- Node declarations
 - changes, 9-4
 - istype, 4-33
- Node keyword, 4-39
- Nodes
 - automatic names, 9-2
 - complement arrays, 5-34
- Non-inverting outputs
 - and preload, 8-25
 - attribute for, 4-11
- NOT, 3-8
 - alternate operator for, 4-14
- Not equal, 3-9
- Notations
 - JEDEC, B-3
 - macrocell trace, 8-12
- Note
 - See* Errors
- Numbers, 3-7
 - changing base, 4-21

O

- \wedge_0 , 3-7
- Obsolete attributes, 9-2
- Obsolete options, 9-5
- Octal, 3-7
- octalf.abl, 4-57
- .OE, 4-3
 - example, 4-7
- Off-set, 5-14
 - See* @DCSET
 - See also* Optimize
- OK button, 7-8
- Old source files, 9-1
- On-set, 5-14
 - See* @DCSET
 - See also* Optimize
- 1 to 8 demultiplexer, example, 6-7
- One-bit changes, 5-33
- @Onset, 4-20
- Open ABEL, 1-2
- Opening a file, 7-9
- Operating system shell, 7-9
- Operation
 - menus, 7-7
 - overview, 1-3
 - summary, 7-3

- with earlier ABEL versions, 9-7
- Operators
 - alternate set of, 4-14
 - arithmetic, 3-9
 - assignment, 3-10
 - complement, 3-12
 - logical, 3-8
 - overview, 3-8
 - priority, 3-11
 - relational, 3-9
 - standard set, 4-22
- Optimization
 - and @DCSET, 5-15
 - reducing product terms, 5-33
 - tutorial, 2-8
 - of XORs, 5-18
- Optimize
 - by pin auto polarity reduction, 7-21
 - D/T flip-flop auto-synthesize, 7-22
 - default reduction, 7-21
 - exact reduction, 7-22
 - fixed polarity, 7-21
 - group reduction, 7-22
 - merge only, 7-22
- Optimize menu, 7-20
- Optimize options dialog box, 7-20
- Optional programs, 7-10
- Options
 - 43, 7-31
 - 50, 7-31
 - args, 7-14
 - batch, 7-14
 - break, 7-19
 - bw, 7-31
 - checksum, 7-24
 - config lock, 7-25
 - config nomiser, 7-25
 - config noturbo, 7-25
 - config ptintact, 7-25
 - device, 7-24
 - document, 7-24
 - errlog, 7-31
 - format, 7-26
 - help, 7-31
 - i, 7-31
 - initial, 7-18
 - ivector, 7-31
 - ivectors, 7-18
 - list, 7-14
 - o, 7-31
 - ovectors, 7-14
 - reduce, 7-20
 - reduce bypin, 7-21
 - reduce choose, 7-21
 - reduce dt, 7-22

- reduce exact, 7-22
- reduce fixed, 7-21
- reduce group, 7-22
- signal, 7-19
- silent, 7-31
- trace, 7-16
- trace brief, 7-18
- trace clock, 7-18
- trace detail, 7-18
- trace macro, 7-17
- trace none, 7-17
- trace pins, 7-17
- trace table, 7-17
- trace wave, 7-17
- ues, 7-26
- usage, 7-31
- vectors, 7-31
- x, 7-18, 7-25
- z, 7-18
- changes, 9-5
- new, 9-6
- summary, 7-31
- See also* Command line
- Options keyword, 4-40
- OR, 3-8
 - alternate operator for, 4-14
- Output enables
 - See* .OE
- Output files
 - choosing format of, 7-26
 - summary, 7-3
- Output pin, 4-41
- Outputs
 - buffered, 8-13
 - stabilizing in simulation, 8-14

P

- .P., 3-5
- P22V10, example test vectors, 8-32
- @Page, 4-21
- PALASM-2 PDS format, 7-26
- par-det.abl, 2-11
- Parameters
 - See* Options
- PARSE, 9-5
- Partitioning, 7-33
- PartMap
 - blow product terms, 7-25
 - checksum, 7-24
 - document, 7-24
 - don't cares, 7-25
 - format of programmer load file, 7-26
 - miser bits, 7-25

- program security fuse, 7-25
- turbo bits, 7-25
- unused OR fuses, 7-25
- user electronic signature word, 7-26
- PartMap menu, 7-23
- PDS
 - PALASM-2 format, 7-26
 - Xilinx, 7-26
- .PIN, 4-3, 5-10
 - istype, 4-33
- Pin assignments, 4-41
 - See also* SmartPart User Manual
- Pin declarations
 - changes, 9-4
- Pin keyword, 4-41
- Pin-to-pin descriptions, 5-3
 - example, 5-5
 - and flip-flops, 5-9
 - resolving ambiguities, 5-3
- pin2pin.abl, 5-11
- 'pin', obsolete, 9-2
- Pins trace, 7-17, 8-10
- Pins, controlling preset/reset, 8-33
- PLA files, 1-2
 - merging, 7-33
 - partitioning, 7-33
 - translating to FPGA format, 7-26
- PLA2DASH, 7-34
- PLA2EQN, 7-34
- PLAMerge, 7-33
- PLAOpt, 7-20
 - reduce, 7-20
 - summary, 7-3
 - timing problems and, 8-1
 - See also* Optimize and Optimization
- PLASim
 - break, 7-19
 - initial, 7-18
 - signal, 7-19
 - trace, 7-16
 - x, 7-18
 - z, 7-18
 - summary, 7-3
 - See also* Simulation
- PLAsplit, 7-33
- PLCC chip diagram, 7-25
- PLDgrade, 7-27
 - See also* PLDgrade User Manual
- PLDmap, 7-23
- PLDmap Options dialog box, 7-24
- PLDs
 - See* Devices
- Plus, 3-9
- POF format, 7-26
- Polarity

- active levels, 5-16
- automatic selection, 5-17
- and FPLAs, 5-17
- positive, 4-11
- Polarity control, 5-16 - 5-17
 - changes, 9-3
- Porting designs
 - See* Architecture independence or Downloading
- 'pos', 4-11
 - changes, 9-3
 - and polarity control, 5-17
- Positive polarity, 4-11
- Powerup state, 5-27, 8-33
 - simulation, 7-18
- .PR, 4-3
 - example, 4-7
- Preload registers, 8-24, B-12
 - assume inverting, 8-25
 - supervoltage, 8-29
- Preset
 - See also* .PR
 - built-in, example, 5-6
 - controlling with pins, 8-33
 - controlling with product terms, 8-32
 - synchronous, 8-32
- Preset registers, 8-24
 - example, 8-25
 - special considerations, 8-25
 - timing diagram, 8-26
- preset.abl, 8-26
- Printing, 7-9
- Priority of operators, 3-11
- Problems
 - See* Errors
- Processing
 - compilation, 7-12
 - fault grading, 7-27
 - FPGA translation, 7-26
 - mapping, 7-23
 - optimization, 7-20
 - selecting a device, 7-22
 - simulation, 7-14
 - tutorial, 2-7
- Processing options, 4-40
- Product terms
 - blow, 7-25
 - controlling preset/reset, 8-32
 - reducing, 5-33
 - reducing with intermediate expressions, 4-9
- Program flow, 1-3
- Program pause, 7-10
- Program summary, 7-3
- Programmable polarity
 - active levels for devices, 5-16
- Programmable polarity and reset, 8-32

- Programmer load files
 - creating, 7-23
 - downloading, 7-29
 - format of, 7-26
 - tutorial, 2-9
- Programmer/tester options, B-13
- Programming fields, B-2
- Propagation delays
 - feedback, 8-14
 - timing problems, 8-1
- Property keyword, 4-42
- Protocol, transmission, B-4

Q

- .Q, 4-3, 5-10

R

- .R, 4-3
- @Radix, 4-21
- .RE, 4-3
 - example, 4-7
- Redraw screen, 7-12
- REDUCE, 9-5
- Reduction
 - See Optimization
 - XOR_factors, 4-56
- Redundancy, intentional, 8-1
- Refresh screen, 7-12
- 'reg', 4-11
- 'reg_d', 4-11
- 'reg_g', 4-11
- 'reg_jk', 4-11
- 'reg_sr', 4-12
- 'reg_t', 4-12
- regfb.abl, 8-6
- Register preload, B-12
- Registered designs
 - pin-to-pin vs. detail descriptions, 5-3
- Registers
 - asynchronous presets, 8-17
 - bit values in state machines, 5-33
 - cleared state in state machines, 5-28
 - dot extensions, 4-4
 - powerup states, 5-27
 - preload, 8-17
 - preset and preload, 8-24
- Relational operators, 3-9
- Repaint screen, 7-12
- Repeat, 4-21
 - @Irp directive, 4-19
 - @Irpc directive, 4-20

- example, 8-21
- Reset, 8-24
 - See also* .RE
 - and architecture-independence, 8-32
 - asynchronous, 8-32
 - controlling with pins, 8-33
 - controlling with product terms, 8-32
 - example, inverted architecture, 5-7
 - example, non-inverted architecture, 5-7
 - and programmable polarity, 8-32
 - resolving ambiguities, 5-7
- reset22a.abl, 8-32
- Resimulate equations, 7-15
- Resimulate JEDEC, 7-16
- Resource allocation, 7-24
- Response file, 7-30
- RS flip-flop
 - See* SR flip-flop

S

- .S, 4-3
- Sample
 - See* examples
- Saving a file, 7-9
- Schematics, 7-34
- Scripts
 - See* Macros
 - See also* Batch processing
- Search, 7-11
- Security fuse, 7-25
- Selecting a device, 7-22
- Selector
 - See* SmartPart User Manual
- sequence.abl, 5-27, 6-28
- Sequential circuits
 - See* Clock inputs
- Set operations, 3-13, C-1
- Sets, 3-13
 - assignment and comparison, 3-14
 - limitations, 3-17
- Setup
 - See* Installation Guide
- 7-segment display decoder, example, 6-23
- Shell, 7-9
- Shift, 3-9
- Signals, 7-19
 - automatic selection of, 8-19
 - multiple assignments, 6-12
 - nodes, 4-39
 - pins, 4-41
- Signature macro, 9-1
- Signetics program logic tables, 7-34
- Signetics snap format, 7-26

- simple.abl, 8-18
- Simplification
 - See* Optimization
- SIMULATE, 9-5
- Simulate equations, 7-15
- Simulate JEDEC, 7-15
- Simulation, 7-14
 - automatic signal selection, 8-19
 - brief trace, 7-18, 8-7 - 8-8
 - and buffered outputs, 8-13
 - clock trace, 7-18
 - detail trace, 7-18, 8-9
 - device, tutorial, 2-9
 - don't care values, 7-18, 8-23
 - equation, tutorial, 2-8
 - fault grading, 7-27
 - and feedback, 8-14
 - high impedance values, 7-18
 - limiting data, 8-6
 - macrocell trace, 8-11
 - macrocell trace notation, 8-12
 - model of, 8-3
 - multiple test vector sections, 8-18
 - no trace, 8-6
 - operation, 8-4
 - pinpointing errors, 8-6
 - pins trace, 8-10
 - powerup state, 7-18
 - register preloads, 8-17
 - stabilizing outputs, 8-14
 - table trace, 8-9
 - and test vectors, 8-17
 - test_vectors, 4-47
 - .tmv file, 7-18
 - trace and break points, 8-5
 - trace keyword, 4-50
 - trace options, 7-16
 - and unspecified inputs, 8-14
 - watch signals, 7-19
 - wave trace, 8-13
- SmartPart, 7-22
 - See also* SmartPart User Manual
- Snap format, 7-26
- Source files
 - ABEL-HDL, 3-1
 - beginning, 4-38
 - declarations, 3-22
 - design considerations, 5-1
 - directives, 3-28
 - header, 3-22
 - introduction to, 2-1
 - logic descriptions, 3-25
 - managing in the ABEL Design Environment, 7-9
 - partitioning, 7-33
 - structure of, 3-19

- test vectors, 3-27
- tutorial, 2-3
- .SP, 4-3
- Spaces to tabs, 7-10
- Special constants, 3-5
- .SR, 4-3
- SR flip-flop
 - and :=, 5-8
 - clocked memory element, 4-12
 - dot extensions, 4-4
- @Standard, 4-22
- Startup
 - See* Installation Guide or "Tutorials"
- State machine example, 4-46, 5-24 - 5-25, 5-27, 8-29
 - @DCSET, 5-31
 - blackjack machine, 6-29
 - no @DCSET, 5-29
 - three-state sequencer, 6-26
- State machines
 - algorithm changes, 9-2
 - case keyword, 4-23
 - cleared register state, 5-28
 - and @DCSET, 5-16, 5-29
 - debugging, 8-17
 - design considerations, 5-20
 - goto, 4-30
 - identifiers in, 5-26
 - identifying states, 5-33
 - if-then-else, 4-31
 - illegal states, 5-28, 8-29
 - Mealy vs. Moore, 5-21
 - powerup register states, 5-27
 - reducing product terms, 5-33
 - state_diagram, 4-43
 - test vectors for, 4-47
 - transition statements, 4-43
 - tutorial, 2-14
 - using state register outputs, 5-33
 - with-endwith, 4-55
- State registers, 5-33
- State_diagram keyword, 4-43
- statema.abl, 4-46
- Strings, 3-8
- Structured vector failure, 8-23
- STX
 - See* checksum
- Subtraction, 3-9
- Sum-of-products, XOR_factors, 4-56
- Supervoltage preload, 8-29
- .SVn., 3-5
- Switches
 - See* Options
- Synchronous preset
 - See* .SP
- Synchronous reset

See .SR
Syntax
 changes, 9-1
 of ABEL-HDL, 3-3
Synthesis
 See D/T flip-flop auto-synthesize

T

.T, 4-3
T flip-flop
 and equations, 5-9
 clocked memory element, 4-12
 dot extensions, 4-4
Table trace, 7-17, 8-9
Tabular
 truth table, 4-51
Temporary files, cleaning up, 7-36
Terms
 product, 8-32
 redundant, 8-1
Test points
 See Nodes
Test vectors
 and branch conditions, 8-31
 conversion of JEDEC, 8-4
 creating, 8-19
 error 75, 8-23
 and illegal states, 8-29
 in .tmv file, 8-3
 multiple sections, 8-18
 output filename, 7-14
 and powerup states, 8-33
 and preset, reset, and preload registers, 8-24
 preset considerations, 8-25
 for reset/preset, 8-32
 and simulation, 8-17
 and state machines, 8-18
 and supervoltage preload, 8-29
 and trace clock, 8-33
 trace keyword, 4-50
Test_vectors keyword, 4-47
Testing
 See Simulation
Testing fields, B-2
Three-state sequencer, example, 6-26
Times, 3-9
Timing problems, 8-1
Title keyword, 4-49
.tmv file, 7-18
Trace, 7-16
 advanced use of, 8-5
 break points, 7-19
 brief, 7-18, 8-7

- clock, 7-18, 8-8
- detail, 7-18, 8-9
- macrocell, 7-17, 8-11
- none, 7-17, 8-6
- pins, 7-17, 8-10
- signal points, 7-19
- table, 7-17, 8-9
- wave, 7-17, 8-13
- Trace keyword, 4-50
- traffic.abl, 5-29
- traffic1.abl, 5-31
- Transferring designs, 5-2
 - See Architecture independence or Downloading
- TRANSFOR, 9-5
- Transition conditions, 5-28
- Transition statements, 4-43
- Transitions
 - case keyword, 4-23
 - if-then-else keywords, 4-31
- Translation, 7-34
- Transmission protocol, B-4
- Troubleshooting
 - See Errors
- Truth tables
 - and @Dcset, 4-52
 - 7-segment display decoder example, 6-23
 - tutorial, 2-16
- Truth_table keyword, 4-51
- tsbuffer.abl, 6-19
- TTL preload, 8-27
- ttlload.abl, 8-28
- Turbo bits, 7-25
- Turbo macro, 9-1
- Tutorial, 2-1
 - architecture independence, 2-11
 - automatic design updating, 2-10
 - compiling a design, 2-8
 - creating a design, 2-3
 - generating a programmer load file, 2-9
 - optimizing a design, 2-8
 - processing a design, 2-7
 - simulating equations, 2-8
 - simulating the device, 2-9
 - state machines, 2-14
 - truth table, 2-16
- 12 to 4 multiplexer, example, 6-4

U

- .U., 3-5
- Understanding ABEL, 1-1
- Unspecified inputs in simulation, 8-14
- Unspecified pins, 8-23
- Unused OR fuses, 7-25

- Updates
 - See* Backward Compatibility
- Updating
 - See* Automatic design updating
- User Electronic Signature Word, 7-26, 9-1
- Utilities, 7-33

V

- Validation
 - See* Simulation
- Vectors
 - displayed in simulation, 7-19
 - See also* Test vectors
- Vectors only, 7-13
- Verification
 - See* Simulation
- Version changes
 - See* Backward Compatibility
- View menu, 7-27
- Viewing files, 7-27

W

- Warning
 - See* Errors
- Watch signals and simulation, 7-19
- Wave trace, 7-17, 8-13
- When-then-else, 4-28
- When-then-else keyword, 4-54
- With-endwith keyword, 4-55

X

- .X., 3-5
- x1.abl, 5-18
- x2.abl, 5-19
- Xilinx PDS, 7-26
- XNOR, 3-8
 - alternate operator for, 4-14
- XOR, 3-8, 4-12
 - alternate operator for, 4-14
- XOR gate
 - attribute for, 4-12
- 'xor'[, 5-18
- XOR_Factors
 - example, 4-57
 - summary, 3-27
- XOR_factors keyword, 4-56
- xorfact.abl, 4-57
- XORs
 - example, 5-18 - 5-19

flip-flop emulation, 5-19
implied, 5-18
and operator priority, 5-19
optimization of, 5-18

Z

.Z., 3-5

ABEL Reference Card

ABEL-HDL

General

Comments	"comment" or "comment <End-of-Line>
Strings	'string'
Blocks	{ block }
Identifiers	Letter or _ followed by letters, digits or _
Number notation	Binary - ^b## Octal - ^o## Hex - ^h## Decimal - [^d]##
Special Constants	.C. .D. .F. .K. .P..sv2. .sv3. .sv4. .sv5. .sv6. .sv7. .sv8. .sv9. .U. .X. .Z

Syntax of Elements

Module	MODULE <i>module_name</i> [(<i>dummy_arg</i> [<i>dummy_arg</i> ...])]
Options	OPTIONS ' <i>option</i> ' [<i>option</i> ...] [;]
Title	TITLE ' <i>string</i> '
Declarations	DECLARATIONS <i>declarations</i> ;
Device	<i>device_id</i> DEVICE <i>real_device</i> ;
Pin	[[<i>pin_id</i>] [[<i>pin_id</i> ...]] PIN [<i>pin#</i> [<i>pin#</i>]] [ISTYPE ' <i>attributes</i> '] ;
Node	[[<i>node_id</i>] [[<i>node_id</i> ...]] NODE [<i>node#</i> [<i>node#</i>]] [ISTYPE ' <i>attributes</i> '] ;
Constant	id [<i>id</i>] ... = <i>expr</i> [<i>expr</i>] ... ;
Macro	<i>macro_id</i> MACRO [(<i>dummy_arg</i> [<i>dummy_arg</i> ...]) { <i>block</i> } ;
Istype	<i>signal</i> [<i>signal</i> ...] [PIN NODE [<i>##s</i>]] ISTYPE ' <i>attr</i> ' [<i>attr</i>] ... ;
Library	LIBRARY ' <i>name</i> '
Equations	EQUATIONS <i>equations</i>
When-Then-Else	WHEN <i>condition</i> THEN [[<i>element=expression</i>], [ELSE <i>equation</i>]; or WHEN <i>condition</i> THEN <i>equation</i> ; [ELSE <i>equation</i>];
Truth Table	TRUTH_TABLE (<i>inputs</i> -> <i>outputs</i>)
State Diagram	STATE_DIAGRAM <i>state_reg</i> [-> <i>state_out</i>] [STATE <i>state_exp</i> : [<i>equation</i>] [<i>equation</i>] : <i>trans_stmt</i> ...]
If-Then-Else	IF <i>expression</i> THEN <i>state_exp</i> [<i>ELSE</i> <i>state_exp</i>] ;
Chained If-Then-Else	IF <i>expression</i> THEN <i>state_expression</i> ELSE IF <i>expression</i> THEN <i>state_expression</i> ELSE <i>state_expression</i> ;

Case CASE [*expression* : *state_exp* ;
[*expression* : *state_exp* ;
[*expression* : *state_exp* ;] ...
ENDCASE ;

Goto GOTO *state_exp* ;

With-endwith transition_stmt *state_exp* WITH
equation
[*equation*] ...
ENDWITH ;

Fuses Section FUSES
fuse_number = *fuse value* ;
or
fuse_number_set = *fuse value* ;

XOR Factors XOR_FACTOR *signal*
name = *xor_factors* ;

Test Vectors TEST_VECTORS [*note*]
(*inputs* -> *outputs*)
[*invalues* -> *outvalues*] ...

Trace TRACE (*inputs* -> *outputs*)

End END [*module_name*]

Attributes

Attr.	Description
'buffer'	No inverter
'com'	Combinatorial
'invert'	Inverter
'neg'	Complement signal
'pos'	Do not complement signal
'reg'	Clocked memory element
'reg_D'	Clocked memory element - D-type flip-flop
'reg_T'	Clocked memory element - T-type flip-flop
'reg_SR'	Clocked memory element - SR-type flip-flop
'reg_JK'	Clocked memory element - JK-type flip-flop
'reg_G'	Memory element - D-type flip-flop with gated clock
'xor'	XOR gate

Directives

@ALTERNATE	Alternate operators
@CONST <i>id</i> = <i>expression</i> ;	Constant definition
@DCSET	Don't Care Set Adjustable
@EXIT	Abort processing in Compiler
@EXPR [<i>block</i>] <i>expr</i> ;	Expression
@IF <i>expr</i> . <i>block</i>	Include block if true
@IFB (<i>arg</i>) <i>block</i>	Include block if blank
@IFDEF <i>id</i> <i>block</i>	Include block if defined
@IFIDEN (<i>arg1</i> , <i>arg2</i>) <i>block</i>	Include block if identical
@IFNB (<i>arg</i>) <i>block</i>	Include block if not blank
@IFNDEF <i>id</i> <i>block</i>	Include block if not defined
@IFNIDEN (<i>arg1</i> , <i>arg2</i>) <i>block</i>	Include block if not identical
@INCLUDE <i>filespec</i>	Include file
@IRP <i>dummy_arg</i> (<i>arg</i> [<i>arg</i>] ...) <i>block</i>	Indefinite repeat
@IRPC <i>dummy_arg</i> (<i>arg</i>) <i>block</i>	Indefinite repeat, character
@MESSAGE ' <i>string</i> '	Print message
@ONSET	Don't Care Set Fixed
@PAGE	Form feed in listing file

@RADIX *expr* ; Change default base numbering
@REPEAT *expr block* Repeat block *n* times
@STANDARD Return to default operators

Dot Extensions

.AP	Asynchronous Preset
.AR	Asynchronous Reset
.CE	Clock-enable input to a gated-clock flip-flop
.CLK	Clock
.D	Data input to a D-type flip-flop
.FB	Registered feedback normalized to pin
.FC	Flip-flop mode control
.J	J input to JK Flip Flop
.K	K input to JK Flip Flop
.LD	Load input for registers
.LE	Active low latch enable
.LH	Active high latch enable
.OE	Output Enable
.PIN	Pin Feedback
.PR	Preset
.Q	Q feedback
.R	R input to SR Flip Flop
.RE	Reset
.S	S input to SR Flip Flop
.SP	Synchronous register preset
.SR	Synchronous register reset
.T	T input to a T-type (toggle) flip flop

Operator 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

Logical Operators

!	NOT: ones complement
&	AND
#	OR
\$	XOR: exclusive OR
!\$	XNOR: exclusive NOR

Relational Operators

==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Arithmetic Operators

-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

Assignment Operators

=	Combinatorial assignment
:=	Registered assignment (D-type flip-flops)

Language Processor

ABEL Batch Processing

abel4bat filename [options]

Global Options

@ response_filename.rsp
-i filename
-o filename
-help
-usage
-silent
-menu
-erlog

AHDL2PLA

ahdl2pla ahdfile [options]
-args arguments
-list [expand]
-ovector filename
-syntax
-vectors

PLASim

plasim plfile [options]
-ivectors filename
-break first_vector [last_vector]
-signal signal_names pin#s
-trace [pins|wave|table|macro|none]
[brief|clock|detail]
-x (0|1|h|l)
-z (0|1|h|l)
-initial (0|1)

PLAOpt

plaopt plfile [options]
-reduce [bypin|group] [(choose|fixed) [dt] [exact]]

Fuseasm

fuseasm plfile [options]
-ivectors filename
-device device
-ues signature_word
-document [brief|long] [plcc]
-checksum [none|full|dummy]
-format [default|82|83|87|88|hex|brief|full|pcf]
-config [ptintact|noturbo|nomiser|lock]
-x (0|1)

JEDSim

jedsim jedecfile [options]
-ivectors filename
-break first_vector [last_vector]
-signal signal_names pin#s
-trace [pins|wave|table|macro|none]
[brief|clock|detail]
-x (0|1)
-z (0|1)
-initial (0|1)

Valid Set Operators

Operators	Expression	--> Result
!,?	![a, b, c]	--> [!a, !b, !c]
-(unary)	-[a, b, c]	--> ![a, b, c] + 1
&, #, \$, !\$	[a, b, c] & [x, y, z]	--> [a & x, b & y, c & z]
all	[x, y, z] & a	--> [x, y, z] & [a, a, a]
.	[x, y, z].d	--> [x.d, y.d, z.d]
==	[a, b, c] == [x, y, z]	--> (a == x) & (b == y) & (c == z)
!=	[a, b, c] != [x, y, z]	--> (a != x) # (b != y) # (c != z)
+	[a3, a2, a1] + [b3, b2, b1]	--> [a3 \$ b3 \$ c2, a2 \$ b2 \$ c1, a1 \$ b1 \$ c0] where c2 = (a2 & b2) # (a2 & c1) # (b2 & c1) c1 = (a1 & b1) # (a1 & c0) # (b1 & c0) c0 = 0
-(binary)	[a, b, c] - [x, y, z]	--> [a, b, c] + (- [x, y, z])
<	[a3, a2, a1] < [b3, b2, b1]	--> c3 where c3 = (!a3 & (b3 # c2) # a3 & b3 & c2) != 0 c2 = (!a2 & (b2 # c1) # a2 & b2 & c1) != 0 c1 = (!a1 & (b1 # c0) # a1 & b1 & c0) != 0 c0 = 0
<=	[a, b, c] <= [x, y, z]	--> !([x, y, z] < [a, b, c])
>=	[a, b, c] >= [x, y, z]	--> !([a, b, c] < [x, y, z])
>	[a, b, c] > [x, y, z]	--> [x, y, z] < [a, b, c]
all	6 & [a, b, c]	--> [1, 1, 0] & [a, b, c]
all	6 & [a, b]	--> [1, 0] & [a, b]
all	6 & [a, b, c, d]	--> [0, 1, 1, 0] & [a, b, c, d]

ABEL Design Environment

Starting

abel4 [filename] [-43|-50|-bw]

Keyboard

[Alt]-letter	Pop up menu
[ALT]-[F1]	Unisite/2900 Terminal Emulator
[F1]/[L2]	Help
[F2]	Pop up options
[F4]	Process through Simulate JEDEC
[F5]	Same as <OK> in dialog box
[F6]	Clear entry field
[Esc]	Same as <Cancel> in dialog box
[Space]	Toggle (•) or select [X]
[Tab] or [Enter]	Next selection in dialog box
[Shift]-[Tab]	Previous selection in dialog box

Editor

[Ctrl]-[D]	Delete current line
[Ctrl]-[R]	Replicate current line
[Ctrl]-[N]	Find next search string
[Ctrl]-[Home]	Go to beginning of file
[Ctrl]-[End]	Go to end of file

Utilities

abellib [library] command { files }
-l name
-e name
-a name
-d name
jed2ahdl -i infile [-dev device] [-o outfile] [-rep map_file]
pla2eqn -i infile [-o outfile]
[-language abel|pds|lca|snap|none]
plasplit plfile -signal name [name ...] -o filename
plamerge -i plfile -a plfile -o filename

pla2dash plfile -o outfile

ifldoc -i jedecfile -o outfile [-device device]

finddev4 string

cleanup4 [all]

Options

SmartPart — Devsel

devsel plfile [options] -manufacturer mfr_name
-database filename
-device devices
-sort range
-power range
-uf1 range
-uf2 range
-pins range
-price range
-speed range
-utilization range
-package [dip|pdip|cdip|sdip|loc|plcc|dcc|sogc|sogp]
-spc [com|mill|ind|std]
-technology [cmos|ttl|ecl|gaas]
-erasable [no|yes, uv, ee]
-jamload [yes|no]
-log filename

range = exact ## | less ## | greater ## | range ## ##

SmartPart — Fit

fit plfile [options] fitter -i filename
-log filename
-device device
-idevice filename
-ad hoc filename
-first
-suppress
-preassign [keep|ignore|try]

PLDgrade

afsim jedecfile [options]
-o filename
-x (0|1)
-z (0|1)
-initial (0|1)
-document [brief|long]