

abelTM

DATA I/O

ABEL™ 3.1



January 1989

096-0037-002

January 1989

096-0037-002

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

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

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

Data I/O and FutureNet are registered trademarks of Data I/O Corporation. Data I/O acknowledges the trademarks of other organizations for their respective products or services identified in this document.

Copyright 1987, 1988, 1989 Data I/O Corporation. All rights reserved.

Table of Contents

1. Introduction

1.1 More About ABEL Features	1-6
1.1.1 Design Checking	1-6
1.1.2 Logic Reduction	1-6
1.1.3 Simulation	1-6
1.1.4 Functional Device Testing	1-6
1.1.5 Standard JEDEC Format Programmer Load File	1-7
1.2 System Requirements	1-7
1.3 Contents of the ABEL Package	1-8
1.4 Documentation	1-10
1.4.1 The ABEL Manual	1-10
1.4.2 System Specific Information	1-11
1.4.3 Logic Diagrams	1-11
1.5 Design Example Files	1-11
1.6 Notation Conventions	1-12
1.7 Software Update Service	1-14
1.8 Field Customer Support	1-14
1.9 License Agreement and Warranty	1-15
1.9.1 License	1-15
1.9.2 Term	1-16
1.9.3 Limited Warranty	1-16
1.9.4 Limitation of Remedies	1-17
1.9.5 General	1-17
1.10 Ordering	1-19

2. Installation

3. ABEL Source Files

3.1 Elements of the ABEL Source File	3-2
3.2 Examining an Example ABEL Source File	3-6
3.2.1 Purpose of the Address Decoder	3-7
3.2.2 The MODULE Statement	3-7
3.2.3 The FLAG Statement	3-8
3.2.4 The TITLE Statement	3-8
3.2.5 The DEVICE Declaration	3-8
3.2.6 PIN and NODE Declarations	3-9
3.2.7 CONSTANT Declarations	3-9
3.2.8 EQUATIONS Statements	3-10
3.2.9 Test Vectors	3-11
3.3 Processing an ABEL Source File	3-12
3.3.1 Entering the Command Line	3-12
3.3.2 New Files Created	3-14
3.4 Downloading the Programmer Load File	3-15

4. The ABEL Language Processor

4.1 ABEL Batch Processing	4-3
4.1.1 ABEL Libraries	4-5
4.1.2 Generated Output Files	4-6
4.1.3 Creating Your Own Batch or Command Files	4-8
4.2 PARSE	4-9
4.2.1 PARSE Listing File	4-13
4.3 TRANSFOR	4-16
4.4 REDUCE	4-18
4.4.1 Level 0 Reduction	4-19
4.4.2 Level 1 Reduction	4-19
4.4.3 Level 2 Reduction	4-20
4.4.4 Level 3 Reduction	4-20
4.4.5 Level 4 Reduction	4-21
4.5 FUSEMAP	4-22
4.6 SIMULATE	4-27
4.6.1 Input Files	4-33
4.6.2 SIMULATE Program Operation	4-34
4.6.3 Devices with Clock Inputs	4-36
4.6.4 SIMULATE Output File, Trace Level 0	4-38
4.6.5 SIMULATE Output File, Trace Level 1	4-39

4.6.6 SIMULATE Output File, Trace Level 2	4-42
4.6.7 SIMULATE Output File, Trace Level 3	4-43
4.6.8 SIMULATE Output File, Trace Level 4	4-47
4.6.9 SIMULATE Output File, Trace Level 5	4-49
4.6.10 SIMULATE Output File, Two Trace Levels	4-51
4.6.11 Simulation and Designs With Buffered Outputs	4-52
4.6.12 Simulation and Unspecified Inputs	4-52
4.6.13 Simulation for Designs With Feedback	4-53
4.6.14 EZSIM - A Batch File for Re-simulation of a Design	4-60
4.7 DOCUMENT	4-61

5. Transferring the Programmer Load File

5.1 Downloading to a Model 29 with LogicPak	5-2
5.2 Downloading to a Unisite Programmer	5-5
5.3 PROM Download (Model 29/UniPak2)	5-5

6. ABEL Utilities

6.1 TOABEL, PALASM to ABEL Converter	6-1
6.2 IFLDOC	6-2
6.3 ABELLIB, Library Manager	6-4
6.4 Library File Usage	6-5
6.5 JEDABEL, JEDEC File to Equations	6-6

7. Language Elements

7.1 Basic Syntax	7-1
7.2 Valid ASCII Characters	7-2
7.3 Identifiers	7-3
7.3.1 Reserved Identifiers	7-4
7.3.2 Choosing Identifiers	7-5
7.4 Strings	7-6
7.5 Comments	7-7
7.6 Numbers	7-8

Table of Contents

7.7 Special Constants	7-11
7.8 Operators, Expressions, and Equations	7-12
7.8.1 Logical Operators	7-13
7.8.2 Arithmetic Operators	7-14
7.8.3 Relational Operators	7-15
7.8.4 Assignment Operators	7-17
7.8.5 Expressions	7-17
7.8.6 Equations	7-20
7.9 Sets	7-23
7.9.1 Set Operations	7-24
7.9.2 Set Assignment and Comparison	7-26
7.9.3 Set Evaluation	7-28
7.9.4 Limitations/Restrictions on Sets	7-30
7.10 Blocks	7-32
7.11 Arguments and Argument Substitution	7-33

8. Language Structure

8.1 Basic Structure	8-1
8.2 MODULE Statement and Structure	8-3
8.3 FLAG Statement	8-6
8.4 TITLE Statement	8-7
8.5 Declarations	8-8
8.5.1 Device Declaration Statement	8-9
8.5.2 Pin Declaration Statement	8-10
8.5.3 Node Declaration Statement	8-12
8.5.4 Constant Declaration Statement	8-14
8.5.5 Macro Declaration Statement and Macro Expansion	8-16
8.5.6 ISTYPE Statement	8-19
8.5.7 LIBRARY Statement	8-24
8.6 Equations Statement	8-25
8.7 Truth Table Statement	8-26
8.7.1 Truth Table Header Syntax	8-26
8.7.2 Truth Table Format	8-28
8.7.3 Programmable Polarity Registers	8-29
8.8 State Diagrams	8-30
8.8.1 STATE_DIAGRAM Statement	8-31
8.8.2 IF-THEN-ELSE Statement	8-34
8.8.3 Chained IF-THEN-ELSE Statements	8-35

8.8.4 CASE Statement	8-36
8.8.5 GOTO Statement	8-38
8.8.6 WITH-ENDWITH Statement	8-39
8.9 Fuses Section	8-40
8.10 Test Vectors	8-42

9. Directives

9.1 @ALTERNATE Directive	9-2
9.2 @CONST (Constant) Directive	9-3
9.3 @EXIT Directive	9-4
9.4 @EXPR (Expression) Directive	9-4
9.5 @IF Directive	9-5
9.6 @IFB (If Blank) Directive	9-6
9.7 @IFDEF (If Defined) Directive	9-7
9.8 @IFIDEN (If Identical) Directive	9-8
9.9 @IFNB (If Not Blank) Directive	9-9
9.10 @IFNDEF (If Not Defined) Directive	9-10
9.11 @IFNIDEN (If Not Identical) Directive	9-11
9.12 @INCLUDE Directive	9-12
9.13 @IRP (Indefinite Repeat) Directive	9-13
9.14 @IRPC (Indefinite Repeat, Character) Directive	9-15
9.15 @MESSAGE Directive	9-16
9.16 @PAGE Directive	9-16
9.17 @RADIX Directive	9-17
9.18 @REPEAT Directive	9-18
9.19 @STANDARD Directive	9-19

10. Design Examples

10.1 6809 Memory Address Decoder	10-5
10.1.1 Design Specification	10-5
10.1.2 Design Method	10-6
10.1.3 Test Vectors	10-8
10.2 12 to 4 Multiplexer	10-9
10.2.1 Design Specification	10-9
10.2.2 Design Method	10-10

Table of Contents

10.2.3 Test Vectors	10-11
10.3 1 to 8 Demultiplexer	10-13
10.3.1 Design Specification	10-13
10.3.2 Design Method	10-14
10.3.3 Test Vectors	10-15
10.4 4-Bit Counter/Multiplexer	10-17
10.4.1 Design Specification	10-17
10.4.2 Design Method	10-19
10.4.3 Test Vectors	10-21
10.4.4 Multiple Assignments to the Same Signal	10-24
10.5 Three-State Sequencer	10-27
10.5.1 Design Specification	10-27
10.5.2 Design Method	10-27
10.5.3 Test Vectors	10-29
10.6 8-Bit Barrel Shifter	10-31
10.6.1 Design Specification	10-31
10.6.1 Design Method	10-32
10.6.3 Test Vectors	10-34
10.7 7-Segment Display Decoder	10-36
10.7.1 Design Specification	10-36
10.7.2 Design Method	10-36
10.7.3 Test Vectors	10-39
10.8 4-Bit Comparator	10-40
10.8.1 Design Specification	10-41
10.8.2 Design Method	10-42
10.8.3 Test Vectors	10-45
10.9 Bi-Directional Three-State Buffer	10-46
10.9.1 Design Specification	10-46
10.9.2 Design Method	10-47
10.10 Blackjack Machine	10-49
10.10.1 Design Specification - MUXADD	10-52
10.10.2 Design Method - MUXADD	10-53
10.10.3 Test Vectors - MUXADD	10-54
10.10.4 Design Specification - BINBCD	10-56
10.10.5 Design Method - BINBCD	10-56
10.10.6 Test Vectors - BINBCD	10-57
10.10.7 Design Specification - BJACK	10-61
10.10.8 Design Method - BJACK	10-61
10.10.9 Test Vectors - BJACK	10-64

11. Design Considerations

11.1 Using State Machines	11-2
11.1.1 Use Identifiers Rather Than Numbers for States	11-3
11.1.2 "Power On" Register States	11-5
11.1.3 Designing With Programmable Polarity Outputs	11-5
11.1.4 Unsatisfied Transition Conditions, D-TypeFlip-Flops	11-5
11.1.5 Unsatisfied Transition Conditions, Other Flip-Flops	11-6
11.1.6 Number Adjacent States for One-Bit Changes	11-7
11.1.7 Use State Register Outputs To Identify States	11-9
11.2 Solving Timing Problems with REDUCE	11-10
11.3 Passing Arguments from the Command Line	11-13
11.4 Effect of Equation Polarity on Reduction Speed	11-17

12. Simulation

12.1 Test Vectors and Simulation	12-2
12.2 Trace Levels and Breakpoints	12-3
12.3 Debugging State Machines	12-5
12.4 Multiple Test Vector Tables	12-6
12.5 Using Macros and Directives to Create Test Vectors	12-8
12.6 Don't Cares in Simulation	12-14
12.7 Preset and Preload Registers	12-17
12.7.1 Special Preset Considerations	12-18
12.7.2 TTL Preload	12-21
12.7.3 Supervoltage Preload	12-23
12.7.4 Preset/Reset Controlled by Product Term	12-28
12.7.5 Preset/Reset Controlled by Pin	12-30
12.7.6 Power-Up States	12-30
12.8 Asynchronous Circuits	12-31

13. Using Features of Advanced Devices

13.1 Output Enables	13-1
13.1.1 Pin Controlled Output Enable	13-1
13.1.2 Term Controlled Output Enable	13-2
13.1.3 Configurable Output Enable	13-4
13.2 Output Macro Cell Control with ISTYPE	13-6
13.2.1 Controlling Macro Cell Polarity	13-6
13.2.2 Controlling Macro Cell Feedback Point	13-8
13.2.3 Selecting or Bypassing Device Registers	13-11
13.2.4 Controlling Register Type	13-12
13.3 Controlling Device Nodes	13-13
13.3.1 Using Node Numbers	13-13
13.3.2 Using Dot Extension Notation	13-15
13.4 More On Feedback	13-17
13.4.1 Selectable Feedback Type	13-17
13.4.2 Multiple Feedback Paths	13-19
13.4.3 The .Q Dot Extension	13-21
13.5 Using Select Multiplexers	13-23
13.6 Using Selectable Register Types	13-25
13.7 4-Bit Shifter/Counter Design	13-26
13.7.1 The F159 Logic Diagram	13-26
13.7.2 Examination of the Source File	13-28
13.7.3 Combining Equations	13-31
13.7.4 Specifying the Flip-Flop Inputs	13-31
13.7.5 Test Vectors	13-33
13.8 Using Complement Arrays	13-37
13.9 Equations for XOR PALs	13-42
13.10 JK Flip-Flop Emulations	13-42

Appendix A. Error Messages

A.1 General Error Messages	A-2
A.1.1 Command Line Errors	A-2
A.1.2 Fatal Errors	A-3
A.1.3 Intermediate File Errors	A-6
A.1.4 Logical Errors	A-8
A.1.5 Preprocessor Errors	A-14
A.1.6 Syntax Errors	A-15
A.1.7 Device File/Internal Errors	A-21

A.2 Non-Fatal Simulation Errors	A-22
A.3 TOABEL Error Messages	A-24
A.4 IFLDOC Errors	A-25
A.4.1 Command Line Errors	A-25
A.4.2 Diagnostic Errors	A-26
A.4.3 JEDEC Input File Errors	A-26
A.5 IFLDOC Warning Messages	A-28
A.6 ABELLIB Error Messages	A-32
A.7 JEDABEL Error Messages	A-33

Appendix B. JEDEC Standard Number 3A

B.1 Introduction	B-1
B.2 Summary of Programming and Testing Fields	B-3
B.3 Special Notations and Definitions	B-4
B.3.1 Notation Conventions	B-4
B.3.2 BNF Rules and Definitions	B-5
B.4 Transitional Protocol	B-7
B.4.1 Protocol Syntax	B-7
B.4.2 Computing the Checksum	B-7
B.4.3 Disabling the Transition Checksum	B-8
B.5 Data Fields	B-9
B.5.1 General Field Syntax	B-9
B.5.2 Field Identifiers	B-9
B.6 Comment and Definition Fields	B-11
B.6.1 Design Specification Field	B-11
B.6.2 Note Field (N)	B-11
B.6.3 Device Definition Field (D) (Obsolete)	B-12
B.6.4 Value Field (QF,QP,QV)	B-12
B.7 Device Programming Fields	B-14
B.7.1 Syntax and Overview	B-14
B.7.2 Fuse Default States	B-15
B.7.3 Fuse List Field	B-15
B.7.4 Fuse Checksum Field	B-16
B.8 Device Testing Fields	B-18
B.8.1 Syntax and Overview	B-18
B.8.2 Default Test Condition Field (X)	B-19
B.8.3 Test Vectors	B-20

Table of Contents

B.8.4	Pin Sequence	B-21
B.8.5	Test Conditions	B-21
B.8.6	Register Preload	B-23
B.9	Programmer/Tester Options	B-25
B.9.1	Security Fuse (G)	B-25
B.9.2	Signature Analysis Test (S,R,T)	B-25
B.9.3	Access Time (A)	B-26
B.10	Data File Example	B-26

Appendix C. Programmable Logic Device Information

C.1	ABEL Support for Specific Devices	C-1
C.1.1	PROM Support	C-1
C.1.2	Devices Supported by TOABEL	C-2
C.1.3	Devices Supported by IFLDOC	C-2
C.2	Specific Device Information	C-3
C.2.1	Altera and Intel EPLDs	C-3
C.2.2	AMD	C-4
C.2.3	Exel 78C800	C-4
C.2.4	Lattices GALs	C-5
C.2.5	MMI P20RA10 and P16RA8	C-6
C.2.6	MMI P32VX10 and P22RX8	C-6
C.2.7	MMI P16X4/P16A4	C-6
C.2.8	Ricoh and VTI EPALs	C-7
C.2.9	Signetics	C-7
C.2.10	F405C and F405D	C-9
C.3	Device Nodes	C-9

Appendix D. Syntax Diagrams

D.1	How to Read Syntax Diagrams	D-1
D.2	ABEL Syntax Diagrams	D-3

Appendix E. ASCII Table

List of Figures

1-1.	Logic Design Steps with ABEL	1-5
3-1.	Source File Template	3-4
3-2.	Block Diagram; 6809 Memory Address Decoder	3-6
4-1.	Processing Flow of the Language Processor	4-2
4-2.	SIMULATE Processing Flow Diagram	4-35
4-3.	Trace Level 3 Simulation Output	4-46
4-4.	Synchronous Feedback Circuit	4-54
4-5.	Asynchronous Feedback Circuit	4-57
5-1.	Cable Configuration for Transfer Between an IBM-XT and a Data I/O Programmer	5-3
5-2.	Cable Configuration for Transfer Between an IBM-AT and a Data I/O Programmer	5-4
6-1.	Sample Output File From IFLDOC	5-3
8-1.	Structure of an ABEL Source File	6-2
8-2.	Feedback Paths for an E0310	8-23
8-3.	Pictorial State Diagram	8-33
10-1.	Block Diagram: 6809 Memory Address Decoder	10-5
10-2.	Simplified Block Diagram: 6809 Memory Address Decoder	10-8
10-3.	Block Diagram: 12 to 4 Multiplexer	10-9
10-4.	Simplified Block Diagram: 12 to 4 Multiplexer	10-10
10-5.	Block Diagram: 1 to 8 Demultiplexer	10-13
10-6.	Simplified Block Diagram: Demultiplexer	10-14
10-7.	Block Diagram: 4-Bit Counter With 2 Input Multiplexer	10-17
10-8.	Simplified Block Diagram: 4 -Bit Counter With 2 Input Multiplexer	10-19
10-9.	State Machine Bubble Diagram	10-28
10-10.	Block Diagram: 8-Bit Barrel Shifter	10-31
10-11.	Simplified Block Diagram: 8-Bit Barrel Shifter	10-32
10-12.	Block Diagram: 7-Segment Display Decoder	10-37
10-13.	Simplified Block Diagram: 7-Segment Display Decoder	10-37
10-14.	Block Diagram: 4-Bit Comparator	10-40
10-15.	Simplified Block Diagram: 4-bit Comparator	10-41
10-16.	Block Diagram: Bidirectional Tri-State Buffer	10-46
10-17.	Simplified Block Diagram: Tri-State Buffer	10-47

Table of Contents

10-18. Schematic: Blackjack Machine	10-50
10-19. Pictorial State Diagram: Blackjack Machine	10-63
11-1. D-Type Register with False Inputs	11-6
11-2. Circuit Using an Input and Its Complement	11-10
11-3. Timing Diagram for $F = B \& !C \# !A \& C$	11-12
12-1. Timing Diagram Showing Test Vector Action	12-20
12-2. Internal Register of the F159	12-21
12-3. A Cross-Coupled Flip-Flop	12-32
13-1. Output Enable Controlled by Device Pin	13-2
13-2. Output Enable Controlled by Product Term	13-3
13-3. Typical Multiplexer for Output Enable Modes	13-5
13-4. Controlling Macro Cells with ISTYPE	13-7
13-5. Macro Cell, Configurable to Combinatorial or Registered Output	13-11
13-6. Registered and Combinatorial Feedback (P16R4)	13-17
13-7. Selectable Feedback Paths (E0310)	13-18
13-8. Configurable Macro Cell (32VX10)	13-19
13-9. Location of the Q Signal in Registered Devices	13-21
13-10. Output Macro-Cell for the E1800	13-24
13-11. Logic Diagram of the F159 FPLS (partial)	13-27
13-12. Flip-Flop Input Notation Examples	13-33
13-13. Abbreviated F105 Schematic	13-41
13-14. JK Flip-Flop Emulation Using T Flip-Flop	13-42
13-15. T Flip-Flop Emulation Using D Flip-Flop	13-43
13-16. JK Flip-Flop Emulation, D Flip-Flop with XOR	13-43

List of Tables

1-1. Files Supplied with ABEL	1-9
1-2. Notation Conventions	1-12
4-1. Data Translation Format Codes and File Extensions	4-26
4-2. Notation Used in Simulation Output Files	4-44
7-1. Number Representation in Different Bases	7-9
7-2. Special Constant Values	7-11
7-3. Logical Operators	7-13
7-4. Arithmetic Operators	7-14
7-5. Relational Operators	7-15
7-6. Assignment Operators	7-17
7-7. Summary of Operators and Priorities	7-18
7-8. Valid Set Operations	7-27
9-1. Directives	9-1
9-2. Alternate Operator Set	9-2
10-1. Design Examples Supplied with ABEL	10-2
10-2. Address Ranges for the 6809 Controller	10-6
10-3. Counter Modes	10-18
10-4. Devices Used in the Blackjack Machine	10-52
10-5. States of the Blackjack State Machine	10-62

Listings

3-1.	Source File Describing an Address Decoder	3-5
3-2.	Messages Displayed During Processing	3-13
4-1.	PARSE Listing File with Errors from M6809ERR.ABL	4-14
4-2.	Corrected Source File, M6809A.ABL	4-15
4-3.	Clock Inputs, Trace Level 2 Output	4-37
4-4.	Trace Level 0 Output for M6809A.ABL	4-38
4-5.	Test Vectors Used to Create Simulation Error	4-39
4-6.	Level 1 Simulation Output, All Vectors	4-40
4-7.	Level 1 Simulation Output, Single Vector	4-41
4-8.	Level 2 Simulation Output, Single Vector	4-42
4-9.	Trace Level 4 Simulation Output	4-47
4-10.	Trace Level 4 Simulation Output	4-48
4-11.	Trace Level 5 Simulation Output	4-50
4-12.	Trace Levels 4 and 5 in the Same Output	4-51
4-13.	Source File: Synchronous Feedback Circuit	4-54
4-14.	Simulation Output, Trace Level 1: Synchronous Feedback Circuit	4-55
4-15.	Simulation Output (partial), Trace Level 2 Synchronous Feedback Circuit	4-56
4-16.	Source File: Async. Feedback Circuit	4-58
4-17.	Simulation Output, Trace Level 1: Asynchronous Feedback Circuit	4-58
4-18.	Simulation Output, Trace Level 2: Asynchronous Feedback Circuit	4-59
4-19.	Documentation Output for M6809A.ABL	4-64
6-1.	"Source" File Generated by JEDABEL	6-7
10-1.	Source File: Memory Address Decoder	10-7
10-2.	Source File: 12 to 4 Multiplexer	10-11
10-3.	1 to 8 Demultiplexer	10-15
10-4.	Source file: 4-bit Counter with 2 Input	10-22
10-5.	Multiple Equations Sections, 4-Bit Counter	10-25
10-6.	Source File: Three-State Sequencer	10-30
10-7.	Source File: 8-Bit Barrel Shifter	10-35
10-8.	Source File: 7-Segment Display Decoder	10-38
10-9.	Source File: 4-Bit Comparator	10-44
10-10.	Source File: Tri-State Bi-Directional	10-48
10-11.	Source File: Multiplexer/Adder/Comparator	10-55

10-12.	Source File: Binary to BCD Converter	10-59
10-13.	Source File: State Machine (Controller)	10-65
11-1.	Source File: State Machine	11-4
11-2.	Device Type Passed From Command Line, Memory Address Decoder	11-16
12-1.	Source File With Multiple Test Vectors Sections	12-7
12-2.	Test Vectors Described With a Macro and @IF and @IRP Directives	12-9
12-3.	PARSE Output Showing Expanded Output for a Macro and @IF and @IRP Directives	12-11
12-4.	Test Vectors Described With a Macro, @CONST and @REPEAT Directives	12-12
12-5.	PARSE Output with Expanded Output for a Macro and @CONST, @REPEAT Directives	12-13
12-6.	Assignment of Don't Care Value (.x.) to Design Outputs	12-15
12-7.	SIMULATE Results with Outputs Specified as Don't Care	12-16
12-8.	Test Vectors for Special Preset Conditions	12-19
12-9.	Invoking the TTL Preload Function	12-22
12-10.	The Illegal States Defined	12-25
12-11.	Test Vectors for Illegal States	12-25
12-12.	Using Test Vectors to Preload A State Machine	12-28
12-13.	Controlling Reset/Preset by Product Term	12-29
12-14.	Using the Input Side of the Test Vector	12-32
13-1.	Controlling Macro Cells with ISTYPE	13-9
13-2.	Using Node Numbers for Reset/Preset Functions	13-14
13-3.	Using Dot Extensions for Reset/Preset Functions	13-16
13-4.	Output Configurations for a P32VX10	13-20
13-5.	A Shifter/Counter in an F159	13-29
13-6.	SIMULATE Output for the Shifter/Counter	13-35
13-7.	Transition Equations for a Decade Counter	13-39

Table of Contents

Preface

Data I/O Device Support Policy/Liability

1. Data I/O strives to achieve more device support verification from semiconductor manufacturers than any other software developer.
2. Every effort is made to program an adequate number of samples according to the manufacturer-supplied specification. We verify the device with test vectors which we apply when programming the device.
3. The objective of Data I/O is to seek and obtain verification on all devices.
4. Data I/O has made every attempt to ensure that the device information (as provided by the device manufacturer) contained in our software and documentation is accurate and complete. However, Data I/O assumes no liability for errors, or for any damages, whether direct, indirect, consequential or incidental, that result from use of documents provided with the software, regardless of whether or not Data I/O has been advised of the possibility of such loss or damage.

Technical Questions?

If you have technical questions, contact the Customer Resource Center at the following telephone numbers:

USA

Outside Washington state	800-247-5700
Inside Washington state	206-867-6899

FAX Numbers	206-881-2215
	206-882-1043

International

Data I/O Japan	(03) 432-6991
Data I/O Europe	+31 (0)20 6622866
Data I/O Canada	416-678-0761
Data I/O Intercontinental	206-881-6444

Bulletin Board Number	206-882-3211
(for new information on	(1200/2400 baud, 8 bits,
Data I/O products)	no parity, 1 stop bit)

You may also contact the Customer Resource Center by sending your written inquiries to:

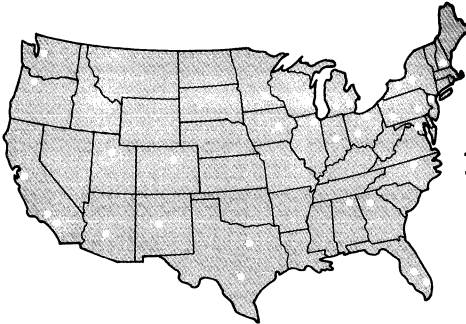
Data I/O Corporation
Customer Resource Center
10525 Willows Road N.E.
Redmond, WA 98073-9746 USA

Sales/Service Questions?

For questions regarding sales and update service agreements, refer to the list of telephone numbers on the facing page.

Telephone numbers are subject to change without notice.

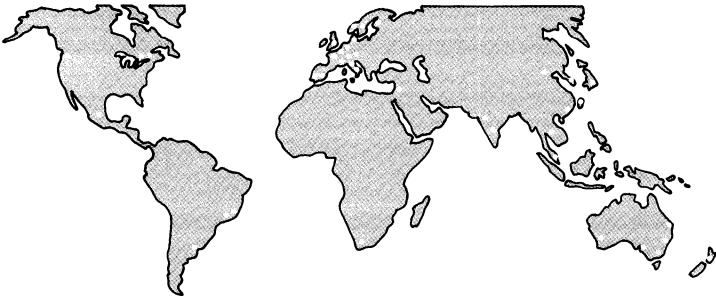
U.S. Sales Offices



1-800-247-5700
Dept. 1000

Data I/O Corporation
10525 Willows Road N.E.
P. O. Box 97046
Redmond, WA 98073-9746
U.S.A.

International Sales Offices and Representatives



INTERNATIONAL SALES OFFICES

EUROPE +31 (0)20-622866
CANADA (416) 678-0761
JAPAN (03) 432-6991

INTERNATIONAL REPRESENTATIVES

ARGENTINA
Reycom Electronica 01 701-4462
AUSTRALIA (Anitech)
Adelaide 08 356-7333
Brisbane 07 275-1766
Melbourne 03 795-9011
Sydney 02 648-1711
Perth 09 277-7000
AUSTRIA
Ing. Ernst Steiner 0222 827474
BELGIUM
Simac Electronics 02 2192451
BRAZIL (Sistronics)
Sao Paulo 011 247-5588
Rio de Janeiro 021 284-1248
CHINA (Dorado Company)
Beijing 507766
Hong Kong 3-770-2021
U.S.A. (206) 583-0000
DENMARK
ITT Multikomponent 02 456645

FINLAND
Instrumarium
Elektronikka 90 5284312
FRANCE
M.B. Electronique 13 9568131
GERMANY, FEDERAL REPUBLIC OF
Instrumatic Electronic
Systems GmbH 089 85802-0
GREECE
Eltronic Ltd. 01 7249511
HONG KONG
Eurotherm (Far East) Ltd. 5-546391
INDIA (Transmarketing Private Ltd.)
Bombay 022 4938585
Bangalore 0812 564389
ISRAEL
Telsys Ltd. 03 494891
ITALY (Sistrel)
Milan 02 6181893
Rome 06 5040273
KOREA
Elcom Systems Inc. 02 555-5222
MEXICO
Christensen 905 595-7594
NETHERLANDS
Simac Electronics 040 582911
NEW ZEALAND
Warburton Franki 0444-2645

NORWAY
Teleinstrument 02 789460
PORTUGAL
Decada 01 4103420
SINGAPORE/MALAYSIA
GEA Technology 2729412
SOUTH AFRICA
Electronic Building
Elements 012 46-9221
SPAIN
Unitronics 01 242-5204
SWEDEN
Teleinstrument 08 380370
SWITZERLAND (Instrumatic)
Thalwil 01 7231410
Geneva 022 360830
TAIWAN
Sertek International 02 5010055
THAILAND
Dynamic Supply
Engineering 02 3925313
UNITED KINGDOM
Microsystem Services 0494 41661

1. Introduction

ABEL™ is a complete logic design tool that lets you easily describe and implement programmable logic designs in PLDs, and PROMs. ABEL consists of a special-purpose, high-level language that is used to describe logic designs, and a language processor that converts logic descriptions to programmer load files. Programmer load files contain the information necessary to program and test programmable logic devices. ABEL may be used with other Data I/O-FutureNet design development tools such as:

PLDtest™; an automatic test vector generator that allows 100% testing of programmed logic parts

PLD-CADAT™; a program that allows integration of PLDs into a CADAT logic simulation. CADAT can then simulate the PLD as if it were a standard off-the-shelf part.

PLD-Linx™; a schematic diagram interface that converts schematic designs to ABEL source files

PROMlink™; a program that permits control of and communication with Data I/O™ programmers by means of a personal computer.

Features of the ABEL design language are:

- Universal syntax for all programmable logic types
- High-level, structured design language
- Flexible forms for describing logic:
 - Boolean Equations
 - Truth Tables
 - State Diagrams
- Test Vectors for simulation and testing
- Time-Saving Macros and Directives

The ABEL language processor also has many powerful features:

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

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

```
truth_table "3-input AND gate"
```

```
([ A, B, C ] -> Y)
```

```
[ 0,.X.,.X.] -> 0 ;
```

```
[.X., 0,.X.] -> 0 ;
```

```
[.X.,.X., 0] -> 0 ;
```

```
[ 1, 1, 1] -> 1 ;
```

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

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

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

Figure 1-1 shows the logic design process and the role ABEL takes in it. Beginning with the design concept, the designer creates the ABEL source file required by the language processor in order for it to generate the programmer load file. The source file is written by you and contains a complete description of your logic design. You can create the source file manually by means of a text editor (or word processor) that generates ASCII files, or you can use PLD-Linx to convert a DASH-generated schematic of the design to an ABEL source file.

The source file is presented to the ABEL language processor which performs several functions to produce a programmer load file (in JEDEC format) and design documentation. The first ABEL function, PARSE, checks the syntax of the source file and flags any errors. TRANSFORM converts the logic description to an intermediate form. REDUCE performs logic reduction, and FUSEMAP creates the programmer load file. The programmer load file can then be downloaded to the logic programmer to program parts, or can be first transmitted to PLDtest, an automatic test vector generator. The SIMULATE function tests the design of the part against your test vectors contained in the source file and reports any functional failure of the design. The DOCUMENT function generates a listing of the source file, a drawing of the logic device pin assignments, and a listing of the programmer load file.

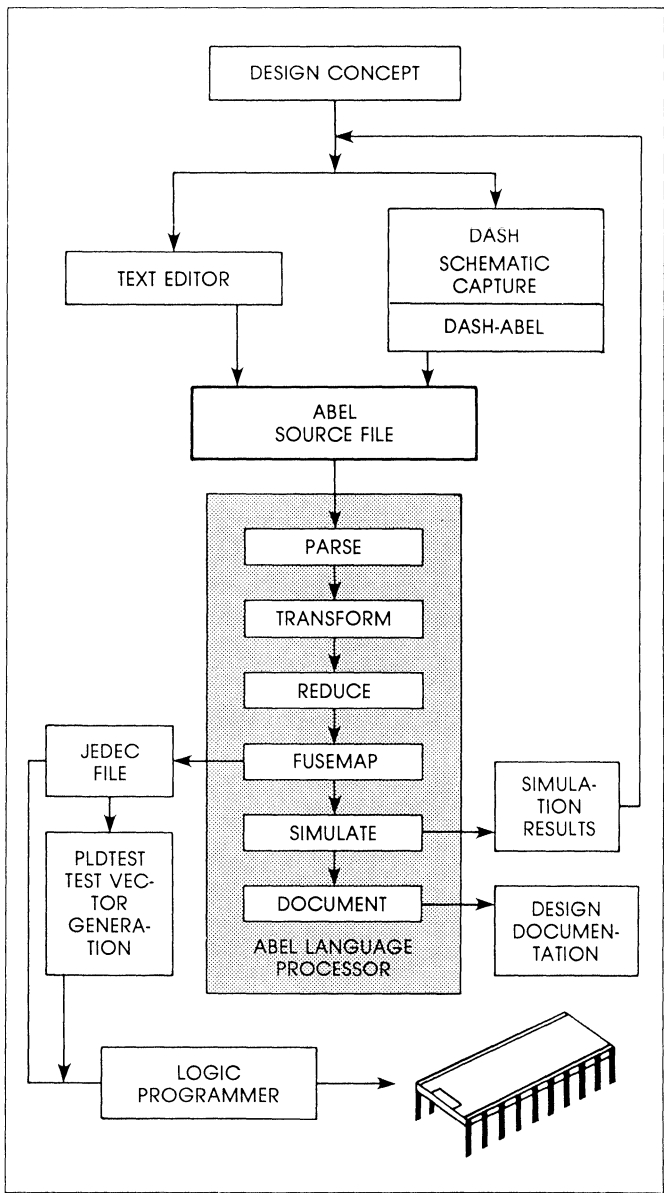


Figure 1-1. Logic Design Steps with ABEL

1.1 More About ABEL Features

1.1.1 Design Checking

The language processor checks your logic design for correct language syntax and explicitly tells you where an error occurs and what the error is. The language processor also checks your design to see if it can be implemented on the chosen device. For example, if a device input pin is used as an output in an equation, the language processor detects and reports the error.

1.1.2 Logic Reduction

The language processor reduces your logic design to a near minimal form, so that you do not have to perform the tedious task of logic reduction by traditional methods such as Karnaugh maps. You may choose different levels of reduction based on the design and the device.

1.1.3 Simulation

Simulation of a design is performed after a logic design has been reduced and converted to a programmer load file. The simulation facility uses device characteristics, a fuse map, and test vectors to simulate the actual operation of the device. The fuse map and test vectors used for simulation are the same as those that will be used to program and test the real device.

1.1.4 Functional Device Testing

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

1.1.5 Standard JEDEC Format Programmer Load File

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

1.2 System Requirements

ABEL presently runs on the following computers and operating systems. Versions for additional systems are under development.

IBM[®]/AT/XT/PS2[®] and MS-DOS[®] compatibles
VAX[™]/VMS[™]
VAX[™]/Unix[™]
Sun[™]
Daisy[™]
Apollo/Mentor Graphics[™]
Apple[®] MacIntosh[®] II or IIfx

The configuration information and installation instructions for ABEL differ for each type of system. To install ABEL in your particular system, refer to the System Specific Information supplied with your ABEL package. In addition to ABEL, you will need an editor or word processor with which to create ABEL source files. This may be any editor of your choice as long as it produces a standard ASCII file.

For downloading programmer load files to a logic programmer, you will need:

- An RS232 port and a cable to connect to the programmer.
- Cable and programmer configuration are discussed in Section 5 of this User's Guide.

1.3 Contents of the ABEL Package

The ABEL software package consists of the following:

- Floppy disks or magnetic tape
- A manual consisting of:

User's Guide
Language Reference
Applications Guide

- Logic Diagrams
- System Specific Information.

If any of the above items is missing, please contact your FutureNet representative. Table 1-1 lists the files residing on the disks or magnetic tape that are common to all versions of ABEL. Additional files that accompany your particular version are listed in the System Specific Information.

Table 1-1. Files Supplied with ABEL

File Name	Description
ABEL	Batch file for a complete ABEL run
PARSE	Language parser
TRANSFOR	Equation transformation program
REDUCE	Equation reduction program
FUSEMAP	Programmer load file generator
DOCUMENT	Documentation generator
IFLDOC	Program that converts JEDEC-format files to FPLA program tables for documentation purposes
TOABEL	PALASM to ABEL Converter
SIMULATE	Simulation facility
ESPRESSO	Equation reduction program used by REDUCE
ABELLIB	Device library librarian
ABEL3LIB.DEV	Device file library
* .ABL	Logic design examples
EXAMPLES.TXT	Describes the logic design examples included on the disk
JEDABEL	JEDEC file to ABEL translator
ABEL3LIB.INC	Library of include macros
DEVICES.TXT	Data base of ABEL supported devices
FINDDEV	Searches the device data base for a specified device
EZSIM	Quick simulator program
CLEANUP	Removes files created by ABEL
PSF	Menu system to run ABEL

1.4 Documentation

1.4.1 The ABEL Manual

The ABEL manual is divided into three major sections: User's Guide, Language Reference, and Applications Guide. The User's Guide describes how to use the ABEL language processor to create source files that contain logic descriptions, and convert the files to programmer load files. All processor options are described in detail and the output files are explained. The procedure for downloading programmer load files is also described.

The Language Reference contains a complete definition of the ABEL design language and is intended for use as both a quick reference and a general introduction to the ABEL design language.

The Applications Guide presents information and examples to help you use ABEL. Specifications, design methods, and complete source files are given for typical programmable logic designs. Advanced designs are also given, as well as suggestions and tips concerning the use of ABEL.

1.4.2 System Specific Information

This document describes how to install ABEL on your system in preparation for logic design. It also contains information regarding supplied software and files that can be run specifically on your system.

1.4.3 User Notes

User notes are provided to insert the latest device information into the manual, plus any other information that may be useful but was not available at the time the manual was printed.

1.4.4 Logic Diagrams

A set of device logic diagrams is provided for reference. Logic diagrams for most supported devices are included in the set.

1.5 Design Example Files

The ABEL manual shows complete logic designs described with ABEL and discusses general topics that you will find useful as you use ABEL. All the logic designs and design features discussed in this manual are contained on the Design Examples disk or tape you received with the ABEL package. The design examples are listed in table 10-1.

You can process these design examples with ABEL, either as they stand, or with your own modifications, to create programmer load files. Many of the design examples listed in table 10-1 are described in detail within this manual. Design examples not described in the manual, but listed in table 10-1, can be examined as necessary.

1.6 Notation Conventions

Table 1-2 lists notation conventions used in the definitions and syntax descriptions contained in this manual.

Table 1-2. Notation Conventions

Notation	Usage
<i>italics</i>	Indicates references by name to items contained in examples, figures, listings and tables.
quotation marks (')	Surround italicized items when the italicized reference contains spaces.
UPPER-CASE	In syntax descriptions and diagrams, indicates a keyword that must be entered in full. The entry may be either upper-case, lower-case, or mixed-case.
lower-case	In syntax descriptions and diagrams, indicates that a value or name is supplied by the user. The user-supplied value can be entered in either upper-case, lower-case, or mixed-case.
square brackets []	Surround optional entries that can be supplied or omitted as necessary.
ellipsis (...)	Indicates that the preceding item can be repeated as necessary.
all other punctuation	Apostrophes, exclamation points, parentheses, quotes and commas must be entered exactly as shown.

An example of italics used in text:

In table 1-2, the word *ellipsis* refers to three periods in a row.

An example of quotation marks used in text:

The file specification, "*m6809a fus*", is incorrect because a space separates the filename from the extension. The correct specification is *m6809a.fus*.

An example of upper-case, lower-case, square brackets, and ellipsis as used in a syntax description:

PARSE [-Iin_file] [-Aarg]...

The command, *PARSE*, must be entered, but it can be entered in either upper-case, lower-case, or mixed-case. An input file can be (but does not have to be) specified with the -I parameter by typing -I in either upper-case or lower-case followed by a file specification in place of *in_file*. Arguments can be supplied following the -A parameter. The -A parameter is typed in either upper-case or lower-case. An argument is supplied by the user in place of *arg*. As many arguments as desired can be entered, but each requires that a new -A parameter be used. The following would be a valid user entry:

parse -Ap16r4 -aGND

1.7 Software Update Service

The first year of software update service is included with your ABEL purchase. This service supplies you with software updates, "bug-fixes," and application ideas. To renew this update service, refer to the list of telephone numbers in the Preface.

1.8 Field Customer Support

FutureNet has Product Support Engineers who can provide you with additional assistance in using this product. The Product Support Engineer can be reached at 800-247-5700 within the Continental U.S. For assistance at other locations, call your nearest Data I/O Sales Office.

1.9 License Agreement and Warranty

The user should completely and carefully read the following terms and conditions before accepting delivery of this product. Possession of this software product indicates the user's acceptance of these terms and conditions. If the user does not agree with them, the user should promptly return the product and the purchase price will be refunded to the original purchaser of this product. Data I/O Corporation, licensor, provides this set of software programs and licenses their use. The user assumes the responsibility for the selection of the programs to achieve the user's intended results, and for the installation, use, and results obtained from said program(s).

1.9.1 License

1. The user may use the licensed program on any single machine.
2. The user may copy the licensed program into any machine readable or printed form for backup or archival purposes in support of the user's use of the program on the single machine.
3. The user recognizes the proprietary nature of the licensed programs and materials and agrees to preserve and protect Data I/O Corporation's interest therein. All information relating to the licensed programs and materials provided to the user shall be retained by the user and shall not be used or disclosed except for the purpose of meeting the user's own internal use.

1.9.2 Term

The license is effective until terminated. The user may terminate the license at any time by destroying the licensed program and materials together with all copies. The license will also terminate if the user fails to comply with any term or condition of this agreement. The user agrees upon such termination to destroy the licensed program and materials together with all copies in whatever form. Within 30 days after termination, the user shall certify in writing that through its best efforts and to the best of its knowledge all copies, in part or in whole, of the terminated and/or discontinued licensed programs and materials relating thereto have been destroyed.

1.9.3 Limited Warranty

1. The ABEL Software programs are provided by Data I/O Corporation "AS IS" without warranty of any kind, including but not limited to, the warranties of merchantability and fitness for a particular purpose. Data I/O Corporation will not be liable to the user of the ABEL software programs for any damages, including lost profits, lost savings, loss of use or other incidental or consequential damages arising out of the use, or inability to use, the provided software programs, even if Data I/O Corporation has been advised of the possibility of such damages, or for any other claims brought by any other party.
2. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to the user.
3. Data I/O Corporation does not warrant that the functions contained in the licensed program(s) will meet the user's requirements or that the operation of the licensed program(s) will be uninterrupted or error free.

4. Data I/O Corporation warrants the diskette (s) or magnetic tape or cassette (s) on which the programs are furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to the user as evidenced by a copy of the packing slip.

1.9.4 Limitation of Remedies

Data I/O Corporation's entire liability and the user's exclusive remedy shall be as follows:

1. The replacement of any diskette (s) or magnetic tape or cassette (s) not meeting Data I/O Corporation's limited warranty, and which materials are returned to Data I/O Corporation or an authorized Data I/O representative with a copy of the packing slip, or
2. If Data I/O Corporation or a representative is unable to deliver a replacement diskette (s), magnetic tape or cassette (s), which is/are free of defects in materials or workmanship, you may terminate this agreement by returning the programs and their related materials and your money will be refunded.

1.9.5 General

1. The user acknowledges that the user has read this agreement, understands it, and agrees to be bound by its terms and conditions. The user further agrees that it is the complete and exclusive statement of the agreement between the user and Data I/O Corporation, which agreement supersedes any proposal or prior

agreement, whether oral or written, and any other communications between the user and Data I/O Corporation relating to the subject matter of this agreement.

2. This license and agreement shall be governed by the laws of the State of Washington.
3. The user may not sub-license, assign, or transfer the licensed program (s) except as expressly provided in this agreement. Any attempt to do otherwise, to sub-license, assign or transfer any of the rights, duties or obligations hereunder is void, except that Data I/O Corporation may assign this license as a part of an assignment of all software assets of its business.
4. The provisions of this agreement are severable and if any one or more of such provisions are judicially determined to be illegal or otherwise unenforceable, whether in whole or in part, those remaining provisions or portions of the license unaffected by such determination shall be binding on and enforceable by and between the licensee, user, and Data I/O Corporation.
5. In the event it becomes necessary to retain the services of an attorney to enforce any provisions of this license agreement, the non-prevailing party to such legal action and/or litigation agrees to pay the prevailing party's costs, including a reasonable attorney's fee, and court costs, if any.

1.10 Ordering

To place an order for software, contact your Data I/O representative. Orders for shipment must include the following:

- Description of the software (see latest Data I/O price list or contact your sales representative for product numbers).
- Purchase order number
- Desired method of shipment
- Quantity of each item ordered
- Shipping and billing address of firm, including zip code
- Name of person ordering software

2. Installation

The procedure for installing ABEL in your system depends on the system (IBM-PC, VAX/VMS, etc.) to be used. Refer to the System Specific Information supplied with the ABEL package for details regarding installation of ABEL on your particular system.

3. ABEL Source Files

In order for the ABEL Language Processor to convert logic descriptions to programmer load files, the logic descriptions must be contained in an source file. That is, before you can create a load file that contains the fusemap of a particular logic design, you must create an ABEL source file that reflects that logic design. The ABEL source file is an ASCII text file that is written according to the requirements of the ABEL Language Processor. These requirements are described in detail in the Language Reference portion of the manual.

3.1 Elements of the ABEL Source File

You can write an ABEL source file using any word processor or text editor that produces ASCII files. The elements of the ABEL source file are shown in the template in Figure 3-1. Figure 3-1 shows the basic element of the ABEL source file, the module. A source may contain any number of modules, each of which contain a different logic description.

Figure 3-1 shows that the source file module (or complete source file in this case) is made up of several elements. These elements are illustrated in listing 3-1, a source file for an address decoder that is to be placed in a P14L4.

The elements of the ABEL source file are:

MODULE STATEMENT- This names the module and also indicates whether any dummy arguments are used. Refer to section 8.2 for additional details.

FLAG- This can be used to control processing of the source file by the language processor, and is optional. Refer to section 8.3 for additional details.

TITLE- The title is optional and is used to describe the module. The text entered will be used as a header for the ABEL output files (see section 8.4 for additional details).

DEVICE DECLARATIONS- The device declaration associates one or more device identifiers with specific programmable logic devices. Refer to section 8.5.1 for additional details.

PIN DECLARATIONS- This defines the pins of the programmable logic device with identifiers used in the source file. Refer to section 8.5.2 for additional details.

NODE DECLARATIONS- If nodes exist on the programmable logic device, they are defined here. Refer section 8.5.4 for additional details.

CONSTANT DECLARATIONS- Refer to section 8.5.4 for details.

MACRO DEFINITIONS- Refer to section 8.5.5 for details.

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

EQUATIONS- Contains the Boolean equations required to describe the logic design. (You can also express the design by means of truth tables or state diagram.) Refer to section 8.6 for details on the Equations section.

TRUTH TABLE- Contains the truth tables required to describe the logic design. (You can also express the design by means of Boolean equations or state diagrams.) Refer to section 8.7 for details on the Truth Table section of the source file.

STATE DIAGRAM- Contains the state diagrams required to describe the logic design. (You can also express the design by means of Boolean equations or truth tables.) Refer to section 8.8 for details on the State Diagram section of the source file.

TEST VECTORS- Contains the optional test vectors that you write to verify the functionality of the logic design. Refer to section 8.9 for additional details. Test vectors can be used in conjunction with PLDtest, which functionally tests the programmed device.

END STATEMENT- Ends the module.

```

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

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

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

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

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

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

[.,.] -> [.,. ,.,. ,.,. ,.,. ,.];
[.,.] -> [.,. ,.,. ,.,. ,.,. ,.];
[.,.] -> [.,. ,.,. ,.,. ,.,. ,.];

end      ....;

```

Figure 3-1. Source File Template

```
module m6809a
title '6809 memory decode
Jean Designer    FutureNet    Redmond WA    24 Feb 1987'

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

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

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

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

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

    !ROM1    = (Address >= ^hF800);

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

end m6809a
```

Listing 3-1. Source File Describing an Address Decoder

3.2 Examining an Example ABEL Source File

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

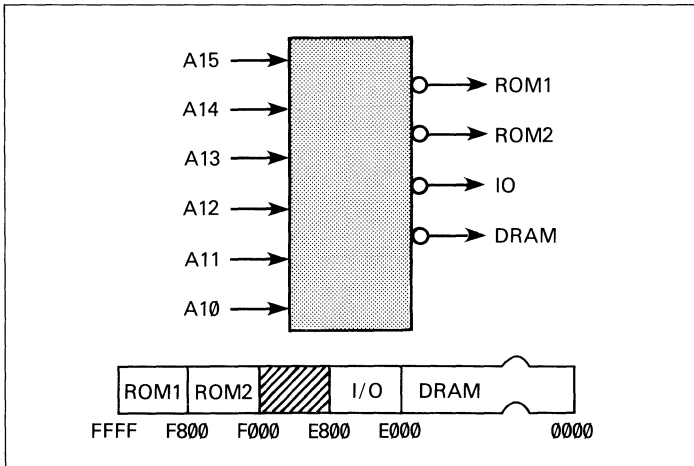


Figure 3-2. Block Diagram; 6809 Memory Address Decoder

To familiarize you with the ABEL source file, the following paragraphs provide an examination of the source file shown in listing 3-1, which describes the address decoder shown in the block diagram, Figure 3-2. This source file is typical of that used with the ABEL language processor to produce a programmer load file for downloading to a logic programmer.

The source file shown in listing 3-1 is typical of that used with the language processor and does not illustrate all the features of ABEL; nor does it show examples of source file syntax rules. For complete source file details, refer to the Language Reference and Applications Guide.

3.2.1 Purpose of the Address Decoder

An address decoder is a logic design problem that is easily solved with programmable logic. In the address decoder of Figure 3-2 the high-order six lines of an address bus are decoded into one of four active-low outputs. The ROM1 output is low for addresses in the range of F800 to FFFF; the ROM2 output is low for addresses in the range of F000 to F7FF; the I/O output is low for addresses in the range of E000 to EFFF; and the DRAM output is low for addresses in the range of 0000 to DFFF. The outputs will be applied to read-only memory, I/O ports, and dynamic RAM within the target environment to enable these functions when the applicable address appears on the address bus.

3.2.2 The MODULE Statement

The MODULE statement is a required element of the ABEL source file. It defines the beginning of the module (since a source file may contain several modules) and must be paired with an END statement. In listing 3-1, the MODULE statement is module m6809a. The keyword is module. The identifier m6809a distinguishes the module from any others in the source file. Refer to section 8.2 for a complete description of the MODULE statement.

3.2.3 The FLAG Statement

The address decoder example in listing 3-1 does not use the FLAG statement. The source file template in Figure 3-1 shows the location of the FLAG statement when used. Refer to section 8.3 for information on the FLAG statement usage.

3.2.4 The TITLE Statement

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

The TITLE statement consists of the keyword title followed by a string, which is the desired title for the module. The string is opened and closed by an apostrophe. In listing 3-1, the title is:

```
'6809 memory decode  
Jean Designer FutureNet Redmond WA 24 Feb 1987'
```

Refer to section 8.4 for details on the TITLE statement.

3.2.5 The DEVICE Declaration

The DEVICE declaration is used to associate the name of the device with the type of programmable logic device into which the logic design will be programmed. The name of the device in listing 3-1 is *U09*, which is the schematic reference in this case. The type of programmable logic device is specified as *'P14L4'*. The device name and device type are located on the same line of the source file but separated by device, the keyword of the DEVICE declaration. Refer also to section 8.5.1.

3.2.6 PIN and NODE Declarations

In listing 3-1, the two lines immediately following the **DEVICE** declaration are the **PIN** declarations. Two lines are used since there is insufficient horizontal space on an 80-column display to arrange the declaration on a single line. You can use as many lines as necessary to make the pin declarations.

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

Attributes can also be assigned to pins, although none are required in this memory decoder example. Refer to section 8.5.2 for complete **PIN** declaration details.

There are no **NODES** to be declared in this example since the P14L4 logic device contains no internal signals that are not accessible at the external pins. Refer to section 8.5.3 for information on device nodes.

3.2.7 CONSTANT Declarations

A constant, described fully in section 8.5.4, is an identifier that retains a constant value through a module. **CONSTANT** declarations are placed with the **PIN** and **NODE** declarations in the source file and use the = sign for their keyword. In listing 3-1, the declaration *H,L,X = 1,0,.X.*; tells the language processor to substitute a logic 1 whenever an upper-case H is encountered, a logic 0 whenever an upper-case L is encountered, and a "don't care" value whenever an upper-case X is encountered. The H, L, and X identifiers are used in subsequent test

vectors. (.X. is a special constant used to denote a don't care condition. Refer to section 7.7.)

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

3.2.8 EQUATIONS Statements

The EQUATIONS statement defines the beginning of a group of equations that specify the logic functions of a device. The actual equations, written in high-level Boolean equations, follow the EQUATIONS statement. In listing 3-1, there are four equations, one for each of the address decoder outputs. These four equations describe the logic function of the address decoder shown in Figure 3-2. Identifiers, defined in the declaration statements, are used in place of P14L4 pin numbers.

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

3.2.9 Test Vectors

Test vectors specify the expected functional operation of a logic device by defining its outputs as a function of its inputs. (Refer to section 8.9 for more detailed information on test vectors.) Listing 3-1 shows a typical set of test vectors and how they are arranged in a table. The form of the test vector table is determined by the header. In listing 3-1, the test vector header is

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

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

Sufficient test vectors should be included to enable the simulator within ABEL to test the intended function of the design. The test vectors provided in listing 3-1 enable testing of the design at several, but not all possible, addresses. The test vectors are sufficient to verify the operation of each output at its lowest address, except for the pin 17 (DRAM) output which is tested at four different addresses.

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

3.3 Processing an ABEL Source File

Once ABEL is installed on your system as described in the accompanying System-Specific Information, you can process a source file to create a programmer load file and, if you wish, download the load file to your logic programmer. This section describes how to process a source file and explains the different processes ABEL executes in order to achieve the load file and accompanying documentation. The following procedures use the address decoder source file previously described and shown in listing 3-1.

3.3.1 Entering the Command Line

The ABEL command line consists of

ABEL filename [parameters]

ABEL (in the command line) is a batch file (or command script) that contains commands to run all six steps of the language processor automatically (although each step can be executed individually on specified files). The filename (without the extension) specifies the input file, which is the source file to be processed into a programmer load file. The allowable parameters, plus a detailed description of the ABEL batch processing, are presented in section 4 of this manual.

The source file for the address decoder is m6809a.abl and is provided on your Design Examples disk or tape supplied as part of the ABEL package. Since this file is provided for you, it is not necessary for you to create this source file. It is necessary however, for you to copy this file, M6809A.ABL, from the examples directory to the current directory in preparation for processing by ABEL.

To enter the command line to process the address decoder example, type (.abl file extension not required):

```
abel m6809a
```

If you followed the above procedure, you have started the language processor. Listing 3-2 shows the processing statistics for each step of the language processor's operation. Processing times vary between systems.

```
echo off
+abel m6809a

PARSE ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a
PARSE complete. Time: 3 seconds

TRANSFOR ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a
..
TRANSFOR complete. Time: 2 seconds

REDUCE ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a
_device U09a
REDUCE complete. Time: 3 seconds

FUSEMAP ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a 'P14L4
_device U09a
_6 of 16 terms used
PLDMAP complete. Time: 3 seconds

SIMULATE ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a 'P14L4
.....
8 of 8 vectors in U09a passed
SIMULATE complete. Time: 2 seconds

DOCUMENT ABEL(tm) Version 3.00 Copyright(C) 1983-1987 FutureNet
module m6809a
_device U09
DOCUMENT complete. Time 2 seconds
```

Listing 3-2. Messages Displayed During Processing

3.3.2 New Files Created

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

filename.ext	Description
M6809A.LST	Listing from PARSE, used to check syntax
U09.JED	Programmer load file for design transfer and input to PLDtest
M6809A.SIM	Simulation output file for error checking
M6809A.DOC	Documentation file for design
M6809A.OUT	Intermediate file for SIMULATE, DOCUMENT, and PLDtest.

The name, U09, of the programmer load file is taken from the device identifier in the source file. You may want to look at the listing, plus the simulation (.SIM) and documentation (.DOC) files to see what the language processor creates. Each of these files is discussed fully in section 4.

3.4 Downloading the Programmer Load File

Downloading the programmer load file differs for each system used to run ABEL, and also for the model of logic programmer used. In general, 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. For MS-DOS systems, PROMlink provides a convenient way to download programmer load files to your Data I/O programmer. Additional information on downloading programmer load files for your particular system is contained in the System-Specific Information provided with the ABEL package and also in Chapter 5 of this manual.

4. The ABEL Language Processor

The ABEL language processor converts logic descriptions to programmer load files that can be downloaded to a logic programmer. The language processor checks your logic description, performs logic reduction, simulates the operation of the programmed device, and creates design documentation. Processing an ABEL source file is a six-step process:

1. **PARSE:** Reads the source file, checks for correct syntax, expands macros, and acts on directives.
2. **TRANSFOR:** Converts the description to an intermediate form.
3. **REDUCE:** Performs logic reduction.
4. **FUSEMAP:** Creates the programmer load file.
5. **SIMULATE:** Simulates the function of a design.
6. **DOCUMENT:** Creates design documentation.

In chapter 3, you saw how the ABEL command can be used to process a source file. The ABEL command is simply a batch file or command script that invokes all six steps of the language processor. You can either run the batch file to process a design, or you can run each step of the language processor separately. The following subsections discuss both of these choices. Section 4.1 describes the batch file in detail. Sections 4.2 through 4.7 describe the individual steps of the language processor, their options, and how to invoke them. Figure 4-1 shows the processing flow of the language processor.

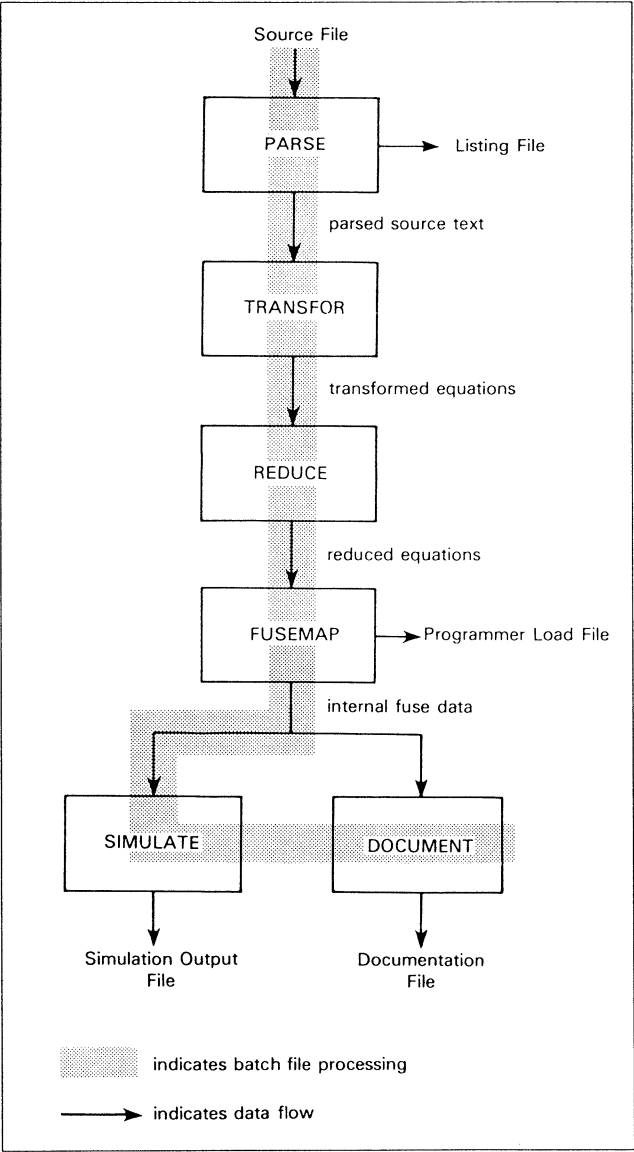


Figure 4-1. Processing Flow of the Language Processor

4.1 ABEL Batch Processing

ABEL filename [parameters]

- | | |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| filename | the name of an ABEL source file. The file extension must be ".ABL" but is not entered. |
| parameters | any of the parameters associated with the individual steps of the language processor (except -I, -O, and -B), as described in sections 4-2 through 4-7. Separate parameters with spaces. |

ABEL is a batch file that contains the commands needed to run all six steps of the language processor automatically. The input file that you specify with filename must be a valid source file. Any devices specified within the source file must have the corresponding device specification file in your library of device specification files. Refer to section 4.1.1, ABEL Libraries.

Any of the parameters associated with the six steps of the language processor, except for -I, -O, and -B, can be entered on the command line to control how a source file is processed. The parameters are discussed in sections 4-2 through 4-7.

If you do not specify any parameters, default values take effect and the following actions are performed:

- A parsed listing file is created. Text included with the `@INCLUDE` directive is shown. Macros, if used, are shown expanded.
- Level 1 logic reduction is performed.
- Programmer load files are created according to the JEDEC standard.
- Level 0 simulation (that shows only simulation errors) is performed.
- Design documentation is created.
- Processing status, statistics and error messages are displayed on the screen.

When any of the ABEL programs are run individually (without being invoked by the ABEL.BAT batch file), using of `-i` or `-o` without a file name will cause the ABEL passes to suppress prompting for input and output file names. This is useful for 'piping' program inputs and outputs.

The SIMULATE program assumes a default file extension of `.OUT` for the infile file name.

If the JEDEC file contains the name of the device, the `-n` flag need not be used with either SIMULATE or IFLDOC. The device name will be included as the default at the device prompt. To accept the default device, press return and processing will continue.

4.1.1 ABEL Libraries

Device specification files are provided in a single library file (abel3lib.dev) that contains specifications on all supported devices. You can obtain individual device (*.DEV) files from the library file as described on page 6-4. By extracting individual device files from the library, you can load only those files you need instead of loading the entire library file each time you use ABEL.

When ABEL is invoked, it will attempt to find a *.DEV device file that corresponds to each device referenced in the source file. However, if you have included the device library abel3lib.dev in the abel3dev directory, ABEL will look for the device specifications in the abel3lib.dev library if no appropriate *.DEV file is found.

The device library file should be located in the "abel3dev" directory on your system. Refer to the System Specific Information provided with your ABEL package for information on installing the device library file.

ABEL "include" files may also be contained in a library. When ABEL requires an "include" file (a file that has been referenced in a @INCLUDE or LIBRARY statement), it will attempt to find the required file. If the file is not found, ABEL will look for the library "abel3lib.inc" and try to find the file within it.

Refer to section 5.3 for information on libraries and for information on the library manager utility (used to add to and delete files from "abel3lib.dev" and "abel3lib.inc").

4.1.2 Generated Output Files

Five types of output files are created during execution of ABEL:

filename.LST	parser listing file
device.JED	programmer load files (JEDEC format)
filename.SIM	simulation output
filename.DOC	design documentation
filename.OUT	intermediate file used by SIMULATE and DOCUMENT

CAUTION

During processing, ABEL creates several temporary files. File with the same filename as the input source file and extensions .TM1, .TM2, and .TM3 are generated; and files named FUSEIN.TMP and FUSEOUT.TMP may be generated. Any files with a filename and extension identical to those of a temporary file will be overwritten.

Parser Listing File.

Contains the source code, error messages, and the effect of @INCLUDE directives. (See chapters 8 and 9 for information on macros and directives.) A complete description of the parser listing file is given in section 4.2.

Programmer Load File.

Contains the fuse states, test vectors and other design information as defined by the JEDEC standard. This file is loaded into a logic programmer to program and test a programmable logic device. (The programmer load file may also be applied to PLDtest.) One programmer load file is created for each device specified in the source file. Complete specification of the JEDEC standard format is given in appendix B.

Simulation Output File.

Output from the simulation step indicating whether simulation was completed successfully, and if not, where predicted and actual results differed. See section 4.6 for more information.

Design Documentation File.

Contains a chip diagram, reduced logic equations and other design information. This file is created by the DOCUMENT step of the language processor.

Intermediate File.

The language processor creates a file named filename.OUT that contains intermediate output data created by FUSEMAP. This intermediate file can be used as input to SIMULATE and DOCUMENT (see sections 4.6 and 4.7).

4.1.3 Creating Your Own Batch or Command Files

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

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

4.2 Parse

PARSE [-Iin_file] [-Oout_file] [-Llistfile] [-E] [-P]
[-Aarg]... [-Hpath] [-Ypath]

- Iin_file ABEL source file is in in_file
- Oout_file intermediate file output from the parser is
 written to out_file. The intermediate file is
 used as input to TRANSFOR.
- Llistfile write the listing file to the file specified by
 listfile (standard filename.ext format)
- E display expanded output resulting from macro
 expansions and directives as part of the parser
 listing
- P display expanded output resulting from macro
 expansions and directives and show the
 directive that added code to the source as part
 of the parser listing
- Aarg the argument following -A is passed to the
 PARSE program to be substituted for dummy
 arguments in the source file
- Hpath files included in the source with the
 @INCLUDE directive are located in the
 directory specified by path
- Ypath any device specification files needed are
 found in the directory specified by path

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

All parameters are optional. If the -I and/or -O parameters are omitted (no file specified), you will be prompted for the input and/or output file names.

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

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

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

Note that the command can be entered in lower-case characters and that the parameters can be specified in any order. Each parameter is discussed in detail below.

-I : Specify Input File

Use -I to specify the file containing your source file. If no input file is specified, the input is assumed to come from your standard input device (usually the keyboard).

-O : Specify Output File

Use -O to specify the name of the output file. If no output file is specified, output will go to your standard output device (usually the monitor). The output file contains the parsed source code and is used as input to the TRANSFOR processor (section 4.3).

-L : Create a Listing File

-L indicates that a listing file is to be created. The file containing the listing is specified directly after the -L parameter. A listing file contains the parsed source code with error messages (if there are any) and with macro expansions and inclusion of code due to directives if the -E parameter was specified. If the -P parameter is specified, the listing also shows the directive that caused the inclusion of code. If -L is not used, no listing file will be created.

-E : Write Expanded Output

The -E parameter causes the parsed and expanded source code to be written to the listing file. Text included by macros and directives is shown. If -E is not specified, the listing contains the source file as it was before processing plus error messages: expanded text is not shown. If no listing file is specified (the -L parameter is omitted), expanded output will be displayed at your terminal.

-P : Display Expanded Output

In addition to the listing information provided by -E, -P lists the directives that caused code to be added to the source. If the -L parameter is not included to produce a listing file, the directive and the expanded output will be displayed at your terminal.

-A : Pass Arguments to Source

The -A parameter lets you pass arguments to a source file. These arguments are substituted for dummy arguments in the source. As many arguments as are needed can be specified, but each argument must be preceded by the -A parameter. Argument substitution is discussed further in chapter 7.

-H : Specify Path for Included Files

The @INCLUDE directive, described in chapter 9, lets you include source text from one file in another source file. By default, the included file is assumed to be in the default directory; -H allows you to override that default value and explicitly specify where included files are to be found. Specify a drive and path immediately after -H. For example, the following specifies that include files are to be found in the directory named examples:

-H/examples

See your operating system manual for more information about path and drive specifications.

-Y : Specify Path for Device Specification Files

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

For example,

```
-Y/devices
```

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

4.2.1 PARSE Listing File

Listing 4-1 shows a listing file created by the PARSE step of the language processor. This listing was created by running PARSE on an altered version of the M6809A.ABL source file described in section 3. Syntax errors were introduced into the M6809A.ABL source file to create the file M6809ERR.ABL, also provided with the design examples. M6809ERR.ABL contains two syntax errors.

The first error is a missing semicolon in the equation:

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

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

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

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

```

0001 | module m6809err
0002 | title '6809 memory decode
0003 | Jean Designer      FutureNet      Redmond WA   24 Feb 1987'
0004 |
0005 |             U09      device 'P14L4';
0006 | A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007 | ROM1,IO,ROM2,DRAM    pin 14,15,16,17;
0008 |
0009 | H,L,X  = 1,0,.X.;
0010 | Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011 |
0012 | equations
0013 |         !DRAM  = (Address <= ^hDFFF)
0014 |
0015 |         !IO    = (Address >= ^hE000) & (Address <= ^hE7FF);
                    ^
? Syntax error: ';' expected

0016 |
0017 |         !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
0018 |
0019 |         !ROM1   = (Address >= ^hF800);
0020 |
0021 | test_vectors Address -> [ROM1,ROM2,IO,DRAM] )
                    ^
? Syntax error: '(' expected

0022 |         ^h0000 -> [ H,  H,  H,  L ];
0023 |         ^h4000 -> [ H,  H,  H,  L ];
0024 |         ^h8000 -> [ H,  H,  H,  L ];
0025 |         ^hC000 -> [ H,  H,  H,  L ];
0026 |         ^hE000 -> [ H,  H,  L,  H ];
0027 |         ^hE800 -> [ H,  H,  H,  H ];
0028 |         ^hF000 -> [ H,  L,  H,  H ];
0029 |         ^hF800 -> [ L,  H,  H,  H ];
0030 | end m6809err

```

Listing 4-1. PARSE Listing File with Errors from M6809ERR.ABL

Listing 4-2 shows the corrected source file with the proper semicolon and left parenthesis added.

```

0001 module m6809a
0002 title '6809 memory decode
0003 Jean Designer      FutureNet      Redmond WA    24 Feb 1987'
0004
0005             U09a    device 'P14L4';
0006     A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007     ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008
0009     H,L,X    = 1,0,.X.;
0010     Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011
0012 equations
0013     !DRAM    = (Address <= ^hDFFF);
0014
0015     !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
0016
0017     !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);
0018
0019     !ROM1    = (Address >= ^hF800);
0020
0021 test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
0022     ^h0000 -> [ H, H, H, L ];
0023     ^h4000 -> [ H, H, H, L ];
0024     ^h8000 -> [ H, H, H, L ];
0025     ^hC000 -> [ H, H, H, L ];
0026     ^hE000 -> [ H, H, L, H ];
0027     ^hE800 -> [ H, H, H, H ];
0028     ^hF000 -> [ H, L, H, H ];
0029     ^hF800 -> [ L, H, H, H ];
0030 end m6809a

```

Listing 4-2. Corrected Source File, M6809A.ABL

4.3 TRANSFOR

TRANSFOR [-Iin_file] [-Oout_file]

-Iin_file in_file is an intermediate output file created by PARSE.

-Oout_file output from TRANSFOR is written to out_file

TRANSFOR reads the equations generated by PARSE and manipulates them to:

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

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

The transformed equations can then be reduced by the REDUCE program. The file specifications given for `in_file` and `out_file` must be valid file specifications per your operating system. (Note that you can assign any legal file extension to the input and output file names.)

Example:

```
TRANSFOR -im6809a -om6809a.tm2
```

This command transforms the parsed source code contained in `m6809a.tm1` (created by PARSE) and creates an output file named `m6809a.tm2`. (If you do not assign a file extension to the output file name, TRANSFOR will assign "tm2" as the file extension.)

4.4 REDUCE

REDUCE [-In_file] [-Oout_file] [-Rn]

- In_file** **in_file** is an intermediate file created by TRANSFOR.
- Oout_file** output from REDUCE is written to out_file
- Rn** **n** = 0, 1, 2, 3, or 4, indicating the level of reduction to be performed (default = 1):
- n=0** : no reduction
 - n=1** : simple reduction
 - n=2** : simple and PRESTO reduction
 - n=3** : simple and PRESTO reduction by pin
 - n=4** : simple and ESPRESSO reduction

REDUCE reduces your logic description so that fewer product terms are used in the programmable logic device. REDUCE reduces the Boolean equations in in_file and writes the reduced equations to the file specified by out_file. The equations in in_file must be in the form produced by TRANSFOR. If in_file is not specified, the input is assumed to come from your standard input device (usually your keyboard). If out_file is not specified the output is written to your standard output device (usually your monitor).

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

When using exclusive-OR parts, such as P20X8, P20X4, etc., and exclusive-OR conditions are encountered in the input file to REDUCE, REDUCE will leave the exclusive-

OR operator intact for that output. If more than one exclusive -OR condition is encountered for one output, only the first will utilize the exclusive-OR operator; others will be converted to multiple ANDs and ORs.

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

4.4.1 Level 0 Reduction.

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

4.4.2 Level 1 Reduction.

If you choose level 1 reduction (-R1), the logic equations are expanded to a sum of products form and then reduced according to the following basic rules of Boolean algebra:

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

Level 1 reduction is the default if no level is explicitly chosen with the -R parameter.

4.4.3 Level 2 Reduction.

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

The following command line performs simple and PRESTO reduction on the transformed equations contained in m6809a.tm2 (created by TRANSFOR) and writes the reduced equations to the file named m6809a.tm3. (Note that you can assign any legal file extension to the input and output file names.)

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

4.4.4 Level 3 Reduction.

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

As mentioned, reduce levels 2 and 3 utilize the PRESTO method. PRESTO is a logic reduction technique that reduces the number of inputs and product terms. In the PRESTO method, two functions are formed. The functions are designated F and FDC and are tabular representations of the original equations. During the operation of ABEL, F and FDC are identical at the beginning of term minimization.

PRESTO tries to eliminate each input literal for each product term by temporarily omitting it from F and checking to see if the result is still covered by FDC. If covered, the literal term is deleted from F. After all input literals have been processed, all output literals are similarly processed, only the literal is temporarily omitted from FDC and then checked for coverage by F. If all output literals are eliminated from a given term, the term is removed. The process is repeated until F stabilizes and represents the minimized function.

4.4.5 Level 4 Reduction.

Level 4 reduction uses the ESPRESSO reduction method. Espresso is a heuristic reduction algorithm developed at the University of California, Berkeley. ESPRESSO's main advantage over PRESTO is its speed, which makes it the best choice for large designs. In some cases, it will also produce better reduction than PRESTO.

4.5 FUSEMAP

FUSEMAP [-Iin_file] [-Oout_file] [-Jpath] [-Cn]
[-Dn] [-Kq]

- Iin_file** intermediate file created by REDUCE.
- Oout_file** intermediate output from FUSEMAP is written to out_file
- Jpath** drive and path specification for programmer load file output
- Cn** checksum parameter. n can be:
- 0 omit STX, ETX, and transmission checksum from the programmer load file
 - 1 include STX, ETX, and a dummy transmission checksum in the load file
 - 2 include full STX, ETX, and transmission checksum (default)
- Dn** programmer load file format specification, where n can be either 0, indicating a standard JEDEC format programmer load file, or any of the microprocessor formats defined in table 4-1. JEDEC format is the default.
- Kq** unused fuses in an FPLA OR array will be disconnected or left connected according to the value of q
- q = Y leave unused fuses connected (default)
- q = N disconnect unused fuses

FUSEMAP processes the output of REDUCE and creates:

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

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

The filename of each load file is the name of the device for which the file contains a fuse map. The file extension is "JED" for JEDEC format programmer load files and ".Pxx" for all other load file formats, where xx corresponds to the number specified with the -D parameter. The load file is written to the path specified by Jpath:, or to the default path if no drive is specified.

Example:

```
FUSEMAP -im6809a -om6809a.out
```

This command invokes FUSEMAP. The file m6809a.tm3 (output of REDUCE) is read, one programmer load file (U09.JED) is created for device U09 in the source file (see listing 4-2 for naming of the programmable logic device), and the output file m6809a.out is created.

If multiple devices are named in a source file, then multiple programmer load files will be created by FUSEMAP. For example, if a source file has three devices named DEVICE1, DEVICE2, and DEVICE3, FUSEMAP would create the programmer load files:

```
DEVICE1.JED  
DEVICE2.JED  
DEVICE3.JED
```

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

-J : Specify drive and path for programmer load file output

The -J parameter allows you to indicate where the programmer load file is written. Specify the desired drive and/or path after -J.

-C : Checksum parameter

Normally, STX is placed at the beginning of the programmer load file and ETX and a checksum are placed at the end of the file in accordance with the JEDEC standard. The -C parameter allows you to control whether and how STX, ETX, and the checksum are written. -C0 omits the STX, ETX, and checksum from the programmer load file. -C1 causes STX and ETX to be written to the programmer load file as usual, but writes a dummy checksum, 0000, thereby disabling checksum checking. -C2 is the default and causes the full STX, ETX, and valid checksum to be written.

-D : Specify Programmer Load File Format

The -D parameter specifies the format of the programmer load file. -D0 is the default and indicates that the programmer load file is to be in the JEDEC format. Other formats can be specified by following -D with the appropriate microprocessor format code (-d83, for example). Supported microprocessor formats, their codes, and the file extension given to the programmer load files are given in table 4-1.

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

-K : Unused Fuses parameter

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

The -K parameter lets you specify what to do with unused fuses. -Ky leaves the fuses connected. -Kn disconnects unused fuses. If -K is not specified, the unused fuses are left connected.

Table 4-1. Data Translation Format Codes and File Extensions

Format	Code	Extension
Motorola Exorciser	82	.P82
Intel Intellec 8/MDS	83	.P83
Motorola Exormax	87	.P87
Intel MCS-86 Hexadecimal Object	88	.P88

4.6 SIMULATE

SIMULATE [-Iin_file] [-Oout_file] [-Tn] [-Ndevice]
 [-Bn1,n2[,n3]] [-Xn] [-Zn] [-Wn,n..n] [-Un]

- Iin_file infile is either an intermediate output file
 created by FUSEMAP or a programmer load
 file in JEDEC format

- Oout_file simulation output file

- Tn n is the simulation trace level, where the trace
 levels are:
 0 : show errors only
 1 : show output and test vectors
 2 : show output and test vectors, all steps
 3 : show complete device internals
 4 : show waveform on specified pins
 5 : show logic levels on specified pins
 default is trace level 0, errors always shown.

- Ndevice industry part number for device; for
 simulation performed independently of ABEL.

- Bn1,n2[,n3] set breakpoints n1 and n2 between which a
 new trace level n3 is in effect. n1 and n2 are
 inclusive.

- Xn set the logical value used for ".X." in load file
 test vectors. n can be 1,0,H,L. Default is 0.

- Zn set the logical value used for ".Z." in load file
 test vectors. n can be 1,0,H,L. Default is 1.

- Wn,n,n..n specify which device pins to "watch" during
 simulation with trace levels 3, 4, and 5.

-Un specify whether registers in a registered device are to be set to 0 or 1 at power on.

If the **-I** and/or **-O** parameters are omitted (no file specified), <filename>.OUT is the default input file and you will be prompted for the output file names. Also, if the device part number (**-Ndevice**) is not specified, you will be prompted. Refer also to section "Input Files" in this section.

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

-T : Set Trace Level

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

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

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

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

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

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

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

-N : Specify a device type to be used for simulation

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

Example:

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

The above example invokes the simulator with the file named U09.jed as input, m6809a.sim as the output file, and P14L4.DEV as the device specification file. If P14L4.DEV

cannot be found, an error message is issued.

NOTE: -N and JEDEC files cannot be used to simulate PROMs.

-Bn1,n2[,n3] : Set breakpoints

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

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

The new trace level can be specified explicitly with n3. If the new trace level is not specified, the trace level between breakpoints will be one level higher than before the break occurred.

Example:

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

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

Example:

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

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

-X and -Z : Set values for "don't care" and "high impedance"

Don't care and high impedance values encountered in test vectors must be given some value during simulation. The -X and -Z parameters allow you to override the default values. As a default, anytime a ".X." is encountered in a test vector the logical value L is substituted for it. As a default, H 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."s and ".Z."s 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 of the input during actual operations. For complete simulation, it is recommended that you run the SIMULATE operation with the don't-cares set to 0 (flag-XO), and then again with them set to 1 (flag-X1). Refer also to Don't Cares in Simulation located in chapter 12.

-Un : Set Register Power-Up State

The power-up state of all registers in a device can be set by means of the -U flag. -Uh sets all registers to 1, while -Ul sets all registers to 0. If no -U parameter is specified, registers are set to the default state specified in the device file.

-Wn,n..n,....: Specify which device pins to "watch" during simulation with trace levels 3, 4, and 5

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

Example:

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

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

```
-w6..1,14..17
```

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

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

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

4.6.1 Input Files

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

SIMULATE does not execute Boolean 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. If a programmer load file is used as the input to **SIMULATE**, part number information must be provided by using the **-N** parameter. If the output file from **FUSEMAP** (specified with **-O** at invocation of **FUSEMAP**) is used for simulation, the part number information is already available and **-N** is not needed.

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

4.6.2 SIMULATE Program Operation

Figure 4-2 shows a flow diagram that depicts SIMULATE operation when evaluating the inputs to the output. SIMULATE gets the first test vector and performs any setup of internal registers (within the PLD) that results from the vector applied to the inputs. SIMULATE then calculates the product terms that result from the test vector, the OR outputs that result from the product terms, any macrocell outputs that result from the OR outputs, and then any feedback functions. As indicated in figure 4-2, the results of the SIMULATE calculations are written to the *.SIM file whenever trace level 2 or trace level 3 are specified.

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. This is further explained in section 4.6.11 and the asynchronous feedback circuit. SIMULATE uses an iterative method to compute the outputs. After the feedback paths have been calculated, SIMULATE checks to see if any changes have occurred with the device since the product terms were last calculated. If changes have occurred due to feedback functions, the calculations are again repeated. This iterative process continues until no changes are detected, or until 20 iterations have taken place. If 20 iterations take place, the design is determined to be unstable and an error is reported. More detailed information on simulating devices with feedback is presented in section 4.6.11.

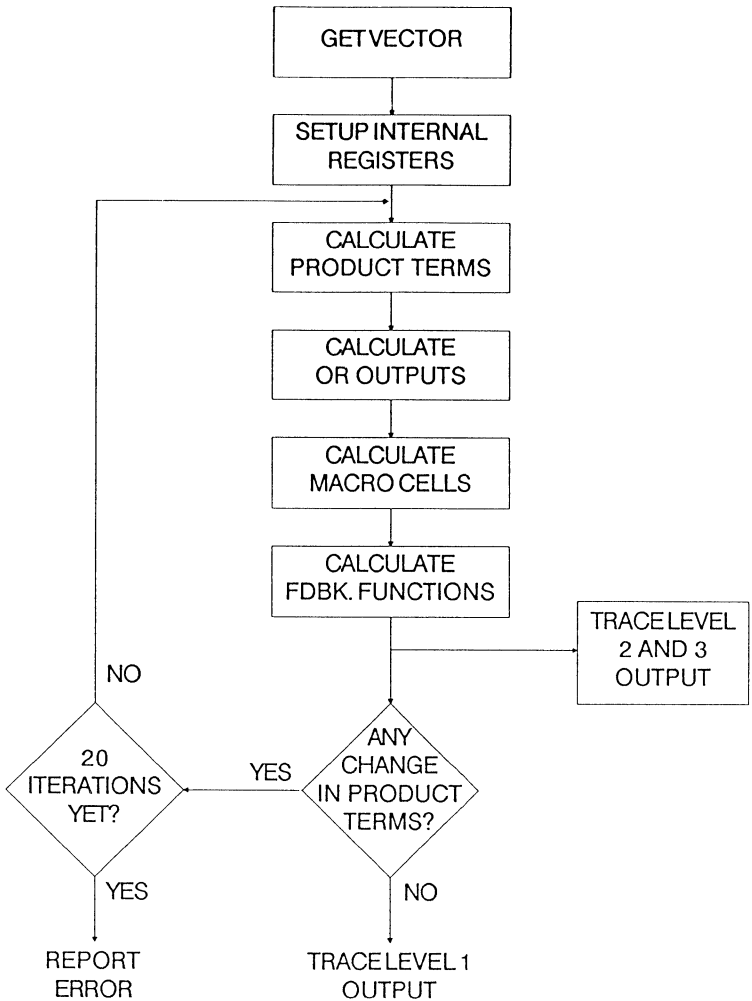


Figure 4-2. SIMULATE Processing Flow Diagram

4.6.3 Devices with Clock Inputs

Since devices with registered outputs must be clocked before the outputs reflect any change in inputs, a clock pulse must be specified as one of the inputs in the test vectors for such devices. A clock input is indicated by a C in the test vector for a low-high-low pulse, and a K for a high-low-high pulse. The clock input in the test vector causes SIMULATE to evaluate the inputs to the outputs prior to the first clock pulse transition (low-to-high or high-to-low depending on the polarity of the clock signal). The evaluation consists of the iterative steps described in section 4.6.2. The inputs to outputs are then evaluated with the clock input at its active state, and then again with the clock input at its inactive state.

When running SIMULATE with trace levels 2 or 3, simulation data will be written to the *.SIM file for all three evaluations. That is, internal test vectors are generated to evaluate the design before the first clock transition, after the first clock transition, and after the second clock transition, thus effectively expanding the number of test vectors. An example of SIMULATE output for a device with a clock input, and using a single test vector, is shown in listing 4-3. This output was generated by the command **simulate regfb -b1,1,2 -w14** which invokes SIMULATE with test vector 1, trace level 2, and output pin 14 of the source file regfb.abl.

In listing 4-3, the clock input is represented by the C on the *Vector In* line. On the four subsequent *Device In* lines, the C input goes from 0 to 1 and back to 0 to provide one complete clock pulse. (Note that the first two *Device In/Device Out* lines of the listing are identical. This is a result of SIMULATE evaluating the outputs against the inputs twice to insure that the first evaluation did not alter functions internal to the the device, and thus affect the outputs. See section 4.6.2.)

Simulate ABEL(tm) 3.xx

Operation of the simulator on devices with feedback
FutureNet 24 Feb 1987

File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'

Vector 1

Vector In [C0111.....0.....]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [10111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Vector Out [.....HH.....]

4 out of 4 vectors passed.

Listing 4-3. Clock Inputs Shown in Trace Level 2 Output

4.6.4 SIMULATE Output File, Trace Level 0

Listing 4-4 shows the output of SIMULATE created during processing of M6809A.ABL (listing 4-2), with pointers to the various parts of the output. For the purposes of this description, one of the test vectors was changed to produce an error, and simulation was run at trace level 0 (only errors are shown). Listing 4-5 shows the test vectors used to create the simulation error, in which the fourth vector has been changed. If the original test vectors shown in listing 4-2 were used, no simulation errors would occur.

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

```
Simulate  ABEL(tm) 3.xx

6809 memory decode
Jean Designer  FutureNet      Redmond WA   24 Feb 1987

File:'m6809er.out'  Module:'m6809er'  Device:'U09er'  Part:'P14L4'

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

7 out of 8 vectors passed.
```

Listing 4-4. Trace Level 0 Output for M6809A.ABL

```
test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
    ^h0000 -> [ H,  H,  H,  L ];
    ^h4000 -> [ H,  H,  H,  L ];
    ^h8000 -> [ H,  H,  H,  L ];
    ^hc000 -> [ L,  H,  H,  L ];
    ^he000 -> [ H,  H,  L,  H ];
    ^he800 -> [ H,  H,  H,  H ];
    ^hf000 -> [ H,  L,  H,  H ];
    ^hf800 -> [ L,  H,  H,  H ];
```

Listing 4-5. Test Vectors Used to Create Simulation Error

4.6.5 SIMULATE Output File, Trace Level 1

Listing 4-6 shows a trace level 1 simulation output for the same the same source file (including the incorrect vector) that produced listing 4-4. Another example of trace level 1 output is shown in listing 4-7. Listing 4-7 was created using the command **simulate add5 -b5,5,1 -w19** to produce an output for test vector 5 while "watching" output pin 19 of the F159 device.

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

Simulate ABEL(tm) 3.xx

6809 memory decode

File:'m6809a.out' Module:'m6809a' Device:'U09a' Part:'P14L4'

Vector 1

Vector In [000000.....]

Device Out [.....HHHL...]

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

Vector 2

Vector In [010000.....]

Device Out [.....HHHL...]

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

Vector 3

Vector In [100000.....]

Device Out [.....HHHL...]

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

Vector 4

Vector In [110000.....]

Device Out [.....HHHL...]

Vector Out [.....LHHL...]

Vector 4

ROM1 14, 'H' found 'L' expected

Vector 5

Vector In [111000.....]

Device Out [.....HLHH...]

Vector Out [.....HLHH...]

Vector 6

Vector In [111010.....]

Device Out [.....HHHH...]

Vector Out [.....HHHH...]

Vector 7

Vector In [111100.....]

Device Out [.....HHLH...]

Vector Out [.....HHLH...]

Vector 8

Vector In [111110.....]

Device Out [.....LHHH...]

Vector Out [.....LHHH...]

7 out of 8 vectors passed.

Listing 4-6. Level 1 Simulation Output, All Vectors

Simulate ABEL(tm) 3.xx

5-bit ripple adder

File:'add5.out' Module:'ADD5' Device:'BJ2A' Part:'F159'

Vector 5

Vector In [C00001....0.....]

Device Out [.....2LLL..HLLLLHHH.LLLLLLLL.....LLLLLLLLLHHLHLL]

Vector Out [.....LLLLH.....]

10 out of 10 vectors passed.

Listing 4-7. Level 1 Simulation Output, Single Vector

(

()

4.6.7 SIMULATE Output File, Trace Level 3

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

Fuse and node numbers shown on the table are numbers assigned by FutureNet 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.

The large size of trace level 3 simulation files is due not only to the complete listing of all device internals, but also due to expanded test vectors caused by clock inputs (see section 4.6.3), and the numerous iterations of the simulator required to stabilize the outputs of some designs (see section 4.6.11). Trace level 3 may be used with a breakpoint to specify a limited number of test vectors, and the **-W** argument to limit the number of outputs included in the simulation output file.

In figure 4-3, the product term line shows the state of 33 product terms. This is because a F159 is used in this example, and the input array has 33 possible terms. (Refer to the F159 logic diagram included with your Logic Diagram Package as part of the ABEL product for details on the F159.) The product terms line is followed by the states of various nodes, input pins, and product terms that exist for indicated states on the product terms line.

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

Table 4-2. Notation Used in Simulation Output Files

Notation	Description
Current Nodes	
OE	Output enable
AR	Asynchronous Reset
SR	Synchronous Reset
AP	Asynchronous Preset
SP	Synchronous Preset
LD	Register Load
CK	Register Clock
FC	Flip/Flop Mode control (F159 or P32VX10)
OR	Normal output OR gate
IN1	First input to a Flip/Flop ("J")
IN2	Second input to a Flip/Flop ("K")
OR Node Types	
PTnnnn	One or more product term, PAL or FPLS (nnnnn = First Fuse)
LOW	Always logic level 0
HIGH	Always logic level 1
Pin nn	Input from pin
Node nn	Input or feedback from a internal node
Pin nn & nn	The AND of two pins
Pin nn # nn	The OR of two pins
PROM nn	Bit nn of a prom output

Table 4-2. (continued)

Notation		Description
PRODUCT Term Display		
Pin nn [1]	Logic level 1 from a pin or node
Pin nn [0]	Logic level 0 from a pin or node
nn & nn [0 & 1]	Logic level 0 from a pin or node
PTnnnnn [TTFFTT]	Multiple product terms
PTnnnnn [TT \$ FT]	XOR of two groups of product terms
PTnnnnn [TT # FT]	OR of two groups of product terms
PTnnnnn [T & !1]	AND of product term and inverted pin (P20RA10)
PTnnnnn [T-FF-T]	Shared product terms ('-' term not connected)
PTnnnnn [TTTTTFTTTTFFFTTF]		Multiple line display of large OR
TTTTTTTTTF]	

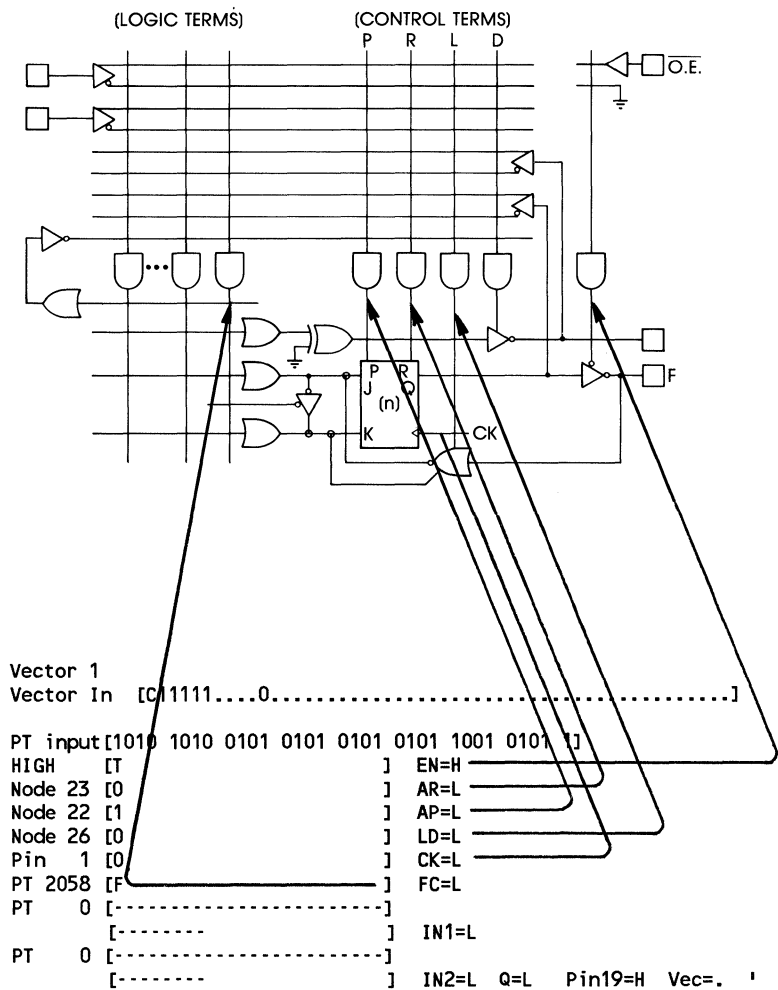
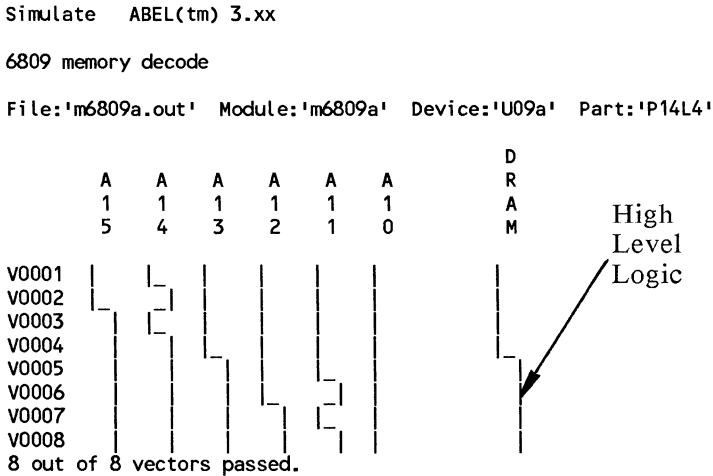


Figure 4-3. Trace Level 3 Simulation Output

4.6.8 SIMULATE Output File, Trace Level 4

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

Listing 4-9 shows the waveform generated by running the same source file used in listings 4-4 and 4-6 (address decoder, m6809a.abl). The command **simulate m6809a -t4 -w1..6,999,17** was used to generate a trace level 4 waveform of pins 1 through 6, and 17 of the address decoder. Each of the specified pins is shown with the logic high and low levels for each of the eight test vectors.



Listing 4-9. Trace Level 4 Simulation Output

Listing 4-10 shows another example of trace level 4. The waveform was generated by simulating the same source file used in listings 4-7 and 4-8 (add5.abl). The command **simulate shiftcnt -b14,30,4 -w1..5,999,12..15** was used to generate a waveform of the signals appearing at pins 1 through 5, and 12 through 15 for test vectors 14 through 30.

```
Simulate  ABEL(tm) 3.xx

universal counter / shift register

File:'shiftcnt.out' Module:'shiftcnt' Device:'IFL4' Part:'F159'

***** Shift left *****

      C      J
      l      K      I      I      I      F      F      F      F
      k      D      1      2      3      0      1      2      3

V0014 | -      |      |      |      |      |      |      |
V0015 | -      |      |      |      |      |      |      |
V0016 | -      |      |      |      |      |      |      |
V0017 | -      |      |      |      |      |      |      |
V0018 | -      |      |      |      |      |      |      |
V0019 | -      |      |      |      |      |      |      |

***** Count up *****

      C      J
      l      K      I      I      I      F      F      F      F
      k      D      1      2      3      0      1      2      3

V0020 | -      |      |      |      |      |      |      |
V0021 | -      |      |      |      |      |      |      |
V0022 | -      |      |      |      |      |      |      |
V0023 | -      |      |      |      |      |      |      |
V0024 | -      |      |      |      |      |      |      |
V0025 | -      |      |      |      |      |      |      |
V0026 | -      |      |      |      |      |      |      |
V0027 | -      |      |      |      |      |      |      |
V0028 | -      |      |      |      |      |      |      |
V0029 | -      |      |      |      |      |      |      |
V0030 | -      |      |      |      |      |      |      |
36 out of 36 vectors passed.
```

Listing 4-10. Trace Level 4 Simulation Output

4.6.9 SIMULATE Output File, Trace Level 5

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

Listing 4-11 shows the level 5 output for the same shifter/counter shown in Listing 4-10. The command used to generate this report is **simulate shiftcnt -b14,30,5 -w1..9,11,999,12..15,999,16..21**. Note that more pins can be shown with level 5 than can with level 4. Also, pins that do not have declared signal names are named simply by their pin numbers and a "P" prefix, such as for pins 6 through 9 and 16 through 19.

universal counter / shift register

***** Count Up and Shift Left *****

***** Shift left *****

***** Count up *****

Listing 4-11. Trace Level 5 Simulation Output

4.6.11 Simulation and Designs With Buffered Outputs

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

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

if the buffer is *enabled*, the active state (H or L) of the output (that results from the levels applied at the input pins by the input test vector)

or

if the buffer is *not enabled*, the same value (1, 0, Z, or X) applied to that output by the input test vector

or

Z if the output is *not enabled* and no 1 or 0 is applied to that output by the input test vector.

4.6.12 Simulation and Unspecified Inputs

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

4.6.13 Simulation for Designs With Feedback

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

SIMULATE determines the final outputs of feedback circuits through iteration, calculating and monitoring the outputs until they stabilize or are clocked out to give the final outputs. (If outputs do not stabilize after 20 iterations, an error message is given.) The iterations, final outputs and the states of the internal register are provided in the simulation output file depending on the trace level you choose to simulate under. Figure 4-4 shows a simple synchronous circuit with feedback. One clock pulse is required after the inputs change to cause a corresponding change in the outputs. The source file describing this circuit and the simulation output for trace levels 1 and 2 are shown in listings 4-13 through 4-15.

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

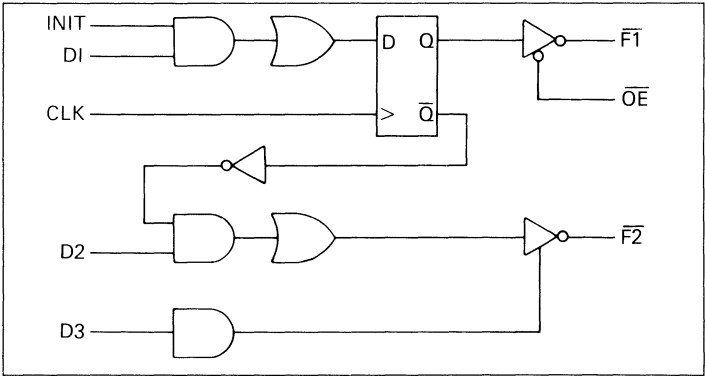


Figure 4-4. Synchronous Feedback Circuit

```
module regfb
title 'Operation of the simulator on devices with feedback
FutureNet 24 Feb 1987'

FB2 device 'P16R4';

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

equations
!F1 := D1 & INIT;
!F2 = D2 & !F1;

ENABLE F2 = D3;

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

end regfb
```

Listing 4-13. Source File: Synchronous Feedback Circuit

Simulate ABEL(tm) 3.xx

Operation of the simulator on devices with feedback
FutureNet 24 Feb 1987

File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'

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

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

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

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

4 out of 4 vectors passed.

**Listing 4-14. Simulation Output, Trace Level 1:
Synchronous Feedback Circuit**

Simulate ABEL(tm) 3.xx

Operation of the simulator on devices with feedback
FutureNet 24 Feb 1987

File:'regfb.out' Module:'regfb' Device:'FB2' Part:'P16R4'

Vector 1

Vector In [C0111.....0.....]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [10111000000011111000]

Device Out [.....ZHHHHHZZ.]

Device In [00111000000011111000]

Device Out [.....ZHHHHHZZ.]

Vector Out [.....HH.....]

-
-
-
-

Vector 4

Vector In [00000.....0.....]

Device In [00000000000000111000]

Device Out [.....ZZLHHHZZ.]

Device In [00000000000000111000]

Device Out [.....ZZLHHHZZ.]

Vector Out [.....ZL.....]

4 out of 4 vectors passed.

Listing 4-15. Simulation Output (partial), Trace Level 2: Synchronous Feedback Circuit

As a second feedback example, figure 4-5 shows an asynchronous circuit that requires more than one simulation iteration before the outputs stabilize. Listing 4-16 shows the source file describing the circuit and listings 4-17 and 4-18 shows the simulation output for trace levels 1 through 3. Trace level 0 output is not shown since there are no simulation errors in this design and level zero only reports errors.

Trace level 1 shows the final outputs after they have stabilized, and also the test vectors. Trace level 2 shows the output values at the different iterations as the outputs stabilize, as well as the final outputs and the test vectors. Notice that for the inputs provided in vector 2, three iterations are needed before the outputs stabilize. Vector 1 requires only one iteration to provide stable outputs. Trace level 3 output is not shown but can be generated by running ABEL with "feedback.abl" to generate the "feedback.sim" file shown in listing 4-12.

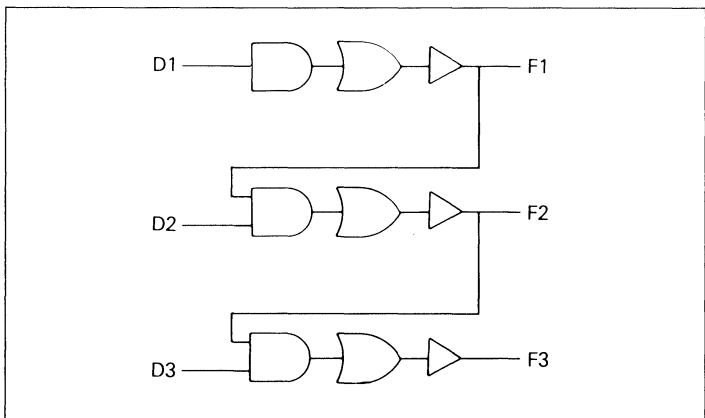


Figure 4-5. Asynchronous Feedback Circuit

```
module feedback
title
'Operation of the simulator on devices with feedback

    FB1 device 'P16HD8';

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

equations

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

test_vectors    ([D1,D2,D3] -> [F1,F2,F3])
                [ 0, 0, 0] -> [ 0, 0, 0];
                [ 1, 1, 1] -> [ 1, 1, 1];

end feedback
```

Listing 4-16. Source File: Asynchronous Feedback Circuit

```
Simulate ABEL(tm) 3.xx

Operation of the simulator on devices with feedback

File:'feedback.out' Module:'feedback' Device:'FB1' Part:'P16HD8'

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

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

2 out of 2 vectors passed.
```

Listing 4-17. Simulation Output, Trace Level 1: Asynchronous Feedback Circuit

Simulate ABEL(tm) 3.xx

Operation of the simulator on devices with feedback
FutureNet 24 Feb 1987

File:'feedback.out' Module:'feedback' Device:'FB1'
Part:'P16HD8'

Vector 1
Vector In [000.....]

Device In [00000000000000000000]
Device Out [.....LLLLLLLL.]

Device In [00000000000000000000]
Device Out [.....LLLLLLLL.]
Vector Out [.....LLL.....]

Vector 2
Vector In [111.....]

Device In [11100000000010000000]
Device Out [.....LHLLLLLL.]

Device In [11100000000011000000]
Device Out [.....LHHLLLLL.]

Device In [11100000000011100000]
Device Out [.....LHHHLLLL.]

Device In [11100000000011100000]
Device Out [.....LHHHLLLL.]
Vector Out [.....HHH.....]

2 out of 2 vectors passed.

Listing 4-18. Simulation Output, Trace Level 2: Asynchronous Feedback Circuit

4.6.14 EZSIM - A Batch File for Re-Simulation of a Design

EZSIM is a batch file that invokes PARSE, TRANSFOR, and SIMULATE only; and omits REDUCE and FUSEMAP. EZSIM allows for simulation on a design that has previously been run through ABEL and further testing is required by changing the test vectors in the source file. Since only the test vectors in the source file are being changed, there is no need to perform the REDUCE and FUSEMAP operations.

EZSIM takes the test vectors from the **.TM2* file generated by TRANSFOR and the fuse information from the **.JED* file, generated during a previous run of FUSEMAP, to run SIMULATE on the design. Performing simulation with EZSIM is faster than rerunning ABEL on the design and allows you to alter test vectors in the source file and run SIMULATE again with no time lost to REDUCE and FUSEMAP.

4.7 DOCUMENT

DOCUMENT [-Iin_file] [-Oout_file] [-V] [-Fn] [-G] [-S]
[-Qwxyz]...

- Iin_file infile is an intermediate output file from FUSEMAP
- Oout_file design documentation file
- V list the test vectors
- Fn list the fuse map and/or terms
 - n=0 list the fuse map and the device utilization information
 - n=1 list only the device utilization information
 - n=2 brief fuse map
- G list the default chip diagram
- Gn n=0 : DIP diagram
 - n=1 : PLCC package diagram
- S list the symbol table
- Qxyz select equations to be listed, where w, x, y, and z can be any combination of the following:
 - 0 : list original equations
 - 1 : list transformed equations
 - 2 : list reduced equations
 - 3 : create the output file in PALASM format

DOCUMENT creates design documentation from information provided by previous steps of the language processor. The documentation is written to the file specified by the **-O** parameter or to the standard output device in the event that no output file is specified. If the **-I** and/or **-O** parameters are omitted (no file specified), you will be prompted for the input and/or output file names.

The design documentation contains the following information for each device in the source file if the appropriate parameter is set:

Symbol table	constant, pin, node, module and macro identifiers listed alphabetically.
Reduced equations	equations produced by REDUCE .
Transformed equations	equations produced by TRANSFOR .
Equations	original equations from the source file and equations generated by PARSE from truth tables and state diagrams.
Test vectors	test vectors described as inputs and outputs, taken from the PARSE intermediate output file.
Fuse map	graphical representation of the fuse states from the programmer load file.
Chip Diagram	a diagram showing the device pinouts and the identifiers assigned to each pin.

If no parameters are supplied in the **DOCUMENT** invocation, no documentation listing will be generated.

-Q : List Equations Parameter

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

-F : List Fuse Map and Terms Utilization

The -F parameter controls the listing of fuse maps and terms. -F0 lists a complete fuse map, the number of terms used, and the utilization of each term with respect to the outputs (utilization report). -F1 lists only the number of terms used and the utilization report. -F2 or -F lists an abbreviated fuse map, where fuse rows that have all their fuses intact are not shown. -F2 eliminates showing many rows of intact fuses in devices where only a small portion of the product terms are used in the design.

Listing 4-19 shows output from DOCUMENT. This output was created by processing the source file, M6809A.ABL (listing 4-2) using the parameters -V, -F, -G, -S, and -Q2. The listing contains a symbol table, the reduced equations, a chip diagram, a fuse map, and the test vectors. In the fuse map, intact connections are shown as "X"s, and blown fuses (no connection) are shown as dashes. The test vector inputs are shown on the left side of the "->" symbol; outputs appear on the right side.

Page 1

ABEL(tm) Version 3.00 - Document Generator 18-Sep-87 12:34PM

6809 memory decode

Jean Designer FutureNet Redmond WA 24 Feb 1987

Symbol list for Module m6809a

A10	Pin 6 pos, com
A11	Pin 5 pos, com
A12	Pin 4 pos, com
A13	Pin 3 pos, com
A14	Pin 2 pos, com
A15	Pin 1 pos, com
Address	([A15,A14,A13,A12,A11,A10,.X.,.X.,.X., .X.,.X.,.X.,.X.,.X.,.X.,.X.])
DRAM	Pin 17 neg, com
H	(1)
IO	Pin 15 neg, com
L	(0)
ROM1	Pin 14 neg, com
ROM2	Pin 16 neg, com
U09a	device P14L4
X	(.X.)
m6809a	Module Name

Page 2

ABEL(tm) Version 3.00 - Document Generator 18-Sep-87 12:34PM

6809 memory decode

Jean Designer FutureNet Corp Redmond WA 24 Feb 1987

Equations for Module m6809a

Device U09a

- Reduced Equations:

DRAM = !(A13 # A14 # A15);

IO = !(A11 & A12 & A13 & A14 & A15);

ROM2 = !(A11 & A12 & A13 & A14 & A15);

ROM1 = !(A11 & A12 & A13 & A14 & A15);

Listing 4-19. Documentation Output for M6809A.ABL
(continued on next page)

Page 3

ABEL(tm) Version 3.00 - Document Generator

18-Sep-87 12:34PM

6809 memory decode

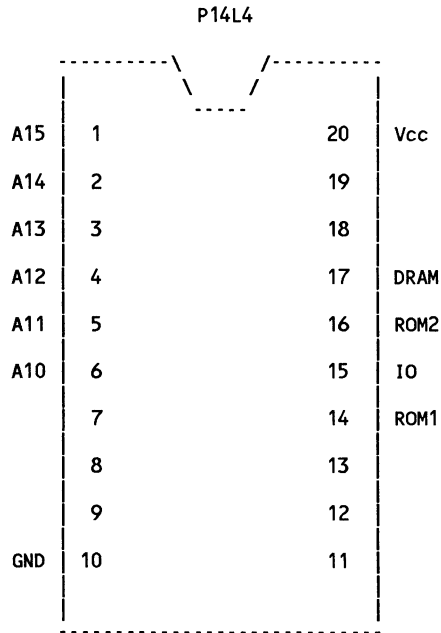
Jean Designer FutureNet

Redmond WA

24 Feb 1987

Chip diagram for Module m6809a

Device U09a



Listing 4-19. Continued.

Page 4
ABEL(tm) Version 3.00 - Document Generator 18-Sep-87 12:34PM
6809 memory decode
Jean Designer FutureNet Redmond WA 24 Feb 1987
Fuse Map for Module m6809a

Device U09a

	0	10	20
0:	-----X----	-----	-----
28:	-X-----	-----	-----
56:	---X-----	-----	-----
84:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
112:	X-X-X---X-	---X-----	-----
140:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
168:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
196:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
224:	X-X-X---X-	---X-----	-----
252:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
280:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
308:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
336:	X-X-X---X-	---X-----	-----
364:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
392:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
420:	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX

Listing 4-19. Continued.

Page 5

ABEL(tm) Version 3.00 - Document Generator

18-Sep-87 12:34PM

6809 memory decode

Jean Designer FutureNet

Redmond WA

24 Feb 1987

for Module m6809a

Device U09a

Device Type: P14L4

Terms Used: 6 out of 16

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

Listing 4-19. Continued.

Page 6

ABEL(tm) Version 3.00 - Document Generator 18-Sep-87 12:34PM

6809 memory decode

Jean Designer FutureNet Redmond WA 24 Feb 1987

Test Vectors for Module m6809a

Device U09a

```
1 [0000 00-- ---- ---- ----] -> [----- ---- ---- -HHH L---];
2 [0100 00-- ---- ---- ----] -> [----- ---- ---- -HHH L---];
3 [1000 00-- ---- ---- ----] -> [----- ---- ---- -HHH L---];
4 [1100 00-- ---- ---- ----] -> [----- ---- ---- -HHH L---];
5 [1110 00-- ---- ---- ----] -> [----- ---- ---- -HLH H---];
6 [1110 10-- ---- ---- ----] -> [----- ---- ---- -HHH H---];
7 [1111 00-- ---- ---- ----] -> [----- ---- ---- -HHL H---];
8 [1111 10-- ---- ---- ----] -> [----- ---- ---- -LHH H---];
```

end of module m6809a

Listing 4-19. (continued)

5. Transferring the Programmer Load File

When you have completed the processing of an ABEL source file, a file named filename.JED will exist in your working directory. This file is the programmer load file, and contains the data necessary for a logic programmer to program a logic device with your design. Transferring the programmer load file amounts to downloading it from your system to the logic programmer over an RS232-compatible communications link. Data I/O's PROMlink, version 2.0 or greater, can be used to transfer programmer load files from MS-DOS systems. IF PROMlink is used, refer to the PROMlink manual for details.

5.1 Downloading to a Model 29 with LogicPak

The following steps are necessary to transfer a programmer load file from an IBM PC/XT/AT (or compatible) to a Data I/O Model 29 programmer with a LogicPak installed. If you are operating with a system other than an IBM PC/XT/AT or compatible, refer to the System Specific Information provided with your ABEL package for variations in the downloading procedure.

1. Connect the cable (see Figures 5-1 and 5-2) to the COM1: port on the IBM PC and to the serial interface of the Model 29.
2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual). Parity should be set with the power off.
3. Configure the IBM PC for 4800 baud (9600 baud could also be used) with no parity by issuing the following command at the DOS prompt on the PC:

```
MODE COM1:4800,n,8
```

4. Enter the family code and pinout code of the device to be programmed into the Model 29 as follows:

```
VERIFY RAM DEVICE START XXXX
```

where XXXX is the 4-digit family and pinout code. A list of these codes may be found in the LogicPak manual or on the Data I/O Wallchart of Programmable Devices.

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

```
SELECT E B START
```

The Model 29 will display a stationary action symbol.

6. At the PC, execute the command:

`COPY d:filename.ext COM1:`

where *d* is the drive specification, and *filename* and *ext* are the filename and extension of the programmer load file. The load file extension is ".JED". The Model 29 action symbol should rotate when the file is being transferred.

7. When the transfer is complete, the Model 29 will display the fuse checksum signifying that the programmer load file is now resident in programmer memory.

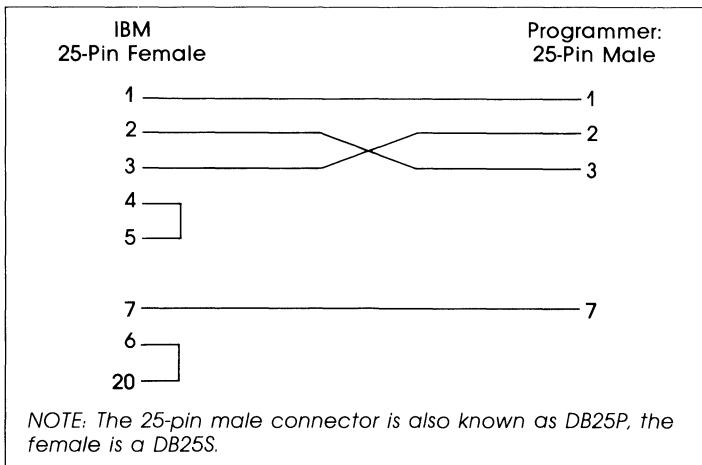


Figure 5-1. Cable Configuration for Transfer Between an IBM-XT and a Data I/O Programmer

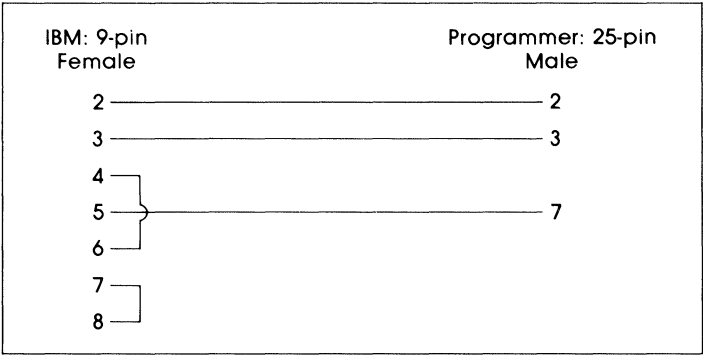


Figure 5-2. Cable Configuration for Transfer Between an IBM-AT and a Data I/O Programmer

5.2 Downloading to a UniSite™ Universal Programmer

Refer to Chapter 2 of your UniSite Operator's Manual for a complete sample procedure.

5.3 PROM Download (Model 29/UniPak2)

Unless you are using PROMlink, the following steps are necessary to transfer a programmer load file from the IBM PC to a DATA I/O Model 29 programmer with a UniPak2:

1. Connect the cable (see Figure 5-1) to the COM1: port on the IBM PC and to the serial interface of the Model 29.
2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual). Parity should be set with the power off.
3. Configure the IBM PC for 4800 baud with no parity by issuing the following command at the DOS prompt on the PC:

```
MODE COM1:4800,n,8
```

4. Select the microprocessor transfer format of the load file. A list of supported microprocessor formats may be found in in table 4-1. For example, the code for the Motorola Exorciser format is 82. To select this format enter the following at the 29:

```
SELECT 8 2 START START
```

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

COPY PORT RAM START

The Model 29 will display a stationary action symbol.

6. At the PC, execute the command:

COPY d:filename.ext COM1:

where d is the drive specification, and filename and ext are the filename and extension of the programmer load file. The load file extension for PROMs is .Pxx where xx is the format code (see table 4-1). The Model 29 action symbol should rotate when the file is being transferred.

7. When the transfer is complete, the Model 29 will display the the fuse RAM sumcheck.

6. ABEL Utilities

This section describes the utilities provided in addition to the ABEL software.

6.1 TOABEL, PALASM to ABEL Converter

TOABEL -Iinfile -Ooutfile

[-Iin_file] PALASM input file

[-Oout_file] ABEL output file

TOABEL is a conversion utility that you can use to convert PALASM (version 1.0 only) logic descriptions to ABEL logic descriptions. TOABEL creates an ABEL source file that can be processed normally with the ABEL language processor. This means that the design can be reduced, simulated (if test vectors are supplied), documented and loaded into a logic programmer just as with any other ABEL source file. Note that test vectors for devices with programmable inputs and outputs (i.e., P16L8) may require editing because TOABEL cannot determine whether a pin is an input or an output.

6.2 IFLDOC

IFLDOC [-In_file] [-Oout_file] [-Ndevice_type]

- | | |
|----------------------|-------------------------------------------------------------------|
| -In_file | JEDEC-format input is in in_file |
| -Oout_file | output in a Signetics program table format is written to out_file |
| -Ndevice_type | device_type indicates the type of device described by in_file |

IFLDOC is a conversion program that converts JEDEC-format files to listing files in a format similar to Signetic Program Logic tables. The listing file created is for documentation purposes only; it will not drive any device programmer.

JEDEC-format input contained in in_file is converted to the Signetics-like form and written to out_file. The device type described by the JEDEC file must be explicitly specified with the -N parameter. If in_file and out_file are not specified, input and output are received and sent from the standard input and output devices. A device type must be specified; there is no default value. (If the programmer load file was generated by ABEL or GATES, the IFLDOC program can extract the device type from a string in the file header.)

For devices with programmable flip-flop types, IFLDOC assumes that the flip-flops are J-K flip-flops unless the Fc line is programmed and the individual FF mode is set to A. (Refer to IFL logic diagrams for the function and placement of these lines.)

Example:

```
ifldoc -Iifl2.jed -Oifl2.tab -nf82s105
```

This command invokes IFLDOC to convert the JEDEC-format data contained in ifl2.jed to a program table format written to ifl2.tab . The -N parameter specifies that the device described by ifl2.jed is an F82s105. The output file resulting from this command is shown in Figure 6-1.

[illegible]

Figure 6-1. Sample Output File From IFLDOC

6.3 ABELLIB, Library Manager

ABELLIB [library] [command] { files }

ABELLIB is used to maintain the ABEL library files. The device library (abel3lib.dev) is a single file that contains all specifications for all devices currently supported by the ABEL software package. The "include" library (abel3lib.inc) is a single file that contains text files that may be included in an ABEL source file via the LIBRARY statement.

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

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

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

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

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

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

1. Make the directory that contains the library the current directory. For MS-DOS systems, use the command:

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

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

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

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

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

The suggested procedure to create a device library that contains only those device you will use is:

1. Make the directory that contains the device files the current directory.
2. Create a device library, such as one named *newlib*, with the following command:

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

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

3. Repeat step 2 as necessary to add the desired device files to the library. The newly created library will be contained in the current directory.

4. Since ABEL looks for "abel3lib.dev" for device specifications, you must rename any existing "abel3lib.dev" library file, then rename your new library "abel3lib.dev." For example, MS-DOS users can use the commands:

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

to ensure that ABEL accesses the newly created library.

6.4 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:

- a. The current directory
- b. The directory indicated in the ABEL3DEV environment variable
- c. Directories indicated in the PATH environment variable

6.5 JEDABEL, JEDEC File to Equations Converter

JEDABEL [-Iinfile] [-Ndevice] [-Ooutfile]

- Infile** ***.JED** file in JEDEC Standard 3A format
- Ndevice** device number (P14L4, P16L8, etc.)
- Ooutfile** output file name; you can use **.ABL** as the file extension in preparation for ABEL processing.
- Mmapfile** map output file name; the default file extension is **.MAP**.

JEDABEL allows you to convert the fuse information in a standard JEDEC file to a set of equations that reflect the design contained in the file. This utility may be used when a previously designed part is to be updated or modified and no ABEL source file exists that was used to create the part. Since the JEDEC file contains no signal name information, the equations generated by JEDABEL use the device pin numbers for names of the inputs and outputs. JEDABEL also produces a detailed report that helps you to determine the precise configuration of the programmed device. Any equations file generated by JEDABEL will require some editing in order to achieve a working ABEL source file. However, the basis for the source file is provided. Listing 6-1 shows a "source" file generated by JEDABEL from the file U09A.JED. (This file U09A.JED is created by running M6809A.ABL on ABEL.) You can compare the equations in listing 6-1 with the original equations shown back in listing 4-2. While the equations in each listing are in a different form, the designs expressed by the equations are the same. Running the source file shown in listing 6-1 through ABEL will result in a fusemap in the JEDEC file functionally equivalent to that contained in U09A.JED.

User's Guide

```
module u09a
"Created by JEDABEL Ver 1.00 on Thu Oct 01 16:31:51 1987

title
'ABEL(tm) Version 3.00 FutureNet/Data I/O Corp. JEDEC file for:
P14L4
Created on: 01-Oct-87 11:17 AM
6809 memory decode
Jean Designer FutureNet Corp Redmond WA 24 Feb 1987'

u09a device 'P14L4';

"Pin Declarations
    PIN01, PIN02, PIN03, PIN04 Pin 1, 2, 3, 4;
    PIN05, PIN06, PIN07, PIN08 Pin 5, 6, 7, 8;
    PIN09, PIN10, PIN11, PIN12 Pin 9,10,11,12;
    PIN13, PIN14, PIN15, PIN16 Pin 13,14,15,16;
    PIN17, PIN18, PIN19, PIN20 Pin 17,18,19,20;

"Node Declarations
    PIN14,PIN15,PIN16,PIN17 IsType 'Neg';
    PIN14,PIN15,PIN16,PIN17 IsType 'Com';

"Feedback nodes
    X,Z,C,P = .X., .Z., .C., .P.;

EQUATIONS

!PIN14 = ( PIN02 & PIN01 & PIN03 & PIN04 & PIN05 );
!PIN15 = ( PIN02 & PIN01 & PIN03 & !PIN04 & !PIN05 );
!PIN16 = ( PIN02 & PIN01 & PIN03 & PIN04 & !PIN05 );
!PIN17 = (!PIN03 # !PIN02 # !PIN01 );

TEST_VECTORS
([PIN01,PIN02,PIN03,PIN04,PIN05,PIN06]->[PIN14,PIN15,PIN16,PIN17])
    [0,0,0,0,0,0]->[1,1,1,0];
    [0,1,0,0,0,0]->[1,1,1,0];
    [1,0,0,0,0,0]->[1,1,1,0];
    [1,1,0,0,0,0]->[1,1,1,0];
    [1,1,1,0,0,0]->[1,0,1,1];
    [1,1,1,0,1,0]->[1,1,1,1];
    [1,1,1,1,0,0]->[1,1,0,1];
    [1,1,1,1,1,0]->[0,1,1,1];

END
```

Listing 6-1. "Source" File Generated by JEDABEL

7. Language Elements

This section describes the various elements of the ABEL design language. These language elements are combined according to the structure described in section 3 to create ABEL logic descriptions. Each element is presented on the following pages. The organization is from the most basic to the most complex topics.

7.1 Basic Syntax

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

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

4. Spaces cannot be imbedded in the middle of keywords, numbers, operators or identifiers. Spaces can appear in strings, comments, blocks and actual arguments. For example, if the keyword `MODULE` is entered as `"MOD ULE"`, it will be interpreted as two identifiers, `MOD` and `ULE`. Similarly, if you enter `"102 05"` (instead of `10205`), it will be interpreted as two numbers, `102` and `5`.
5. Keywords (words defined as part of the language and that have specific uses) can be typed in either upper-case or lower-case with no difference in effect.
6. Identifiers (user-supplied names and labels) can be typed in either upper-case, lower-case, or mixed- case, but are case-sensitive: the identifier output, typed in all lower-case letters, is not the same as the identifier Output with an upper-case `"O"`.

7.2 Valid ASCII Characters

All upper-case and lower-case alphabetic characters and most other characters on common keyboards are valid. Valid characters are listed or shown below.

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

7.3 Identifiers

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

The rules governing identifiers are:

1. Identifiers must begin with an alphabetic character or with an underscore.
2. Identifiers may be up to 31 characters long. Anything longer than 31 characters is considered an error and is flagged by the language processor.
3. Other than the first character (see rule 1), identifiers may contain upper-case and lower-case alphabetic characters, digits and underscores.
4. Spaces cannot be used in an identifier. Use underscores to provide separation between words.
5. Identifiers are case-sensitive: upper-case letters and lower-case letters are not the same.

Some valid identifiers are:

HELLO	hello
_K5input	P_h
This_is_a_long_identifier	

Note the use of underscores to separate words. Note also that lower-case letters are not the same as upper-case letters. Thus hello is a different identifier than HELLO. 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
b6 kj	Contains a space

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

7.3.1 Reserved Identifiers

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

CASE	FUSES	PIN
DEVICE	GOTO	STATE
ELSE	IF	STATE_DIAGRAM
ENABLE	IN	TEST_VECTORS
END	ISTYPE	THEN
ENDCASE	LIBRARY	TITLE
ENDWITH	MACRO	TRUTH_TABLE
EQUATIONS	MODULE	WHEN
FLAG	NODE	WITH

7.3.2 Choosing Identifiers

The right choice in identifiers can make a source file easy to read and understand. This is especially important in an environment where more than one person may work on the same project, or where one designer will take up another's earlier work. The following suggestions are given to help make logic descriptions self-explanatory, thus eliminating the need for extensive documentation.

- Choose identifiers that match the function of that which they describe. For example, the pin to be used as the carry-in on an adder could be named `Carry_In`. For a simple OR gate, the two input pins might be given the identifiers `IN1` and `IN2`, and the output might be named `OR`.
- Avoid large numbers of similar identifiers. For example, do not name the outputs of a 16 bit adder:
`ADDER_OUTPUT_BIT_1`
`ADDER_OUTPUT_BIT_2`
and so on. Such grouping of names makes the source file difficult to read.
- Use underscores to separate words in your identifier. `THIS_IS_AN_IDENTIFIER` is much easier to read than `THISISANIDENTIFIER`.
- Mixed-case identifiers can help make your source file readable; for example, `CarryIn`

7.4 Strings

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

Valid strings:

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

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

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

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 (’).

7.5 Comments

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

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

Comments cannot be imbedded within keywords.

Valid comments (shown in boldface):

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

Note that the information inside single quotation marks (apostrophes) is not a comment but is part of the statement.

7.6 Numbers

All operations in ABEL involving numeric values are done with 32-bit accuracy. Thus, valid numeric values fall in the range 0 to $2^{32} - 1$. 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 7-1 gives the four bases supported by ABEL and their accompanying symbols. The base symbols can be typed in upper-case or lower-case.

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

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

Table 7-1. Number Representation in Different Bases

Base Name	Base	Symbol
binary	2	<code>^b</code>
octal	8	<code>^o</code>
decimal	10	<code>^d</code>
hexadecimal	16	<code>^h</code>

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

Specification	Decimal Value
<code>75</code>	<code>75</code>
<code>^h75</code>	<code>117</code>
<code>^b101</code>	<code>5</code>
<code>^o17</code>	<code>15</code>
<code>^h0F</code>	<code>15</code>

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

Numbers may also be specified by alphabetic characters. In this case, the numeric ASCII code of the letter is used as the numeric value. For example, the character "a" is

decimal 97, and hexadecimal 61 in ASCII coding. The decimal value 97 would be used if "a" were specified as a number.

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

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

7.7 Special Constants

Constant, non-changing values can be used in ABEL 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," section 8.5, and "Module Statement," section 8.2). Constant values may be either numeric or one of the non-numeric special constant values. The special values are listed in table 7-2.

Table 7-2. Special Constant Values

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

When one of the special constants is used, it must be entered as shown in table 7-2 with surrounding periods. The periods indicate that a special constant is being used; without the periods, .C. would appear to be an identifier named C. Special constants can be entered in either upper-case or lower-case.

7.8 Operators, Expressions, and Equations

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

ABEL 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.

7.8.1 Logical Operators

Logical operators are used in Boolean expressions. ABEL incorporates the standard logical operators used in most logic designs; these operators are listed in table 7-3. Logical operations involving operands of more than one bit are performed bit by bit. Thus, 2 minus 4 equals 6. For alternate operators, refer to the @ALTERNATE Directive section in this Language Reference manual.

Table 7-3. Logical Operators

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

7.8.2 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 7-4 lists the arithmetic operators.

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

Table 7-4. Arithmetic Operators

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

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

7.8.3 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 7-5 lists the relational operators.

Table 7-5. Relational Operators

Operator	Example	Description
==	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

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

Some examples of relational operators in expressions follow:

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

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

For example:

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

means that:

A will equal the complement of D if B equals C

A will equal D if B does not equal C.

7.8.4 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 section 7.8.6 for a complete discussion of equations. There are two assignment operators, unclocked and clocked. Unclocked or immediate assignment occurs without any delay as soon as the equation is evaluated. Clocked assignment occurs at the next clock pulse from the clock associated with the output. Table 7-6 shows the assignment operators.

Table 7-6. Assignment Operators

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

7.8.5 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 7-7 summarizes the logical, arithmetic and relational operators, presented in groups according to their priority.

Table 7-7. Summary of Operators and Priorities

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

When operations of the same priority exist in the same expression, they are performed in the order found from left to right in that expression. Parentheses may be used as in normal mathematics to change the order of evaluation, with the operation in the innermost set of parentheses performed first. Some examples of valid expressions are given on the following page. 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	
$2 == ^\text{HA}$	0	false
$14 == ^\text{HE}$	-1	true

7.8.6 Equations

**[WHEN condition THEN][!...] [ENABLE] element
= expression; [ELSE equation]
or
[WHEN condition THEN][!...] [ENABLE] element
:= expression; [ELSE equation]**

condition	any valid expression
element	an identifier naming a signal or set of signals, or an actual set, to which the value of expression will be assigned
expression	any valid expression
= and :=	unlocked and clocked assignment operators

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 "=" (unlocked) and "==" (clocked) described in section 7.8.4.

The keyword, **ENABLE**, is used to enable tri-state output buffers. The identifier must be a tri-state type output, and the value assignment applies only to the buffer enable rather than to the signal itself.

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

logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

Examples of equations:

$X = A \& B;$ " unclocked assignment to X

$ENABLE\ Y = C \# D;$ " Y is enabled if C or D is true

$Y: = A \& B;$ " clocked assignment to Y

$!A = B \& C \# D;$ " same as $A = !(B \& C \# D);$

$WHEN\ B\ THEN\ A=B; ELSE\ A=C;$

Multiple Assignments to the Same Identifier

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

Examples:

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.

7.9 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 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 square 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]	decremting range
[X.,X.,X.,X.,X.,X.,A10..A7]	range within a larger set

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

7.9.1 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 7-8 lists the operators that may be used with sets. Appendix F describes how these operators are applied to sets.

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

Some examples of set usage are given here.

The Boolean equation,

$$\text{Chip_Sel} = \text{A15} \ \& \ !\text{A14} \ \& \ \text{A13};$$

represents an address decoder where A15, A14 and A13 are the three high-order bits of a 16-bit address. The decoder can easily be implemented with set operations. First, a constant set that holds the address lines is defined so that the set can be referenced by name. This definition is done in the constant declaration section of a module (described in chapter 8).

The declaration is:

$$\text{Addr} = [\text{A15}, \text{A14}, \text{A13}];$$

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]).

7.9.2 Set Assignment and Comparison

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

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

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

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

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

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

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

Thus, the following two assignments are equivalent:

```
[a,b] = ^B101011;  
[a,b] = ^B11;
```

And so are these two:

```
[d,c] = ^B01;  
[d,c] = ^B1;
```

The set assignment,

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

is the same as the two assignments:

`a = c & d;`

`b = c & d;`

Table 7-8. Valid Set Operations

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

7.9.3 Set Evaluation

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

When a set is written out:

[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 one's) or FALSE (all zero's), 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.

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

Both sel1 and sel2 will be true when the value of the four "a" lines (taken as a binary number) are greater than 2. Note that the width of the "sel" set and the "a" set are different. The "2" will be expanded to four bits (of binary) to match the size of the "a" set. The result of the

comparison (which will be all ones or all zeros) will then be truncated to two bits to match the size of the "sel" set.

$$[\text{out3}..\text{out0}] = [\text{in3}..\text{in0}] \ \& \ \text{enable}$$

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

7.9.4 Limitations/Restrictions on Sets

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

$$\text{count} = [\text{a}, \text{b}, \text{c}]$$

Then the following expression has a width of one:

$$9 \ \& \ (\text{count} == 0)$$

It will be expanded as follows:

$$\begin{aligned} &9 \ \& \ (\ !\text{a} \ \& \ !\text{b} \ \& \ !\text{c}) \\ &1 \ \& \ !\text{a} \ \& \ !\text{b} \ \& \ !\text{c} \end{aligned}$$

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.

$[x1, y1] = [a, b] \& 1 \& d$	$[x2, y2] = 1 \& d \& [a, b]$
$= ([a, b] \& 1) \& d$	$= (1 \& d) \& [a, b]$
$= ([a, b] \& [0, 1]) \& d$	$= d \& [a, b]$
$= ([a \& 0, b \& 1]) \& d$	$= [d, a] \& [d, b]$
$= [0, b] \& d$	$= [d \& a, d \& b]$
$= [0, b] \& [b, d]$	
$= [0 \& d, b \& d]$	
$= [0, b \& d]$	

$x1 = 0$	$x2 = a \& d$
$y1 = b \& d$	$y2 = b \& d$

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

1. Fully parenthesize your equation. Most errors are simply caused by ignoring the precedence rules in table 5.1.
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.

The following restriction applies to sets:

Because set assignment is applied to all elements of a set, sets with mixed combinational and registered outputs cannot be used on the left side of an equation. Assignments are either clocked (combinational) or unclocked (registered) according to the assignment operator used (= or :=) and a clocked assignment to a combinational output or an unclocked assignment to a registered output constitutes an error and will be flagged by the language processor.

7.10 Blocks

Blocks are sections of ASCII text enclosed in braces, "{" and "}". Blocks are used in macros and directives. The text contained in a block can be all on one line or can span many lines. Some examples of blocks follow.

```
{ this is a block }  
{  
  this is also a block, and it  
  spans more than one line.  
}
```

```
{ A = B # C;  
D = [0, 1] + [1, 0];  
}
```

Blocks can be nested within other blocks, as shown below, where the block { D = A } is nested within a larger block:

```
{ A = B $ C;  
  { D = A; }  
  E = C;  
}
```

Blocks and nesting of blocks can be useful in macros and when used with directives. (See "Macro Declarations" in Chapter 8, and "Directives" in Chapter 9.)

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.

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

dummy argument	an identifier that is used to indicate where an actual argument is to be substituted in the macro, module, or directive.
actual argument	the argument (value) used in the macro, directive or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL.

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 identifiers occurring in the source file.

Take for example, the following macro declaration arguments (macros are discussed fully in Chapter 8 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:

$$D = \text{OR_EM} (x,y,z\&1);$$

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,

$$D = \text{OR_EM} (x,y,z \& 1)$$

results in the equation:

$$D = x \# y \# z \& 1;$$

Argument substitution occurs before the source file is checked for syntactic or logical correctness. This means that the code is checked for correctness with the actual arguments in place. Thus, if an actual argument violates a syntactic or logical rule, the parser will detect and report the error.

In review:

- **Dummy arguments are place holders for actual arguments.**
- **A question mark preceding the dummy argument indicates that an actual argument is to be substituted.**
- **Actual arguments replace dummy arguments before the source file is checked for correctness.**
- **Spaces are significant in actual arguments.**

Further discussion and examples of argument usage are given in the Applications Guide, in the explanations of modules and macros in Chapter 8, and also in Chapter 9.

8. Language Structure

This section describes the structure of a complete ABEL design description. The language elements described in chapter 7 are combined within the proper structures to define Boolean equations, state machines and truth tables. The device or devices being described must be indicated and initial declarations must be made. Definitions of expected output from simulation of the device may also be made.

8.1 Basic Structure

ABEL source files can be broken into independent parts called modules. Each module contains one or more complete logic descriptions. At the simplest level, an input file to the ABEL language processor consists of only one module; at the most complex level, an indefinite number of modules may be combined into one source file and processed at the same time.

Every module consists of several different sections, each with its own unique function. These sections are:

- Declarations of devices, pins, nodes, constants,
attributes, and macros
- Boolean logic equations
- Truth tables
- State diagrams
- Explicit fuse declarations
- Test vectors

Some or all of these parts of a module may exist for any given application. Declarations must always be made. Figure 8-1 shows the structure of an ABEL source file. Because a source file is a collection of one or more modules, each with its own beginning and end, different source files can be combined to form complete system designs in one source file.

```
1st Module Start
  Flags
  Title
  Declarations
    Constant Declarations
    Macro Definitions
    Device Declarations
    Pin and Node Assignments
    Attribute Declarations
    Library References
  Boolean Equations
  Truth Tables
  State Diagrams
  Fuse Declarations
  Test Vectors
1st Module End
2nd Module Start
  .
  .
  .
2nd Module End
  .
  .
```

Figure 8-1. Structure of an ABEL Source File

8.2 MODULE Statement and Structure

```
MODULE modname [ ( dummy_arg [, dummy_arg] ... ) ]  
    [ FLAG statement ]..  
    [ TITLE statement ]  
    declarations  
    [ EQUATIONS ]..  
    [ TRUTH_TABLE ] ...  
    [ STATE_DIAGRAM ]..  
    [ TEST_VECTORS ]..  
END [ modname ] [ ; ]
```

modname	a valid identifier naming the module
dummy_arg	a dummy argument
FLAG	defines processing parameters to be passed to the language processor
TITLE	defines the title of the module
declarations	declarations section of the module in which pin, node, device, attribute, and constant declarations are made
EQUATIONS	Boolean logic equations section
TRUTH_TABLE	truth table specification
STATE_DIAGRAM	state machine specification
FUSES	explicit fuse declarations
TEST_VECTORS	specification of simulation test vectors

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

Each module in a source file should have a unique name. The module contains the different declarations, equations, truth tables, state diagrams and simulation tables necessary to complete the logic description(s). The following three rules apply to module structure:

1. If the FLAG statement is used, it must be the first keyword used after the MODULE statement.
2. If the TITLE statement is used, it must be the first keyword used after the FLAG statement (if one exists). If FLAG is not used, the TITLE statement must be the first keyword used after the MODULE statement.
3. One declarations section must exist in a module. All other sections can occur in the module as many times as needed and in any order.

The optional dummy arguments in the module statement allow the actual arguments to be passed to the module when it is processed by the language processor. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value will be substituted by the parser.

For example:

```
MODULE MY_EXAMPLE (A,B)
.
.
C = ?B + ?A
.
.
END
```

In the module named MY_EXAMPLE, C will take on the value of "A + B" where A and B contain actual arguments passed to the module when the language processor is invoked. For further information about dummy arguments, see section 7.11.

8.3 FLAG Statement

FLAG parameter[, parameter]...

parameter a string containing valid command line parameters, excepting -I, -O, -N, and all PARSE parameters

The FLAG statement provides an alternate method of defining processing parameters that affect the way in which the source file is processed by the language processor. These parameters are normally passed from the command line or from a batch file when the language processor is invoked. FLAG allows the explicit statement of processing parameters directly in the source file. Parameters entered from the command line override parameters specified with the FLAG statement. PARSE parameters, and the -I, -O, and -N parameters are not allowed. Complete information on command line parameters is contained in Chapter 9.

FLAG is useful when a source file requires a specific type or level of processing for that processing to be successful. For example, a design may be large or complex enough that it will generate too many product terms for the specified device unless the PRESTO reduction algorithm is used in the reduction step of the processing. You might also want to simulate the design at trace level 3. This is done with the following command:

```
flag '-r2,-t3'
```

8.4 TITLE Statement

TITLE string

The title statement is used to give a module a title that will appear as a header in both the programmer load file and documentation file created by the language processor. The title is specified in the string following the keyword, **TITLE**.

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.

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 23 Jan 1987'
```

8.5 Declarations

The declarations section of a module specifies the device being described and the correspondence between the names used in a module and the pins and nodes of the device. Constants, attributes, and macros are also defined in the declarations section. Declarations stay in effect only in the module in which they are defined. Each module must have its own declarations section. There are six types of declaration statements:

- Attribute
- Constant
- Device
- Library
- Macro
- Node
- Pin

The syntax and use of each of these types is presented in the subsections that follow.

8.5.1 Device Declaration Statement

device_id [, device_id]... DEVICE real_device ;

device_id an identifier used in a module for
references to a device

real_device a string describing the industry part
number of the real device represented
by device_id

The device declaration statement associates device names used in a module with actual programmable logic devices on which designs are implemented. The device name used in the logic description is specified with `device_id`. Device identifiers used in device declarations should be valid file names since JEDEC files are created by appending the extension ".JED" to the identifier. The industry part number of the programmable logic device is indicated by the string, `real_device`.

These are the only devices that can be specified by `real_device`.

The ending semicolon is required.

The following example specifies two device names, `u14` and `u15`, that represent F159 IFLs:

```
u14, u15 device 'F159' ;
```


8.5.2 Pin Declaration Statement

```
[!]pin_id [, [!]pin_id[... PIN [IN device_id]
                pin#[='attr[,attr]...'] [,pin#[='attr[,attr]...']]...
```

pin_id	an identifier used to refer to a pin throughout a module
device_id	a declared device name identifier indicating the device associated with these pins
pin #	the pin number on the real device
attr	a string that specifies pin attributes for devices with programmable pins. Valid strings are:

pos	positive polarity
neg	negative polarity
reg	registered signal
reg_d	D-type register
reg_g	G-type (clock enabled) register
reg_jk	JK-type register
reg_rs	RS-type register
reg_t	T-type register
reg_jkd	JK/D controllable register
com	combinational signal
latch	latch input pin
feed_pin	feedback from pin
feed_reg	feedback from register
feed_or	feedback from OR-gate

The pin declaration statement indicates the correspondence between identifiers used in a module and the pins on a real device. The declaration can also specify pin attributes for devices with programmable pin characteristics.

When lists of pin_ids and pin #s are used in one pin declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one pin number associated with each identifier listed.

When more than one device is declared in one module, the optional IN portion of the pin declaration must be used so that the pins are associated with the proper device. A device declaration must be made before the pin declarations. The ending semicolon is required. An example of a simple pin declaration follows:

```
!Clock, Reset, S1 PIN IN U12 12,15,3 ;
```

This pin declaration assigns the pin names, Clock, to pin 12 of device U12, Reset to pin 15, and S1 to pin 3.

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

The pin attribute, attr, specifies pin attributes. Attributes can be defined in this way or with the ISTYPE statement. The ISTYPE statement and attributes are discussed in section 8.5.6. The pin declaration,

```
F0 pin 13 = 'neg, reg';
```

specifies that equations for F0, which corresponds to pin 13, should be optimized for a negative polarity registered output.

8.5.3 Node Declaration Statement

**[!node_id [,!node_id]... NODE [IN device_id] node#
[= 'attr[,attr]... '][,node#|= 'attr[,attr]...']]**...

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

device_id a declared device name identifier indicating
 the device associated with these nodes

node # the node number on the real device

attr a string that specifies node attributes for
 devices with programmable nodes. Valid
 strings are:

pos	positive polarity
neg	negative polarity
reg	registered signal
reg_d	D-type register
reg_g	G-type (clock enabled) register
reg_jk	JK-type register
reg_rs	RS-type register
reg_t	T-type register
reg_jkd	JK/D controllable register
com	combinational signal
latch	latch input pin
feed_pin	feedback from pin
feed_reg	feedback from register
feed_or	feedback from OR-gate
fuse	single fuse node
pin	pin controlled node
eqn	array controlled node

The node declaration statement indicates the correspondence between identifiers used in a module and the internal nodes of a real device. Nodes are "pseudo-pins": internal

signals that are not accessible on the device's external pins, but that are needed to program otherwise inaccessible fuses. The device nodes supported by ABEL are listed in appendix C. The declaration can also specify node attributes for devices with programmable node characteristics.

When lists of `node_ids` and `node #s` are used in one node declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one node number associated with each identifier listed.

When more than one device is declared in one module, the optional `IN` portion of the node declaration must be used so that the nodes are associated with the proper device.

A device declaration must be made before the node declarations. The ending semicolon is required. The following example declares three nodes A, B, and C in the device U15.

```
A, B, C NODE IN U15 21,22,23 ;
```

The node attribute string, `attr`, specifies node attributes. Attributes can be defined in this way or with the `ISTYPE` statement. The `ISTYPE` statement and attributes are discussed in section 8.5.6.

The node declaration,

```
B NODE 22 = 'pos,com' ;
```

specifies that node 22 has positive polarity and is a combinational node.

If shorthand notation is used to refer to internal device nodes, it is not necessary to declare the node in a declaration statement. For examples of shorthand notation of nodes, refer to Chapter 13. Appendix C lists the shorthand notation for all supported nodes.

8.5.4 Constant Declaration Statement

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

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

expr an expression defining the constant value

The constant declaration statement defines constants to be used within a module. A constant is an identifier that retains a constant value throughout a module.

The identifiers listed on the left side of the equals sign in the constant declaration statement are assigned the values listed on the right side of the equals sign.

There is a one-to-one correspondence between the identifiers and the expressions listed and there must be one expression for each identifier.

The ending semicolon is required.

Constants are useful when a value must be used many times throughout a module and especially when the value is used many times but might need to be changed during the logic design process. Rather than changing the value throughout the module, the value can be changed once in the declaration of the constant.

Constant declarations may not be self-referencing; for example:

`X = X;`

will cause errors, as will the declarations

`a = b;`
`b = a;`

Some examples of valid constant declarations follow:

<code>ABC = 3 * 17;</code>	" ABC is assigned the value 51
<code>Y = 'Bc' ;</code>	" Y = + H4263 ;
<code>X =. X.;</code>	" X means 'don't care'
<code>ADDR = [1,0,15];</code>	" ADDR is a set with 3 elements
<code>A,B,C = 5,[1,0],6;</code>	" 3 constants declared here
<code>D pin 6;</code>	" see next line
<code>E = [5 * 7,D];</code>	" signal names can be included
<code>G = [1,2]+[3,4];</code>	" set operations are legal
<code>A = B & C;</code>	" operations on identifiers are valid
<code>A = [!B,C];</code>	" set and identifiers on right

8.5.5 Macro Declaration Statement and Macro Expansion

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

macro_id an identifier naming the macro

dummy_arg a dummy argument

block a block (see chapter 7)

The macro declaration statement defines a macro. Macros are used to include ABEL code in a source file without typing or copying the code everywhere it is needed. A macro is defined once in the declarations section of a module and then used anywhere within the module as frequently as needed. Macros can be used only within the module in which they are declared.

Wherever the macro_id occurs, the text in the block associated with that macro will be substituted. With the exception of dummy arguments, all text in the block (including spaces, end-of-lines, etc.) is substituted exactly as it appears in the block.

When debugging your source file, you can use the -E and -P flags to examine macro statements. The -E parameter causes the parsed and expanded source code to be written to the listing file (or to the display if the -L flag is omitted). In addition to the listing information provided by the -E flag, the -P flag lists the directives that caused code to be added to the source.

The dummy arguments used in the declaration of the macro allow different actual arguments to be used in the macro each time it is invoked in the module. Within the macro,

dummy arguments are preceded by a "?" to indicate that an actual argument will be substituted for the dummy by the ABEL parser. This is best shown by example.

The equation,

```
NAND3 MACRO (A,B,C) { !(?A & ?B & ?C) } ;
```

declares a macro named NAND3 with the dummy arguments A, B and C. The macro defines a three-input NAND gate. When the macro identifier occurs in the source, actual arguments for A, B and C will be supplied.

For example, the equation,

```
D = NAND3 (Clock,Hello,Busy) ;
```

brings the text in the block associated with NAND3 into the code, with Clock substituted for ?A, Hello for ?B and Busy for ?C. This results in:

```
D = !( Clock & Hello & Busy ) ;
```

which is the three-input NAND.

The macro NAND3 has been specified by a Boolean equation, but it could have been specified using another ABEL 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 in section 8.7.

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 they occur in logic descriptions:

```
A
X = W + B (inp) ;
Y = W + B( )C ; "note the blank actual argument
```

resulting in:

```
"note leading space from block in A
W = S1 & S2 & S3 ;
X = W + ! inp ;
Y = W + ! C ;
```

Circular macro references (when a macro refers to itself within its own definition) cause the PARSE program to terminate abnormally with errors. This error can often be detected by examining the PARSE listing file. If errors appear after the first use of a macro, and the errors cannot be easily explained otherwise, check for a circular macro reference.

8.5.6 ISTYPE Statement

signal [,signal]... ISTYPE [IN device_id] 'attr [,attr]...';

signal a pin or node identifier

device_id a declared device name identifier indicating
the device associated with these nodes

attr	pos	positive polarity
	neg	negative polarity
	reg	registered signal
	reg_d	D-type register
	reg_g	G-type (clock enabled) register
	reg_jk	JK-type register
	reg_rs	RS-type register
	reg_t	T-type register
	reg_jkd	JK/D controllable register
	com	combinational signal
	latch	latch input signal
	feed_pin	feedback from pin
	feed_reg	feedback from register
	feed_or	feedback from OR-gate
	pin	select node from pin
	eqn	select node from equation
	fuse	select node from fuse
	share	product term sharing

The **ISTYPE** statement defines attributes or characteristics of pins and nodes for devices with programmable characteristics. The attributes are used to form correct logic for the device and to optimize equations for the device. If no attributes are defined for a device with programmable characteristics the device defaults are applied. If attributes are defined for a device without programmable characteristics, an error is reported by the language processor. An example of **ISTYPE** statement usage is presented in the design example file **ALTERA.ABL**.

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

When more than one device is declared in one module, the optional **IN DEVICE_id** portion of the **ISTYPE** must be used so the signals (pins or nodes) are associated with the correct device. A device declaration must be made before the **ISTYPE** statement appears in the source file. Declarations of the pin and node names used in the **ISTYPE** statement must be made before the **ISTYPE** statement.

An example of the **ISTYPE** statement follows:

```
FO, A istype 'neg, latch' ;
```

This declaration statement defines **F 0** and **A** as negative polarity latches. Both **F 0** and **A** had to have been defined previously in the module.

Definitions of each of the attributes follows:

Pos (Positive Polarity)

Pos indicates that the associated input or output has positive polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for that polarity. Pos may be entered in upper-, lower-, or mixed-case letters.

Neg (Negative Polarity)

Neg indicates that the associated input or output has negative polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for negative polarity. Neg may be entered in upper-, lower-, or mixed-case letters.

Reg (Registered Signal)

Reg indicates that the associated input or output is registered rather than combinational. The device will be programmed for this condition, and the signal will change on application of the clock.

Reg_(type) (Registered Type)

This attribute indicates the type of register to use for a signal that has programmable register types. The device will be programmed for the indicated register type.

Com (Combinational Signal)

Com indicates that the associated input or output is a combinational or combinatorial signal. The device will be programmed for this condition.

Latch

The latch attribute indicates that the associated input or output is latched. Latches have an enable line and operate as follows:

1. If the enable is high, the input signal to the latch is passed through to the output.
2. If the enable is low, the last value present on the input line before the high-to-low transition of the enable is held on the output line.

Feed_pin, Feed_reg, and Feed_or (Feedback Specifications)

Some available devices allow definition of the internal feedback paths. Feedback can occur from the output pin, the output of an internal register, or from the OR-gate output, as shown in figure 8-2.

The `feed_pin`, `feed_reg`, or `feed_or` attributes specify which feedback path to use: from the pin, the register output, or the OR-gate, respectively. Only one of the feedback attributes can be in effect at any given time; if more than one feedback attribute is specified for the same signal, the last attribute specified takes effect. Also, feedback attributes are valid only for devices with feedback paths that can be defined by the user. Feedback attributes are valid only for outputs. The attribute may be specified in upper-, lower-, or mixed- case characters.

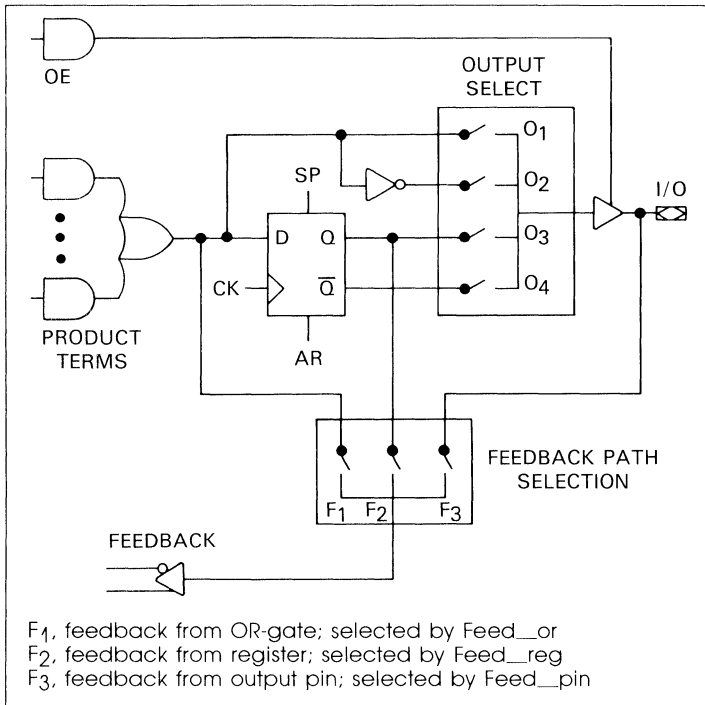


Figure 8-2. Feedback Paths for an E0310

**Pin, Eqn, or Fuse
(selectable node type)**

Some devices allow internal device features, such as output enables or clocks to be configured in one of several ways. The Pin, Eqn, and Fuse attributes specify which type of feature is desired for the indicated node. For example.

Q0.oe istype pin

specifies that the output enable is pin controlled.

8.5.7 LIBRARY Statement

LIBRARY 'name'

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

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

The file extension of '.inc' is appended to the name specified, and the resulting file name is searched for. If no file is found, ABEL will attempt to find the file in the abel3lib.inc library file. Refer to sections 6.3 and 6.4 for more information on libraries.

8.6 Equations Statement

EQUATIONS [IN device_id]

device_id a previously declared device identifier
 that indicates the device associated with
 these equations

The equations statement defines the beginning of a group of equations associated with a device. Equations specify logic functions with Boolean algebra.

The equations statement is followed by the equations associated with the device indicated by the device identifier. In modules with only one device, the device identifier is optional; in all other cases it must be specified so that the equations will be applied toward the correct device.

The equations following the equation statement are any valid ABEL equations as described in section 2.8.6.

A sample equations section follows:

```
EQUATIONS IN IC13
  A = B & C # A ;
  [W,Y] = 3 ;
  !F = B == C ;
```


8.7 Truth Table Statement

Truth tables are another way to describe logic designs with ABEL and may be used in lieu of, or in addition to, Boolean equations and state diagrams. Truth tables specify outputs as functions of different input combinations in a tabular form. A truth table is specified with a header describing the format of the table and with the table itself.

8.7.1 Truth Table Header Syntax

TRUTH_TABLE [IN device_id] (inputs -> outputs)

or

TRUTH_TABLE [IN device_id] (inputs := reg_outs)

or

TRUTH_TABLE [IN device_id] (inputs := reg_outs -> outputs)

device_id an identifier naming the device associated with the truth table

inputs the inputs to the logic function

outputs the outputs from the logic function

reg_outs the registered (clocked) outputs

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

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

The truth table header can have one of the three forms shown above, depending on whether the device has registered or combinational outputs or both.

In all three forms, the device identifier is required when more than one device is declared in a module. An example of truth table usage is presented in the design example file LED7.ABL.

The inputs and outputs (both registered and combinational) of a truth table are either single signals, or, more frequently, sets of signals. If only one signal is used as either the input or output, its name is specified. Sets of signals used as inputs or outputs are specified in the normal set notation with the signals surrounded by square brackets and separated by commas (see section 7.9).

The syntax shown in the first form defines the format of a truth table with simple combinational outputs. The values of the inputs determine the values of the outputs.

The second form describes a format for a truth table with registered outputs. The symbol ">" preceding the outputs distinguishes these outputs from the combinational ones. Again the values of the inputs determine the values of the outputs, but now the outputs are registered or clocked: they will contain the new value (as determined by the inputs) after the next clock pulse.

The third form is more complex, defining a table with both combinational and registered outputs. It is important in this format to make sure the different specification characters ">" and ">" are used for the different types of outputs.

Examples of headers along with their accompanying truth tables are given after the format of the table is discussed.

8.7.2 Truth Table Format

The actual truth table (as opposed to the header) is specified according to the form described within the parentheses in the header. The truth table is a list of input combinations and resulting outputs. All or some of the possible input combinations may be listed.

As an example, the following truth table defines an exclusive-OR function with two inputs (A and B), one enable (en), and one output (C):

```
TRUTH_TABLE IN IC16 ( [en,A,B] -> C )
```

```
[0,.X.,.X.] -> .X.; " don't care w/ enable off  
[1, 0 , 0 ] -> 0 ;  
[1, 0 , 1 ] -> 1 ;  
[1, 1 , 0 ] -> 1 ;  
[1, 1 , 1 ] -> 0 ;
```

All values specified in the table must be constants, either declared, numeric, or the special constant, ".X.". Each line of the table (each input/output listing) must end with a semicolon.

Whereas the header defines the names of the inputs and outputs, the table defines the values of inputs and the resulting output values.

The following example shows a truth table description of a simple state machine with four states and one output. The current state is described by signals A and B, which are put into a set. The next state is described by the registered outputs C and D, which are also collected into a set. The single combinational output is signal E. The machine simply counts through the different states, driving the output E low when A equals 1 and B equals 0.

```
TRUTH_TABLE IN IC17 ( [A,B] := [C,D] -> E )
```

```
0 := 1 -> 1 ;  
1 := 2 -> 0 ;  
2 := 3 -> 1 ;  
3 := 0 -> 1 ;
```

Note that the input and output combinations are specified by a single constant value rather than by set notation. This is equivalent to:

```
[0,0] := [0,1] -> 1 ;  
[0,1] := [1,0] -> 0 ;  
[1,0] := [1,1] -> 1 ;  
[1,1] := [0,0] -> 1 ;
```

8.7.3 Programmable Polarity Registers

When using state diagrams and truth tables for programmable polarity devices, the default polarity used by ABEL (if no polarity is specified via the ISTYPE statement) is now negative, rather than positive. Existing ABEL source files that assume positive polarity for these devices may require the addition of an ISTYPE statement to force positive polarity. See example LED1.ABL.

8.8 State Diagrams

An alternative to describing logic with Boolean equations and truth tables is to use a state diagram. The state diagram easily describes the operation of a sequential state machine implemented with programmable logic.

The specification of a state diagram requires the use of the `STATE_DIAGRAM` construct, which defines the state machine, and the `IF-THEN-ELSE`, `CASE`, and `GOTO` statements which determine the operation of the state machine.

The `STATE_DIAGRAM` construct is discussed first, and then the syntaxes of the `IF-THEN-ELSE`, `CASE`, and `GOTO` statements are presented.

Syntax of the `WITH-ENDWITH` statement, which permits the specification of equations in terms of transitions, is also presented.

8.8.1 STATE_DIAGRAM Statement

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

device_id	an identifier specifying the associated device
state_reg	an identifier or set of identifiers specifying the signals that determine the current state of the machine.
state_out	an identifier or set of identifiers that determine the next state of the machine (for designs with external registers)
state_exp	an expression giving the current state
equation	a valid equation that defines the state machine outputs
trans_stmt	an IF-THEN-ELSE, CASE or GOTO statement, optionally followed by WITH_ENDWITH transition equations.

The STATE_DIAGRAM construct defines a state machine associated with a device in a module. The state machine starts in one of the states indicated by state_exp. The equations listed after that state_exp are evaluated, and the transition statement (trans_stmt) is evaluated after the next clock, causing the machine to advance to the next state.

Equations associated with a state are optional. Each state must have a transition statement. If none of the transition conditions for a state is met, the next state is undefined. (For some devices, undefined state transitions cause a transition to the cleared register state.) As many states as are needed can be specified.

When there is more than one device declared in a module, the "IN device_id" section is required.

Following is an example of a simple state machine that advances from one state to the next, setting the output to the current state, and then starting over again. Note that the states do not need to be specified in any particular order. Note also that state 2 is identified by an expression rather than by a constant. The state register is composed of the signals a and b.

```
state__diagram in u15 [a,b]
    state 3      : y = 3 ;
                  goto 0 ;
    state 1      : y = 1 ;
                  goto 2 ;
    state 0      : y = 0 ;
                  goto 1 ;
    state 1 + 1   : y = 2 ;
                  goto 3 ;
```

The next state diagram specifies a more complex state machine where the state_reg is specified with a constant set containing the signals a and b. Assuming that the state machine starts in state 1 (a = 0, b = 1), the sequence of states will be 1,4,2,3,2,4, 1,4,2,3,2,4, 1...

```

current_state = [a, b] "constant declaration
STATE_DIAGRAM current_state
state 1:  w = 1 ;
          y = 1 ; GOTO 4 ;
state 2 : IF y==3 THEN 3
          ELSE 4 ;
state 3 : w = 2 ;
          y = w ;
          GOTO 2 ;
state 4 : y = 3 ;
          CASE   w==1: 2;
                w==2: 1;
          ENDCASE ;

```

Figure 8-3 shows the pictorial state diagram for this same state machine.

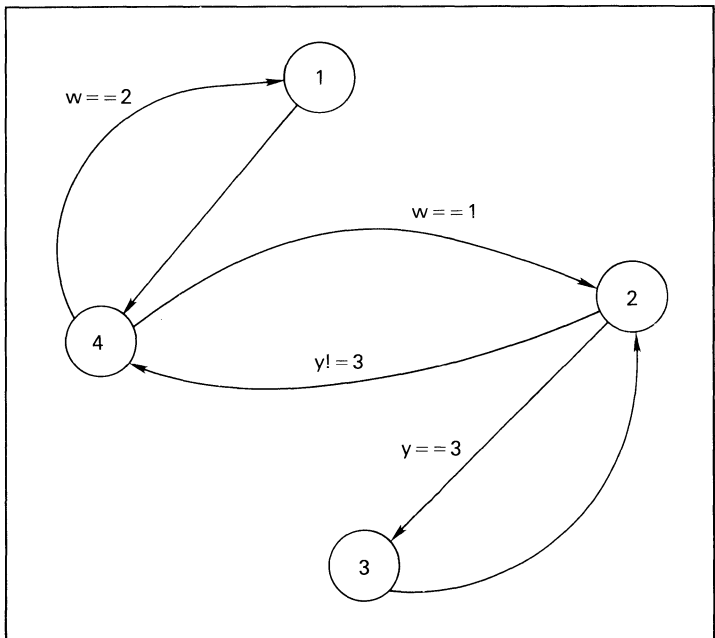


Figure 8-3. Pictorial State Diagram

8.8.2 IF-THEN-ELSE Statement

IF expression THEN state__exp [ELSE state__exp] ;

expression any valid expression

state__exp an expression identifying the next state,
 optionally followed by WITH__ENDWITH
 transition equations.

The IF-THEN-ELSE statement is an easy way to describe the progression from one state to another in a state machine. The expression following the IF keyword is evaluated, and if the result is true, the machine goes to the state indicated by the state__exp following the THEN keyword. If the result of the expression is false, the machine advances to the state indicated by the ELSE keyword.

Any number of IF statements may be used in a given state, and the use of the ELSE clause is optional.

Examples:

```
if A==B then 2 ; "if A equals B goto state 2
if x-y then j else k; "if x-y is not 0 goto j, else goto k
if A then b*c; "if A is true (non-zero) goto state b*c
```

8.8.3 Chained IF-THEN-ELSE Statements

```
IF expression THEN state__expression  
ELSE  
  IF expression THEN state__expression  
  ELSE  
    IF expression THEN state__expression  
    ELSE state__expression ;
```

Additional IF-THEN-ELSE statements can be chained to the ELSE clause of an IF-THEN-ELSE statement. Any number of IF-THEN-ELSE statements can be chained, but the final statement must end with a semicolon. An example of chained IF-THEN-ELSE statements follows:

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

Often, chains of mutually exclusive IF-THEN-ELSE statements can be more clearly expressed with a CASE statement. The chained IF-THEN-ELSE statement is intended for situations (such as the preceding example) where the conditions are not mutually exclusive.

See section 8.8.1 for examples of IF-THEN-ELSE usage as they are used in a state diagram.

8.8.4 CASE Statement

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

expression any valid ABEL expression

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

The CASE statement is an easy way to indicate the transitions of a state machine when there are multiple possible conditions that affect the state transitions.

The expressions contained within the CASE-ENDCASE keywords must be mutually exclusive, meaning that only one of the expressions can be true at any given time. If two or more expressions within the same CASE statement are true, the resulting equations are undefined.

The state machine will advance to the state indicated by the state_exp following the expression that produces a true value. If no expression is true, the result is undefined, and the resulting action depends on the device being used. (For devices with D flip-flops, the next state will be the cleared register state.) For this reason, you should be sure to cover all possible conditions in the CASE statement expressions. If the expression produces a numeric rather than a logical value, 0 is false, and any non-zero value is true.

Example:

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

More CASE statement examples are presented in section 8.8.1.

8.8.5 GOTO Statement

GOTO state_exp ;

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

The GOTO statement causes an unconditional transition to the state indicated by state_exp.

Example:

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

8.8.6 WITH-ENDWITH Statement

```

transition__stmt state__exp WITH equation
                                [equation]
                                .
                                .
                                ENDWITH ;

```

transition__stmt IF, ELSE, or CASE statement

state__exp the next state

equation an equation for state machine outputs

The WITH-ENDWITH statement, when used in conjunction with the IF-THEN or CASE statement, allows output equations to be written in terms of transitions; for example:

```

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

```

8.9 Fuses Section

FUSES [IN device_id]

fuse__number = fuse value ;

or

fuse__number__set = fuse value ;

device_id	identifier specifying associated device
fuse__number	fuse number obtained from logic diagram of device
fuse__number__set	set of fuse numbers contained in square brackets
fuse__value	number indicating state of fuse(s)

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

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

See also examples ORXOR.ABL and CNT10ROM.

Fuse values appearing on the right side of the = symbol can be any number. In the case of only a single fuse number

being specified on the left side of the = symbol, the least significant (LSB) bit of the fuse value is assigned to the fuse; a 0 indicates a fuse intact, and a 1 indicates a fuse blown. In the case of multiple fuse numbers, the fuse value is expanded to a binary number and truncated or given leading zeros to obtain fuse values for each fuse number.

When fuse states are specified using the FUSES-section syntax, the resulting fuse values supercede the fuse values obtained through the use of equations, truth tables and state diagrams, and will effect device simulation accordingly. While a high number of fuses can be specified in the FUSES section (typically over 2,000) there is some limit. The maximum number of fuses you can specify varies but is well beyond what is practical when using the FUSES section.

The PC/MS-DOS versions have a limit of 15 fuses **per statement** (e.g. FUSES[1000..1015] = 0;).

8.10 Test Vectors

TEST_VECTORS [IN device_id] [note] (inputs -> outputs)

[invalues -> outvalues;]

.
.
.

device_id an identifier indicating the device associated with this table

note a string used to describe the test vectors

inputs identifier or set of identifiers specifying the names of the input signals to the device, or feedback output signals

outputs identifier or set of identifiers specifying the output signals from the device

invalues input value or set of input values

outvalues output value or set of output values resulting from the given inputs

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs. Test vectors are used for simulation of an internal model of the device and functional testing of the real programmed device.

A special simulation utility, SIMULATE, is provided as part of the ABEL software package. SIMULATE simulates the operation of the device model by applying the inputs specified in the test vectors to the fuse states created by the language processor. SIMULATE is discussed further in chapter 4.

Functional testing of the real device is performed by a logic programmer after a device has been programmed. This can be done because the test vectors become part of the programmer load file that is loaded into the logic programmer.

Test vectors are written for each different device in the module, so that the different characteristics of each device can be taken into account separately during simulation.

Test vectors for a device are specified in a table. The table consists of a header and the vectors themselves. The header indicates that test vectors are to follow and defines the format of the table. The vectors specify the input-to-output function.

The form of the test vectors is determined by the header. Each vector is specified in the format described within the parentheses in the header statement. An optional note string can be specified in the header. This note string is often used to describe what the vectors test, and is included as output in the simulation output file, the document output file, and the JEDEC programmer load file.

The table lists input combinations and their resulting outputs. All or some of the possible input combinations can be listed.

All values specified in the table must be constants, either declared, numeric or the special constant, ".X.". Each line of the table (each input/output listing) must end with a semicolon.

Following is a simple test vectors table:

```
TEST_VECTORS
  ( [A,B] -> [C, D] )

    [0,0] -> [1,1] ;
    [0,1] -> [1,0] ;
    [1,0] -> [0,1] ;
    [1,1] -> [0,0] ;
```

The following test vector table is equivalent to the table specified above because values for sets can be specified with numeric constants.

```
TEST_VECTORS
  ( [A,B] -> [C,D] )

    0 -> 3 ;
    1 -> 2 ;
    2 -> 1 ;
    3 -> 0 ;
```

If the signal identifiers used in the test vector header were declared as active-low in the declaration section, then constant values specified in the test vectors will be inverted accordingly.

9. Directives

Directives provide many options that can affect the contents of a source file when it is processed. Sections of ABEL 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.

Directives are for the designer who understands the basics of ABEL and who wants to use more complex structures. Table 9-1 lists the available directives, which are described in detail in later subsections. Examples of directive usage to create test vectors are given in chapter 12.

Some of the directives take arguments that are used to determine conditions. When this is the case, the argument can be an actual argument, or it can be a dummy argument preceded by a question mark. The rules applying to actual and dummy arguments are presented in section 7.11.

When debugging your source file, you can use the -E and -P flags to examine statements. The -E parameter causes the parsed and expanded source code to be written to the listing file (or to the display if the -L flag is omitted). Text included by directives is written (or displayed). In addition to the listing information provided by the -E flag, the -P flag lists the directives that caused code to be added to the source file.

Table 9-1. Directives

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

9.1 @ALTERNATE Directive

@ALTERNATE

@ALTERNATE brings an alternate set of operators into effect that duplicate the normal ABEL 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.

The alternate operator set is listed in table 9-2.

Table 9-2. Alternate Operator Set

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

Note that the use of the alternate operator set precludes use of the ABEL addition, multiplication and division operators because they represent the OR, AND and NOT logical operators in the alternate set.

9.2 @CONST (Constant) Directive

@CONST id = expression ;

id a valid identifier

expression a valid expression

@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 so that they can define their own internal constants. Constants defined with @CONST override any previous constant declarations. Declaring an identifier as a constant in this manner constitutes an error if the identifier was used earlier in the source file as something other than a constant (i.e., a macro, pin, device).

Example:

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

9.3 @EXIT Directive

@EXIT

The @EXIT directive causes PARSE to abort processing of the source file with error bits set. (Error bits allow the operating system to determine that a processing error has occurred.)

9.4 @EXPR (Expression) Directive

@EXPR [block] expression ;

block a block

expression a valid expression

@EXPR evaluates the given expression, and converts it to a string of digits in the default base numbering system. This string and the block are then inserted into the source file at the point at which the @EXPR directive occurs. The expression must produce a valid number.

Example:

```
@expr {ABC} ^B11 ;
```

Assuming that the default base is base ten, this example causes the text ABC3 to be inserted into the source file.

9.5 @IF Directive

@IF expression block

expression	a valid expression that produces a numeric value
block	a valid block of text as described in section 2

@IF is used to include sections of ABEL source code based on the value resulting from an expression. If the expression is non-zero (logical true), the block of code is included as part of the source.

Dummy argument substitution is valid in the expression.

Example:

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


9.6 @IFB (If Blank) Directive

@IFB (arg) block

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

block a valid block of text

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

Examples:

```
@IFB ()  
{ text here will be included  
with the rest of the source file.  
}
```

```
@IFB ( hello )  
{ this text will not be included }
```

```
@IFB (?A)  
{ this text will be included if no value is substituted  
for A. }
```

9.7 @IFDEF (If Defined) Directive

@IFDEF id block

id an identifier

block a valid block of text

@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"
```

9.8 @IFIDEN (If Identical) Directive

@IFIDEN (arg1,arg2) block

arg1,2 either an actual argument, or a dummy
 argument name preceded by a "?"

block a valid block of text

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

Example:

```
@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.

9.9 @IFNB (If Not Blank) Directive

@IFNB (arg) block

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

block a valid block of text

@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 source here will not be included  
  with the rest of the source file.  
}  
@IFNB ( hello )  
{ this text will be included }  
@IFNB (?A)  
{ this text will be included if a value is  
  substituted for A}
```

9.10 @IFDEF (If Not Defined) Directive

@IFDEF id block

id an identifier

block a valid block of text

@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.

Example:

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

9.11 @IFNIDEN (If Not Identical) Directive

@IFNIDEN (arg1,arg2) block

arg1,2 either an actual argument, or a dummy argument name preceded by a "?"

block a valid block of text

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

Example:

```
@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.

9.12 @INCLUDE Directive

@INCLUDE filespec

filespec a string specifying the name of a file, where the specification follows the rules of the operating system being used

@INCLUDE causes the contents of the file identified by the file specification to be placed in the ABEL source file. The inclusion will begin at the location of the **@INCLUDE** directive. The file specification can include an explicit drive or path specification that indicates where the file is to be found. If no drive or path specification is given, the file is expected to be on either the default drive or path, or on the drive or path specified by the **-H** parameter.

Example:

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

9.13 @IRP (Indefinite Repeat) Directive

@IRP dummy_arg (arg [,arg]...) block

dummy_arg a dummy argument

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

block a block of text

@IRP causes the block to be repeated in the source file *n* times, where *n* equals the number of arguments contained in the parentheses. Each time the block is repeated, the dummy argument takes on the value of the next successive argument.

For example:

```
@IRP A (1, ^H0A,0)
{B = ?A ;
}
```

results in:

```
B = 1 ;
B = ^H0A ;
B = 0 ;
```

which is inserted into the source file at the location of the **@IRP** directive. Multiple assignments to the same identifier cause an implicit OR to occur (see section 2.8.6).

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.

9.14 @IRPC (Indefinite Repeat, Character) Directive

@IRPC dummy_arg (arg) block

dummy_arg a dummy argument

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

block a block

@IRPC causes the block to be repeated in the source file *n* times, where *n* equals the number of characters contained in **arg**. Each time the block is repeated, the dummy argument takes on the value of the next successive character.

For example:

```
@IRPC A (Cat)
{B = ?A ;
}
```

results in:

```
B = C ;
B = a ;
B = t ;
```

which is inserted into the source file at the location of the **@IRPC** directive.

9.15 @MESSAGE Directive

@MESSAGE

string string any valid string

@MESSAGE prints a message specified in string at the terminal. This can be used to monitor the progress of the PARSE step of the language processor.

Example:

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

9.16 @PAGE Directive

@PAGE

Send a form feed to the parser listing file. If no listing is being created, **@PAGE** has no effect.

9.17 @RADIX Directive

@RADIX *expr* ;

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

@RADIX is used to change the default base numbering system. The normal default is base 10 (decimal). This directive is useful when many numbers need to be specified in a base other than 10, say base 2. The **@RADIX** directive can be issued and all numbers that do not have their base explicitly stated are assumed to be in the new base, in this case, base 2. (See section 7.6.)

The newly specified default base stays in effect until another **@RADIX** directive is issued or until the end of the module is reached.

When the default base is set to 16, all numbers in that base that begin with an alphabetic character must begin with leading zeroes.

Examples:

```
@radix 2 ;           "change default base to binary
@radix 11011/11 + 1 ; "change back to decimal
```

9.18 @REPEAT Directive

@REPEAT *expr* *block*

expr a valid expression that produces a number

block a block

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

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 Applications Guide.

9.19 @STANDARD Directive

@STANDARD

@STANDARD switches the operators in effect back to the ABEL standard operators from the alternate set. The alternate set is chosen with the @ALTERNATE directive.

10. Design Examples

The logic designs and design features discussed in the following sections are contained on the Design Examples disk or tape you received with the ABEL package; and are listed in table 10-1.

You can process these design examples with ABEL, either as they stand, or with your own modifications, to create programmer load files. Many of the design examples listed in table 10-1 are described in detail within this manual, and the section reference is given for these design examples. Design examples not described in the manual, but listed in table 10-1, can be examined as the need arises.

Table 10-1 lists the design examples alphabetically by filename of the ABEL source file and contains columns for:

- **THE NAME OF THE SOURCE FILE.** The source file uses the .ABL file extension (e.g., ADD5 is ADD5.ABL). The ABEL batch file requires that the file extension not be entered for the source file. The command line for any of individual programs defaults to .ABL for a source file extension.
- **PROGRAMMABLE LOGIC DEVICE TYPE.** The type of programmable logic device type used in the design example is listed to assist you in locating meaningful design examples.
- **SECTION REFERENCE.** Where applicable, a reference to a specific section in this manual is given that provides some detailed information.
- **TYPE OF EXAMPLE.** A brief description of the design example.

Table 10-1. Design Examples Supplied with ABEL

Filename (*ABL)	Device Type	Section Refer- ence.	Type of Example
ADD5	F159	4.6	Simulation trace levels
ALARM	P16R4	N/A	Macros (state diagram)
BARREL	P20R8	10.6	Sets
BCD7PAL	P16L8	N/A	PALASM example for TOABEL
BCD7ROM	RA5P8	10.7	Truth table
BGATES	P12H6	N/A	ABEL version of PALASM basic gates
BGEQN	F100	N/A	@repeat directive used w/equations
BINBCD	P16L8	10.10	Truth table
BJACK	P16R6	10.10	State machine
BUFFER	F153	10.9	Three-state output control
COMP4A	F153	10.8	Relational operators
COUNT4	P16R4	10.4	Equations
COUNT4A	P16R4	10.4	Multiple equations for one output
DECADE	F105	13.5	Transition equations complement array
DEMO1800	E1800		Macro cell control
DMUX1T8	P16L8	10.3	Sets
FEEDBACK	P16R4	4.6	Simulator example
FPLS	F157	N/A	Logic sequencer features
GALS	16U8		Selection of GAL models
GATES	P12H6	N/A	PALASM example for TOABEL
LCUCROM	RA8P8	N/A	Macros, compiler directives
MACROCEL	E0310	13.2	Output macro cells
M6809A	P14L4	10.1	Sets, relational operators
M6809B	P14L4	12.5	Macros with test vectors
M6809C	P14L4	12.5	Macros with test vectors
M6809D	P16L8	11.3	Device type from command line

Table 1-1. (cont.)

Filename (* .ABL)	Device Type	Section Refer- ence.	Type of Example
M6809ER	P14L4	4.6	Demonstrates simulation error
M6809ERR	P14L4	4.2	Demonstrates syntax error
MUXADD	P22V10	10.10	Ripple adder
MUX12T4	P14H4	10.2	Sets
OCTAL	P20X8	N/A	Exclusive OR equations
REGFB	P16R4	4.6	Simulator example
SHIFT40	32R16	N/A	84-pin PAL example
TI1SHOT	P19R8	N/A	Input registers
TRAFFIC	F167	N/A	State machine RS flip/flops
SEQUENCE	P16R4	10.5	State Diagram
SHIFTCNT	F159	13.2	Controls JK & D flip-flops

*This table represents a partial list of design examples available at the time of printing. To verify the design examples provided on your ABEL disk or tape, use the operating system to print the file named EXAMPLES.TXT.

The following logic design examples are believed to be representative of typical programmable logic applications. These serve to illustrate significant ABEL features. Use them to learn more about ABEL and to start designing with ABEL. We encourage you to use these examples directly by modifying them to suit your needs, or by incorporating them into larger system designs.

The titles of the designs indicate the logic design problem being solved, the major features of ABEL used to solve the problem, and the type of programmable logic device used. Each design has accompanying block diagrams and source file listings.

All the examples presented in this section are included on your ABEL distribution disk or tape. Additional examples not explained here are also included. 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; and also in table 1-1. You can use the operating system to display the list of design examples contained in EXAMPLES.TXT.

10.1 6809 Memory Address Decoder (Equations P14L4)

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL implementation of such a design.

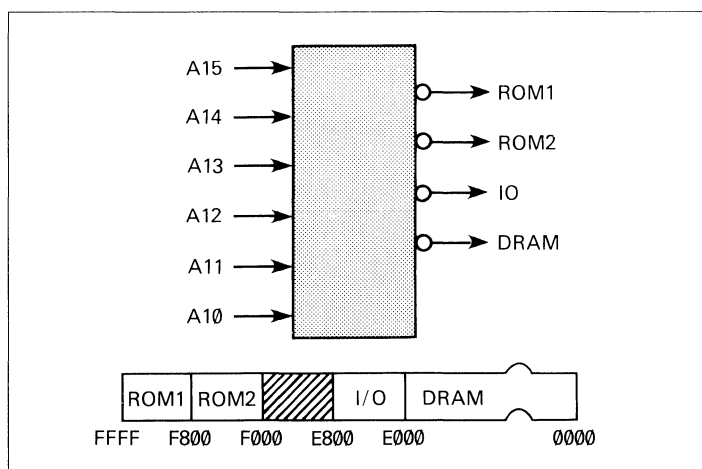


Figure 10-1. Block Diagram: 6809 Memory Address Decoder

10.1.1 Design Specification

Figure 10-1 shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (*DRAM*), I/O (*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.

Table 10-2 shows the address ranges associated with each section of memory. These address ranges can also be seen in figure 10-1.

Table 10-2. Address Ranges for the 6809 Controller

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

10.1.2 Design Method

Figure 10-2 shows a simplified block diagram for the address decoder. The address decoder is implemented with simple Boolean equations employing both relational and logical operators as shown in listing 10-1. A significant amount of simplification is achieved by grouping the address bits into a set named *Address*. The lower-order ten address bits that are not used for the address decode are given "don't care" values in the address set. In this way, the designer indicates that the address in the overall design (that beyond the decoder) contains sixteen bits, but that bits 0-9 do not affect the decode of that address. This is opposed to simply defining the set as

$$\text{Address} = [\text{A15}, \text{A14}, \text{A13}, \text{A12}, \text{A11}, \text{A10}]$$

which ignores the existence of the lower-order bits. Specifying all 16 address lines as members of the address set also allows full 16-bit comparisons of the address value against the ranges shown in table 10-2.

```

module m6809a
title '6809 memory decode
Jean Designer      Data I/O      Redmond WA    24 Feb 1987'

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

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

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

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

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

!ROM1  = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
^h0000 -> [ H,  H,  H,  L ];
^h4000 -> [ H,  H,  H,  L ];
^h8000 -> [ H,  H,  H,  L ];
^hc000 -> [ H,  H,  H,  L ];
^hE000 -> [ H,  H,  L,  H ];
^hE800 -> [ H,  H,  H,  H ];
^hF000 -> [ H,  L,  H,  H ];
^hF800 -> [ L,  H,  H,  H ];

end m6809a

```

Listing 10-1. Source File: 6809 Memory Address Decoder

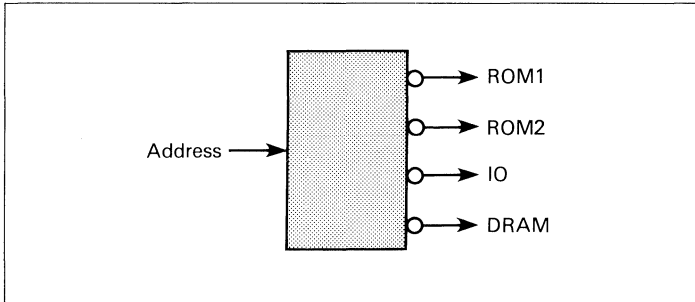


Figure 10-2. Simplified Block Diagram: 6809 Memory Address Decoder

10.1.3 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.

10.2 12 to 4 Multiplexer (Equations/P14H4)

Following is a description of a 12-input to 4-output multiplexer implemented with one Boolean equation.

10.2.1 Design Specification

Figure 10-3 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.

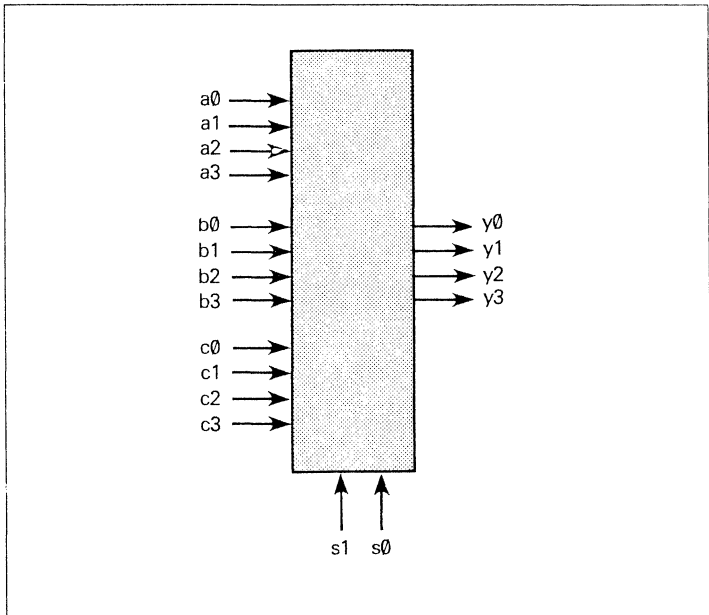


Figure 10-3. Block Diagram: 12 to 4 Multiplexer

10.2.2 Design Method

Figure 10-4 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 listing 10-2.

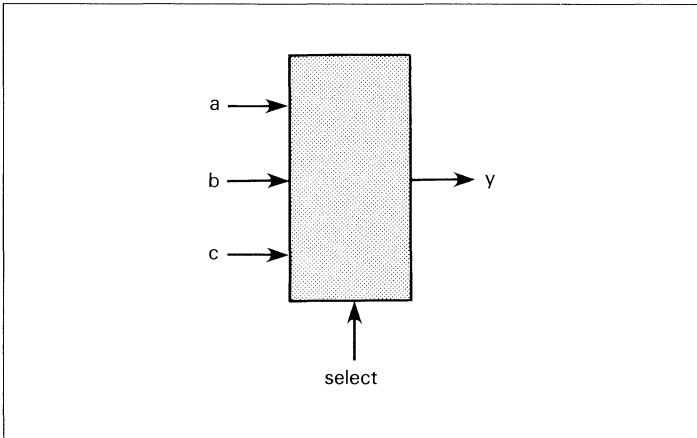


Figure 10-4. Simplified Block Diagram: 12 to 4 Multiplexer

Once the sets have been declared, specification of the design is made by means of the following four equations that use WHILE-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*. 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.

10.2.3 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.

```
module _mux12t4
title '12 to 4 multiplexer  11 Nov 1987
Charles Olivier & Dave Pellerin  Data I/O Corp.  Redmond WA'
```

```
    mux12t4      device  'P14H4';

    a0,a1,a2,a3  pin     1,2,3,4;
    b0,b1,b2,b3  pin     5,6,7,8;
    c0,c1,c2,c3  pin     9,11,12,13;
    s1,s0        pin     18,19;
    y0,y1,y2,y3  pin     14,15,16,17;

    H      =      [1,1,1,1];
    L      =      [0,0,0,0];
    X      =      .x.;
    select  =      [s1, s0];
    y       =      [y3,y2,y1,y0];
    a       =      [a3,a2,a1,a0];
    b       =      [b3,b2,b1,b0];
    c       =      [c3,c2,c1,c0];
```

equations

```
    when (select == 0) then y = a;
    when (select == 1) then y = b;
    when (select == 2) then y = c;
    when (select == 3) then y = c;
```

test_vectors

```
    ([select, a, b, c] -> y)
    [0      , 1, X, X] -> 1; "select = 0, gates lines a to output
    [0      ,10, H, L] -> 10;
    [0      , 5, H, L] -> 5;

    [1      , H, 3, H] -> 3; "select = 1, gates lines b to output
    [1      ,10, 7, H] -> 7;
    [1      , L,15, L] -> 15;

    [2      , L, L, 8] -> 8; "select = 2, gates lines c to output
    [2      , H, H, 9] -> 9;
    [2      , L, L, 1] -> 1;

    [3      , H, H, 0] -> 0; "select = 3, gates lines c to output
    [3      , L, L, 9] -> 9;
    [3      , H, L, 0] -> 0;
```

```
end _mux12t4
```

Listing 10-2. Source File: 12 to 4 Multiplexer

10.3 1 to 8 Demultiplexer (Equations/P16L8)

Following is the design for a simple one line to eight line demultiplexer with an enable. The design is implemented with Boolean equations and a P16L8 PAL.

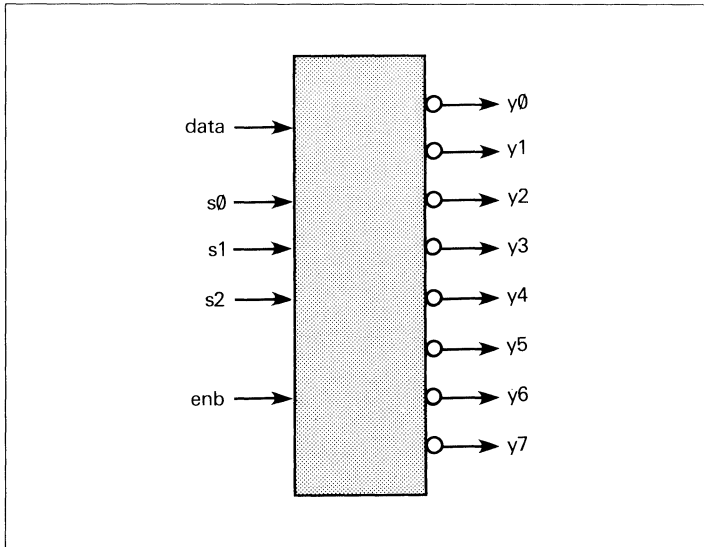


Figure 10-5. Block Diagram: 1 to 8 Demultiplexer

10.3.1 Design Specification

Figure 10-5 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 three bit binary number whose value ranges from 0 to 7, thus selecting one

of the outputs. *enb* is an enable line. When *enb* is low (0), the outputs go to high impedance. When *enb* is high, the outputs are determined by the demultiplexing function.

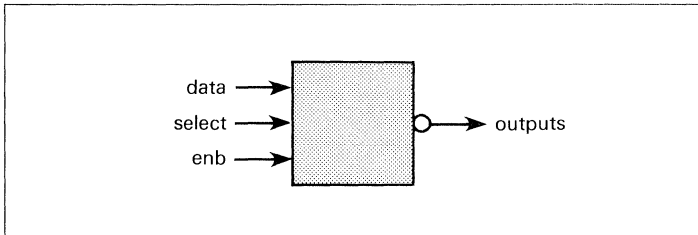


Figure 10-6. Simplified Block Diagram: Demultiplexer

10.3.2 Design Method

Figure 10-6 shows a simplified block diagram of the demultiplexer. The select lines and outputs are collected into sets named *select* and *outputs* to simplify the Boolean equations with which the demultiplexing function is described. The ABEL description of this logic is shown in listing 10-3. Each output is determined by ANDing the data line with a relational expression that checks for equivalence between the select lines and one of the eight possible select values. Because *select* can have only one value at any given time, only one of the outputs is selected for an AND with the data value. The selected output shows the inverse of the data line.

The enable function is implemented with a separate equation using the set *outputs* along with the *ENABLE* keyword. Thus the equation,

```
enable outputs = enb ;
```

assigns the enables on each of the output lines to the input, *enb*.

10.3.3 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.

```

module dmux1t8
title '1 to 8 line demultiplexer
Mark Kuenster    FurureNet Division, Data I/O Corp.    19 Nov 1987'

        DM1      device  'P16L8';

        y0,y1,y2,y3,y4,y5,y6,y7 pin    12,13,14,15,16,17,18,19;
        s0,s1,s2,data,enb      pin    1,2,3,4,5;

        H,L,Z    =      1,0,.Z.;
        select    =      [s2, s1, s0];
        outputs    =      [y7, y6, y5, y4, y3, y2, y1, y0];

equations
!y0    = (select == 0) & data;
!y1    = (select == 1) & data;
!y2    = (select == 2) & data;
!y3    = (select == 3) & data;
!y4    = (select == 4) & data;
!y5    = (select == 5) & data;
!y6    = (select == 6) & data;
!y7    = (select == 7) & data;

enable outputs = 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, L]; "Select y0
[ H ,  1 , H ] -> [H, H, H, H, H, H, L, H]; "Select y1
[ H ,  2 , H ] -> [H, H, H, H, H, L, H, H]; "Select y2
[ H ,  3 , H ] -> [H, H, H, H, L, H, H, H]; "Select y3
[ H ,  4 , H ] -> [H, H, H, L, H, H, H, H]; "Select y4
[ H ,  5 , H ] -> [H, H, L, H, H, H, H, H]; "Select y5
[ H ,  6 , H ] -> [H, L, H, H, H, H, H, H]; "Select y6
[ H ,  7 , H ] -> [L, H, H, H, H, H, H, H]; "Select y7
[ L ,  0 , H ] -> [Z, Z, Z, Z, Z, Z, Z, Z];

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 dmux1t8

```

Listing 10-3. 1 to 8 Demultiplexer

10.4 4-Bit Counter/Multiplexer (Equations/P16R4)

Following is a design for a 4-bit synchronous counter implemented with a P16R4. Counter features include carry-in and carry-out, 2 input multiplexing and a hold state. The counter is described by Boolean equations.

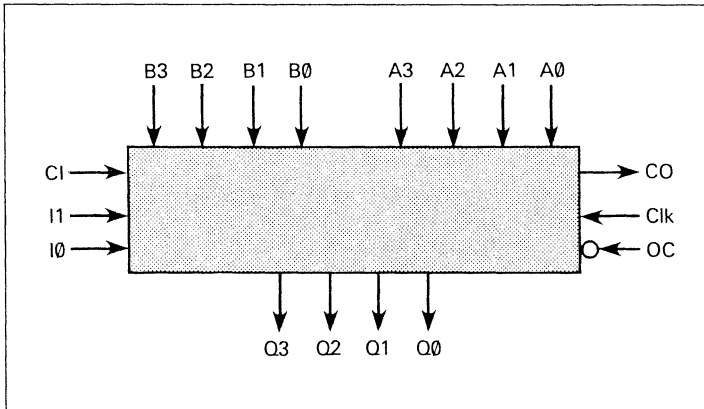


Figure 10-7. Block Diagram: 4-Bit Counter With 2 Input Multiplexer

10.4.1 Design Specification

Figure 10-7 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 table 10-3. 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 carry-out 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).

Table 10-3. Counter Modes

Mode	I1	I0	Description
Hold	0	0	count remains unchanged
Load A	0	1	load A0-A3 into the count registers
Load B	1	0	load B0-B3 into the count registers
Increment	1	1	increment the count by 1 on clock pulse

The carry-in and carry-out lines operate such that two or more of the counters can be chained together to form a wider counter. To do this, the carry-out 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 carry-out bit is one, causing the next counter's increment to be one.

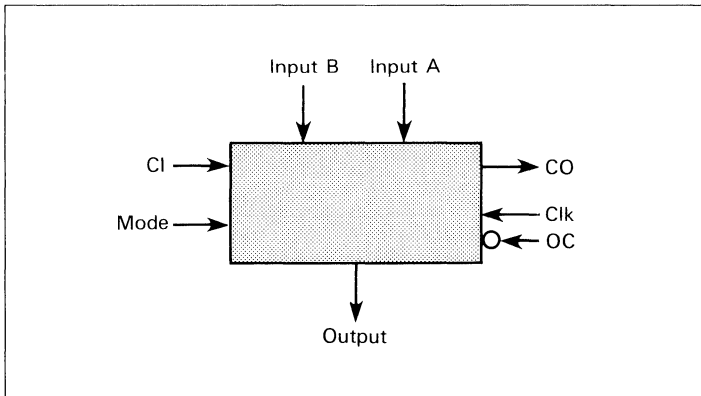


Figure 10-8. Simplified Block Diagram: 4 -Bit Counter With 2 Input Multiplexer

10.4.2 Design Method

The counter is described with Boolean equations. Figure 10-8 shows the simplified block diagram corresponding to the ABEL implementation of the counter design. Listing 10-4 shows the source file.

The design is simplified by grouping the input bits, output bits and mode selectors into sets, so that they can be referenced by name. The inputs *A0-A3* are assigned to the set *InputA*, the inputs *B0-B3* to the set *InputB*, the outputs *Q0-Q3* and *CO* to the set *Output*, and the mode selectors *I0* and *I1* to the set *Mode*.

Notation in the source file can be further simplified by some simple constant assignments. *H* represents a 1, *L* represents a 0, *X* represents the special constant *.X.*, 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 *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 Boolean equations for a 4-bit adder with carry-in, carry-out and multiplexed load.

10.4.3 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 listing 10-4 test for proper loading of *InputA* and *InputB*, for increments after loads, for hold states, for correct operation of the carry-out, 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) to observe the outputs. The test vectors in listing 10-4 include the output enable pin (OC).

module count4
 title '4-bit counter with 2 input mux 19 Nov 1987
 based on an example by Birkner/Coli in the MMI PAL Handbook
 Dan Burrier & Mike McGee FutureNet Division, Data I/O Corp.'

```
P7022    device 'P16R4';

Clk,OC,CO,I1,I0,CI    pin   1,11,12,13,18,19;
A0,A1,A2,A3,B0,B1,B2,B3 pin   2,3,4,5,6,7,8,9;
Q3,Q2,Q1,Q0           pin   14,15,16,17;
H,L,X,Z,C            = 1,0, .X.,.Z.,.C.;
InputA               = [A3,A2,A1,A0];
InputB               = [B3,B2,B1,B0];
Output                = [CO,Q3,Q2,Q1,Q0];
Mode                  = [I1,I0];
Hold,LoadA,LoadB,Incr = 0,1,2,3;            " define Modes
```

equations

```
!Q0        := (Mode==Hold ) & !Q0
            # (Mode==LoadA) & !A0
            # (Mode==LoadB) & !B0
            # (Mode==Incr ) & !CI & !Q0                "Hold if no carry
            # (Mode==Incr ) & CI & Q0 ;

!Q1        := (Mode==Hold ) & !Q1
            # (Mode==LoadA) & !A1
            # (Mode==LoadB) & !B1
            # (Mode==Incr ) & !CI & !Q1                "Hold if no carry
            # (Mode==Incr ) & !Q0 & !Q1                "Hold if Q0=L
            # (Mode==Incr ) & CI & Q0 & Q1 ;

!Q2        := (Mode==Hold ) & !Q2
            # (Mode==LoadA) & !A2
            # (Mode==LoadB) & !B2
            # (Mode==Incr ) & !CI & !Q2                "Hold if no carry
            # (Mode==Incr ) & !Q0 & !Q2                "Hold if Q0=L
            # (Mode==Incr ) & !Q1 & !Q2                "Hold if Q1=L
            # (Mode==Incr ) & CI & Q0 & Q1 & Q2 ;

!Q3        := (Mode==Hold ) & !Q3
            # (Mode==LoadA) & !A3
            # (Mode==LoadB) & !B3
            # (Mode==Incr ) & !CI & !Q3                "Hold if no carry
            # (Mode==Incr ) & !Q0 & !Q3                "Hold if Q0=L
            # (Mode==Incr ) & !Q1 & !Q3                "Hold if Q1=L
            # (Mode==Incr ) & !Q2 & !Q3                "Hold if Q2=L
            # (Mode==Incr ) & CI & Q0 & Q1 & Q2 & Q3 ;

!CO        = !CI # !Q0 # !Q1 # !Q2 # !Q3 ;
```

Listing 10-4. Source file: 4-bit Counter with 2 Input Mux
(continued on next page)

```

@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

```

Listing 10-4. (continued)

10.4.4 Multiple Assignments 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.

Listing 10-5 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.

This is the same 4-bit counter described in section 10.4, but in that section the count and multiplexing functions are described together in one equations section. The function of each counter is identical, and the same equations are produced regardless of the method of description chosen.

```

module count4a
title '4-bit counter with 2 input mux          19 Nov 1987
based on an example by Birkner/Coli in the MMI PAL Handbook
Dan Burrier & Mike McGee   FutureNet Division, Data I/O Corp.'

```

```

P7022A    device 'P16R4';

Clk,OC,CO,I1,I0,CI    pin  1,11,12,13,18,19;
A0,A1,A2,A3,B0,B1,B2,B3 pin  2,3,4,5,6,7,8,9;
Q3,Q2,Q1,Q0          pin  14,15,16,17;

H,L,X,Z,C              = 1,0, .X.,.Z.,.C.;
InputA                 = [A3,A2,A1,A0];
InputB                 = [B3,B2,B1,B0];
Output                 = [CO,Q3,Q2,Q1,Q0];
Mode                   = [I1,I0];
Hold,LoadA,LoadB,Incr  = 0,1,2,3;      " define Modes

```

```

equations      " input multiplexer

```

```

!Q0    := (Mode==Hold ) & !Q0
        # (Mode==LoadA) & !A0
        # (Mode==LoadB) & !B0;

!Q1    := (Mode==Hold ) & !Q1
        # (Mode==LoadA) & !A1
        # (Mode==LoadB) & !B1;

!Q2    := (Mode==Hold ) & !Q2
        # (Mode==LoadA) & !A2
        # (Mode==LoadB) & !B2;

!Q3    := (Mode==Hold ) & !Q3
        # (Mode==LoadA) & !A3
        # (Mode==LoadB) & !B3;

```

```

" 4 bit counter

```

```

!Q0    := (Mode==Incr ) & !CI & !Q0          "Hold if no carry
        # (Mode==Incr ) & CI & Q0 ;

!Q1    := (Mode==Incr ) & !CI & !Q1          "Hold if no carry
        # (Mode==Incr ) & !Q0 & !Q1         "Hold if Q0=L
        # (Mode==Incr ) & CI & Q0 & Q1 ;

!Q2    := (Mode==Incr ) & !CI & !Q2          "Hold if no carry
        # (Mode==Incr ) & !Q0 & !Q2         "Hold if Q0=L
        # (Mode==Incr ) & !Q1 & !Q2         "Hold if Q1=L
        # (Mode==Incr ) & CI & Q0 & Q1 & Q2 ;

```

Listing 10-5. Multiple Equations Sections, 4-Bit Counter
(continued on next page)


```

@page
!Q3      := (Mode==Incr ) & !CI & !Q3      "Hold if no carry
          # (Mode==Incr ) & !Q0 & !Q3      "Hold if Q0=L
          # (Mode==Incr ) & !Q1 & !Q3      "Hold if Q1=L
          # (Mode==Incr ) & !Q2 & !Q3      "Hold if Q2=L
          # (Mode==Incr ) & CI & Q0 & Q1 & Q2 & Q3 ;

!CO      = !CI # !Q0 # !Q1 # !Q2 # !Q3 ;

test_vectors ' test Load A and B'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
  [ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;
  [ C, L, LoadA, ^h1 , ^h7 ,X ] -> ^h1;
  [ C, L, LoadB, ^h1 , ^h7 ,X ] -> ^h7;
  [ C, L, LoadA, ^h2 , ^hB ,X ] -> ^h2;
  [ C, L, LoadB, ^h2 , ^hB ,X ] -> ^hB;
  [ C, L, LoadA, ^h4 , ^hD ,X ] -> ^h4;
  [ C, L, LoadB, ^h4 , ^hD ,X ] -> ^hD;
  [ C, L, LoadA, ^h8 , ^hE ,X ] -> ^h8;
  [ C, L, LoadB, ^h8 , ^hE ,X ] -> ^hE;
  [ C, L, LoadA, ^h0 , ^hF ,X ] -> ^h0;
  [ C, L, LoadB, ^h0 , ^hF ,L ] -> ^hF;

test_vectors ' test increment'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, LoadB, X , ^h1 ,X ] -> ^h1;
  [ C, L, Incr , X , X ,H ] -> ^h2;
  [ C, L, LoadB, X , ^h3 ,X ] -> ^h3;
  [ C, L, Incr , X , X ,H ] -> ^h4;
  [ C, L, LoadA, ^h7 , X ,X ] -> ^h7;
  [ C, L, Incr , X , X ,H ] -> ^h8;
  [ C, L, LoadA, ^hF , X ,L ] -> ^hF;
  [ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
  [ C, L, LoadB, X , ^hC ,X ] -> ^hC;
  [ C, L, Incr , X , X ,H ] -> ^hD;
  [ C, L, Hold , X , X ,H ] -> ^hD;

test_vectors ' test carry'
  ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
  [ C, L, Incr , X , X ,H ] -> ^hE;
  [ C, L, Incr , X , X ,H ] -> ^h1F;      "carry out
  [ C, L, Incr , X , X ,H ] -> ^h0;      "roll over
  [ C, L, Incr , X , X ,H ] -> ^h1;
  [ C, L, Incr , X , X ,L ] -> ^h1;      "no carry in
  [ C, L, Incr , X , X ,H ] -> ^h2;
  [ L, H, Hold , X , X ,X ] -> [X,Z,Z,Z,Z];
end count4a

```

Listing 10-5. (continued)

10.5 Three-State Sequencer (State Diagram/16R4)

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 exact limit on number of State Diagram states that can be processed by ABEL, but 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.

10.5.1 Design Specification

Figure 10-9 shows the sequencer design, with a bubble 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.

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.

10.5.2 Design Method

The sequencer is described by using a `STATE_DIAGRAM` section in the ABEL source file. Listing 10-6 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. The FLAG statement is used to select level 3 reduction. 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.

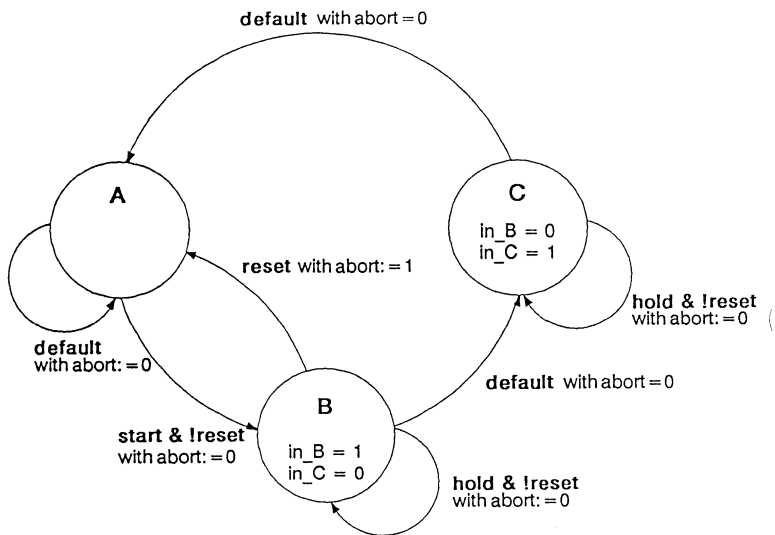


Figure 10-9. State Machine Bubble Diagram

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 it's previous value if the machine stays in state A.

10.5.3 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.

```

module sequence          flag '-r3'
title 'State machine example'      D. B. Pellerin - FutureNet';

    d1      device 'p16r4';

    q1,q0
    clock,enab,start,hold,reset    pin    14,15;
                                   pin    1,11,4,2,3;
    abort                                   pin    17;
    in_B,in_C                             pin    12,13;
    sreg                                   =    [q1,q0];

    A = 0;          B = 1;          C = 2;  "State Values

state_diagram sreg;
    State A:          " Hold in state A until start is active
        in_B = 0;
        in_C = 0;
        IF (start & !reset) THEN B WITH abort := 0;
        ELSE A WITH abort := abort;

    State B:          " Advance to state C unless reset
        in_B = 1;          " or hold is active. Turn on abort
        in_C = 0;          " indicator if reset.
        IF (reset) THEN A WITH abort := 1;
        ELSE IF (hold) THEN B WITH abort := 0;
        ELSE C WITH abort := 0;

    State C:          " Go back to A unless hold is active
        in_B = 0;          " Reset overrides hold.
        in_C = 1;
        IF (hold & !reset) THEN C WITH abort := 0;
        ELSE A WITH abort := 0;

test_vectors([clock,enab,start,reset,hold]->[sreg,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

```

Listing 10-6. Source File: Three-State Sequencer

10.6 8-Bit Barrel Shifter (Equations/P20R8)

This design for an 8-bit barrel shifter includes a shift amount selector, an output control, and a device enable. The design is specified with one Boolean equation for a P20R8.

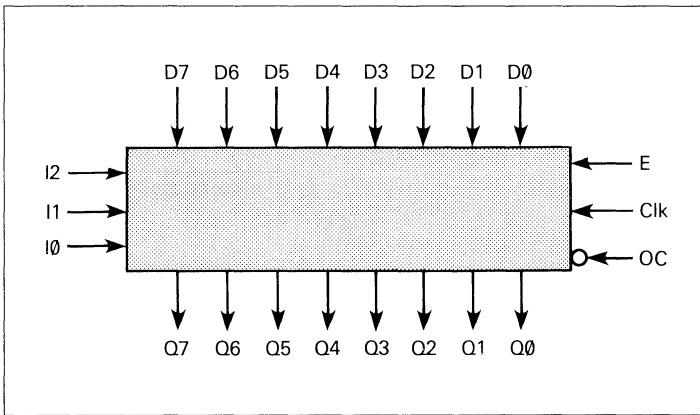


Figure 10-10. Block Diagram: 8-Bit Barrel Shifter

10.6.1 Design Specification

Figure 10-10 shows a block diagram for the barrel shifter. The shifter has eight inputs ($D0-D7$), eight outputs ($Q0-Q7$), three select lines ($I0-I2$), a clock (Clk), an output control (OC), and an enable (E). On each clock pulse when E is high, the outputs show the inputs shifted by n bits to the right, where n is specified by the select lines. The bit shifted out of the barrel shifter on the right is shifted in on the left, actually performing a rotate. When E is low, the shifter outputs are preset to 1.

The output control, when high, sets all outputs to high impedance, without affecting the shift. This means that if a shift is selected while the output control is high, the shift still occurs, but it is not seen at the outputs. If the *OC* is then set low, the shifted data will appear on the outputs.

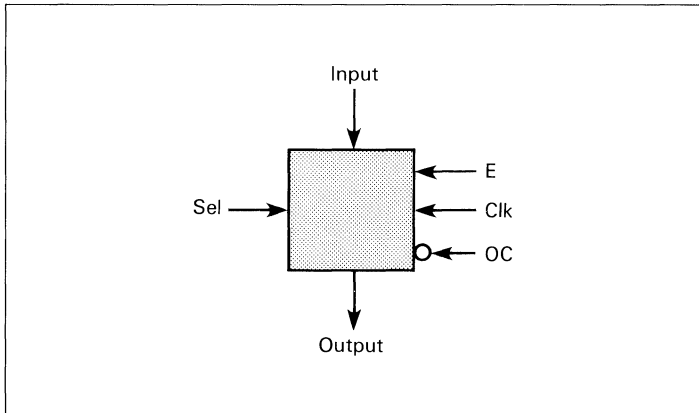


Figure 10-11. Simplified Block Diagram: 8-Bit Barrel Shifter

10.6.2 Design Method

Figure 10-11 and listing 10-7 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 Boolean 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,

$$(\text{Sel} == 0) \& \![D7,D6,D5,D4,D3,D2,D1,D0]$$

defines that for a shift of 0, the inputs are transferred without a shift directly to the outputs. Similarly, the product term,

$$(\text{Sel} == 5) \& \![D4,D3,D2,D1,D0,D7,D6,D5]$$

defines that for a shift of 5, output $Q7$ gets the value of input $D4$, $Q6$ gets the value of $D3$, and so on, corresponding to the correct shift of 5 places. Notice that the low-order input bits have been "wrapped around," shifted out of the right side and into the left side.

Sel can have only one value at a time, thus only one of the " $\text{Sel} ==$ " 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. The inverse of the inputs is available on the device.

10.6.3 Test Vectors

The test vectors are written to 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, as a further test, 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 that the wrap-around 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. That is, the *Z* becomes [*Z,Z,Z,Z,Z,Z,Z,Z*] to cover all eight outputs.

```

module barrel
title '8-bit barrel shifter
Gerrit Barrere    Data I/O Corp  Redmond WA    17 Oct 1987'

P7095    device    'P20R8';

D7,D6,D5,D4,D3,D2,D1,D0      Pin 2,3,4,5,6,7,8,9;
Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0      Pin 15,16,17,18,19,20,21,22;
Clk,OC,E,I2,I1,I0            Pin 1,13,23,10,11,14;

Input      = [D7,D6,D5,D4,D3,D2,D1,D0];
Output     = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
Sel        = [I2,I1,I0];
H,L,C,Z    = 1,0, .C., .Z.;

equations
    Enable Output = !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

```

Listing 10-7. Source File: 8-Bit Barrel Shifter

10.7 7-Segment Display Decoder (Truth Table/RA5P8)

This display decoder decodes a four-bit binary number to display the decimal equivalent on a seven segment LED display. The design incorporates a truth table.

10.7.1 Design Specification

Figure 10-12 shows a block diagram for the design of a 7-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.

10.7.2 Design Method

Figure 10-13 and listing 10-8 show the simplified block diagram and the source file for the ABEL implementation of the display decoder. The FLAG statement is used to make sure that the programmer load file is in the Motorola Exorciser format. The binary inputs and the decoded outputs are grouped into the sets *bcd* and *led* to simplify notation. The constants ON and OFF are declared so that the design can be described in terms of turning a segment on or off. To turn a segment on, the appropriate line must be driven low, thus we declare ON as 0 and OFF as 1.

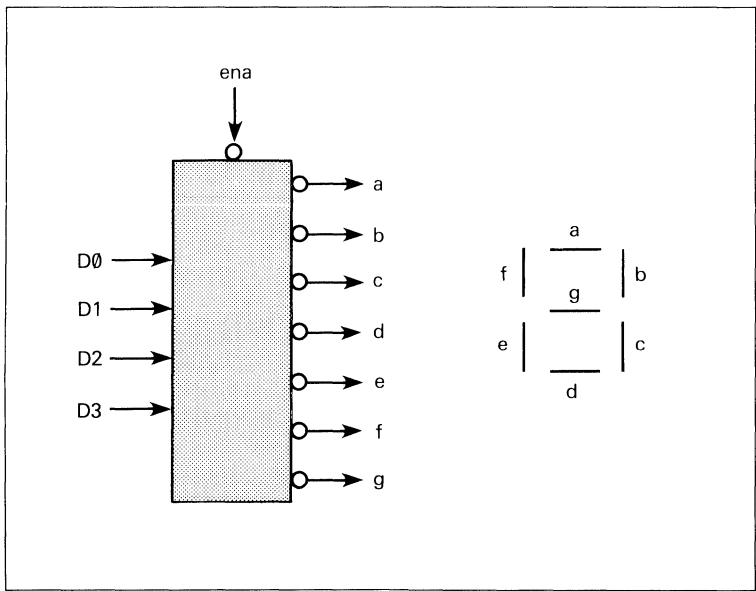


Figure 10-12. Block Diagram: 7-Segment Display Decoder

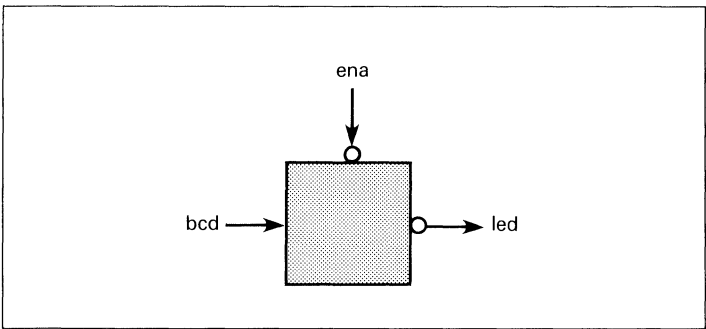


Figure 10-13. Simplified Block Diagram: 7-Segment Display Decoder

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

```
module bcd7rom flag '-d82'
title 'seven segment display decoder    16 Mar 1987
Walter Bright  FutureNet Division Data I/O Corp  Redmond WA'
"
"      a
"      ---      BCD-to-seven-segment decoder similar to the 7449
"      f| g |b
"      ---      segment identification
"      e| d |c
"      ---
"      U6      device  'RA5P8';

      D3,D2,D1,D0      pin 10,11,12,13;
      a,b,c,d,e,f,g    pin 1,2,3,4,5,6,7;
      ena              pin 15;

      bcd      = [D3,D2,D1,D0];
      led      = [a,b,c,d,e,f,g];
      ON,OFF   = 0,1;
      L,H,X,Z  = 0,1,.X,.Z.;

      " for common anode LEDs

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

test_vectors ([ena,bcd] -> led)
"
      a      b      c      d      e      f      g
[L,1] -> [OFF,  ON,  ON,  OFF, OFF, OFF,  OFF];
[L,2] -> [ ON,  ON,  OFF, ON,  ON, OFF,  ON];
[L,3] -> [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
[L,4] -> [OFF,  ON,  ON, OFF, OFF, ON,  ON];
[L,5] -> [ ON,  OFF, ON,  ON, OFF, ON,  ON];
[L,6] -> [ ON,  OFF, ON,  ON, ON,  ON,  ON];
[L,7] -> [ ON,  ON,  ON, OFF, OFF, OFF,  OFF];
[L,8] -> [ ON,  ON,  ON, ON,  ON,  ON,  ON];
[L,9] -> [ ON,  ON,  ON, ON,  OFF, ON,  ON];
[L,0] -> [ ON,  ON,  ON, ON,  ON,  ON,  OFF];
[H,5] -> [ Z,   Z,   Z,   Z,   Z,   Z,   Z];
end      bcd7rom
```

Listing 10-8. Source File: 7-Segment Display Decoder

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.

10.7.3 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.

10.8 4-Bit Comparator (Macros, Equations/F153)

This is a design for a 4-bit comparator that provides an output for "equal to", "less than", "not equal to", and "greater than", as well as intermediate outputs. The design is implemented with Boolean equations.

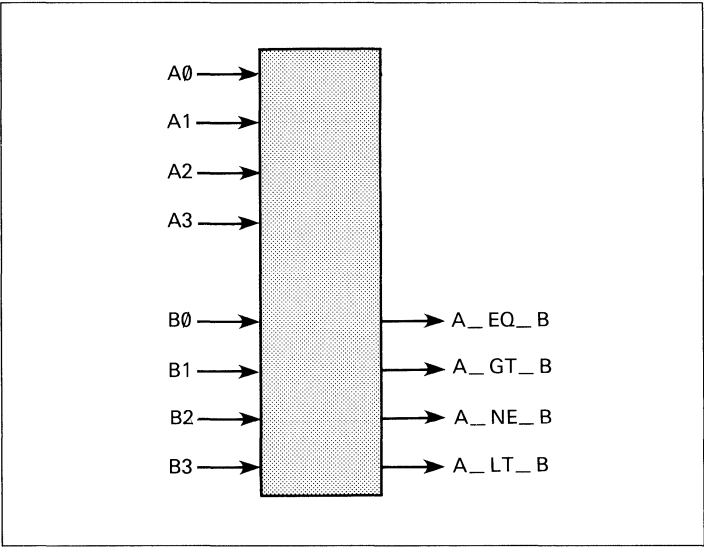


Figure 10-14. Block Diagram: 4-Bit Comparator

10.8.1 Design Specification

The comparator, as shown in figure 10-14, compares the values of two four-bit inputs ($A0-A3$ and $B0-B3$) and determines whether A is equal to, not equal to, less than, or greater than B . The result of the comparison is shown on the output lines, A_EQ_B , A_GT_B , A_NE_B , and A_LT_B .

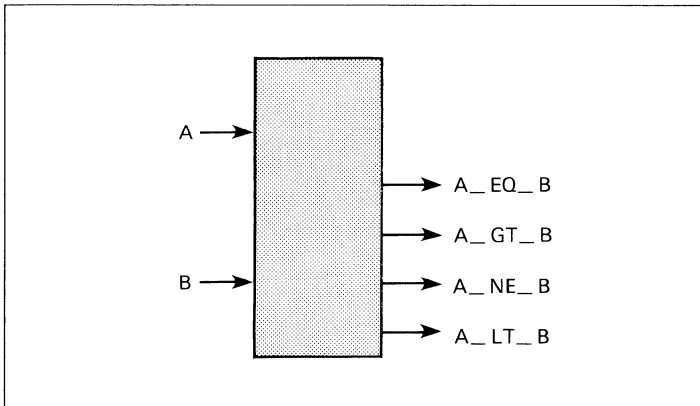


Figure 10-15. Simplified Block Diagram: 4-bit Comparator

10.8.2 Design Method

Figure 10-15 and listing 10-9 show the simplified block diagram and source file listing for the comparator. The inputs *A0-A3* and *B0-B3* are grouped into the sets *A* and *B*. *YES* and *NO* are defined as 1 and 0, to be used in the test vectors.

The equations section of the source file contains the following equations:

```
A_EQ_B = A == B;  
A_NE_B = !(A == B);  
A_GT_B = A > B;  
A_LT_B = !((A > B) # (A == B));
```

You could also use the following equations for the design of this comparator, however, many more product terms are used in the FPLA:

```
A_EQ_B = A == B;  
A_NE_B = A != B;  
A_GT_B = A > B;  
A_LT_B = A < B;
```

The first set of equations takes advantage of product term sharing within the target FPLA, while the latter set requires a different set of product terms for each equation. For example, the equation

$$A_NE_B = \neg(A == B);$$

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

$$A_LT_B = \neg((A > B) \# (A == B));$$

uses the same product terms as equations

$$\begin{aligned} A_EQ_B &= A == B; \\ A_GT_B &= A > B; \end{aligned}$$

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.

```

module_comp4a flag '-r4'
title '4-bit look-ahead comparator' 16 Nov 1987
Steve Weil & Gary Thomas FutureNet Division, Data I/O Corp.'

    comp4a device 'F153';

    A3,A2,A1,A0    pin 1,2,3,4;  A = [A3,A2,A1,A0];
    B3,B2,B1,B0    pin 5,6,7,8;  B = [B3,B2,B1,B0];

    A_NE_B,A_EQ_B,A_GT_B,A_LT_B    pin 14,15,16,17;

    No,Yes  = 0,1;

equations
    A_EQ_B  =  A == B;

    A_NE_B  =  !(A == B);

    A_GT_B  =  A > B;

    A_LT_B  =  !((A > B) # (A == B));

test_vectors 'test for A = B'
    ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
    [ 0, 0] -> [ Yes , No , No , No ];
    [ 1, 1] -> [ Yes , No , No , No ];
    [ 2, 2] -> [ Yes , No , No , No ];
    [ 5, 5] -> [ Yes , No , No , No ];
    [ 8, 8] -> [ Yes , No , No , No ];
    [10,10] -> [ Yes , No , No , No ];
    [15,15] -> [ Yes , No , No , No ];

test_vectors 'test for A > B'
    ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
    [ 1, 0] -> [ No , Yes , No , Yes ];
    [ 2, 1] -> [ No , Yes , No , Yes ];
    [ 4, 3] -> [ No , Yes , No , Yes ];
    [ 8, 7] -> [ No , Yes , No , Yes ];
    [15,14] -> [ No , Yes , No , Yes ];
    [ 6, 2] -> [ No , Yes , No , Yes ];
    [ 5, 0] -> [ No , Yes , No , Yes ];

test_vectors 'test for A < B'
    ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
    [ 3, 9] -> [ No , No , Yes , Yes ];
    [14,15] -> [ No , No , Yes , Yes ];
    [ 7, 8] -> [ No , No , Yes , Yes ];
    [ 3, 4] -> [ No , No , Yes , Yes ];
    [ 2, 8] -> [ No , No , Yes , Yes ];
end_comp4a

```

Listing 10-9. Source File: 4-Bit Comparator

10.8.3 Test Vectors

Three separate test vectors sections are written to test three of the four possible conditions. (The fourth and untested condition of NOT EQUAL TO is simply the inverse of EQUAL TO.) Each test vectors table includes a test vector message that helps make output from the documentation generator (DOCUMENT) and the simulator (SIMULATE) easier to read. The three tested conditions are not mutually exclusive, so one or more of them can be met by a given A and B. In the test vectors table, the constants YES and NO are used rather than 1 and 0, just for ease of reading. YES and NO are declared in the declaration section of the source file.

10.9 Bi-Directional Three-State Buffer (F153)

A four-bit bidirectional buffer with 3-state 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.

10.9.1 Design Specification

Figure 10-16 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.

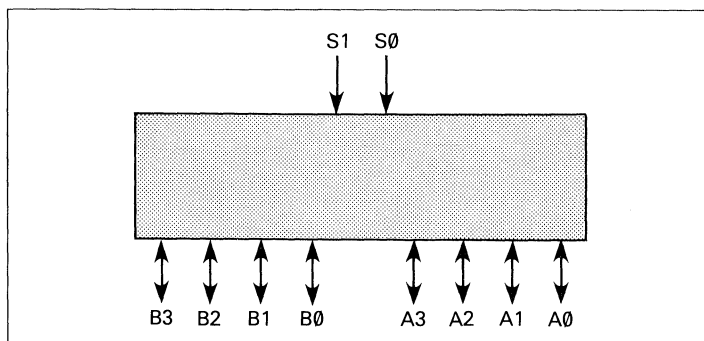


Figure 10-16. Block Diagram: Bidirectional Tri-State Buffer

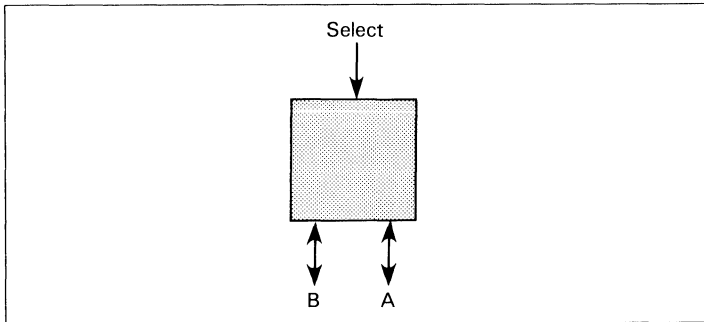


Figure 10-17. Simplified Block Diagram: Tri-State Buffer

10.9.2 Design Method

A simplified block diagram for the buffer is shown in figure 10-17. The A and B inputs/outputs are grouped into two sets, A and B. The select lines are grouped into the select set. Listing 10-10 shows the source file that describes the design.

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.

```
module tsbuffer
title 'bi-directional three state buffer      8 Nov 1987
Brenda French & Mary Bailey  FutureNet Division, 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;

    Enable A = (Select == 1);
    Enable B = (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
```

Listing 10-10. Source File: Tri-State Bi-Directional Buffer

10.10 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).

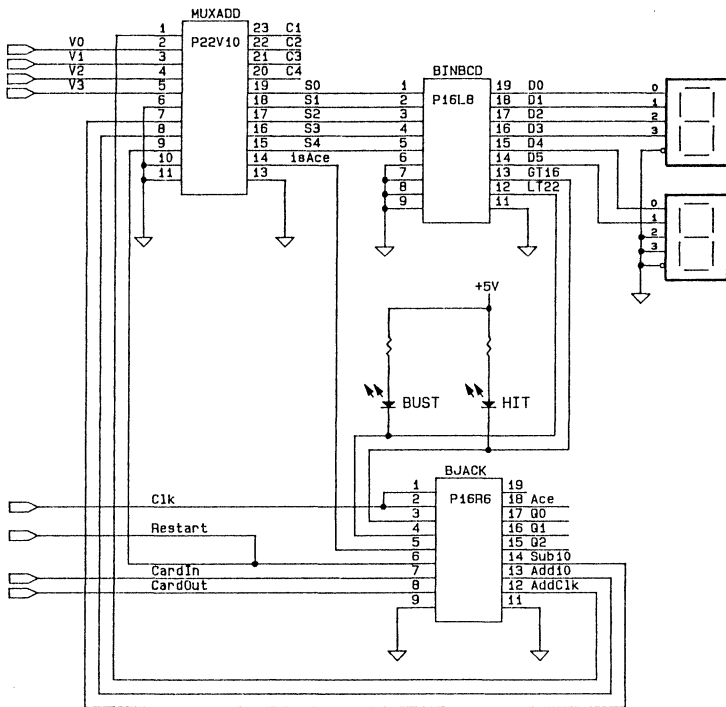


Figure 10-18. Schematic of a Blackjack Machine Implemented in Three PLDs

As Figure 10-18 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.

Circuits that are a straight-forward function of a set of inputs and outputs are often most easily expressed in equations; the adder is such a circuit. The PLD for the adder function (identified as MUXADD in Figure 10-18) includes three elements: a multiplexer, the adder itself, and a comparator. The multiplexer selects either the value of the newly dealt card or one of the two fixed values used for the ace (*ADD10* or *SUB10*). The adder adds the value selected by the multiplexer to the previous score when triggered by the clock signal, *ADDCLK*. The comparator detects when an ace is present and passes this information on to the blackjack controller, *BJACK*.

Designs with outputs that do not follow a specific pattern are most easily expressed as truth tables. Such is the case with the binary-to-BCD converter (identified in the schematic, figure 10-18, 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 blackjack machine is implemented with three programmable logic devices. The device types and their functions are listed in table 10-4. 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.

Table 10-4. Devices used in the Blackjack Machine

Device	Function in the Blackjack Machine
P22V10	Multiplexer/Adder/Comparator
P16L8	Binary-BCD converter
P16R4	State machine

10.10.1 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.

10.10.2 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 listing 10-11. 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 and 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.

10.10.3 Test Vectors - MUXADD

The test vectors shown in listing 10-11 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.

```

module MuxAdd flag '-r3'
title '5-bit ripple adder with input multiplex
Michael Holley   FutureNet Division, Data I/O Corp.
Redmond WA   14 July 1987'

    MuxAdd device 'P22V10';

    AddClk,Clr,Add10,Sub10,is_Ace  pin  1, 9, 8, 7,14;
    V4,V3,V2,V1,V0                pin  6, 5, 4, 3, 2;
    S4,S3,S2,S1,S0                pin 15,16,17,18,19;
    C4,C3,C2,C1                  pin 20,21,22,23;
    Reset                          node 25;

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

    Card      = [V4,V3,V2,V1,V0];
    Score     = [S4,S3,S2,S1,S0];
    CarryIn   = [C4,C3,C2,C1, 0];
    CarryOut  = [ X,C4,C3,C2,C1];
    ten       = [ 0, 1, 0, 1, 0];
    minus_ten = [ 1, 0, 1, 1, 0];

" Input Multiplexer
    Data      = Add10 & Sub10 & Card
               # !Add10 & Sub10 & ten
               # Add10 & !Sub10 & minus_ten;

equations
    Score     := Data $ Score $ CarryIn;

    CarryOut  = Data & Score # (Data # Score) & CarryIn;

    Reset     = !Clr;          "Async reset node for registers

    is_Ace    = Card == 1;

test_vectors
    ([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

```

Listing 10-11. Source File: Multiplexer/Adder/Comparator

10.10.4 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.

10.10.5 Design Method - BINBCD

The design of BINBCD is shown in the source file of listing 10-12. 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 31 decimal.

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 listing 10-12 and described in paragraph 10.10.6, 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` and `*.OUT` files show the truth table created from the macro.

The BINBCD design also provides the outputs *LT22* and *GT16* to control the *Bust* and *Hit* LEDs. A pair of equations generate an active-high *LT22* signal to turn off the *Bust* LED whenever *Score* is less than 22, and an active-high *GT16* signal to turn off the *Hit* LED whenever *Score* is greater than 16.

10.10.6 Test Vectors - BINBCD

The test vectors shown in listing 10-12 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 `macro (a) {@const ?a=0};` (contained in the BINBCD.ABL source file shown in listing 10-12), followed by 31 additional 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 listing 10-12) 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). In the BINBCD.DOC file, the test vectors appear as follows when BINBCD.ABL is run on ABEL with the -v option.

```

12 [0000 0--- ---- ----] -> [---- ---- ---- -LLL LLL-];
13 [1000 0--- ---- ----] -> [---- ---- ---- -LLL LLH-];
14 [0100 0--- ---- ----] -> [---- ---- ---- -LLL LHL-];
15 [1100 0--- ---- ----] -> [---- ---- ---- -LLL LHH-];
16 [0010 0--- ---- ----] -> [---- ---- ---- -LLL HLL-];
17 [1010 0--- ---- ----] -> [---- ---- ---- -LLL HLH-];
18 [0110 0--- ---- ----] -> [---- ---- ---- -LLL HHL-];
19 [1110 0--- ---- ----] -> [---- ---- ---- -LLL HHH-];
20 [0001 0--- ---- ----] -> [---- ---- ---- -LLH LLL-];
21 [1001 0--- ---- ----] -> [---- ---- ---- -LLH LLH-];
22 [0101 0--- ---- ----] -> [---- ---- ---- -LLH LLL-];
23 [1101 0--- ---- ----] -> [---- ---- ---- -LLH LLH-];
24 [0011 0--- ---- ----] -> [---- ---- ---- -LLH LHL-];
25 [1011 0--- ---- ----] -> [---- ---- ---- -LLH LHH-];
26 [0111 0--- ---- ----] -> [---- ---- ---- -LLH HLL-];
27 [1111 0--- ---- ----] -> [---- ---- ---- -LLH HLH-];
28 [0000 1--- ---- ----] -> [---- ---- ---- -LHL HHL-];
29 [1000 1--- ---- ----] -> [---- ---- ---- -LHL HHH-];
30 [0100 1--- ---- ----] -> [---- ---- ---- -LHH LLL-];
31 [1100 1--- ---- ----] -> [---- ---- ---- -LHH LLH-];
32 [0010 1--- ---- ----] -> [---- ---- ---- -LHL LLL-];
33 [1010 1--- ---- ----] -> [---- ---- ---- -LHL LLH-];
34 [0110 1--- ---- ----] -> [---- ---- ---- -LHL LHL-];
35 [1110 1--- ---- ----] -> [---- ---- ---- -LHL LHH-];
36 [0001 1--- ---- ----] -> [---- ---- ---- -LHL HLL-];
37 [1001 1--- ---- ----] -> [---- ---- ---- -LHL HLH-];
38 [0101 1--- ---- ----] -> [---- ---- ---- -LHL HHL-];
39 [1101 1--- ---- ----] -> [---- ---- ---- -LHL HHH-];
40 [0011 1--- ---- ----] -> [---- ---- ---- -HLH LLL-];
41 [1011 1--- ---- ----] -> [---- ---- ---- -HLH LLH-];
42 [0111 1--- ---- ----] -> [---- ---- ---- -HHL LLL-];
43 [1111 1--- ---- ----] -> [---- ---- ---- -HHL LLH-];

```

The use of macros and directives to create test vectors is described in more detail in Chapter 12.

```

module _binbcd flag '-r3'
title
'comparator and binary to bcd decoder for Blackjack Machine
Michael Holley Data I/O Corp 25 June 1987'

" The 5 -bit binary (0 - 31) score is converted into two BCD outputs.
" The integer division '/' and the modulus operator '%' are used to
" extract the individual digits from the two digit score.
" 'Score % 10' will yield the 'units' and
" 'Score / 10' will yield the 'tens'
"
" The 'GT16' and 'LT22' outputs are for the state machine controller.

    binbcd device 'P16L8';

    S4,S3,S2,S1,S0 pin 5,4,3,2,1;
    score          = [S4,S3,S2,S1,S0];

    LT22,GT16      pin 12,13;

    D5,D4          pin 14,15;
    bcd2           = [D5,D4];

    D3,D2,D1,D0    pin 16,17,18,19;
    bcd1           = [D3,D2,D1,D0];

" Digit separation macros
binary          = 0;                "scratch variable
clear macro (a) (@const ?a=0);
inc macro (a) (@const ?a=?a+1;);

equations
    LT22 = (score < 22);             "Bust
    GT16 = (score > 16);             "Hit / Stand

test_vectors ( score -> [GT16,LT22])
    1 -> [ 0 , 1 ];
    6 -> [ 0 , 1 ];
    8 -> [ 0 , 1 ];
    16 -> [ 0 , 1 ];
    17 -> [ 1 , 1 ];
    18 -> [ 1 , 1 ];
    20 -> [ 1 , 1 ];
    21 -> [ 1 , 1 ];
    22 -> [ 1 , 0 ];
    23 -> [ 1 , 0 ];
    24 -> [ 1 , 0 ];

```

Listing 10-12. Source File: Binary to BCD Converter
(continued on next page)

```
@page
truth_table ( score -> [bcd2,bcd1])
    0 -> [ 0 , 0 ];
    1 -> [ 0 , 1 ];
    2 -> [ 0 , 2 ];
    3 -> [ 0 , 3 ];
    4 -> [ 0 , 4 ];
    5 -> [ 0 , 5 ];
    6 -> [ 0 , 6 ];
    7 -> [ 0 , 7 ];
    8 -> [ 0 , 8 ];
    9 -> [ 0 , 9 ];
   10 -> [ 1 , 0 ];
   11 -> [ 1 , 1 ];
   12 -> [ 1 , 2 ];
   13 -> [ 1 , 3 ];
   14 -> [ 1 , 4 ];
   15 -> [ 1 , 5 ];
   16 -> [ 1 , 6 ];
   17 -> [ 1 , 7 ];
   18 -> [ 1 , 8 ];
   19 -> [ 1 , 9 ];
   20 -> [ 2 , 0 ];
   21 -> [ 2 , 1 ];
   22 -> [ 2 , 2 ];
   23 -> [ 2 , 3 ];
   24 -> [ 2 , 4 ];
   25 -> [ 2 , 5 ];
   26 -> [ 2 , 6 ];
   27 -> [ 2 , 7 ];
   28 -> [ 2 , 8 ];
   29 -> [ 2 , 9 ];
   30 -> [ 3 , 0 ];
   31 -> [ 3 , 1 ];

" This truth table could be replaced with the following macro.
"      clear(binary);
"      @repeat 32 {
"          binary -> [binary/10,binary%10]; inc(binary);}
"
" The test vectors will demonstrate the use of the macro.
"
test_vectors ( score -> [bcd2,bcd1])
    clear(binary);
    @repeat 32 {
        binary -> [binary/10,binary%10]; inc(binary);}
end_binbcd
```

Listing 10-12. (continued)

10.10.7 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.

10.10.8 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. The easiest way to express a state machine design is with a state diagram. Any other form of design expression would be tedious to develop and would be likely to contain errors.

In describing a state machine, the first step is to develop a bubble diagram. Figure 10-19 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.

You must then express the bubble diagram in a the form shown in *state_diagram* portion of listing 10-13. There is a one-to-one correlation between the bubble diagram and the state diagram described in the source file (listing 10-13). Table 10-5 provides a more detailed description of the state identifiers (state machine states) illustrated in the bubble diagram and listed in the source file.

Table 10-5. States of the Blackjack State Machine

State Identifier	Description
Clear	Clear the state machine, adder, and displays.
ShowHit	Indicate that another card is needed. <i>Hit</i> indicator is lit.
AddCard	Add the value present at the adder input to the current count.
Add10	Add the fixed value 10 to the current count, effectively giving an ace a value of 11.
Wait	Wait until a card is taken out of the reader.
Test17	Test the current count for a value less than 17.
Test22	Test the current count for a value less than 22.
Sub10	Add the fixed value -10 to the current count, effectively subtracting 10 and restoring an ace to 1 after it was an 11.
ShowBust	Indicate that no more cards are needed. <i>Bust</i> indicator is lit.
ShowStand	Indicate that another card is needed. Neither <i>Hit</i> nor <i>Bust</i> indicators are lit.

Note that in listing 10-13, each of the state identifiers (*Clear*, *ShowHit*, etc.) are defined as sets having binary values. These values were chosen to minimize the number of product terms used in the P16R6.

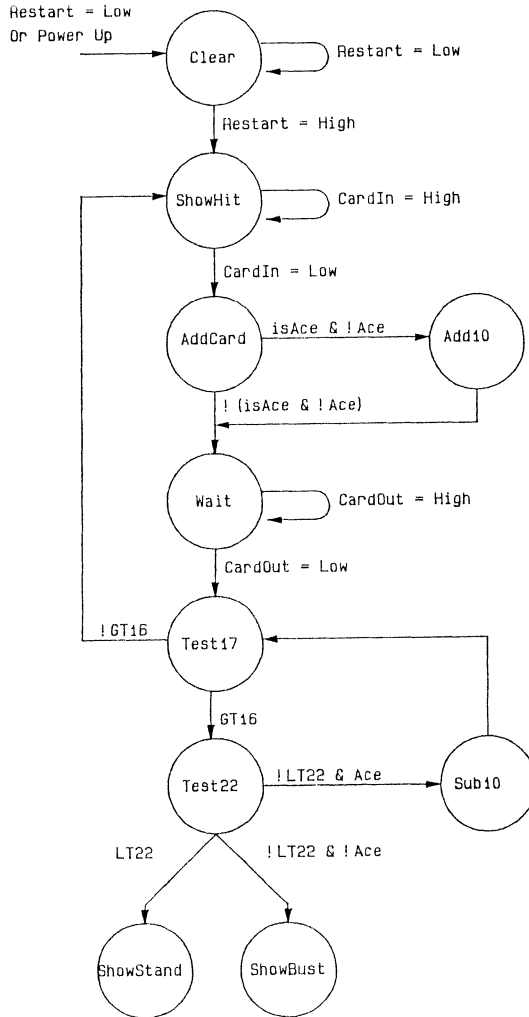


Figure 10-19. Pictorial State Diagram: Blackjack Machine

Operation of the state machine proceeds as follows if no aces are drawn: If a card is needed from the reader, the state machine goes to state *ShowHit*. When *CardIn* goes low, meaning that a card has been read, a transition to state *AddCard* is made. The card value is added to the current score or count value. The machine goes to *Wait* state until the card is withdrawn from the reader. The machine goes to *Test17* state. If the score is less than 17, another card is drawn. If the score is greater than or equal to 17, the machine goes to state *Test22*. If the score is less than 22, the machine goes to the *ShowStand* state. If the score is 22 or greater, a transition is made to the *ShowBust* state. In either the *ShowStand* or *ShowBust* state, a transition is made to *Clear* (to clear the state register and adder) when *Restart* goes low. When *Restart* goes back to high, the state machine returns to *ShowHit* state and the cycle begins again.

Operation of the state machine when an ace is drawn is essentially the same. A card is drawn and the score is added. If the card is an ace and no ace has been drawn previously, the state machine goes to state *Add10*, and ten is added to the count, in effect making the ace an 11. Transitions to and from *Test17* and *Test22* proceed as before. However, if the score exceeds 21, and an ace has been set to 11, the state machine goes to state *Sub10*, 10 is subtracted from the score, and the state machine goes to state *Test17*.

10.10.9 Test Vectors - BJACK

Listing 10-13 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.

```

module _Bjack      flag '-r3'
title 'BlackJack state machine controller
Michael Holley & Kyu Lee  Data I/O Corp.  23 Mar 1987'

    Bjack  device  'P16R6';

"Inputs
    Clk,ClkIN      pin 1,2;      "System clock
    GT16,LT22      pin 3,4;      "Score less than 17 and 22
    is_Ace          pin 5;        "Card is ace
    Restart         pin 6;        "Restart game
    CardIn,CardOut  pin 7,8;      "Card present switches
    Ena             pin 11;

    Sensor          = [CardIn,CardOut];
    _In              = [ 0 , 1 ];
    InOut            = [ 1 , 1 ];
    Out              = [ 1 , 0 ];

"Outputs
    AddClk          pin 12;        "Adder clock
    Add10            pin 13;        "Input Mux control
    Sub10            pin 14;        "Input Mux control
    Q2,Q1,Q0         pin 15,16,17;
    Ace              pin 18;        "Ace Memory

    High,Low         = 1,0;
    H,L,C,X          = 1,0,.C.,.X.; "test vector charactors

    Qstate           = [Add10,Sub10,Q2,Q1,Q0];
    Clear             = [ 1 , 1 , 1, 1, 1];
    ShowHit           = [ 1 , 1 , 1, 1, 0];
    AddCard           = [ 1 , 1 , 0, 0, 0];
    Add_10            = [ 0 , 1 , 0, 0, 0];
    Wait             = [ 1 , 1 , 0, 0, 1];
    Test_17           = [ 1 , 1 , 0, 1, 0];
    Test_22           = [ 1 , 1 , 0, 1, 1];
    ShowStand         = [ 1 , 1 , 1, 0, 0];
    ShowBust          = [ 1 , 1 , 1, 0, 1];
    Sub_10            = [ 1 , 0 , 0, 0, 1];

equations
    Qstate := Clear & !Restart;

```

Listing 10-13. Source File: State Machine (Controller)
(continued on next page)

@page
state_diagram Qstate

```
State Clear:  AddClk  = !ClkIN;
               Ace     := Low;
               goto ShowHit;

State ShowHit: AddClk  = Low;
               Ace     := Ace;
               if (CardIn==Low) then AddCard else ShowHit;

State AddCard: AddClk  = !ClkIN;
               Ace     := Ace;
               if (is_Ace & !Ace) then Add_10 else Wait;

State Add_10:  AddClk  = !ClkIN;
               Ace     := High;
               goto    Wait;

State Wait:    AddClk  = Low;
               Ace     := Ace;
               if (CardOut==Low) then Test_17 else Wait;

State Test_17: AddClk  = Low;
               Ace     := Ace;
               if !GT16 then ShowHit else Test_22;

State Test_22: AddClk  = Low;
               Ace     := Ace;
               case
                 LT22           : ShowStand;
                 !LT22 & !Ace    : ShowBust;
                 !LT22 & Ace     : Sub_10;
               endcase;

State Sub_10:  AddClk  = !ClkIN;
               Ace     := Low;
               goto    Test_17;

State ShowBust: AddClk  = Low;
               Ace     := Ace;
               goto ShowBust;

State ShowStand: AddClk  = Low;
               Ace     := Ace;
               goto ShowStand;
```

Listing 10-13. (continued)

apage

test_vectors 'Assume two cards that total between 16 and 21'

```
([Ena,Ck,ClkIN,GT16,LT22,is_Ace,Restart,Sensor] -> [Ace,Qstate,AddClk])
[ L , C , L , L , H , L , L , Out ] -> [ X ,Clear , H ];
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];

[ L , C , L , L , H , L , H , InOut ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];

[ L , C , L , L , H , L , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , L , H , _In ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ L ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,Test_22 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowStand , L ];
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
```

test_vectors 'Assume 2 Aces and another card that total between 16 and 21'

```
([Ena,Ck,ClkIN,GT16,LT22,is_Ace,Restart,Sensor] -> [Ace,Qstate,AddClk])
[ L , C , L , L , H , L , L , Out ] -> [ L ,Clear , H ];
[ L , C , L , L , H , L , H , Out ] -> [ L ,ShowHit , L ];

[ L , C , L , L , H , H , H , InOut ] -> [ L ,ShowHit , L ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,AddCard , H ];
[ L , C , L , L , H , H , H , _In ] -> [ L ,Add_10 , H ];
[ L , C , L , L , H , H , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];

[ L , C , L , L , H , H , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , L , H , H , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
[ L , C , L , L , H , L , H , Out ] -> [ H ,ShowHit , L ];
```

Listing 10-13. (continued)

```

[ L , C , L , L , H , L , H , _In ] -> [ H ,AddCard , H ];
[ L , C , L , H , H , L , H , _In ] -> [ H ,Wait , L ];
[ L , C , L , H , H , L , H , InOut ] -> [ H ,Wait , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,Test_17 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,Test_22 , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , H , H , L , H , Out ] -> [ H ,ShowStand , L ];
[ L , C , L , H , H , L , L , Out ] -> [ H ,Clear , H ];

@page
test_vectors 'Assume an Ace and 2 cards that total between 16 and 21'
([Ena,Ck,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

```

Listing 10-13. (continued)

11. Design Considerations

General information and considerations that you may find useful as you design logic with ABEL are contained in this chapter. This chapter contains:

- 11.1 Using State Machines
- 11.2 Solving Timing Problems with REDUCE
- 11.3 Passing Arguments From the Command Line
- 11.4 Effect of Equation Polarity on Reduction Speed

11.1 Using State Machines

State machines are one of the more powerful features of ABEL, but some care should be taken in organizing the state diagram and in ordering the states within it. This section describes some of the steps you can take to make state diagrams easy to read and maintain and to avoid some potential problems. The most common problem encountered with state machines is that too many product terms are created for the chosen device. This problem arises because state machines often have many different states and complex state transitions. The topics discussed in the following subsections will help you avoid this problem by reducing the number of required product terms.

The following subsections provide information related to state machines:

- 11.1.1 Use Identifiers Rather Than Numbers for States
- 11.1.2 "Power On" Register States
- 11.1.3 Design With Registers, not With Outputs
- 11.1.4 Unsatisfied Transition Conditions, D-Type Flip-Flops
- 11.1.5 Unsatisfied Transition Conditions, Other Flip-Flops
- 11.1.6 Number Adjacent States for One Bit Changes
- 11.1.7 Use State Register Outputs to Identify States

11.1.1 Use Identifiers Rather Than Numbers for States

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.

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.

In writing a state diagram with ABEL, 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 listing 11-1, which lists the source file for a state machine named "sequence." (This state machine is also discussed section 10.5.) In the state diagram, identifiers (*A*, *B*, and *C*) are used to specify the states. These identifiers are assigned a constant decimal value in the declaration section of the source file that identify the bit values in the state register for each state. Note that *A*, *B*, and *C* are only identifiers, they do not indicate the bit pattern of the state machine. It is their declared values that define the value of the state register (*sreg*) for each state. The declared values are 0, 1, and 2.

```
module sequence          flag '-r3'
title 'State machine example'      D. B. Pellerin - FutureNet';

    d1      device 'p16r4';

    q1,q0
    clock,enab,start,hold,reset    pin    14,15;
    abort                          pin    1,11,4,2,3;
    in_B,in_C                      pin    17;
    sreg                          pin    12,13;
                                =      [q1,q0];

    "State Values...
    A = 0;      B = 1;      C = 2;

state_diagram sreg;
    State A:      " Hold in state A until start is active.
        in_B = 0;
        in_C = 0;
        IF (start & !reset) THEN B WITH abort := 0;
        ELSE A WITH abort := abort;

    State B:      " Advance to state C unless reset
        in_B = 1;      " or hold is active. Turn on abort
        in_C = 0;      " indicator if reset.
        IF (reset) THEN A WITH abort := 1;
        ELSE IF (hold) THEN B WITH abort := 0;
        ELSE C WITH abort := 0;

    State C:      " Go back to A unless hold is active
        in_B = 0;      " Reset overrides hold.
        in_C = 1;
        IF (hold & !reset) THEN C WITH abort := 0;
        ELSE A WITH abort := 0;

test_vectors([clock,enab,start,reset,hold]->[sreg,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
```

Listing 11-1. Source File: State Machine

11.1.2 "Power On" Register States

Your state machine design might depend on a specific starting state, and, therefore, you must know what the "power-on" state of a device's register is. If the register powers on in a state that is not defined in the state diagram description of the state machine, the next state is undefined and unknown unless your design specifically accounts for this situation. Therefore, you should account for the power-on state of the device you are using or make sure that your design goes to a known state at power-on time.

11.1.3 Designing With Programmable Polarity Outputs

With devices featuring programmable polarity on the outputs of the registers, you must be keep in mind that the register outputs may be programmed to the complement of the device outputs. This is important because all the sets and clears are on the registers rather than on the programmable device outputs, and because you are concerned with the states (register outputs) themselves rather than with the outputs at the device pins. The final outputs can be handled separately.

11.1.4 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 11-1, is cleared to all 0's on the next clock pulse. This action causes the state machine to go to the state that corresponds to the cleared state register, or, the "cleared-register state." This can either be used to your advantage, or cause problems, depending on your design.

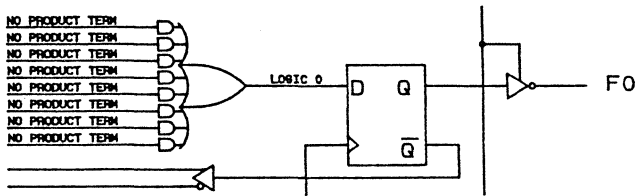


Figure 11-1. D-Type Register with False Inputs

You can use this action to eliminate some conditions in your state diagram and some product terms in the converted design. Rather than having a condition for a transition to the cleared-register state, that transition can be left implicit. If none of the other transition conditions is met, the machine will go to the cleared state. This same fact can cause problems if the cleared state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine will go to the undefined cleared state, and remain in that state.

In general, this means that you should always have a state assigned to the cleared-register state. Or, you must define every possible condition so that some condition is always met for each state. But, the automatic transition to the cleared-register state can also be used to your advantage by eliminating product terms and explicit definitions of transitions.

11.1.5 Unsatisfied Transition Conditions, Other Flip-Flops

If none of the state conditions is met in a state machine that employs J-K, R-S, and T-type flip-flops, the state machine does not advance to the next state as it should, but holds its present state. This is due to the low input to the register

from the OR array output. In such a case, the state machine can "get stuck" and will not change state. (This holding action can be used to advantage in some designs.)

To prevent this condition from occurring, you can use the complement array provided in some devices, such as the F105 and F159, to detect a "no conditions met" situation and reset the state machine to a known state. An example of using the complement array is shown in listing 13-7. (In this example, the feedback afforded by the array resets a counter to zero, but could be used to set a state machine.)

11.1.6 Number Adjacent States for One-Bit Changes

The number of product terms produced by a state diagram can be reduced greatly by a careful choice of state register bit values. Your state machine should be described with symbolic names for the states, as described in section 11.1.1. 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.

One specialized case occurs when the language processor reports that too many product terms were produced for one of the state register bits. If this occurs, 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.

11.1.7 Use State Register Outputs To Identify States

Sometimes it is necessary to identify specific states of a state machine and signal on 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 Name	State Register Bit Values		
	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 section 11.1.6.

11.2 Solving Timing Problems with REDUCE

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

Figure 11-2 shows a circuit that has a timing problem because both the complement of the signal C and the uncomplemented signal are used. Notice that the part of the circuit using !C has one additional unit of propagation delay.

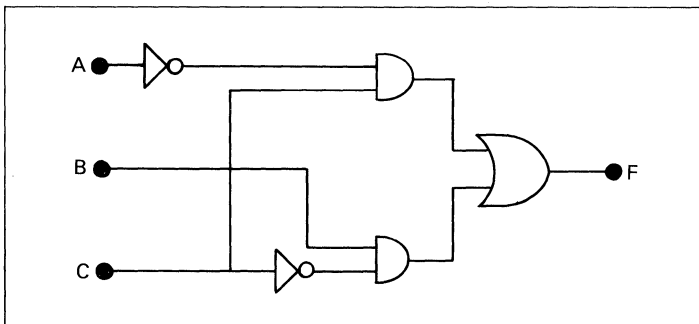


Figure 11-2. Circuit Using an Input and Its Complement

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 11-3.

The hazard can be eliminated by the introduction of a redundant term⁽¹⁾, !A & B, so that the full equation becomes:

$$F = B \& !C \# !A \& C \# !A \& B$$

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

Information on the language processor, the reduction pass, and the different reduction levels is presented in sections 4.1 and 4.4.

(1) Harris Programmable Logic (HPL TM) Application Note 106, Hazard Free Logic Design, Steven Bennett.

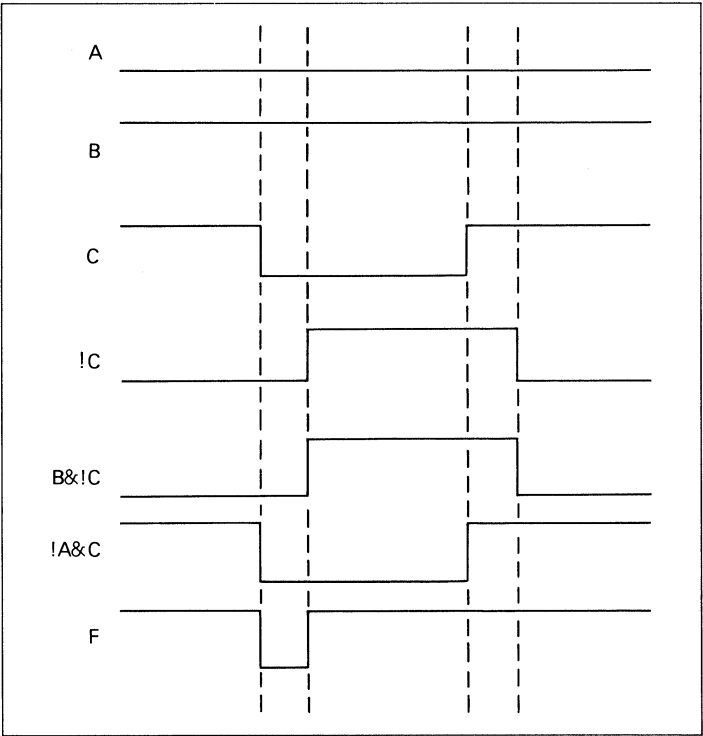


Figure 11-3. Timing Diagram for $F = B \& !C \# !A \& C$

11.3 Selecting Devices at the Command Line

Dummy arguments can be used within a source file so that variable values can be passed from the command line when the ABEL language processor is invoked. This section describes such a use of dummy arguments to provide a variable device type to the memory address decoder presented in section 10.1.

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

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

This idea is used in the example logic design, M6809D, shown in listing 11-2. M6809D allows the user to specify the device type from the language processor command line rather than in the source file itself. Thus, the same source file can be used to create many different programmer load files for different devices. M6809D also provides a default device type so that if none is specified, a P14L4 is used by

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

The variable device type is implemented in three of the first eleven lines of the source file. These three lines are discussed in detail here; the remainder of the source file is described in section 10.1.

Line 1:

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

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

```
abel m6809d -aP16L8
```

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

Line 8:

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

Line 8 uses the "IF NOT BLANK" directive, `@IFNB`, to insert the appropriate device declaration into the source file. If the value passed for `dev` is not blank, the text enclosed by the left and right braces is inserted into the source with the value of `dev` substituted. The text within the braces is a standard device declaration followed by the

@MESSAGE directive. **@MESSAGE** causes the text within single quotes to be written to the terminal. In this case, the message indicates what device is being used.

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

```
U09 device 'P16L8';
```

to be inserted into the source for processing.

Line 10:

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

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

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

```
module m6809d (dev)
title '6809 memory decoder
Jean Designer    Data I/O Corp Redmond WA    24 Feb 1987'

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

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

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

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

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

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

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

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

    !ROM1    = (Address >= ^hF800);

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

end m6809d
```

**Listing 11-2. Device Type Passed From Command Line,
Memory Address Decoder**

12. Simulation

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

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

SIMULATE does not evaluate the Boolean equations, truth tables, or state diagrams contained in the source file. SIMULATE uses the test vectors that you provide, the programmer load file created by previous steps of the language processor, and device characteristics to simulate the operation of the programmed device.

The information contained in this section explains some of the finer points of using the simulation facility and gives some suggestions concerning its use. Each subsection covers a separate topic. The subsections are:

- 12.1 Test Vectors and Simulation
- 12.2 Trace Levels and Breakpoints
- 12.3 Debugging State Machines
- 12.4 Multiple Test Vector Tables
- 12.5 Using Macros and Directives to Create Test Vectors
- 12.6 Don't Cares in Simulation
- 12.7 Preset, Reset and Reload Registers
- 12.8 Asynchronous Circuits

12.1 Test Vectors and Simulation

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

NOTE: The nodes shown on ABEL logic diagrams are outputs for writing equations; for example, the OR terms for the JK flip/flops in an F159. These nodes cannot be used as direct flip/flop inputs for simulation test vectors.

12.2 Trace Levels and Breakpoints

Through careful use of trace levels and breakpoints, you can easily control the amount of information that SIMULATE provides and make the analysis of a faulty design much easier. SIMULATE provides everything from simple error messages indicating that the actual outputs differ from the outputs you predicted in your test vectors to detailed information about the states of internal registers and gates of a device during simulation. It is suggested that you first simulate a design at the lowest trace level (level 0) to determine whether any errors exist. Once you find errors, use the higher trace levels (2, 3, 4, or 5) successively to increase the amount of information provided until you have enough information to solve the problem. Examples of the output produced by the different trace levels are given in section 4.6.

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

Detailed information on the different trace levels and on using breakpoints is provided in section 4.6. In summary:

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

12.3 Debugging State Machines

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

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

In summary, to test and debug larger state machines:

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

12.4 Multiple Test Vector Tables

More than one set of test vectors can be used to simulate the function of a device. This may be useful in the representation of the test in the source file. Take for example, the source file presented in listing 12-1 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, anytime you have two or more distinct functions being performed by the same device, you may want to describe the vectors in separate sections for each function. You are not, however, required to do so.

```
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
```

Listing 12-1. Source File With Multiple Test Vectors Sections

12.5 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.

Listing 12-2 shows a modified version of the memory address decoder presented in section 10.1. 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.

```

module m6809b
title '6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1987'

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

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

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

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

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

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

    !ROM1    = (Address >= ^hF800);

test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
    @IRP addr ( ^h0000, ^h8000, ^hE100, ^hE800, ^hF100, ^hFC00) {

        ?addr -> [ between (?addr, ^h0000, ^hDFFF),    " DRAM
                    between (?addr, ^hE000, ^hE7FF),    " IO
                    between (?addr, ^hF000, ^hF7FF),    " ROM2
                    between (?addr, ^hF800, ^hFFFF)];    " ROM1}

end m6809b

```

Listing 12-2. Test Vectors Described With a Macro and @IF and @IRP Directives

Now that 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 *addr*s declared in the `@IRP` statement. This means that the block will be repeated six times, and *addr*s will take on six different values each time *between* is invoked in the block. *addr*s is substituted for *a* in the macro and the values for *b* and *c* are given explicitly. The resulting source is shown in the expanded listing produced by the PARSE step of the language processor. This listing, shown in listing 12-3, shows how the directive and macro were expanded to produce the test vectors. The listing was produced by PARSE with the -E parameter.

In the second source file, shown in listing 12-4, 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 *addr*s before the next repetition of the block. Listing 12-5 shows the expanded output produced by the PARSE step of the language processor invoked with the -E parameter. Notice that in the test vectors shown in the listing, the constant *addr*s appears instead of the actual numeric value that the constant represents. The correct value is retained internally, and these test vectors are functionally the same as those shown in listing 12-3.

```

0001e|module m6809b
0002e|title '6809 memory decode
0003e|Jean Designer Data I/O Corp Redmond WA 24 Feb 1984'
0004e|
0005e|                U09b    device 'P14L4';
0006e|                A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007e|                ROM1,IO,ROM2,DRAM    pin 14,15,16,17;
0008e|
0009e|                H,L,X    = 1,0,.X.;;
0010e|                Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e|
0012e|" This macro will return an 'L' if the address (a) is between the
0013e|" two limits (b,c), otherwise it returns an 'H'.
0014e|between macro (a,b,c)
0015e|    {@if ((?a>=?b)&(?a<=?c))(L){@if!((?a>=?b)&(?a<=?c))(H)};
0016e|
0017e|equations
0018e|    !DRAM    = (Address <= ^hDFFF);
0019e|
0020e|    !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e|
0022e|    !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);
0023e|
0024e|    !ROM1    = (Address >= ^hF800);
0025e|
0026e|test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
0027e|
0028e|
0029e|                ^h0000 -> [ L,    " DRAM
0030e|                        H,    " IO
0031e|                        H,    " ROM2
0032e|                        H];   " ROM1
0033e|
0034e|                ^h8000 -> [ L,    " DRAM
0035e|                        H,    " IO
0036e|                        H,    " ROM2
0037e|                        H];   " ROM1
0038e|
0039e|                ^hE100 -> [ H,    " DRAM
0040e|                        L,    " IO
0041e|                        H,    " ROM2
0042e|                        H];   " ROM1
0043e|
0044e|                ^hE800 -> [ H,    " DRAM
.
.
.
.

```

Listing 12-3. PARSE Output (partial) Showing Expanded Vectors for a Macro and @IF/@IRP Directives

Applications Guide

```
module m6809c
title '6809 memory decode
Jean Designer      Data I/O Corp Redmond WA   24 Feb 1987'

        U09c    device 'P14L4';
A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
ROM1,IO,ROM2,DRAM    pin 14,15,16,17;

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

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

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

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

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

    !ROM1    = (Address >= ^hF800);

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

    @CONST  addr = addr + ^h800;}
end m6809c
```

**Listing 12-4. Test Vectors Described With a Macro,
@CONST and @REPEAT Directives**

```

0001e|module m6809c
0002e|title '6809 memory decode
0003e|Jean Designer      Data I/O Corp Redmond WA   24 Feb 1984'
0004e|
0005e|                U09c    device 'P14L4';
0006e|                A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007e|                ROM1,IO,ROM2,DRAM      pin 14,15,16,17;
0008e|
0009e|                H,L,X   = 1,0,.X.;;
0010e|                Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e|
0012e|" This macro will return an 'L' if the address (a) is between the
0013e|" two limits (b,c), otherwise it returns an 'H'.
0014e|between macro (a,b,c)
0015e|        {&if ((?a=?b)&(a<=?c)){L}&if(!((?a=?b)&(a<=?c)){H}};
0016e|
0017e|equations
0018e|        !DRAM   = (Address <= ^hDFFF);
0019e|
0020e|        !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e|
0022e|        !ROM2     = (Address >= ^hF000) & (Address <= ^hF7FF);
0023e|
0024e|        !ROM1     = (Address >= ^hF800);
0025e|
0026e|test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
0027e|
0028e|
0029e|                addr> -> [ L,      " DRAM
0030e|                           H,      " IO
0031e|                           H,      " ROM2
0032e|                           H];     " ROM1
0033e|
0034e|                addr> -> [ L,      " DRAM
0035e|                           H,      " IO
0036e|                           H,      " ROM2
0037e|                           H];     " ROM1
0038e|
0039e|                addr> -> [ H,      " DRAM
0040e|                           L,      " IO
.
.
.
.

```

Listing 12-5. PARSE Output (partial) with Expanded Output for a Macro and @CONST/@REPEAT Directives

12.6 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). The default value used in the simulator for the don't care inputs is zero; however, the don't care flag (`-Xn`) can be used in the `SIMULATE` command line to specify zero or one (0 or 1) for the don't care value. (Refer also to section 4.6.)

Input pins that are not specified in the test vectors are given the default don't care value zero (0) by the simulator unless the don't care flag is set to `-X1`. In this case, all unspecified pins will be assigned a value of 1 in the test vectors.

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

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

Also, the simulator checks the design with a single level for the don't care inputs, while the target circuit may place other levels on the input during actual operation of the device. For complete simulation, you must run the `SIMULATE` operation with the don't cares set to 0 (flag `-X0`), and then again with them set to 1 (flag `-X1`).

```
module findout flag '-t1'
title 'The ABEL simulator will find the output levels
Ngoc Nicholas FutureNet - Data I/O 8 Nov 1987'

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
```

Listing 12-6. Assignment of Don't Care Value (.x.) to Design Outputs

Output pins that are not specified in the test vectors are disregarded by the simulator and no error will be indicated due to conflict between a specified value and the value determined by the simulator. The .X. constant at an output pin tells the simulator not to compare the outputs (the output produced by the design and the output specified in the test vector) but still allow them to be displayed. Listing 12-6 shows how the .X. value can be assigned to the outputs prior to the SIMULATE step of the language processor.

Simulate ABEL(tm) 3.00

The ABEL simulator will find the output levels
Ngoc Nicholas FutureNet - Data I/O 8 Nov 1985

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

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

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

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

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

4 out of 4 vectors passed.

Listing 12-7. SIMULATE Results with Outputs Specified as Don't Care

Using trace level 1, 2, or 3, you can observe the actual output values determined by the simulator. In listing 12-7, the X entries (in the test vectors) for pins 14 and 15 allow the simulator to display an H or L to indicate the output value for the specified inputs. (All unused inputs are set to the default condition, 0.) The N entries indicate outputs that are not to be tested.

12.7 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.

Several programmable logic devices contain registers that can be preset, reset, or preloaded. To verify the operation of these devices, appropriate test vectors must be written and placed in the source file. These test vectors allow the SIMULATE step of the language processor to verify operation of the design by performing the required operations of these registers.

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

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

12.7.1 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 listing 12-8, along with the test vectors required to properly verify the design. The equation for the preset condition is written using the dot extension for the two registers (see section 13.3). 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 12-1 is a timing diagram that shows the action of the test vectors in listing 12-8. As indicated in figure 12-1, 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.

```

module _preset flag '-KY' "leave unused OR terms connected
title '2-bit counter to demonstrate power on preset  21 Nov 1987
Michael Holley                                     FutureNet Division, Data I/O Corp'

    preset device 'F167';

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

    Ck            = .C.;

equations

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

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

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

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

```

Listing 12-8. Test Vectors for Special Preset Conditions

The next six test vectors provide clock inputs to increment the counter through all states and back to state one. As indicated in figure 12-1, 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.

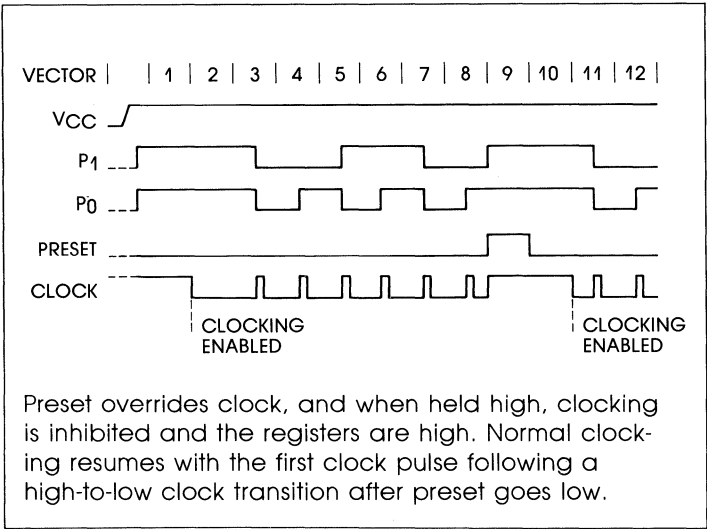


Figure 12-1. Timing Diagram Showing Test Vector Action

12.7.2 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 12-2 shows a typical FPLS layout. To preload the output register in such a device, four conditions must be present:

1. The output is placed in the high-impedance state.
2. The desired register state is placed on the output pin.
3. The load control term is activated.
4. A clock pulse is applied to the clock input.

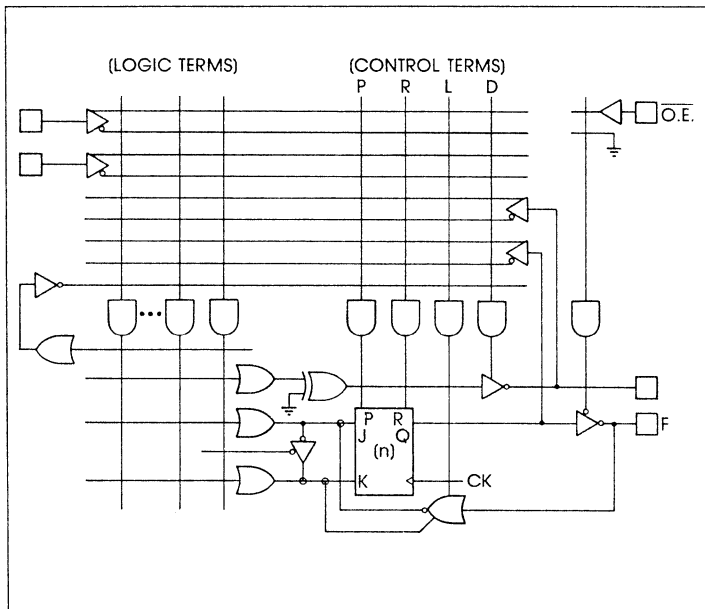


Figure 12-2. Internal Register of the F159

The fifth test vector in listing 12-9 shows how each of the above conditions are met. In the fifth test vector:

- the OE (*Ena*) pin is held at 1
- the output (*FO*) 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.

```
module TTLload
title 'TTL load example
Brenda French      FutureNet - Data I/O    21 Nov 1987'

    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';

equations
    F0.OE      = !Ena;
    !F0        := J_IN;
    F0.K       = K_IN;
    F0.L       = LOAD;

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
```

Listing 12-9. Invoking the TTL Preload Function

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 (*FO* = X on the input side and 0 on the output side of the vector) while holding the clock input at 0.

12.7.3 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 power-up 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 design of this decade counter should be such that whenever 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 define the illegal states for a decade counter, it is necessary to define the illegal states (i.e., 10 through 15). The example in listing 12-10 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 as shown in listing 12-11. This group of test vectors preloads the device to each possible illegal state and then verifies that the device clock to state S0. The test vectors preload the counter by means of the .P. special constant applied to the clock pin and a logic high applied to the output enable (OE) pin. The test vector that follows the preload test vector verifies the result of the preload operation, while the following test vector verifies the clocking of the counter from the illegal state to state S0.

```
module cnt10p flag '-r3'
title 'decimal counter
Note: preload the data on pins into the registers
Denny Siu FutureNet Division, Data I/O Corp 4 Oct 1987'

    U10p        device 'P16R4';

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

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

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

Listing 12-10. The Illegal States Defined

```
module cnt10p flag '-r3'
title 'decimal counter
Note: preload the data on pins into the registers
Denny Siu FutureNet Division, Data I/O Corp 4 Oct 1987'

U10p      device 'P16R4';

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

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

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

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

State S1: IF !Clr THEN S2 ELSE S0;

State S2: IF !Clr THEN S3 ELSE S0;

State S3: IF !Clr THEN S4 ELSE S0;

State S4: IF !Clr THEN S5 ELSE S0;

State S5: IF !Clr THEN S6 ELSE S0;

State S6: IF !Clr THEN S7 ELSE S0;

State S7: IF !Clr THEN S8 ELSE S0;

State S8: IF !Clr THEN S9 ELSE S0;

State S9: GOTO S0;
```

Listing 12-11. Test Vectors for Illegal States
continued on next page

```

apage
test_vectors
( [Clk,OE,Clr] -> [q3,q2,q1,q0] )
[ Ck , 0, 1 ] -> S0;
[ Ck , 0, 0 ] -> S1;
[ Ck , 0, 0 ] -> S2;
[ Ck , 0, 0 ] -> S3;
[ Ck , 0, 0 ] -> S4;
[ Ck , 0, 0 ] -> S5;
[ Ck , 1, 0 ] -> Z ;
[ Ck , 0, 0 ] -> S7;
[ 0 , 0, 0 ] -> S7;
[ Ck , 0, 0 ] -> S8;
[ Ck , 0, 0 ] -> S9;
[ Ck , 0, 0 ] -> S0;
[ Ck , 0, 0 ] -> S1;
[ Ck , 0, 0 ] -> S2;
[ Ck , 0, 1 ] -> S0;

test_vectors 'preload to illegal states'
( [Clk,OE,Clr,[q3,q2,q1,q0]] -> [q3,q2,q1,q0] )
[ P , 1, 0 , S10 ] -> X ;
[ 0 , 0, 0 , X ] -> S10;
[ Ck , 0, 0 , X ] -> S0 ;
[ P , 1, 0 , S11 ] -> X ;
[ 0 , 0, 0 , X ] -> S11;
[ Ck , 0, 0 , X ] -> S0 ;
[ P , 1, 0 , S12 ] -> X ;
[ 0 , 0, 0 , X ] -> S12;
[ Ck , 0, 0 , X ] -> S0 ;
[ P , 1, 0 , S13 ] -> X ;
[ 0 , 0, 0 , X ] -> S13;
[ Ck , 0, 0 , X ] -> S0 ;
[ P , 1, 0 , S14 ] -> X ;
[ 0 , 0, 0 , X ] -> S14;
[ Ck , 0, 0 , X ] -> S0 ;
[ P , 1, 0 , S15 ] -> X ;
[ 0 , 0, 0 , X ] -> S15;
[ Ck , 0, 0 , X ] -> S0 ;
end cnt10p

```

Listing 12-11 (continued)

An example of a state machine that contains branch conditions is given in section 10.10 (blackjack machine). 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 listing 12-12 show how this state machine can be preloaded to test these three branches of the design.

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

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

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

Listing 12-12. Using Test Vectors to Preload A State Machine

12.7.4 Preset/Reset Controlled by Product Term

In programmable logic devices such as the P22V10, preset and reset functions can be controlled by product terms placed in the "equations" portion of the ABEL source file. (Note that in devices that have programmable polarity at the output of the registers, preset and reset functions may complement the output pins.) An example of controlling the preset and reset functions is given in listing 12-13. In this listing, two nodes within the P22V10 are declared as the reset and preset functions. (Nodes are internal signal lines defined by Data I/O to aid in the programming of parts with internal signals. Refer to section 5.4 for additional details on nodes.) The declared nodes are nodes 25 and 26. Appendix C lists nodes 25 and 26 of the P22V10 as useable for the reset and preset functions.

```

module _reset22
title 'Demonstrates Asynchronous Reset and Synchronous Preset
Dave Pellerin FutureNet Division, Data I/O Corp 8 Nov 1987'

reset22 device 'P22V10';

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

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

equations
Output := Input; "Registered buffer

reset = R & !T;

preset = S & !T;

test_vectors
([Clk,Input,R,S,T] -> Output)
[ Ck, 0 ,0,0,0] -> 0;
[ Ck, 1 ,0,0,0] -> 1;
[ Ck, 2 ,0,0,1] -> 2;
[ 0 , 3 ,0,0,1] -> 2; "Hold
[ Ck, 3 ,0,0,1] -> 3;

[ 0 , 3 ,1,0,1] -> 3; "Reset = R & !T
[ 0 , 3 ,1,0,0] -> 0; "Async Reset

[ 0 , 0 ,0,1,0] -> 0; "Preset requires clock
[ Ck, 0 ,0,1,0] -> 3; "Sync Preset

end

```

Listing 12-13. Controlling Reset/Preset by Product Term

In the P22V10, the reset is asynchronous while preset is synchronous (to the clock input). The test vectors in listing 12-13 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).

12.7.5 Preset/Reset Controlled by Pin

Some devices, such as the P32R16, have a direct Preset coming from a pin. Be sure to include this in the test vectors or simulation errors may occur.

12.7.6 Power-Up States

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

12.8 Asynchronous Circuits

In some asynchronous circuits, the ABEL simulator may not maintain the output state and will report an error. In these cases, the device outputs should be specified on both sides of the test vector. Figure 12-3 shows a common cross-coupled flip-flop, an asynchronous circuit that can cause erroneous error reporting by the simulator.

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

Listing 12-14 shows two sets of test vectors that could be used to test the design of a cross-coupled flip-flop such as that shown in figure 12-3. The first set of test vectors defines the inputs only, and tests the outputs for the desired results. Because of the circuit's asynchronous nature, the simulator may not be able to maintain the output state and may report an error.

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

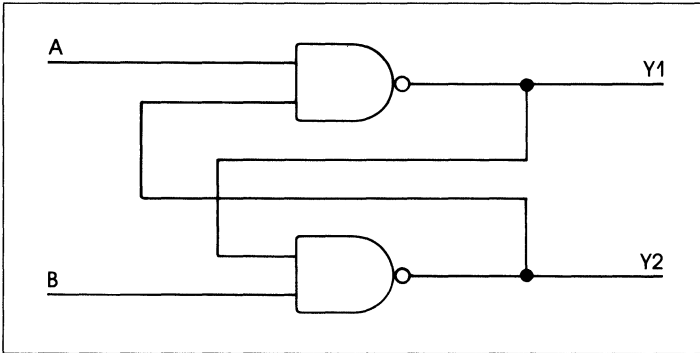


Figure 12-3. A Cross-Coupled Flip-Flop

```

module _flipflop
title 'Simulation of an asynchronous flip flop
Lloyd Hyden   Data I/O Corp   21 Nov 1987'

    flipflop      device 'P16L8';

    A,B,Y1,Y2     pin 1,2,13,14;

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

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

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

end
  
```

Listing 12-14. Using the Input Side of the Test Vectors to Define Outputs

13. Using Features of Advanced Devices

This chapter contains information that will help you utilize the features of the more advanced programmable logic devices. Subjects dealt with in this section include devices with enable-type outputs, configurable output macro cells, and the use of internal device nodes.

13.1 Output Enables

There are three types of output enables used in programmable logic devices; pin controlled, term controlled, and configurable. Each of these output enables, and how it is implemented in ABEL, is discussed in the following paragraphs.

13.1.1 Pin Controlled Output Enable

The pin controlled output enable is illustrated in figure 13-1. Here, the output enable input is a specific device pin; FO is high-impedance for OE, and is active for !OE. The pin controlled output enable is not included in the design equations but should be included in all test vectors.

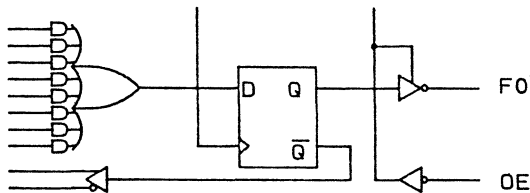


Figure 13-1. Output Enable Controlled by Device Pin

13.1.2 Term Controlled Output Enable

An example of a term controlled output enable is shown in Figure 13-2. In this example, the product term that enables pin F0 is $A \& B$. The equation in the ABEL source file would be:

```
enable F0 = A & B;
```

or, optionally

```
F0.OE = A & B;
```

For output pins that select a multiplexer, .OE must be used to specify the output enable. For other output pins, .OE or ENABLE can be used. See section 13.3.2 for more information on dot extension notation.

For devices that have a negative output enable, a ! can be placed on either side of the equation as follows

$\text{!F0.OE} = A \& B;$
 $\text{F0.OE} = \text{!(A \& B)};$

$\text{!enable F0} = A \& B;$
 $\text{enable F0} = \text{!(A \& B)};$

When the ENABLE keyword is used, any ! operator placed after the ENABLE keyword, but before the equation is ignored, for example

$\text{enable !F0} = A \& B;$

is equivalent to

$\text{enable F0} = A \& B;$

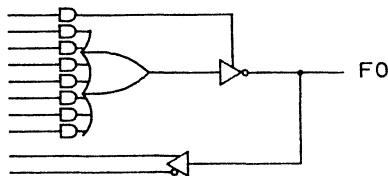


Figure 13-2. Output Enable Controlled by Product Term

13.1.3 Configurable Output Enable

The configurable output enable is one that, through the use of a device feature called a 'select multiplexer' can be made to operate in any of four distinct modes:

- All related outputs are controlled by a device pin.
- All related outputs are controlled by a device term.
- All related outputs are enabled at all times.
- All related outputs are disabled at all times.

Figure 13-3 shows a typical select multiplexer that can select any one of the four modes, by means of a pair of fuses. For devices with this type of output enable, you may specify the type of enable you wish to have on the device with a combination of `ISTYPE` statements and equations.

To configure the output enable for pin *F0* to be controlled by a device pin, add the following `ISTYPE` statement to the declarations section of your source file:

```
F0.OE  istype  'pin';
```

and, if the device allows you to choose a pin or pin polarity for the enable, write an equation to specify which pin is to be used, and it's polarity:

```
F0.OE = !OE;
```

To specify that an output enable is to be controlled by a term in the device, declare the enable as follows:

```
F0.OE  istype  'eqn';
```

and write an equation for the enable

$$F0.OE = A \& B;$$

To specify that an output is to be always enabled or disabled, declare the enable with

$$F0.OE \text{ istype 'fuse';}$$

and indicate the desired state of the output enable by writing an equation of the form

$$F0.OE = 1;$$

or

$$F0.OE = 0;$$

For most devices and designs, the use of the ISTYPE statement to configure the selectable enable is optional since ABEL can normally determine the type of output enable required from the form of equation you write.

More detailed information on the use of select multiplexers can be found in section 13.5.

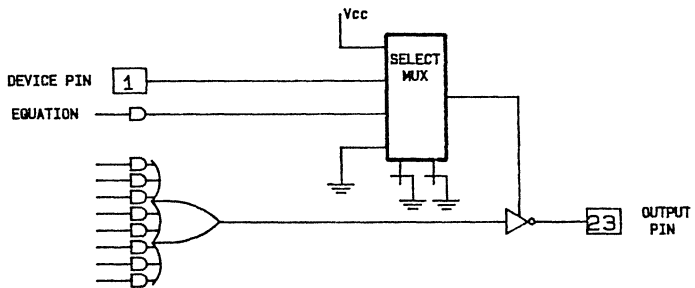


Figure 13-3. Typical Multiplexer for Output Enable Modes

13.2 Output Macro Cell Control with ISTYPE

As we have seen, the ISTYPE statement can be used to specify the exact configuration of programmable output enables. The ISTYPE statement is also used to precisely configure other device features, such as output polarity, selectable feedback, register bypass and type, and other macro cell features. Figure 13-4 shows how various ISTYPE statements affect the macro cell configuration.

13.2.1 Controlling Macro Cell Polarity

Output polarity is controlled by entering "pos" or "neg" in the ISTYPE statement. The ISTYPE statement for Q18 (see listing 13-1) is an example of controlling the output polarity to negative. For a positive output, this statement would be

```
Q18 IsType 'pos,reg,feed_reg';
```

The output polarity (positive or negative) of the macro cell device will affect the DeMorgan's equation inversion, and can result an excess of product terms. For example, a typical equation, such as

$$Y1 = (A \# B) \& (C \# D) \& (E \# F);$$

will reduce to the following three product terms for a negative output.

$$\begin{aligned} !Y1 = (&!A \& !B \\ &\# !C \& !D \\ &\# !E \& !F); \end{aligned}$$

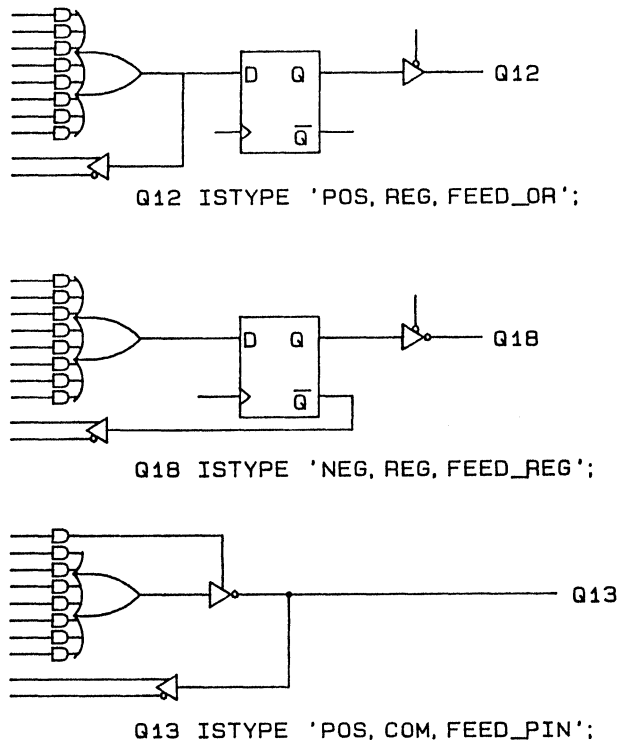


Figure 13-4. Controlling Macro Cells with ISTYPE

However, if a positive output is specified, the same equation requires eight product terms, as follows

```

Y1 = (A & C & E
      #A & C & F
      #A & D & E
      #A & D & F
      #B & C & E
      #B & C & F
      #B & D & E
      #B & D & F);
    
```

When dealing with programmable polarity devices, where the output can be programmed as negative or positive, some consideration should be given to polarity. For most programmable polarity devices, ABEL defaults to a negative selection when the polarity is not specified. In some devices this can mean a design will not fit into the device (causing a "too many terms for output" error message), although it would if positive polarity was specified for the outputs.

13.2.2 Controlling Macro Cell Feedback Point

Listing 13-1 also shows that the ISTYPE statement can be used to explicitly state the macro cell configuration, such as in the case of output Q12. Or, the ISTYPE statement can be used to state the feedback point only, leaving the polarity and register/combinatorial selection to be determined by ABEL from the equations. In the example listing, the ISTYPE statement defines the Q14 output as a feedback from the OR function only. The equation

!Q14 := D2 & D3

further defines Q14 as a negative, registered output.

```
module _mc flag '-f'
title 'Controlling output macro cells
Brian Durwood FutureNet 29 July 1987'

mc device 'E0310';

    Clk,D2,D3,D4,D5,D6,D7    pin 1,2,3,4,5,6,7;
    D8,D9,D11,Q12,Q13,Q14    pin 8,9,11,12,13,14;
    Q15,Q16,Q17,Q18,Q19      pin 15,16,17,18,19;

"Explicitly state macro cell configuration
    Q12      IsType      'pos,reg,feed_or';
    Q13      IsType      'pos,com,feed_pin';
    Q18      IsType      'neg,reg,feed_reg';

"Let equations determine resister/combinatorial and polarity
    Q14      IsType      'feed_or';
    Q15,Q16,Q17 IsType    'feed_pin';

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

equations
    Q12.RE = D6;          "Async Reset
    Q12.PR = D7;          "Sync Preset

test_vectors    ([Clk,D6,D7] -> [Q12,Q14,Q18])
    [ 0, 1, 0] -> [ 0, 1, 1 ]; "Reset
    [ 0, 0, 1] -> [ 0, 1, 1 ];
    [ Ck, 0, 1] -> [ 1, 0, 0 ]; "Preset with clock
    [ 0, 1, 0] -> [ 0, 1, 1 ]; "Reset

equations
    Q12 := D2 & D3;      "Feedback from the OR
    Q13 = Q12 & D4;

test_vectors    ([Clk,D2,D3,D4] -> [Q12,Q13])
    [ Ck, 1, 1, 1] -> [ 1, 1 ];
    [ 0, 1, 0, 1] -> [ 1, 0 ];
    [ Ck, 0, 1, 1] -> [ 0, 0 ];
    [ 0, 1, 1, 1] -> [ 0, 1 ];
```

Listing 13-1. Controlling Macro Cells with ISTYPE
(continued on next page)

```

@page

equations
    !Q14 := D2 & D3;      "Registered      NegativeE
    Q15  = D3 & D4;      "Combinatorial Positive

test_vectors    ([Clk,D2,D3,D4] -> [!Q14,Q15])
    [ Ck, 1, 1, 1] -> [ 1 , 1 ];
    [ Ck, 1, 0, 1] -> [ 0 , 0 ];
    [ Ck, 1, 1, 0] -> [ 1 , 0 ];
    [ Ck, 0, 1, 1] -> [ 0 , 1 ];

equations      "bidirectional buffer
    Q16 = Q17;      enable Q16 = D4;
    Q17 = Q16;      enable Q17 = !D4;

test_vectors    ([D4,Q16,Q17] -> [Q16,Q17])
    [ 1, X, 0] -> [ 0, X];
    [ 1, X, 1] -> [ 1, X];
    [ 0, 0, X] -> [ X, 0];
    [ 0, 1, X] -> [ X, 1];

equations
    Q18 := D5 & !Q18;
    Q19 = Clk;

test_vectors    ([Clk,D5] -> [Q18,Q19])
    [ Ck, 0] -> [ 0, 0];
    [ Ck, 1] -> [ 1, 0];
    [ Ck, 1] -> [ 0, 0];
    [ 0, 1] -> [ 0, 0];
    [ 1, 1] -> [ 1, 1];

end
    
```

Listing 13-1. (continued)

13.2.3 Selecting or Bypassing Device Registers

Figure 13-5 shows an output macro cell that allows the specification of whether the output register is bypassed to provide a combinatorial output, or selected to provide a registered output (other selectable features have been omitted for clarity). ABEL will normally choose whether to bypass or select the register based on the form of equation written for that output; the use of "=" in the equation indicates that the output is to be combinatorial (register bypassed,) and the use of ":=" indicates that the output is to be registered.

For situations where you need to explicitly declare the configuration of the register (as in the case of devices with multiple feedback or configurable input registers) you can use the statement

```
F0 istype 'reg';
```

to indicate that the register for pin F0 is to be selected, or

```
F0 istype 'com';
```

to indicate that the register is to be bypassed.

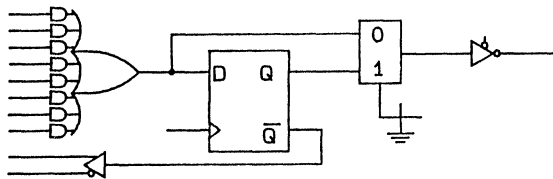


Figure 13-5. Macro Cell, Configurable to Combinatorial or Registered Output.

13.2.4 Controlling Register Type

For devices with selectable register types, such as the F159, the register type may be selected by using one of the attributes "reg_D", "reg_JK", "reg_RS", "reg_T", "reg_G", or "reg_JKD" to indicate which register type is desired. For example, to select a JK type register for an output in the F159, use the statement:

```
F0 istype 'reg_JK';
```

The selection of a register type may be made independently of whether the register has been bypassed (istype 'com') or selected (istype 'reg').

13.2.5 Selectable Product Term Sharing

Some devices allow you to specify whether an output may share product terms from another output or internal node. To specify that the terms are to be shared, use the statement

```
F0,F1 istype 'share';
```

13.3 Controlling Device Nodes

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

13.3.1 Using Node Numbers

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

In listing 13-2, the declaration statements assign the identifiers "reset" and "preset" to nodes 25 and 26. Once declared, the identifiers can be used in equations to specify the logic for each signal.

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


```
module _reset22
title 'Demonstrates Asynchronous Reset and Synchronous Preset
Dave Pellerin FutureNet Division, Data I/O Corp 8 Nov 1987'

    reset22    device    'P22V10';

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

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

equations
    Output      := Input;    "Registered buffer

    reset       = R & !T;

    preset      = S & !T;

test_vectors
    ([Clk,Input,R,S,T] -> Output)
    [ Ck, 0 ,0,0,0] -> 0;
    [ Ck, 1 ,0,0,0] -> 1;
    [ Ck, 2 ,0,0,1] -> 2;
    [ 0 , 3 ,0,0,1] -> 2;    "Hold
    [ Ck, 3 ,0,0,1] -> 3;

    [ 0 , 3 ,1,0,1] -> 3;    "Reset = R & !T
    [ 0 , 3 ,1,0,0] -> 0;    "Async Reset

    [ 0 , 0 ,0,1,0] -> 0;    "Preset requires clock
    [ Ck, 0 ,0,1,0] -> 3;    "Sync Preset

end
```

Listing 13-2. Using Node Numbers for Reset/Preset Functions

13.3.2 Using Dot Extension Notation

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

$$Q2.RE = R \& !T;$$

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

$$Q2.PR = S \& !T;$$

When dot extension notation is used, there is no need to declare the node in the declaration section of your ABEL source file. Listing 13-3 shows the P22V10 design example using dot extension notation for the reset and preset functions. Note that node numbers are not used in this example to declare the reset and preset nodes. The dot extensions supported in the ABEL language are listed below, and detailed information about which devices can use these dot extensions is contained in Appendix C.

.AP	asynchronous preset
.AR	asynchronous reset
.C	clock input
.FC	function (mode) control
.J	J input of J-K flip-flop
.K	K input of J-K flip-flop
.L	load input for registers
.OE	output enable
.PR	register preset
.Q	register feedback
.R	R input of RS flip-flop
.S	S input of RS flip-flop
.RE	register reset

```
module _reset22a
title 'Demonstrates Asynchronous Reset and Synchronous Preset
Dave Pellerin FutureNet - Data I/O Corp 8 Nov 1985'

reset22a device 'P22V10';

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

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

equations

Output := Input; "Registered buffer

Q2.RE = R & !T;

Q2.PR = S & !T;

test_vectors
([Clk,Input,R,S,T] -> Output)
[ Ck, 0 ,0,0,0] -> 0;
[ Ck, 1 ,0,0,0] -> 1;
[ Ck, 2 ,0,0,1] -> 2;
[ 0 , 3 ,0,0,1] -> 2; "Hold
[ Ck, 3 ,0,0,1] -> 3;

[ 0 , 3 ,1,0,1] -> 3; "Reset = R & !T
[ 0 , 3 ,1,0,0] -> 0; "Async Reset

[ 0 , 0 ,0,1,0] -> 0; "Preset requires clock
[ Ck, 0 ,0,1,0] -> 3; "Sync Preset

end
```

Listing 13-3. Using Dot Extensions for Reset/Preset Functions.

13.4 More On Feedback

Most programmable logic devices provide feedback from some or all of their outputs. In simple cases, you need only refer to the output pin name as an input in your design equations, and the feedback will be properly used. Figure 13-6 shows a portion of a P16R4 logic diagram and illustrates how the feedback is routed in the device for registered and combinatorial output pins. We have seen in earlier examples such as the four-bit counter/multiplexer described in section 10.4, how the feedback is used for devices such as this. In some cases, however, you will need to use more explicit syntax to indicate what kind of feedback you require.

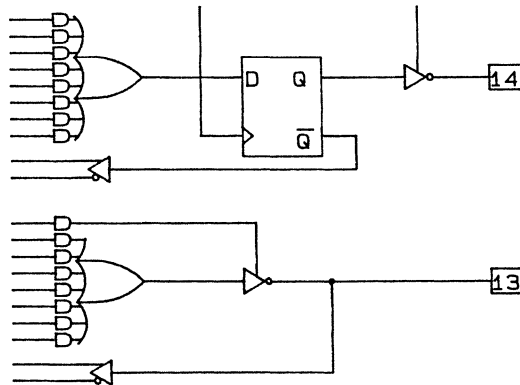


Figure 13-6. Registered and Combinatorial Feedback (P16R4)

13.4.1 Selectable Feedback Type

Some devices, such as the E0310, allow the selection of which feedback path you wish to use for each device output. Figure 13-7 shows the logic diagram for one output

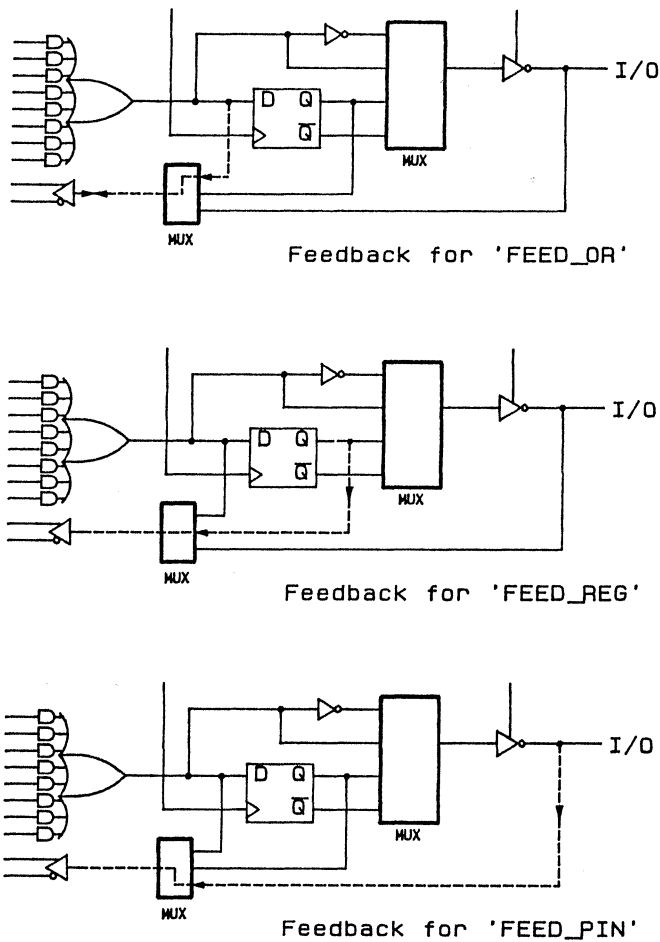


Figure 13-7. Selectable Feedback Paths (E0310)

of a E0310 and illustrates the three selectable feedback paths. Section 13.2.2 describes how to configure the feedback for devices such as this using the `ISTYPE` statement. Once the feedback type has been specified in your design, no special equation syntax is required to use the feedback in your equations.

13.4.2 Multiple Feedback Paths

Devices with multiple feedback paths, such as the P32VX10 device shown in figure 13-8, require that you not only configure the device to indicate the register bypass and feedback paths; you must also use special dot extension notation to indicate which feedback path you are referring to in your equations.

For example, listing 13-4 shows an ABEL source file that demonstrates how to configure and use the outputs of a P32VX10. In the declarations section of the ABEL source file, output pins are declared and the `ISTYPE` statement is used to configure the output polarity of pins Q14 through Q17.

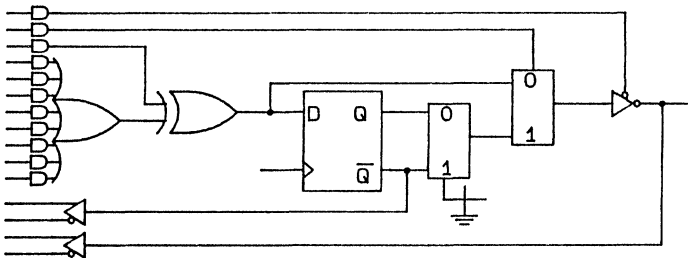


Figure 13-8. Configurable Macro Cell (32VX10)

```

module _test32vx
title 'controlling feedback for MMI 32VX10
Kim-Fu Lim      FutureNet      29 Nov 1987'

    test32vx      device      'P32VX10A';
    Clk,I2,I3,I4,I5,I6      pin  1,2,3,4,5,6;
    I7,I8,I9,I10,I11,I13    pin  7,8,9,10,11,13;
    Q14,Q15,Q16,Q17,Q18    pin  14,15,16,17,18;
    Q19,Q20,Q21,Q22,Q23    pin  19,20,21,22,23;
    AP,SR              node 25,26;

    Q14  istype 'neg';
    Q15  istype 'pos';
    Q16  istype 'neg';
    Q17  istype 'neg';

equations
    Q14.OE      = !I13;
    Q14.FC      = I4;
    !Q14        := I2 & I3;

    Q15.OE      = !I13;
    Q15.FC      = I4;          "Bypass register with I4
    Q15         := I2 & I3;

    Q16.FC      = 1;          "Bypass register always
    Q16         := Q15;       "Feedback from pin

    Q17.FC      = 1;          "Bypass register always
    Q17         := Q15.Q;     "Feedback from Q

test_vectors
    ([Clk,I2,I3,I4,I13,Q15,Q16] -> [Q14,Q15,Q16,Q17])
    [.C., 0, 0, 0, 0, .X.,.X.] -> [ 1, 0, 0, 0 ];
    [.C., 1, 1, 0, 0, .X.,.X.] -> [ 0, 1, 1, 1 ];
    [.C., 0, 0, 0, 0, .X.,.X.] -> [ 1, 0, 0, 0 ];
    [ 0, 0, 0, 0, 1, .X.,.X.] -> [.Z.,.Z.,.X., 0 ];
    [ 0, 0, 0, 0, 1, 0, .X.] -> [.Z.,.X., 0, 0 ];
    [ 0, 0, 0, 0, 1, 1, .X.] -> [.Z.,.X., 1, 0 ];
    [ 0, 0, 0, 1, 0, .X.,.X.] -> [ 1, 1, 1, 0 ];
    [ 0, 1, 1, 1, 0, .X.,.X.] -> [ 0, 0, 0, 0 ];
end

```

Listing 13-4. Output Configurations for a P32VX10

The P32VX10 register bypass feature allows the bypass of the register to be controlled via a product term. The equations for $Q16.FC$ and $Q17.FC$ cause the control terms for these pins to be programmed to always select the register. Register bypass for pins $Q14$ and $Q15$ are controlled dynamically in the design equations by input pin $I4$.

In the design equations, pin feedback is specified for pin $Q15$ by referring to $Q15$ as an input, and register feedback is specified by referring to $Q15.Q$ as an input.

13.4.3 The .Q Dot Extension

In the previous example, register feedback was indicated in the design by using the $.Q$ dot extension when referring to a signal as an input in the equations. Figure 13-9 shows a typical registered output, and where you should assume that the $.Q$ signal originates when you use it in equations; although this is not necessarily where the device feedback actually comes from.

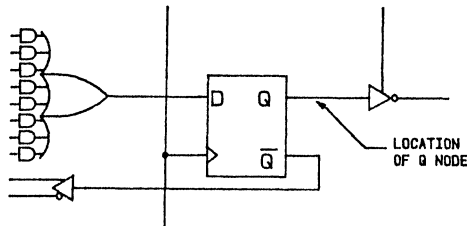


Figure 13-9. Location of the Q Signal in Registered Devices

In most devices, as in figure 13-9, the register feedback does not actually originate at the Q output of the register, but at the complement output. The PARSE module will invert the feedback if it comes from the complement output. This can be seen in the "original equations" section of the DOCUMENT output (*.DOC) when the -q0 parameter is specified on the ABEL command line.

For the purpose of writing equations, however, you should assume the polarity of the .Q signal is the same as the polarity of the Q output of the register, regardless of any inversion the output may have after the register.

13.5 Using Select Multiplexers

Section 13.1 showed how a configurable output enable can be controlled using a combination of ISTYPE statements and equations. These ABEL language features may be used to control a variety of configurable features in complex devices.

Figure 13-10 shows an output macro cell for an E1800 device. This device has selectable clock and enable lines, allowing each macro cell to operate in one of the following modes:

- Clocked from dedicated pin, enable from a term
- Clocked from a term, always enabled

As seen in figure 13-10, two select multiplexers are controlled by a single fuse. This means that only one ISTYPE statement is required to configure both select multiplexers. For example, to indicate that the macro cell for pin FO is to be configured for clocking from term/enabled always, you could write the statement

```
F0.OE  istype  'fuse';
```

in combination with an equation such as

```
F0.OE = 1;
```

Because the both select multiplexers are controlled by one fuse, you could indicate the same configuration with the declaration

```
F0.C    istype  'eqn';
```

(

1

13.6 Using Selectable Register Types

Many devices, such as the F159/7/5 and E0600/900, allow you control the register type for outputs. There are two types of register controls; fuse controlled and term controlled. The fuse controlled register is specified by using the ISTYPE statement. The following register types may be specified in an ISTYPE statement:

reg_D	D type register
reg_JK	JK type register
reg_RS	RS type register
reg_T	T type register
reg_G	G type latch/gated clock
reg_JKD	D/JK controllable register

For example, the statement

```
F0 istype 'reg_D';
```

will configure the register for output F0 as a D type flip-flop.

For term controlled register types, the .FC dot extension is used to write equations for register type control. For example, the equation

```
F0.FC = I2 & I3;
```

indicates that the register type is controlled by pins I2 and I3, and the equation

```
F0.FC = 1;
```

or

```
F0.FC = 0;
```

indicates that the register type should be fixed. The actual type of register that results from this equation depends on what device is being used.

13.7 4-Bit Shifter/Counter Design

When writing equations for outputs with multiple input registers (JK and RS register types) you must use dot extension syntax to access the second input to the register. For example, to write an equation for the K input to a JK register, use the dot extension `.K`.

The following example, a shifter/counter, uses the controllable register type features of an F159 device.

13.7.1 The F159 Logic Diagram

Figure 13-11 is a logic diagram of the F159 FPLS and shows the details of the eight registered-output macro cells. The shifter/counter design used in this example employs the lower four of these outputs. Note that the registers in this device are dynamically programmable as either J-K or D-type flip-flops by means of the FC line. Dynamic programming of the flip-flops is possible as long as the fuses on the *M0* through *M3* lines remain intact and are not blown. (If any of the *Mx* fuses are blown, the associated register becomes a J-K flip-flop permanently.)

The dynamic programming of the output registers allows the use of J-K flip-flops when the counter mode is selected, and D-type when the shift function is selected. This is ideal since J-Ks are better suited to counters and D-types are better suited to shifters.

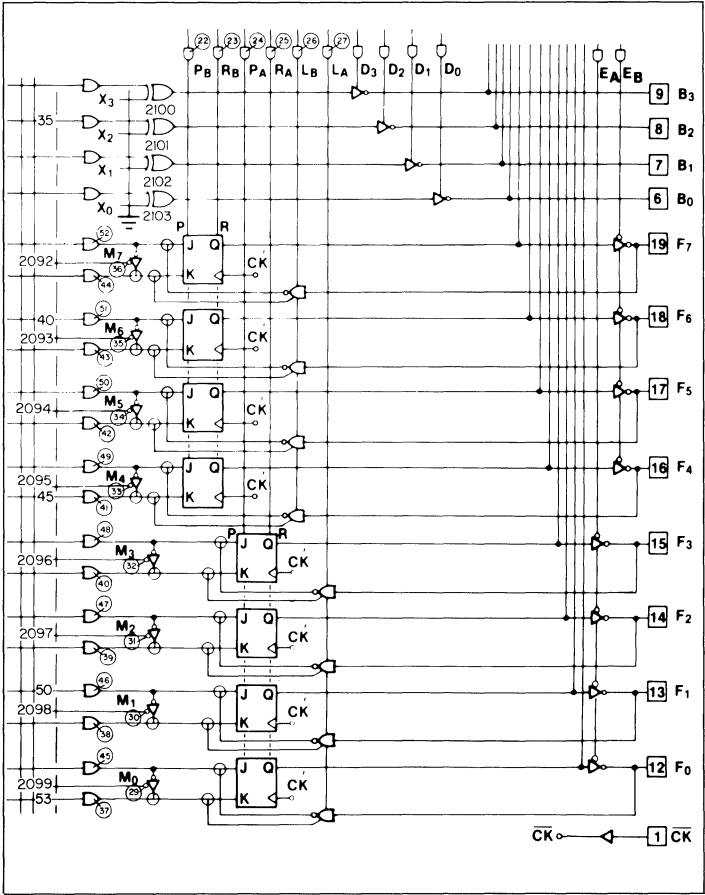


Figure 13-11. Logic Diagram of the F159 FPLS (partial)

13.7.2 Examination of the Source File

The source file for the shifter/counter is shown in listing 13-5. Pin declarations are made to define inputs and outputs; and a node declaration is made to declare the RA node (to provide the clear function for the counter). An ISTATE statement is used to configure the four outputs (F0-F3) as *reg_JKD*.

Sets are used to define the mode control lines and the configuration of these lines for each of the shifter/counter modes. Inputs *JKD*, and *I1* through *I3* are defined as "mode". Under "mode", each shifter/counter function is defined as specific inputs to *JKD* and *I1* through *I3* (pins 2 through 5). The four outputs (F0-F3) are defined as "count", which is used in the first equation to configure the output enable input. (Refer also to section 13.1.3.)

The next equation defines the *FC* (J-K/D-type flip-flop function control) line as equal to input line *JKD* (pin 2). Only *F0.FC* is used, since all flip-flops in the F159 share a common flip-flop control. The remaining equations define the operation of the shifter/counter. Some rules apply to these latter equations and are explained in the following sections 13.7.3 and 13.7.4. The final equation defines the reset input to the flip-flops (RA, node 25) as equal to input line *I3* pulled high. That is,

$$RA = (\text{mode} == \text{Clear})$$

which is equal to

$$\begin{aligned} RA &= [I3, I2, I1, JKD] == [1, X, X, X] \\ &= I3 \end{aligned}$$

```

module shiftcnt  flag '-t1'
title 'universal counter / shift register
Bjorn Benson   Data I/O Corp   25 Nov 1987'

IFL4      device 'F159';

    Clk,JKD,I1,I2,I3,OE      pin 1,2,3,4,5,11;
    F0,F1,F2,F3             pin 12,13,14,15;
    F3,F2,F1,F0             istype 'reg_JKD';
    H,L,X,Z,Ck              =    1,0,.X.,.Z.,.C.;

" Define counter/shift modes
mode      = [I3,I2,I1,JKD];          " [I3,I2,I1,JKD]
Down      = [ 0, 1, 0, 1 ];          Left   = [ 0, 1, 0, 0 ];
Up        = [ 0, 0, 1, 1 ];          Right  = [ 0, 0, 1, 0 ];
Clear     = [ 1, X, X, X ];

count     = [F3,F2,F1,F0];          " Group register outputs into set

equations
" Count Down      JK (T) Flip/Flop
    !F0           := (mode == Down) ;
    F0.K          = (mode == Down) ;
    !F1           := (mode == Down) & F0;
    F1.K          = (mode == Down) & F0;
    !F2           := (mode == Down) & F1 & F0;
    F2.K          = (mode == Down) & F1 & F0;
    !F3           := (mode == Down) & F2 & F1 & F0;
    F3.K          = (mode == Down) & F2 & F1 & F0;

" Count Up        JK (T) Flip/Flop
    [!F0,F0.K]    := (mode == Up) ;
    [!F1,F1.K]    := (mode == Up) & !F0;
    [!F2,F2.K]    := (mode == Up) & !F1 & !F0;
    [!F3,F3.K]    := (mode == Up) & !F2 & !F1 & !F0;

" Shift Left      D Flip/Flop
    !F0           := (mode == Left) & !F3;
    !F1           := (mode == Left) & !F0;
    !F2           := (mode == Left) & !F1;
    !F3           := (mode == Left) & !F2;

" Shift Right     D Flip/Flop
    !F0           := (mode == Right) & !F1;
    !F1           := (mode == Right) & !F2;
    !F2           := (mode == Right) & !F3;
    !F3           := (mode == Right) & !F0;

```

Listing 13-5. A Shifter/Counter in an F159
(continued on next page)


```

@page
equations
    [F3.OE,F2.OE,F1.OE,F0.OE] = !OE;
    [F3.FC,F2.FC,F1.FC,F0.FC] = JKD;
    [F3.RE,F2.RE,F1.RE,F0.RE] = (mode == Clear);

test_vectors
    ([Clk,OE,mode] -> count)
    [ Ck, L,Clear] -> !0 ; " Count Up and Shift Left
    [ Ck, L,Up ] -> !1 ;
    [ Ck, L,Up ] -> !2 ;
    [ Ck, L,Up ] -> !3 ;
    [ Ck, L,Left ] -> !6 ;
    [ Ck, L,Left ] -> !12 ;
    [ Ck, H,Left ] -> Z ;

    [ Ck, L,Clear] -> ^b1111; " Shift right
    [ Ck, L,Down ] -> ^b0000;
    [ Ck, L,Down ] -> ^b0001;
    [ Ck, L,Right] -> ^b1000;
    [ Ck, L,Right] -> ^b0100;
    [ Ck, L,Right] -> ^b0010;
    [ Ck, L,Right] -> ^b0001;

    [ Ck, L,Clear] -> ^b1111; " Shift left
    [ Ck, L,Up ] -> ^b1110;
    [ Ck, L,Left ] -> ^b1101;
    [ Ck, L,Left ] -> ^b1011;
    [ Ck, L,Left ] -> ^b0111;
    [ Ck, L,Left ] -> ^b1110;

    [ Ck, L,Clear] -> !0 ; " Count up
    [ Ck, L,Up ] -> !1 ;
    [ Ck, L,Up ] -> !2 ;
    [ Ck, L,Up ] -> !3 ;
    [ Ck, L,Up ] -> !4 ;
    [ Ck, L,Up ] -> !5 ;
    [ Ck, L,Up ] -> !6 ;
    [ Ck, L,Up ] -> !7 ;
    [ Ck, L,Up ] -> !8 ;
    [ Ck, L,Up ] -> !9 ;
    [ Ck, L,Up ] -> !10 ;
    [ Ck, L,Up ] -> !11 ;
    [ Ck, L,Up ] -> !12 ;
    [ Ck, L,Up ] -> !13 ;
    [ Ck, L,Up ] -> !14 ;
    [ Ck, L,Up ] -> !15 ;
    [ Ck, L,Up ] -> !0 ;
end

```

Listing 13-5. (continued)

13.7.3 Combining Equations

The equations in the "Count Down" portion of the source file can be written as shown, or you can combine pairs of equations that are identical on the righthand side. That is, the two equations

```
!F0 := (mode == Down);  
F0.K = (mode == Down);
```

may be written as the single equation

```
[!F0,F0.K] := (mode == Down);
```

The latter equation shows how the two equations have been ORed into a single equation without altering the specified design of the device. Similarly,

```
!F2 := (mode == Down) & F1 & F0;  
F2.K = (mode == Down) & F1 & F0;
```

may be written as

```
[!F2,F2.K] := (mode == Down) & F1 & F0;
```

13.7.4 Specifying the Flip-Flop Inputs

To specify the flip-flop inputs, the dot extension notation given in the Device Nodes appendix is used directly in the equations. For example, the J and K inputs of the flip-flop associated with the F1 output of the F159 may be noted as F1.J and F1.K, as shown in the upper portion of figure 13-12. The inputs for other flip-flops in the design may be noted in a similar manner as shown below.

```
[F1.J,F1.K] := (mode == Down)  
[F2.J,F2.K] := (mode == Down) & F1 & F0;  
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

This method of flip-flop input notation, shown in the following equations, eliminates the need to declare the nodes for the flip-flop inputs.

Instead of being noted as described above, the J input (but not the K input) can also be noted in terms of the registered output, as shown in the lower portion of figure 13-12. This is due to the fact that whatever logic level is placed on the J input of the register appears at its output. (In figure 13-12, the complement of the output is used due to the inverter in the F159 output.) For example, the equations

```
[F1.J,F1.K] := (mode == Down)
[F2.J,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

could be written as

```
[!F1,F1.K] := (mode == Down)
[!F2,F2.K] := (mode == Down) & F1 & F0;
[!F3,F3.K] := (mode == Down) & F2 & F1 & F0;
```

If a J input is noted in one equation as "F1.J" and in another as "!F1" (the name of the registered output), an inconsistency is introduced into the design equations that will result in an error. The equations

```
[F2.J,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

consistently define the J inputs, however the equations

```
[!F2,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

note the J input in the two different ways.

In the last example, a J input is noted in terms of its J input node and another as the inverse of its registered output. This type of inconsistency will result in a fusemap error when the design is processed by ABEL. It is important to decide on the form used to specify the J inputs and use only that form.

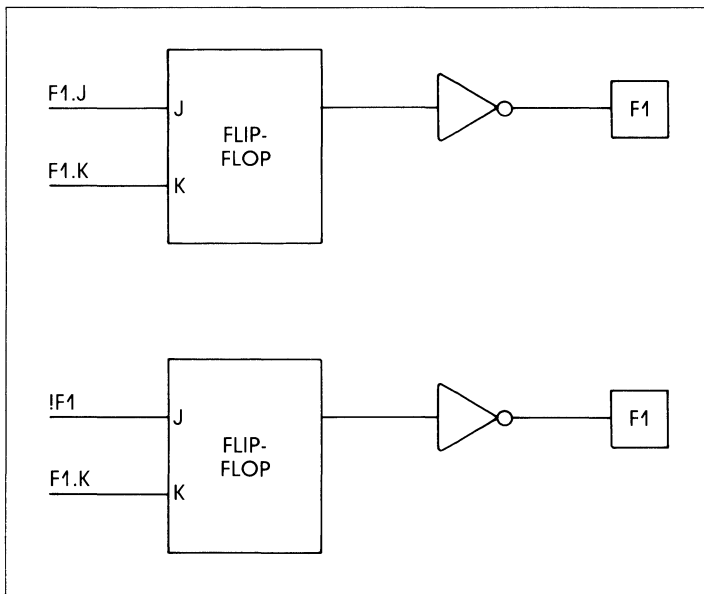


Figure 13-12. Flip-Flop Input Notation Examples

13.7.5 Test Vectors

The test vectors in listing 13-5 are divided into four groups for the sake of functional clarity. The first group simply clears the device and then makes it count up three increments and then shift left twice. This test is provided to show the operation of the design when it is run through the ABEL simulator and then displayed as shown in listing

13-6 (described in text which follows). The second group of test vectors tests the shift right function by first clearing the counter to all positive outputs and then decrementing it to a single positive output. The last four vectors shift the positive bit across the outputs. The third group of vectors operates in a similar manner to test the shift left function. The last group of test vectors test the counter from state 0 through state 15 and then wraparound back to state 0.

The first group of test vectors provide a test, which when the source file of listing 13-5 is run through the ABEL simulator, generate the trace information shown in listing 13-6. (Listing 13-6 is a partial printout of the "shftcnt.SIM" file generated by the SIMULATE element of the ABEL language processor. Listing 13-6 shows the SIMULATE results of the first six test vectors contained in listing 13-5.) The input side of the first test vector shows that pin 5 is high to clear the counter. On the output side of the vector, node 25 (the RA reset register bank A function) is high to clear the counter registers; outputs F0 through F3 show that the counter outputs are all high, indicating the reset condition (the F159 has negative register outputs).

Simulate ABEL(tm) 3.xx

universal counter / shift register

Bjorn Benson Data I/O 9 Mar 1987

File:'shiftcnt.out' Module:'shiftcnt' Device:'IFL4' Part:'F159'

***** Count Up and Shift Left *****

Vector 1

Vector In [CXXX1.....0.....]

Device Out [.....ZZZZ..HHHHZZZZ.LLLLLHLLL.....LLLLLLLLLLLLLLLL]

Vector Out [.....HHHH.....]

Vector 2

Vector In [C1100.....0.....]

Device Out [.....ZZZZ..LHHHZZZZ.HLLLLLLL.....HHLLLLLLHHLLLLLL]

Vector Out [.....LHHH.....]

Vector 3

Vector In [C1100.....0.....]

Device Out [.....ZZZZ..HLHHZZZZ.HLLLLLLL.....HLLLLLLLHLLLLLLL]

Vector Out [.....HLHH.....]

Vector 4

Vector In [C1100.....0.....]

Device Out [.....ZZZZ..LLHHZZZZ.HLLLLLLL.....HHLLLLLLHHHLLLLLL]

Vector Out [.....LLHH.....]

Vector 5

Vector In [C0010.....0.....]

Device Out [.....ZZZZ..HLLHZZZZ.LLLLLLLL.....LLLLLLLLLLHHHLLLLLL]

Vector Out [.....HLLH.....]

Vector 6

Vector In [C0010.....0.....]

Device Out [.....ZZZZ..HHLLZZZZ.LLLLLLLL.....LLLLLLLLLLHLLHLLLLLL]

Vector Out [.....HHLL.....]

Listing 13-6. SIMULATE Output (Partial) for the Shifter/Counter

The second vector of listing 13-6 shows the change in the mode inputs to the count-up function, i.e., pins 4 and 5 at logic high. On the output side of the vector, the F0 pin (pin 12) is negative and the FC line (node 21) is now active to place the flip-flops in the J-K mode as described in section 13.7.1. The third and fourth vectors show that the mode selected on the input side remains the count-up function while the F0 through F3 outputs increment in a binary fashion to a count of three (decimal).

The fifth vector shows that the mode is changed to the shift-left function and the F0 through F3 outputs shift as a result to provide an output of six (decimal). The sixth vector provides a second shift-left function, resulting in a shifter output equal to 12 (decimal) or C (hexadecimal).

13.8 Using Complement Arrays

The complement array is a unique feature that is found in some logic sequencers. The following example shows a typical use; i.e., termination of a counter sequence.

Transition equations may be used to express the design of counters and state machines in some devices that contain J-K and/or R-S flip-flops. A transition equation expresses a state of the circuit as a variation of, or an adjustment to, the previous state. This type of equation eliminates the need to specify each node of the circuit, but only those that require a transition to the opposite state.

An example of transition equations usage is shown in listing 13-7, a source file for a decade counter having a single (clock) input and a single latched output. The purpose of this counter is to divide the clock input by a factor of ten and generate a 50% duty-cycle squarewave output. The device used for this design is an F105 FPLS. In addition to its registered outputs, this device contains a set of "buried" (or feedback) registers whose outputs are fed back to the product term inputs. *COMP*, *P0*, *P1*, *P2*, and *P3* are default names for the outputs that are defined in the F105 device file for nodes 49 and 37 through 40. If no other name for these nodes is declared (and none other is in listing 13-7) the default names are used.

Before examining the equations in listing 13-7, note that the decade counter design is shown in figure 13-13. Figure 13-13 is an abbreviated logic diagram of the F105 that shows the "buried" (no output pins) registers arranged as a counter having feedback paths back to the inputs by means of product terms. The counter registers are designated *P0* through *P3* by default in the F105. These node designations can then be used in the equations. The output register is declared as *F0*.

Node 49, the complement array feedback, is declared (as *COMP*) so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero in the event an illegal counter state is encountered. Any illegal state (and also state 9) will result the absence of an active product term to hold node 49 at a logic low. When node 49 is low, figure 13-13 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 listing 13-7, 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 13-13 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.

```

module decade
title 'Decade Counter - Uses Complement Array and Default Node Names
Michael Holley   FutureNet Division, Data I/O Corp   8 Dec 1987'

    TE10          device 'F105';
    Clk,Clr,F0,PR pin 1,8,18,19;

    _State        = [P3,P2,P1,P0]; "State Registers
    H,L,Ck,X      = 1, 0, .C., .X.;

equations
    [P3.AP,P2.AP,P1.AP,P0.AP,F0.AP] = PR; "Async Preset Option

    "Output      Next State      Present State      Input
[F0 , COMP,      P0 ] := !P3 & !P2 & !P1 & !P0 & !Clr; "0 to 1
[   COMP,      P1 ,P0.R] := !P3 & !P2 & !P1 & P0 & !Clr; "1 to 2
[   COMP,      P0 ] := !P3 & !P2 & P1 & !P0 & !Clr; "2 to 3
[   COMP,      P2 ,P1.R,P0.R] := !P3 & !P2 & P1 & P0 & !Clr; "3 to 4
[   COMP,      P0 ] := !P3 & P2 & !P1 & !P0 & !Clr; "4 to 5
[F0.R, COMP,      P1 ,P0.R] := !P3 & P2 & !P1 & P0 & !Clr; "5 to 6
[   COMP,      P0 ] := !P3 & P2 & P1 & !P0 & !Clr; "6 to 7
[   COMP,P3,P2.R,P1.R,P0.R] := !P3 & P2 & P1 & P0 & !Clr; "7 to 8
[   COMP,      P0 ] := P3 & !P2 & !P1 & !P0 & !Clr; "8 to 9
[   COMP,      P3.R,P2.R,P1.R,P0.R] := !COMP; "Clear

"After Preset, inhibit clock until a High-to-Low clock transition.

test_vectors ([Clk,PR,Clr] -> [_State,F0 ])
[ 1 , 0 , 0 ] -> [ ^b1111, H]; " Power-on Preset
[ Ck, 0 , 1 ] -> [ 0 , H]; " Clear to 0
[ Ck, 0 , 0 ] -> [ 1 , H];
[ Ck, 0 , 0 ] -> [ 2 , H];
[ 1 , 1 , 0 ] -> [ ^b1111, H]; " Preset high
[ 1 , 0 , 0 ] -> [ ^b1111, H]; " Preset low
[ Ck, 0 , 0 ] -> [ 0 , H]; " COMP forces to State 0
[ Ck, 0 , 0 ] -> [ 1 , H];
[ Ck, 0 , 0 ] -> [ 2 , H];
[ Ck, 0 , 0 ] -> [ 3 , H];
[ Ck, 0 , 0 ] -> [ 4 , H];
[ Ck, 0 , 0 ] -> [ 5 , H];
[ Ck, 0 , 0 ] -> [ 6 , L];
[ Ck, 0 , 0 ] -> [ 7 , L];
[ Ck, 0 , 0 ] -> [ 8 , L];
[ Ck, 0 , 0 ] -> [ 9 , L];
[ Ck, 0 , 0 ] -> [ 0 , L];
[ Ck, 0 , 0 ] -> [ 1 , H];
[ Ck, 0 , 0 ] -> [ 2 , H];
[ Ck, 0 , 1 ] -> [ 0 , H]; " Clear
end

```

Listing 13-7. Transition Equations for a Decade Counter

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 described in section 13.3.2 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 13-13. Figure 13-13 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.

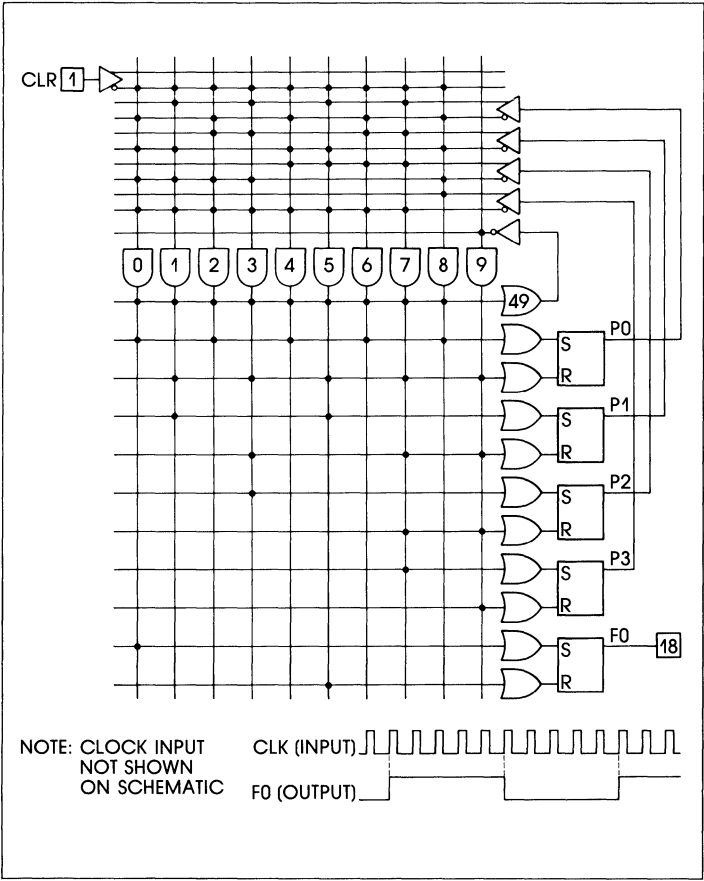


Figure 13-13. Abbreviated F105 Schematic

13.9 Equations for XOR PALs

When writing equations for XOR PALs, you should use parentheses to group those parts of the equation that go on either side of the XOR. This is because the XOR operator (\$) and the OR operator (#) have the same priority in ABEL. See example OCTAL.ABL.

13.10 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 13-14 illustrates the circuitry and the boolean expression:

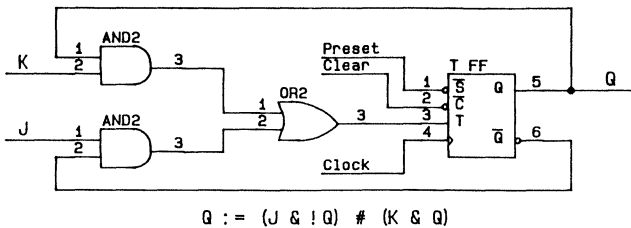


Figure 13-14. JK Flip-Flop Emulation Using T Flip-Flop

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 MMI 22RX8 or the AMD P20XRP8. The circuitry and boolean expression is as shown below in figure 13-15:

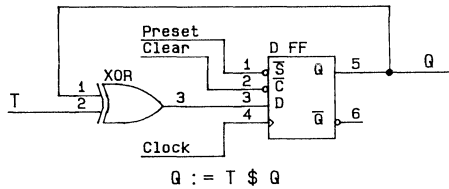
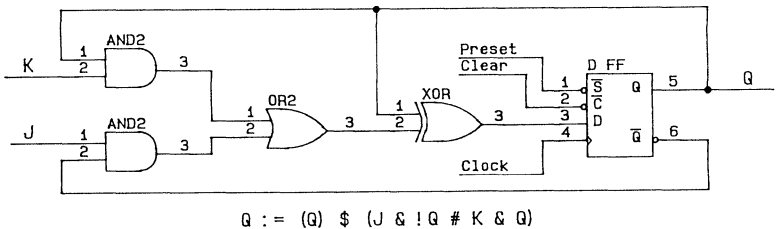


Figure 13-15. T Flip-Flop Emulation Using D Flip-Flop

Finally, a JK flip-flop can also be emulated by combining the D flip-flop emulation of a T flip-flop in figure 13-15 with the circuitry of figure 13-14. Figure 13-16 illustrates this concept:



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

Figure 13-16. JK Flip-Flop Emulation, D Flip-Flop With XOR

The above logic circuitry was derived from *Integrated Circuits in Digital Electronics*, Arpad Barna and Dan Porat, John Wiley and Sons, 1973.

Appendix A. Error Messages

This appendix contains listings of the error messages you may encounter while using the ABEL language processor or any of the utilities provided with the ABEL software package. The error messages are divided into four subsections:

- A.1 General Error Messages
- A.2 Non-fatal Simulation Error Messages
- A.3 TOABEL Error Messages
- A.4 IFLDOC Error Messages
- A.5 IFLDOC Warning Messages
- A.6 ABELLIB Error Messages
- A.7 JEDABEL Error Messages

Within the subsections, the actual error messages appear in smaller typeface. Most of the error messages are listed with a brief description; however those that are self-explanatory are just listed. The symbol 'xxx' in an error message indicates that some actual text or numeric value appears in that location in the message. That text or information may vary depending on the condition causing the error.

Additional information regarding a particular error message can usually be obtained by using the index located at the back of the manual.

A.1 General Error Messages

The following error messages may be displayed on the screen during execution of ABEL programs.

A.1.1 Command Line Errors

Command line errors indicate that an error was made on the command line when one of the language processor programs was invoked. These errors can occur when a program is invoked directly, or from a batch file.

GLOBAL COMMAND LINE ERRORS

illegal argument 'xxx'

An invalid argument (flag) has been specified on the command line.

unable to open input file 'xxx'

unable to open output file 'xxx'

The output file specified can not be created, due to an invalid file name, path or drive specification, or lack of disk space.

input and output file names may not be the same

All file names specified to ABEL programs must have unique names to avoid over-writing input files.

illegal value for flag 'xxx'

A command line argument (flag) that requires a numeric value has been specified with an invalid value.

list and input file names may not be the same

list and output file names may not be the same

All file names specified to ABEL programs must have unique names to avoid over-writing input files.

only one input file allowed, 'xxx'

You have specified more than one input file, either by using the '-I' command line flag, or by omitting the dash ('-') character from another command line argument. Any command line argument not prefixed with a dash is assumed to be an input file name.

only one output file allowed, 'xxx'

More than one output file name was specified using the '-O' flag.

only one listing file allowed, 'xxx'

More than one list file was specified using the '-L' flag.

file name too long

File names specified to ABEL programs must not exceed the file name length limits of the operating system, and the full file name (including path and drive specification) must not exceed 127 characters (64 characters on MS-DOS systems.)

FUSEMAP COMMAND LINE ERRORS

illegal value for unused terms 'xxx', valid = y,n

The value specified using the '-K' flag is invalid; valid values are 'y' or 'n'.

illegal value for checksum 'xxx', valid = 0,1 or 2

The value specified using the '-C' flag is invalid; value may be '0', '1', or '2'.

illegal type of output file 'xxx', valid = 0,82,83,87 or 88

The value specified using the '-D' flag is invalid.

illegal type of output file for wide PROM's with wordsize>8

The value specified using the '-D' flag is not appropriate for the device specified in the design.

SIMULATE COMMAND LINE ERRORS

no device specified

The device type must be specified to SIMULATE by using the '-N' command line flag.

illegal trace level 'xxx', values = 0, 1, 2, 3, 4 or 5

The value specified using the '-T' flag is invalid.

illegal default X value 'xxx', values = 0 or 1

The value specified using the '-X' flag is invalid.

illegal default Z value 'xxx', values = 0 or 1

The value specified using the '-Z' flag is invalid.

illegal breakpoint

The breakpoint information specified using the '-B' flag is incorrect. On MS-DOS systems, you must use a period('.') to separate breakpoint fields, instead of a comma.

A.1.2 Fatal Errors

Can't find device file 'xxx'

The specified device file cannot be found.
ABEL programs attempt to find device files by first searching for the device file itself, and if no file is found, attempt to find the device file in the device file library 'abel3lib.dev'.

The search is performed first in the current (default) directory, then in the directory indicated in the 'abel3dev' variable and, if the device file is still not found, all directories in your 'path' variable.

Equation for 'xxx' needs reduction

Use the next highest level of reduction to eliminate this error.

Equation for 'xxx' is not sums-of-products

There are two reasons for these errors:

1. The input file to FUSEMAP is not the output file from REDUCE.
2. You used operators other than &, # and ! (and \$ for PROMs) and did only a "-R0" reduction in REDUCE. Run REDUCE with a stronger reduction level.

Error writing 'xxxx' file

The output file can no longer be written to. This can arise when a disk is too full, or if the disk data has been damaged.

Illegal operator in equation for 'xxx'

Use the next highest level of reduction to eliminate this error.

Illegal term element at 'xxx'

Use the next highest level of reduction.

Memory overflow

Memory overflow can occur for any of the following reasons:

1. Too many equations or test vectors.
2. Equations are too complex.
3. Equations have too many XOR (\$) operators.
4. Insufficient free memory available.
5. Too many symbols.

(Refer to section 11.6 information on reducing the number of product terms.)

PALs and FPLAs can only be output to a JEDEC file

The output file type must be specified with the "-D0" option (JEDEC format) or left unspecified for PALs and FPLAs.

Premature end of source file

The input file specified is incomplete, possibly due to errors in a previous ABEL program.

PROMs can not be output to a JEDEC file

For PROMs, the output file type must be specified with the "-Dxx" option where xx is not 0.

Too many errors

PARSE terminates execution when many errors are detected in a source file.

Wrong version of intermediate file

The intermediate file specified as input to an ABEL program was generated by an incompatible (older) version of an ABEL program.

A.1.3 Intermediate File Errors

Intermediate files are created by one program in the ABEL language processor and used by one or more of the other programs. If there are errors in an intermediate file, it is likely that subsequent programs needing that file will not operate correctly.

File not produced by PARSE

The intermediate file specified as input to TRANSFOR was not produced by PARSE.

File not produced by TRANSFOR

The intermediate file specified as input to REDUCE was not produced by TRANSFOR.

File not produced by FUSEMAP

The intermediate file specified as input to SIMULATE was not produced by FUSEMAP.

File not produced by REDUCE

The intermediate file specified as input to FUSEMAP was not produced by REDUCE

File not produced by ABEL 3.x

The file was not produced by ABEL 3.x versions of the ABEL programs.

Intermediate file error: TGxxx

An error was detected in the input to TRANSFOR. This may be caused by:

1. An input file generated by PARSE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: RGxxx

An error was detected in the input to REDUCE. This may be caused by:

1. An input file generated by PARSE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: FGxxx

An error was detected in the input to PLDMAP. This may be caused by:

1. Using an input file generated by REDUCE that contained errors.
2. A hardware failure.
3. The file has been modified.

Intermediate file error: SGxxx

These errors occur in SIMULATE for the following reasons:

1. Error in a previous pass.
2. Hardware failure.
3. The file has been modified.

Intermediate file error: DGxxx

These errors in DOCUMENT mean one of the following:

1. Error in an earlier pass.
2. Hardware failure.
3. The file has been modified.

A.1.4 Logical Errors

Logical errors are device-related errors and point to elements of your design that cannot be implemented on the chosen device, or which are implemented incorrectly.

Logical errors may be created by mistakes in the source file or by errors in your design.

Attribute for 'xxx' is 'latch' but pin is register

This input has the wrong attribute assigned to it.
Rewrite your attribute definitions or use a different part.

Attribute for 'xxx' is 'register' but pin is latch

This input has the wrong attribute assigned to it.
Rewrite your attribute definitions or use a different part.

can't map equation for 'xxx' with ISTYPE 'xxx'

The device has a select multiplexer type internal node that can not be programmed using the combination of attribute and equation specified in the design.

can't map equation for select mux 'xxx' - use ISTYPE

The device has a select multiplexer type internal node that can not be programmed with the equation specified in the design. Use an ISTYPE statement to configure the multiplexer properly, or rewrite the equation to match the type of node available in the device.

Can't use an XOR for 'xxx'

The XOR operator is not supported for pin 'xxx'. Use a stronger reduction to eliminate the XORs.

Enable for 'xxx' is a pin, not equation

Enable for this pin is not term controllable.

Enable of 'xxx' can only be a constant or a pin

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Enable of 'xxx' has already been programmed

The enable for this pin is also the enable for other pins and it has already been programmed. Rewrite your equations to eliminate repeat programming of the enable.

Enable of 'xxx' has already been set for this group

The enable for this pin can only be done in groups and the enable for the group that includes this output has already been set. Rewrite your equations so that the enable of this pin is not reprogrammed.

Illegal constant for output 'xxx'

The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Illegal pin for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Inverted input not allowed from 'xxx'

The pin (or node) 'xxx' does not allow the given type of input. Rewrite your equations to use the other type.

Node number 'xxx' is not defined for this device

You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL package to identify device pins and nodes.

Number must be 1 or !0 for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Output 'xxx' used more than once

Some pins can be accessed in more than one way. You have equations referencing this pin in more than one way. Rewrite your equations to use only one method.

Pin can only be negative logic for enable of 'xxx'

The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Pin number 'xxx' is not defined for this device

You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL package to identify device pins and nodes.

Pin or node 'xxx' is already defined as 'xxx'

You have defined the pin or node with multiple names.

Positive input not allowed from 'xxx'

The input from this pin/node can be only of one type. Rewrite your equations to use the other type.

Select mux 'xxx' respecified

The indicated select multiplexer type internal node has been specified with a type that conflicts with a previous usage. Examine the logic diagram for the device and use ISTYPE statement or rewrite equations to properly configure the select multiplexer node.

Signal 'xxx' is not in device 'xxx'

You have specified a pin or node that is not declared in the device.

Signal 'xxx' used or defined as combinational

The indicated signal has been declared as a combinational signal, or is a programmable type and has been used previously as a combinational signal.

Signal 'xxx' used or defined as registered

The indicated signal has been declared as a registered signal, or is a programmable type and has been used previously as a registered signal.

Single fuse equation for 'xxx' must be a constant

The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Status of 'xxx' register has already been set for this group.

The selection of registered/non-registered pins in this part can only be done in groups, and it has already been done for the group that includes this pin. Rewrite your equations so that all the outputs in this group are of the same type ('=' or ':=').

Too many terms for FPLA array at output 'xxx'

The equation for 'xxx' required too many terms programmed into the FPLA. Use a stronger reduction.

Too many terms for output 'xxx'

There are not enough logic terms in the part to program your equation. Either use a stronger reduction (-r2 for example) or rewrite your equations to use fewer terms.

'xxx' attribute not allowed on this device

You have specified a programmable attribute that is not supported on the device.

'xxx' cannot be enabled

You have written an enable equation for a pin that cannot have an enable equation.

'xxx' cannot be programmed inverted

The equation for 'xxx' has the wrong polarity for this pin and the pin can not be programmed to change polarities. Either rewrite your equations or use a stronger reduction which will automatically convert your equations to the correct polarity.

'xxx' has no enable

'xxx' is pin or node identifier.

'xxx' has no registered feedback

You have used pin 'xxx' as an input or feedback pin but it does not have those features. Rewrite your equations so they do not use this pin as an input.

'xxx' is a registered pin

You have used the wrong type of assignment for this pin. Use "[:=" for registered pins and "=" for non-registered pins.

'xxx' is an invalid output

'xxx' is pin or node identifier

'xxx' is not an input pin

Your equation attempts to make use of input or feedback from a pin in the part that is not an input pin and/or has no feedback.

'xxx' is not an input/programmable pin

You have used a pin, that is not an input, as an input in an equation.

'xxx' is not an output pin

You have specified a pin that is not an output pin as an output pin.

'xxx' is not a registered pin

You have used the wrong type of assignment for this pin. Use "!=" for registered pins and "=" for non-registered pins.

'xxx' is not in sums-of-products form

Specify a stronger reduction level and process again.

A.1.5 Preprocessor Errors

Preprocessor errors are syntax errors detected by the preprocessor step of PARSE. The ABEL source is preprocessed before it is actually parsed; the preprocessor expands macros and acts on directives. To correct a preprocessor error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below. Refer to the ABEL Language Reference Manual for the correct syntax for the ABEL source files.

(' expected

';' expected

'=' expected

'xxx' actual arguments expected

Block expected

Constant label expected

Dummy argument expected

Identifier expected

Label expected

Number expected

Number is too large

Radix 'xxx' is not one of 2, 8, 10 or 16

String expected

A.1.6 Syntax Errors

Syntax errors are detected by PARSE and indicate missing, incorrect, or incomplete elements and structures in the ABEL source file. Correct syntax errors by eliminating the error in the source file. The PARSE listing file may aid you in this. The listing file contains pointers that indicate approximately where in a line the error occurs, followed by the error message describing the error. To correct a syntax error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below, with a brief explanation where the message may not self-explanatory. Refer to the ABEL Language Reference Manual for the correct syntax for the ABEL source files.

') ' expected

' -> ' expected

' : ' expected

' : + : ' or ' : * : ' expected

' : > ' expected

' : > ' or ' -> ' expected

' ; ' expected

' = ' or ' : = ' expected

'] ' expected

' THEN ' expected

'xxx' actual arguments specified on command line

Number of arguments found after the MODULE keyword is less than that on the command line.

Actual argument length exceeds 'xxx' chars

Bad Element 'xxx' in range

An intermediate identifier name produced as a result of a range is undefined.

Block expected

Can't have letters imbedded in a number

Can't map set onto a different sized set

Can't map set onto a non-set element

Sets cannot be assigned to numbers, signals or special constants; they can only be assigned to other sets.

Cannot operate on signal 'xxx'

Can only operate on constants in this context.

Closing " of string not found

Closing '}' of block not found

Declaration keyword expected

The valid declaration keywords are:

PIN
NODE
MACRO
DEVICE
ISTYPE
LIBRARY

Device_label expected

Digit not in radix 'xxx'

Don't know what device to use

Either an "IN device__id" is necessary or no device was ever specified.

Dummy argument 'xxx' not recognized

The dummy argument found after the "?" was not specified previously.

Element expected

A set, identifier, number, special constant, or parenthetical expression was expected here.

Enables are not registered

'GOTO', 'IF' or 'CASE' expected

Fuse number expected

Identifier expected

Identifier length exceeds 'xxx' chars

Illegal character 'xxx' in source file

'xxx' is the decimal value of the bad character.

Illegal operation on special constant

Illegal set in state diagram header

Check for illegal nested sets in header

Inconsistency in number of parameters

The number of labels on the left side of a declaration keyword did not match the number of declarations on the right side of the declaration keyword.

Industry part number string expected

Invalid equation for 'fuse' type output, must specify value

Invalid equation for 'pin' type output, must specify signal name

You have written an equation for a select multiplexer type node that does not match the type specified in an ISTYPE statement.

Invalid flag string 'xxx'

Invalid set range

Keyword 'xxx' is out of context

Label 'xxx' is already defined

Label expected

Max of 'xxx' elements in set was exceeded

Mismatch in number of set elements

Sets with different numbers of elements in them cannot be operated on with a binary operator.

'MODULE' expected

Module label doesn't match 'xxx'

If a label is used after the END statement at the end of a module, that label must match the label used with the MODULE statement that began the module.

Multiple mapping to signal 'xxx'

An attempt was made to assign more than one test condition to a single signal.

Negative declaration not allowed

Negative declarations are only allowed for pins and nodes.

No more than 'xxx' args are declared

More actual arguments are used than there were dummy arguments specified.

Number expected

Numeric overflow

A number which requires more than 32 bits to represent it was specified.

Obsolete dot extension - use pin attribute instead

You have used the '.M' dot extension, which has been obsoleted by the ISTYPE 'reg_D' or 'reg_JK' pin attribute. Use an ISTYPE statement or control the register type with the '.FC' dot extension.

Only ^B, ^D, ^H, or ^O radix allowed

Only one label allowed

Operator not allowed

Section keyword or 'END' expected

A section begins with one of the following keywords:

EQUATIONS
TRUTH_TABLE
STATE_DIAGRAM
TEST_VECTORS
FUSES

Signal not allowed

Signal number for 'xxx' is way too large

Signal or .X. expected

Source line length exceeds 'xxx' chars

Special constant must end with a '.'

Special constant not allowed

String expected

String length exceeds 'xxx' chars

Suffix xxx not legal in this context

A shorthand node suffix has been specified that is not supported for the signal specified.

Undefined compiler directive '@'xxx"

Undefined label 'xxx'

Undefined label 'xxx', maybe 'xxx' was meant

Undefined operation on sets

Undefined operation on signal

Undefined special constant 'xxx'

Undefined token

A character was found outside of a valid context in the source file.

Unrecognized attribute 'xxx'

Unrecognized industry part number 'xxx'

The indicated device is not supported by ABEL.

Use '#' for OR instead of '|'

A.1.7 Device File/Internal Errors

Device file and internal errors appear as follows:

Device file error: 'xxx'

"xxx" is a number indicating the type of device file error. Any of these errors point to an error in a device specification file, to a hardware error, or possibly to a bug in the program. If a device file error occurs, note the number of the device file error and contact your Data I/O representative.

Internal error: YYxxx

YY is a two character code indicating the program that failed and xxx is the number of the internal error. If a device file error occurs, and all hardware is operating normally, note the number of the device file error and contact your Data I/O representative.

Internal error: 'xxx', line 'xxx'

If this error occurs, note the file and line numbers indicated and contact your Data I/O representative.

A.2 Non-Fatal Simulation Errors

Non-fatal simulation errors indicate that something is not quite right in the design, but the condition is not severe enough to terminate the simulation process. Pay close attention to these errors, they may point to errors in the design that could cause problems in a programmed device.

Device unstable

The device did not stabilize within 20 iterations.

Both inputs high on RS flip-flop : 'xxx'

Both 'R' and 'S' inputs to the flip-flop were true, resulting in unknown output value.

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

Flip-flop inputs during load : 'xxx'

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

K input high while flip-flop in D mode : 'xxx'

The 'K' input to a JK-D type flip-flop was found to be true when the flip-flop was operating in D mode.

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

Output not High-Z during load : 'xxx'

See data sheet for correct operation of device. Note the states of nodes controlling flip-flops.

A.3 TOABEL Error Messages

Fatal error 701 : Invalid command line argument

Fatal error 702 : Duplicate input and output file name

Fatal error 703 : Error opening input file

Fatal error 704 : Error opening output file

Fatal error 705 : Device not supported

Fatal error 706 : Unexpected end of file encountered

Fatal error 707 : Error writing output file

Syntax error 710 : Invalid symbol ':'

Syntax error 711 : Not a valid pin identifier

Syntax error 712 : Duplicate pin identifier

Syntax error 713 : No device found in input file

Syntax error 714 : Missing '('

Syntax error 715 : Too many elements in enable expression

Syntax error 716 : Expected pin identifier

Syntax error 717 : ':' expected

Syntax error 718 : Undefined pin identifier

Syntax error 719 : Invalid symbol in test vector

A.4 IFLDOC Errors

A.4.1 Command Line Errors

Too many input options at 'xxx'

This error is caused by specifying more than three input parameters. This also can be caused by having unintentional spaces inside a parameter.

Missing - in 'xxx';

This error occurs if a "-" is not the first character of a parameter entered on the command line.

Unrecognized parameter, 'xxx'

This error occurs if a character other than "i", "I", "o", "O", "n", or "N" is entered as a parameter.

Multiple input files

More than one input file was specified on the command line. A maximum of one is valid.

Multiple output files

More than one output file was specified on the command line. A maximum of one is valid.

Multiple devices

More than one device was specified on the command line. A maximum of one is valid.

No device type specified

A device type must be specified.

Input and output files have the same name

The input and output files must have different names.

Can't open the file named 'xxx'

The input or output file named 'xxx' specified with the -I or -O flags cannot be opened.

Unknown device type, 'xxx'

The device, 'xxx', is not supported, or the device type was entered incorrectly.

A.4.2 Diagnostic Errors

Diagnostic errors in IFLDOC indicate that either a hardware failure has occurred or there is a bug in the program. If you can't locate a hardware failure, contact your Data I/O representative. Diagnostic errors are presented in the following form:

Diagnostic Error: 'xxx'

A.4.3 JEDEC Input File Errors

Unrecognized field identifier, 'xxx'

This error occurs when an illegal field identifier is encountered in the JEDEC input file.

Missing '*' or End of File

IFLDOC was unable to find a "*" or EOF marker at the end of a JEDEC field.

The JEDEC file contains references to fuses beyond the maximum for this device

This error occurs when the JEDEC file references more fuses than are available in the specified device. For example, if the JEDEC file references fuse number 5000 in a device with only 200 fuses, this error occurs. Check to see that the device type specified on the command line matches the JEDEC file.

Unrecognized character in JEDEC field, 'x'

A character other than "1" or "0" was found in the fuse field.

Default fuse value specified after fuse field reached

To prevent the loss of fuse information, the fuse default value must be specified before any fuse fields ("L" fields) in the JEDEC file. This error occurs when a default value is found after a fuse field.

A.5 IFLDOC Warning Messages

The following messages are warnings that potential errors may exist in the JEDEC input file, but that those errors are not of a severity that warrants terminating the program.

The number of fuses in this device is not equal to the number given in the JEDEC QF field

You may have the wrong JEDEC file for the device specified on the command line. Check for a match between the JEDEC file and the specified device type.

The JEDEC file may contain less information than is necessary

You may have the wrong JEDEC file for the device specified on the command line. Check for a match between the JEDEC file and the specified device type.

A.6 ABELLIB Error Messages

bad library -- no directory count

bad library -- no directory offset

bad library -- directory entry

The library file specified contains invalid data. Possible causes include: using a library file that was generated on a different operating system; using a library that has been modified; hardware (disk) problems.

can't create file 'xxxxxxx'

The indicated file could not be created when attempting to extract it from the library, due to a lack of disk space, or hardware problems.

can't read file 'xxxxxxx'

The indicated file could not be found or read when attempting to add it to the library.

can't create library file

can't write library file

ABELLIB was unable to create or write the file due to a lack of disk space, hardware problems, or incorrectly specified library file name or path.

file 'xxxxxx' not in library

The indicated file was not found in the library file.

A.7 JEDABEL/SIMUALTE Error Messages

The following errors can occur while reading a defective JEDEC file. These errors will terminate execution of JEDABEL and display the message

JEDEC File Error at line xx.

followed by one of the following:

Fuse address 'nnn' exceeds fuse limit 'nnn'.

While processing a string of fuses, the calculated fuse address exceeded the maximum allowable fuse address for the device. This error could be caused by selecting the wrong device.

L field address exceeds fuse limit 'nnn'.

The fuses specified in the L field exceeded the maximum allowable fuse address for the device. This error could be caused by selecting the wrong device.

QF value 'nnn' doesn't match number of fuses in the device file

The number of fuses specified in the JEDEC QF field doesn't agree with the device file. This error could be caused by selecting the wrong device.

QP value 'nnn' doesn't match number of pins in the device file

The number of pins specified in the JEDEC QP file doesn't agree with the device file. This error could be caused by selecting the wrong device.

QV value 'nnn' doesn't is greater than maximum number of vectors allowed

There is insufficient memory allocated by JEDABEL for the number of test vectors in the JEDEC file.

Illegal fuse state 'x' at fuse address 'nnn'.

The allowable fuse states are "0" and "1".

Illegal default fuse state 'x'.

The allowable default fuse states are "0" and "1".

File Fuse Checksum = nnnn

Hex number expected

The fuse checksum contains illegal characters. The fuse checksum can consist only of hexadecimal characters (i.e., 0-F).

RAM Fuse Checksum = nnnn

Hex number expected

The fuse checksum contains illegal characters. The RAM checksum can consist only of hexadecimal characters (i.e., 0-F).

File Fuse Checksum = nnnn

Hex number too large

The fuse checksum contains too many characters.

RAM Fuse Checksum = nnnn

Hex number too large

The fuse checksum contains too many characters.

The following warnings can occur while reading a defective JEDEC file. Warnings will be displayed with the message

JEDEC File Warning at line xx.

followed by one of the following:

Skipping unknown JEDEC field 'x'.

A reserved field identifier was used.

Illegal default test condition 'x'.

The allowable default test conditions are "0" and "1".

APPENDIX B. JEDEC Standard Number 3A

B.1 Introduction

This appendix defines a format for the transfer of information between a data preparation system and a logic device programmer. This format provides for, but is not limited to, the transfer of fuse, test, identification, and comment information in an ASCII representation. This format 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.

The standard includes a simple transmission protocol based on traditional PROM formats that allow a device programmer to share a computer serial port with a terminal. This simple 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 simple 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.

```
<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*
L0000 11111011111111111111111111111111*
L0028 10111111111111111111111111111111*
L0056 11101111111111111111111111111111*
L0112 01010111011110111111111111111111*
L0224 01010111101110111111111111111111*
L0336 01010111011101111111111111111111*
V0001 000000XXNXXHHLXXN*
V0002 010000XXNXXHHLXXN*
V0003 100000XXNXXHHLXXN*
V0004 110000XXNXXHHLXXN*
V0005 111000XXNXXHLHXXN*
V0006 111010XXNXXHHLHXXN*
V0007 111100XXNXXHHLHXXN*
V0008 111110XXNXXHLHHLHXXN*
C124E*<ETX>8A76
```

Figure B-1. Example of a PLD Data File

B.2 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

B.3 Special Notations and Definitions

B.3.1 Notational Conventions

In addition to the descriptions and examples, this document 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

B.3.2 BNF Rules and Definitions

The following standard definitions are used throughout the rest of this document:

<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>}

 ::= <space> | <carriage return>

<delimiter> ::= {}

<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>

B.4 Transmission Protocol

B.4.1 Protocol Syntax

This simple 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>`

B.4.2 Computing the Transmission Checksum

The transmission checksum is the 16 bit sum ,i.e., modulo 65,535, of all ASCII characters transmitted between and including the STX and ETX. (Figure B-2.) The parity bit is excluded in the calculation.

Syntax of the transmission checksum:

```
<xmit checksum> ::= <hex-digit>:4

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
```

Figure B-2. Computing the Transmission Checksum

B.4.3 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.

B.5 Data Fields

B.5.1 General Field Syntax

In general, each field in the format starts with an identifier, followed by the information, and terminated with an asterisk. For example, "C1234*" specifies that the checksum of the fuse data is 1234. The design specification header does not have an identifier and must be the first field in the transmission, immediately followed by the STX character.

Syntax of fields:

<field> ::= [<delimiter>] <field identifier>
 {<field character>} '*'

<field identifier> ::= | 'A' | 'C' | 'D' | 'F' | 'G'
 | 'L' | 'N' | 'P' | 'Q' | 'R'
 | 'S' | 'T' | 'V' | 'X'

<reserved identifier> ::= | 'B' | 'E' | 'H' | 'I' | 'J'
 | 'K' | 'M' | 'O' | 'U' | 'W'
 | 'Y' | 'Z'

B.5.2 Field Identifiers

Each field begins with a single character identifier that identifies the field type. Multiple character identifiers can be used to create sub-fields (for example, "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 (* reserved for future use)

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 - *

B.6 Comment and Definition Fields

B.6.1 Design Specification Field

The design specification is the first field in the format. It must be included and it does not have an identifier to signal its start. An asterisk terminates the field. The contents of the design specification are not defined but should consist of:

- User's name and company
- Date, part number, and revision
- Manufacturer's device number
- Other information

Syntax of the Design Specification:

`<design specification> ::= {<field character>} '*'`

Example:

```
File for PLD 12S8
Created on 8-Feb-85 3:05PM
6809 memory decode 123-0017-001
Joe Engineer Advance Logic Corp *
```

If none of the above information is required, a blank field consisting of the terminating asterisk is a valid design specification.

B.6.2 Note Field (N)

The note field is used to place notes and comments in the data file. The note field(s) may appear anywhere in the file and the receiving equipment may ignore this field.

Syntax of the Note Field:

<note> ::= 'N' <field characters> '**

Example:

N Following vectors were modified for ACME 123 tester*

B.6.3 Device Definition Field (D) (Obsolete)

This field is now obsolete. It has been eliminated to ensure that the format is device and technology independent.

B.6.4 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 allocate memory efficiently and perform certain calculations. The QF field tells the receiving equipment how much memory to reserve for fuse data, the number of fuses to set to the default condition, and the number of fuses to include in the fuse checksum.

The value fields must occur before any device programming or testing fields in the data file. Files with only testing fields do not require the QF field and files with only programming fields do not require the QP and QV fields.

Syntax for Value Fields:

<fuse limit>	::= 'QF' <number> '**'
<number of pins>	::= 'QP' <number> '**'
<vector limit>	::= 'QV' <number> '**'

Example:

QF1024*	Indicates device has 1024 fuses
QP24*	Indicates device has 24 pins
QV250*	Indicates a maximum of 250 test vectors

B.7 Device Programming Fields

B.7.1 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 '**'
```

Example:

```
F0*
L0000 01001110 00001000 11110000 11111111 01010001*
C021A*
```

B.7.2 Fuse Default States field (F)

The F field defines the states of fuses that are not explicitly defined in the L fields. If no F field is specified, all fuse states must be defined after the QF field and before the first L field.

Example:

F0* Set default to 0

B.7.3 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. For example, "L12" and "L0012" are the same. A space and/or a carriage return must separate the fuse number from the fuse states. The stream of fuse states can be as long as desired (up to the maximum allowable fuse number).

If the state for a fuse is specified more than once, the last state replaces all previous states specified for that fuse. This allows a file to be modified or "patched" by appending new fuse states to the file.

Example:

```
L0000
11111011111111111111111111111111
10111111111111111111111111111111
11101111111111111111111111111111
000000000000000000000000000000*
```

Example:

```
L0000
1111101111111111111111111011111111
1111111111111111
11101111111
1111111111111111
0000000000000000000000000000*
```

Example:

```
L00 1111101111111111111111111111*
L28 1011111111111111111111111111*
L56 1110111111111111111111111111*
L84 0000000000000000000000000000*
```

B.7.4 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, i.e., 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.

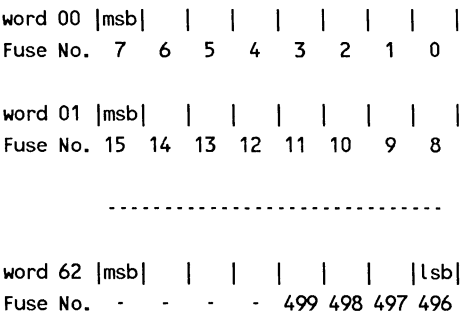


Figure B-3. 8-bit Words Formed From Fuse States for Checksum

QF500*
F0*
L0000 01001110 00001000 11110000 11111111 01010001*
C0212A*

Fuse Number	MSB	LSB
0000	0 1 1 1 0 0 1 0	72
0008	0 0 0 1 0 0 0 0	10
0016	0 0 0 0 1 1 1 1	0F
0024	1 1 1 1 1 1 1 1	FF
0032	1 0 0 0 1 0 1 0	8A
0040	0 0 0 0 0 0 0 0	00
0048	0 0 0 0 0 0 0 0	00
	- - - - -	
0488	0 0 0 0 0 0 0 0	00
0496	- - - - 0 0 0 0	00
	Fuse checksum	021A

Figure B-4. Computing the Fuse Checksum

B.8 Device Testing Fields

B.8.1 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

B.8.2 Default Test Condition Field (X)

The X field defines the input logic level for test vectors not explicitly defined for the "don't care" test condition. The X field will set test vectors 1 through the maximum (set by QV) to the default input test condition. If the X field is used, it must be specified after the QV and QP fields and before the first test vector.

Example:

x1* Set default test condition to 1

In the following example vectors 2 and 5 would default to the "don't care" value of 0 and no outputs would be tested for vectors 2 and 5.

Example:

```
QV5*
QP20*
X0*
V0001 101010000NOZLLHHZ11N*
V0003 111XXXXXXNOZHLLZ11N*
V0004 011XXXXXXNOZLHLHZ11N*
```

B.8.3 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.

Example:

```
V0001 000000XXXNXXXHHHLXXN*
V0002 010000XXXNXXXHHHLXXN*
V0003 100000XXXNXXXHHHLXXN*
V0004 110000XXXNXXXHHHLXXN*
```

The vectors are applied in numerical order to the device being tested. The highest numbered vector to be applied is defined by the QV field. If a vector is not specified during a data transfer, the default value or a vector from a previous transfer will be used. If the same numbered vector is specified more than once, the data in the last vector replaces any data contained in previous vectors with that number. This allows the set of test vectors to be modified or "patched" without transferring the entire set.

B.8.4 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 leftmost condition is applied to pin 1, and the rightmost condition is applied to pin 20 of a 20 pin device, for example. The timing sequence is not defined. A test condition may be applied to pin 5 before or after pin 4.) The P field indicates an alternative correspondence between the test conditions and the pin numbers.

Example:

P 1 2 3 4 5 6 14 15 16 17 7 8 9 10 11 12 13 18 19 20*

V0001 111000HLHHNNNNNNNNNN*

V0002 100000HHLLNNNNNNNNNN*

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).

B.8.5 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 bi-directional (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 XX00XXXXXXNXXXXXXXXXN*  
V02 XX01XXXXXXNXXXXXXXXXN*  
V03 XX11XXXXXXNXXXXXXXXXN*
```

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.

B.8.6 Register Preload

Register Preload means forcing or "jam loading" a register to a known state. Three types of register preloading, "in-circuit", "output register", and "buried register" are defined. The "in-circuit" preload is accomplished with dedicated input pins or internal control logic and uses normal in-circuit logic levels. The standard input and clock test conditions may be used to preload the registers in these devices. The "output register" and "buried register" preload use non-standard levels or "super voltages" to access special modes to preload the registers.

Because super voltages are unique for each device, the following generic methods will allow one set of test vectors to work with "second source" devices. The device programmer/tester will apply the specific super voltage algorithm for each device type.

The "output register" method is used for devices with registers connected to device pins. A P test condition is used to "jam load" registers to a desired state. When the P test condition is applied to the clock pin, the logic level on the register output pin is loaded into the register according to the logic configuration of the device. During preload certain device pins may have to be in a defined state, such as an output enable control pin.

For devices with separate banks of registers, the P test condition is applied to the each clock pin. For example, if pin 1 clocks bank A and pin 2 clocks bank B, a P on pin 2 would preload bank B.

The 0 and 1 input conditions should be used instead of the L and H output test conditions. If the preload must be verified, use a separate test vector to test the outputs.

Example: (16R4 type programmable array logic device)

V1 PXXXXXXXXN1XX1101XXN*	Preload
V2 OXXXXXXXXN0XXHHLHXXN*	Test (don't clock)
V3 CXXXXXXXXN0XXHLHLXXN*	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 010101001N0XXHLLXXN*	(normal vector)

The digit in 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 B111010010001XXXXXXXX*	(preload next 12)

The H and L test conditions can be used to verify the state of the buried registers.

B.9 Programmer/Tester Options

B.9.1 Security Fuse (G)

The security fuse(s) of certain logic devices may be enabled for programming by sending a 1 in the G field. The security fuse prevents the reading of the fuse states. Syntax for the Security Fuse Field:

`<security fuse> ::= 'G' <binary-digit> '**`

Example:

`G1*` Enable security fuse programming.

B.9.2 Signature Analysis Test (S, R, T)

Signature Analysis tests are specified by the S, R, and T fields. The S field defines the starting vector for the test. The possible states are 0 and 1. The R field contains the resulting vector or test-sum. The T field denotes the number of test cycles to be run.

Syntax for Signature Analysis Test:

`<starting vector> ::= 'S' <test condition>:N '**`

`<resulting vector> ::= 'R' <hex-digit>:8 '**`

`<test cycles> ::= 'T' <number> '**`

`N ::= number of pins on device`

Example:

```
S010001000011100011110110*
R5BCD34A7*
T01*
```

B.9.3 Access Time (A)

The A field defines the propagation delay for test vectors in one nanosecond increments. This field may include optional subfields.

Syntax for Access Time

`<access time> ::= 'A'{<field characters>} <number> '**`

Example:

A25*

APD25*

B.10 Data File Examples

[illegible]

Figure B-5. Minimum File for Device Programmer as Defined in JEDEC Standard No. 3, October 1983

```

File for PLD 12S8      Created on 8-Feb-85  3:05PM
6809 memory decode 123-0017-001
Joe Engineer  Advanced Logic Corp *
QF0448*
F0*
L000 11111011111111111111111111111111*
L028 10111111111111111111111111111111*
L056 11101111111111111111111111111111*
L112 01010111011101111111111111111111*
L224 01010111011101111111111111111111*
L336 01010111011101111111111111111111*
C124E*

```

Figure B-6. Data File for Device Programming

```

File for PLD 12S8      Created on 8-Feb-85  3:05PM
6809 memory decode 123-0017-001
Joe Engineer  Advanced Logic Corp *
QP20* QV8*
V0001 000000XXNXXHHHLXXN*
V0002 010000XXNXXHHHLXXN*
V0003 100000XXNXXHHHLXXN*
V0004 110000XXNXXHHHLXXN*
V0005 111000XXNXXHHLHXXN*
V0006 111010XXNXXHHHLHXXN*
V0007 111100XXNXXHHLHXXN*
V0008 111110XXNXXHLHHLHXXN*

```

Figure B-7. Data File for Device Testing

```
File for PLD 12S8      Created on 8-Feb-85  3:05PM
6809 memory decode 123-0017-001
Joe Engineer  Advanced Logic Corp *
QP20*      N Number of pins*
QF0448*    N Number of fuses*
QV8*       N Number of vectors*
G1*        N Program security fuse*
F0*        N Default fuse state*
X0*        N Default test condition*

N Fuse RAM Data*
L0000
11111011111111111111111111111111
10111111111111111111111111111111
11101111111111111111111111111111*
L0112
010101110111101111111111111111*
L0224
010101111011101111111111111111*
L0336
010101110111011111111111111111*
N Test Vectors*
V0001 000000XXXNXXXHHHLXXN*
V0002 010000XXXNXXXHHHLXXN*
V0003 100000XXXNXXXHHHLXXN*
V0004 110000XXXNXXXHHHLXXN*
V0005 111000XXXNXXXHLHXXN*
V0006 111010XXXNXXXHHHXXN*
V0007 111100XXXNXXXHHLHXXN*
V0008 111110XXXNXXXLHHHXXN*
N Fuse RAM checksum*
C124E*
N Signature Analysis test information*
T01*
S00000000000000000000*
R95E4B822*
```

**Figure B-8. 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*  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 111010000N000HHHHOON*
V7 111100000N000HHLHOON*
V5 111000000N000HLHHOON*
V8 111110000N000LHHHOON*
```

**Figure B-9. Data File Showing Position Independence
of fields**

Appendix C. Programmable Logic Device Information

This appendix provides information that pertains to specific types of programmable logic devices. The types of information provided in this appendix includes:

- programming support for specific devices

- special ABEL considerations for certain devices

- device node numbers (assigned by Data I/O) and their functions

C.1 ABEL Support for Specific Devices

C.1.1 PROM Support

ABEL supports PROMs based on memory size (32 x 8 or 1024 x 4). Ignoring the pin numbers on the chip diagram, it is possible to use ABEL's RA9P8 file (512 x 8) for AMD's 27S15 (24 pins), 27S27 (22 pins) or 27S29 (20 pins). The chip diagram will match the 20 pin device. See the CNT10ROM.ABL example on disk or tape for a state diagram design. This example also shows how to program the power-up initialize words available on the RA10R8 and RA11R8.

C.1.2 Devices Supported by TOABEL

The PALASM (version 1.0 only) to ABEL conversion utility, TOABEL, supports the following devices:

P10H8	P10L8	P12H6	P12L6
P14H4	P14L4	P16C1	P16H2
P16H8	P16HD8	P16L2	P16L8
P16LD8	P16P8	P16R4	P16RP4
P16R6	P16RP6	P16R8	P16RP8
P12H10	P12L10	P14H8	P14L8
P16H6	P16L6	P18H4	P18L4
P20C1	P20H2	P20L10	P20L2
P20L8	P20R4	P20RS4	P20R6
P20R8	P20RS8	P20X10	P20X4
P20X8	P22V10	P20RS10	

C.1.3 Devices Supported by IFLDOC

The SigneticsTM-style program table generator supports the following devices:

F100	F153	F161	F168
F103	F155	F162	F173
F105	F157	F163	F839
F151	F159	F167	F2605
F506	F507	F179	F253
F273	F405		

C.2 Specific Device Information

C.2.1 Altera and Intel EPLDs

The following devices are supported.

<u>ABEL</u>	<u>Altera</u>	<u>Intel</u>	<u>Turbo Bits</u>	<u>Miser Bits</u>
E0310	EP300		--	--
E0310	EP310	5C031	--	--
E0320	EP320	5C032	2914,2915	2912,2913
E0600	EP600	5C060	6480,6481	--
E0900	EP900	5C090	17400,17401	--
E1210	EP1200	5C121		
E1800	EP1800	5C180		

The E0600 and E0900 register mode may be controlled with the `ISTYPE` statement. This version only supports the `D` (`reg_D`) and `T` (`reg_T`) register modes. Emulation for `JK` and `RS` may be done with equations. Refer to the Language section in User Notes for more information about emulation of `JK` flip flops.

Turbo bits allow selection of either speed or power consumption tradeoffs within a device. The default mode is to program turbo bits. To avoid programming the turbo bits, use the following syntax (using the E0600 as an example):

```
FUSES[6480,6481]=^b00
```

Miser bits (in devices that have them) are active by default. When turbo bits are programmed, miser bits must be disabled. Each bit pair must be programmed identically, that is, both bits must be 1 or both bits must be 0.

C.2.2 AMD

Due to conflicting device names, the following AMD parts have been assigned alternative names for use with ABEL:

AMD Name	ABEL Name	AMD Name	ABEL Name
22P10	P22AP10	22XP10	P22XP10
20RP4	P20ARP4	20XRP4	P20XRP4
20RP6	P20ARP6	20XRP6	P20XRP6
20RP8	P20ARP8	20XRP8	P20XRP8
20RP10	P20ARP10	20XRP10	P20XRP10

C.2.3 Exel 78C800

For support of this device contact EXEL directly at the address below. Exel provides MultiMaptm software which, when used with ABEL, supports this device.

Exel Microelectronics, Inc.
2150 Commerce Drive
San Jose, CA 95131

C.2.4 Lattices GALs

The Lattice series 20 and 24 GALs each have three device files. Selection of the correct one is based on three simple rules. For most designs the registered model can be used.

1. If the design requires registers, use the P16V8R, P16Z8R, or P20V8R.
2. If the combinatorial design requires feedback, use the P16V8C, P16Z8C, or P20V8C.
3. If the combinatorial design requires eight product terms per output, use the P16V8S, P16Z8S, or P20V8S.

The models are controlled by two fuses, AC0 and SYN, as show in the following table.

<u>Devices</u>			<u>Model</u>	<u>AC0</u>	<u>SYN</u>
P16V8R	P16Z8R	P20V8R	Registered	1	0
P16V8C	P16Z8C	P20V8C	Combinatorial	1	1
P16V8S	P16Z8S	P20V8S	Small PAL	0	1

The User Signature Word can be programmed with the FUSES statement. See DEMO16V8.ABL for a complete GAL design example.

C.2.5 MMI P20RA10 and P16RA8

The P20RA10 and P16RA8 outputs will be registered unless you set the preset and reset nodes high for combinatorial operation. This differs from previous version of ABEL. The P20RA10 and P16RA8 register nodes may be accessed by the "." extensions to the signal names.

Node Extensions

Preset	.PR	Reset	.RE
Enable	.OE	Clock	.C

The enable can be specified either with the ".OE" or an enable statement. See the example, PORT.ABL.

C.2.6 MMI P32VX10A and P22RX8A

In these devices, the XOR gates can be used either as a conventional gate that XORs two product terms, or as a programmable inverter. The method for specifying the dual feedback and macro cell configuration with ABEL 3.0 differs from that of previous versions. Designs developed using an earlier version of ABEL may not work without some changes. Refer to chapter 13 of this manual for information on how to use multiple feedback paths and how to configure the macro cell using ISTYPE statements.

C.2.7 MMI P16X4/P16A4

The example, LIMIT.ABL, demonstrates the use of macros to control the input pairs for arithmetic PALs, such as the P16X4.

C.2.8 Ricoh and VTI EPALs

The VTI and RICOH EPALs allow product term sharing between two adjacent outputs and the product terms can be used by both outputs at the same time. However, this feature is not directly supported by ABEL. The example file ORXOR.ABL demonstrates one method of utilizing this feature.

C.2.9 Signetics

Signetics has changed their product numbering system. All 82 device prefixes were replaced with PL. For example, 82S159 became PLS159 with the numbering change. For ABEL 3.0, the numbering of these device has been changed further. In ABEL 3.0, the 82S has been dropped from the device name so that, for example, the device file for the F82S159 is now named simply F159. To support the old

numbering system (that may appear in existing ABEL source files) each of the following devices is provided with duplicate device files, one for each device number:

Old Device Name	New Device Name
F82S100	F100
F82S103	F103
F82S153	F153
F82S173	F173

All other devices with the F82S prefix (such as the F82S159) are no longer supported by ABEL 3.0 and only the new device files are provided, such as F159, F163, etc.

Also changed for ABEL 3.0 is the way in which "output enables" are specified. Thus, designs for these devices developed using an earlier version of ABEL may require some changes. The ENABLE keyword should no longer be used to specify enable equations; instead, you should write an equation for the .OE extension of the signal name. The example SHIFTCNT.ABL demonstrates how to write enable equations. Refer also to chapter 13 of this manual for information on term controlled output enables.

For the F159 family, the flip/flop control term cannot be used in equations as an input to the complement array (fuse 2091 is always blown). This limitation should not affect most designs and, in those rare cases where it does, the fuse may be controlled with the FUSES statement.

The Truth Table and State Diagram sections default to D flip/flop equations. This default may be changed by setting the pin attribute to 'Reg_JK' with an ISTYPE statement to configure the register as a JK type, or 'Reg_JKD' to configure the register as a controllable type (assumed to be JK). The control term may be referenced by the .FC dot extension.

The .M dot extension should not be used; instead, use and ISTYPE statement to configure the register.

Most of the internal nodes can be accessed with "." extensions appended to the signal names.

Node Extension Notation

Load	.L	J input	.J
Preset	.PR	K input	.K
Reset	.RE	Mode Fuse	.M
Enable	.OE	FF control	.FC

The complement term may be addressed with nodes shown below:

	Node Number	Name
F179	32	comp
F405	29,30	comp1, comp2
F473	28	comp
F105	49	comp
F155	28	comp
F157	28	comp
F159	28	comp
F167	43	comp
F168	45	comp

The complement term may be referred to in your ABEL source file with the name COMP, with no need to declare COMP in the declarations section of the source file.

For the F105 family of devices, the buried registers may be referred to as P0 through P5, with no node declaration required.

C.2.10 F405

The F405 is supported with a single device file in ABEL 3.0. This device features a select multiplexer that is used to configure the clocks. The example DEMO405.ABL demonstrates the use of this device. Refer to chapter 13 of this manual for more information on the select multiplexer.

C.3 Device Nodes

Table C-1 lists the nodes used by ABEL along with the names given to those nodes by the manufacturer. Refer also to the manufacturer's logic diagrams or those supplied in the Programmable Logic Device Schematics publication. Device names are given in boldface followed by the number of pins on the device; node numbers follow. Detailed information about device nodes used in an ABEL design can be obtained by specifying the DOCUMENT flags -S and -F1 on the ABEL command line. Refer also to the ABEL Language Reference and Applications Guide for information on the use of nodes.

Table C-1. Nodes Used by ABEL

Node No.	Notation	Name	Function
F105 (28 Pins)			
29	.R	RF0	R f/f input for F0 (pin 18)
30	.R	RF1	R f/f input for F1 (pin 17)
31	.R	RF2	R f/f control for F2 (pin 16)
32	.R	RF3	f/f input for F3 (pin 15)
33	.R	RF4	R f/f input for F4 (pin 13)
34	.R	RF5	R f/ input for F5 (pin 12)
35	.R	RF6	R f/f input for F6 (pin 11)
36	.R	RF7	R f/f input for F7 (pin 10)
37	-	P0	S f/f input (and output) for P0
38	-	P1	S f/f input (and output) for P1
39	-	P2	S f/f input (and output) for P2
40	-	P3	S f/f input (and output) for P3
41	-	P4	S f/f input (and output) for P4
42	-	P5	S f/f input (and output) for P5
43	.R	RP0	R f/f input for P0
44	.R	RP1	R f/f input for P1
45	.R	RP2	R f/f input for P2
46	.R	RP3	R f/f input for P3
47	.R	RP4	R f/f input for P4
48	.R	RP5	R f/f input for P5
49	-	COMP	Complement term at offset 44
F155 (20 Pins)			
21	.FC	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	.RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load

Table C-1. (cont.)

Node

No.	Notation	Name	Function
28	-	COMP	complementary node at offset 32
33	.K	K0	K f/f input for F0 (pin 14)
34	.K	K1	K f/f input for F1 (pin 15)
35	.K	K2	K f/f input for F2 (pin 16)
36	.K	K3	K f/f input for F3 (pin 17)
37	.J	K4	J0 f/f alt. input for F0 (pin 14)
38	.J	J1	J f/f alt. input for F1 (pin 15)
39	.J	J2	J f/f alt. input for F2 (pin 16)
40	.J	J3	J f/f alt. input for F3 (pin 17)

F157 (20 Pins)

21	.FC	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load
28	-	COMP	complement node at offset 32
35	.K	K0	K f/f input for F0 (pin 13)
36	.K	K1	K f/f input for F1 (pin 14)
37	.K	K2	K f/f input for F2 (pin 15)
38	.K	K3	K f/f input for F3 (pin 16)
39	.K	K4	K f/f input for F4 (pin 17)
40	.K	K5	K f/f input for F5 (pin 18)
41	.J	J0	J f/f alt. input for F0 (pin 13)
42	.J	J1	J f/f alt. input for F1 (pin 14)
43	.J	J2	J f/f alt. input for F2 (pin 15)
44	.J	J3	J f/f alt. input for F3 (pin 16)
45	.J	J4	J f/f alt. input for F4 (pin 17)
46	.J	J5	J f/f alt. input for F5 (pin 18)

Table C-1. (cont.)

Node No.	Notation	Name	Function
F159 (20 Pins)			
21	.FC	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	.RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load
28	-	COMP	complement node at offset 32
37	.K	K0	K f/f input for F0 (pin 12)
38	.K	K1	K f/f input for F1 (pin 13)
39	.K	K2	K f/f input for F2 (pin 14)
40	.K	K3	K f/f input for F3 (pin 15)
41	.K	K4	K f/f input for F4 (pin 16)
42	.K	K5	K f/f input for F5 (pin 17)
43	.K	K6	K f/f input for F6 (pin 18)
44	.K	K7	K f/f input for F7 (pin 19)
45	.J	J0	J f/f alt. input for F0 (pin 12)
46	.J	J1	J f/f alt. input for F1 (pin 13)
47	.J	J2	J f/f alt. input for F2 (pin 14)
48	.J	J3	J f/f alt. input for F3 (pin 15)
49	.J	J4	J f/f alt. input for F4 (pin 16)
50	.J	J5	J f/f alt. input for F5 (pin 17)
51	.J	J6	J f/f alt. input for F6 (pin 18)
52	.J	J7	J f/f alt. input for F7 (pin 19)
F167 (24 Pins)			
25	.R	RF0	R f/f input for F0 (pin 9)
26	.R	RF1	R f/f input for F1 (pin 10)
27	.R	RF2	R f/f input for F2 (pin 11)
28	.R	RF3	R f/f input for F3 (pin 13)
29	.R	RP0	R f/f input for P0 (pin 14)
30	.R	RP1	R f/f input for P1 (pin 15)

Table C-1. (cont.)

Node

No.	Notation	Name	Function
31	-	P7	S f/f input (and output) for P7
32	-	P6	S f/f input (and output) for P6
33	-	P5	S f/f input (and output) for P5
34	-	P4	S f/f input (and output) for P4
35	-	P3	S f/f input (and output) for P3
36	-	P2	S f/f input (and output) for P2
37	.R	RP7	R f/f input for P7
38	.R	RP6	R f/f input for P6
39	.R	RP5	R f/f input for P5
40	.R	RP4	R f/f input for P4
41	.R	RP3	R f/f input for P3
42	.R	RP2	R f/f input for P2
43	-	COMP	Complement term at offset 45

EP300

21	-	R	Reset term at fuse 2628
22	-	P	Preset term at fuse 2592

P22V10 (24 Pins)

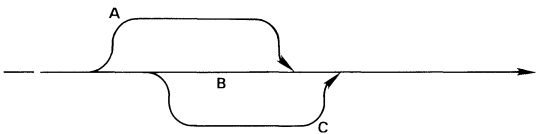
25	-	R	Reset term at fuse 0000
26	-	P	Preset term at fuse 5764

Appendix D. Syntax Diagrams

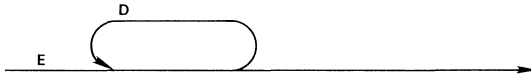
This appendix contains syntax diagrams that formally and completely define the ABEL design language. Section D.1 explains how to read the syntax diagrams. The diagrams begin in section D.2.

D.1 How to Read Syntax Diagrams

Each diagram is preceded on the left side by the name of the language element defined. Follow the diagram in the direction of the arrows to the far right where the diagram ends with a right arrow. In general, the flow is from left to right and top to bottom. Where an intersection exists with a choice of two or more paths, you may follow any one of them. In the example below, you can follow path A, B, or C.

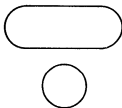


When a path diverges from the main line path and loops back on it, you can follow this loop as many times as desired. For example,

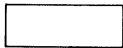


indicates that you can circle around loop D as many times as needed before proceeding along the straight line path E to the right.

Certain shapes in the diagram are significant. The are:

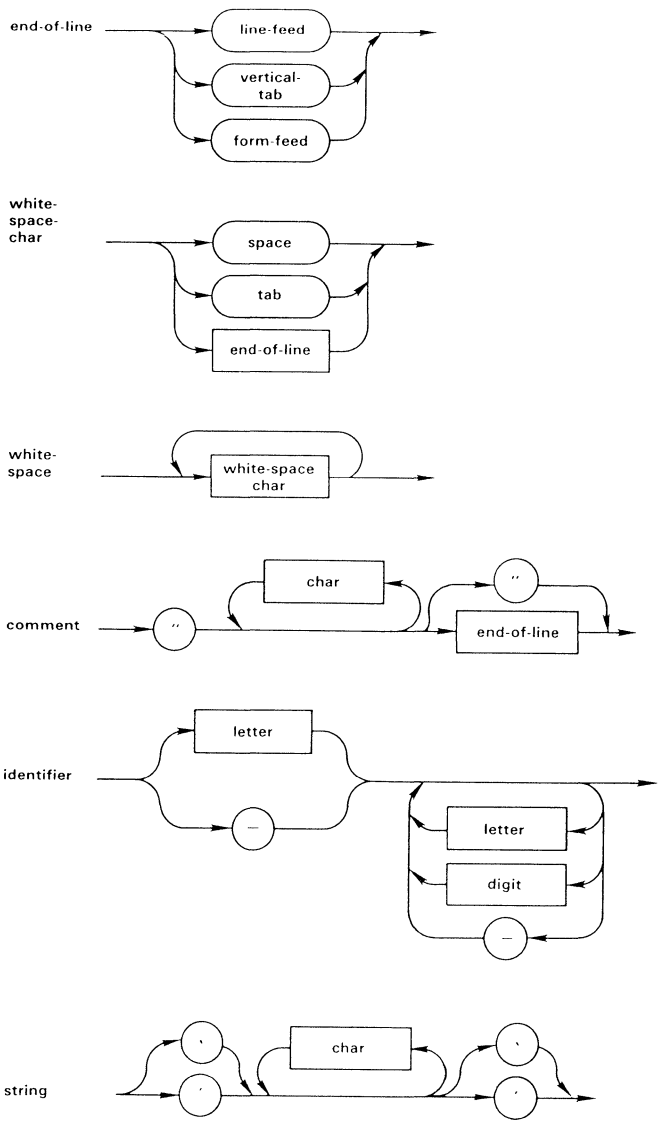


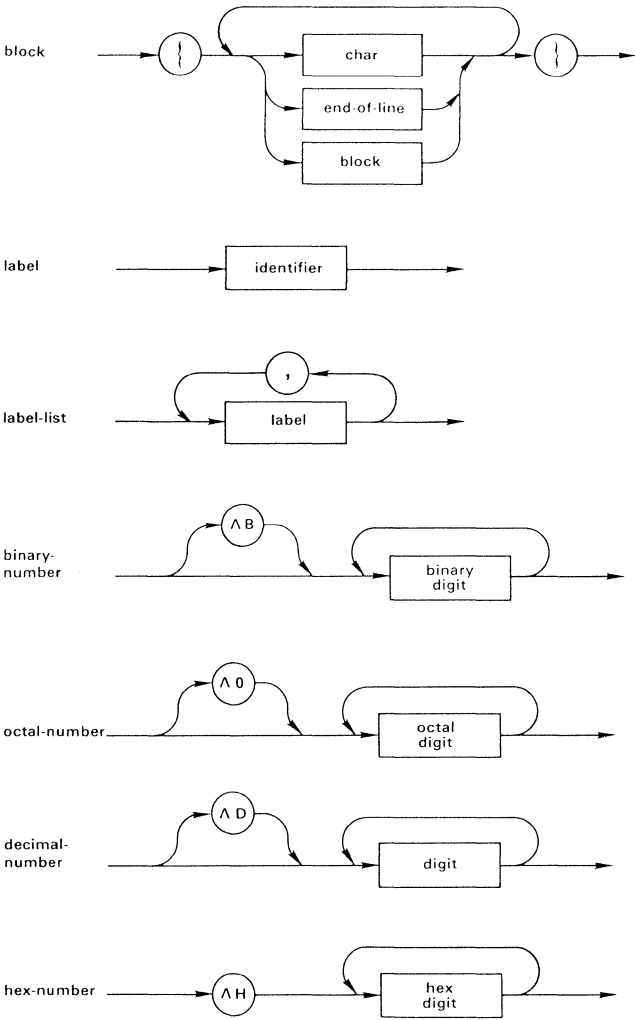
Oblong shapes and circles indicate keywords and punctuation that must be entered as shown.

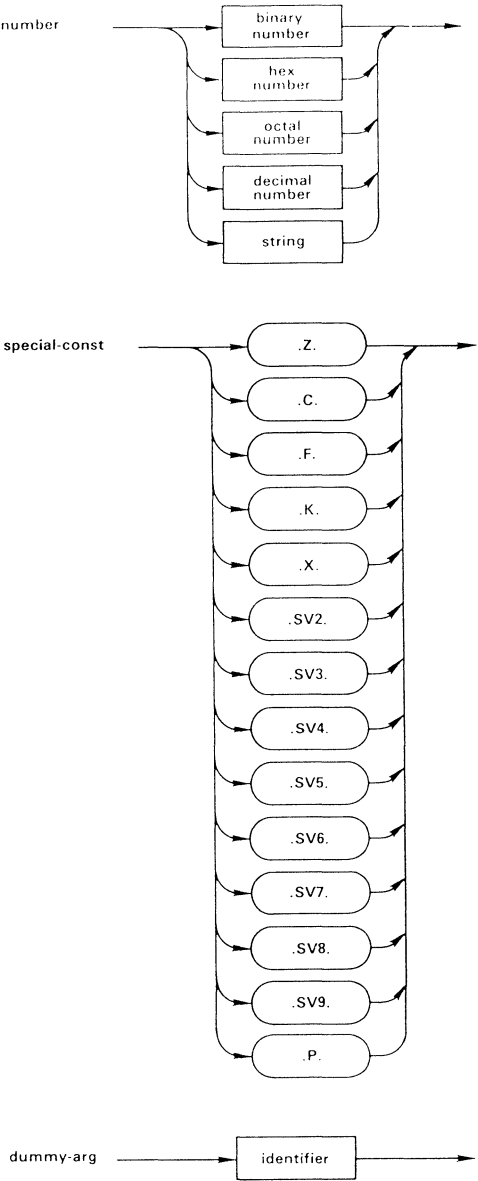


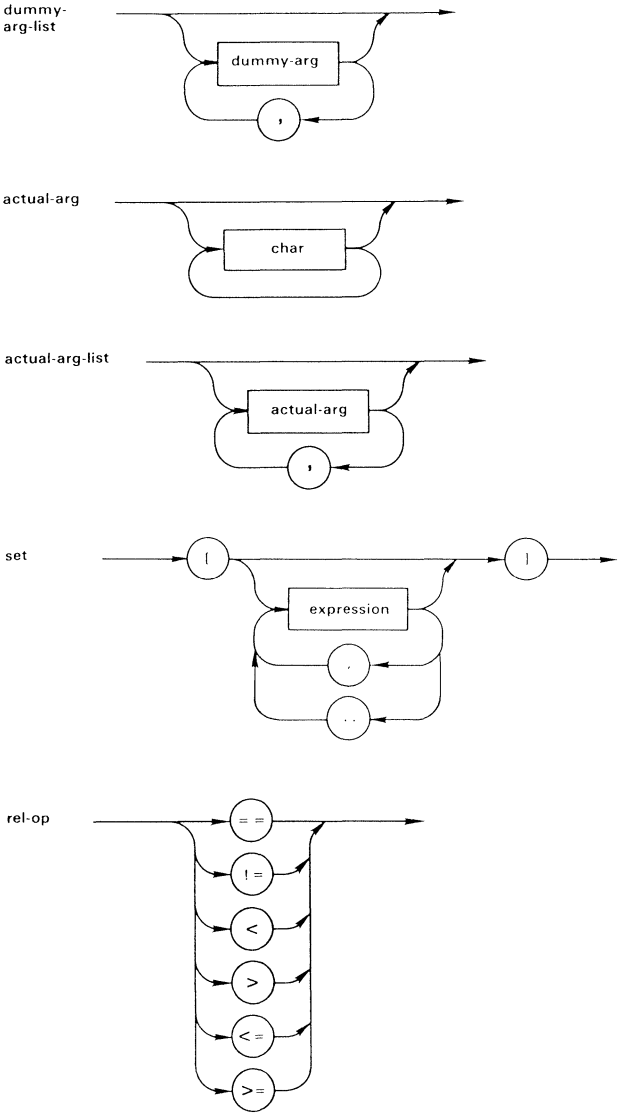
Rectangles indicate elements that are defined by other diagrams.

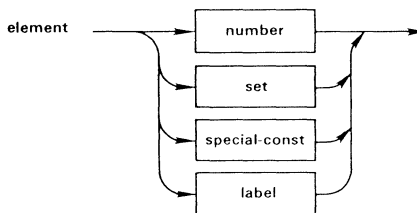
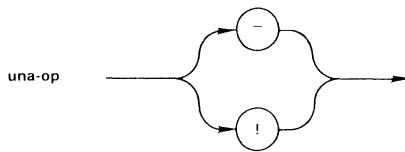
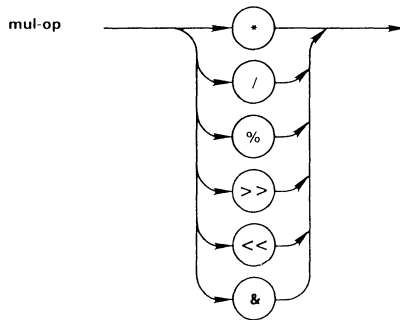
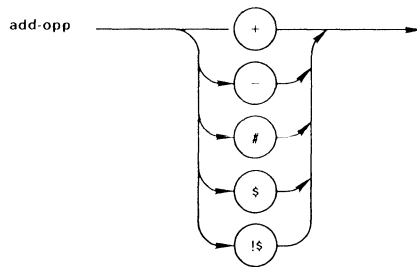
D.2 ABEL Syntax Diagrams

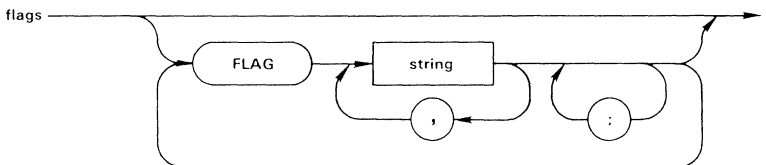
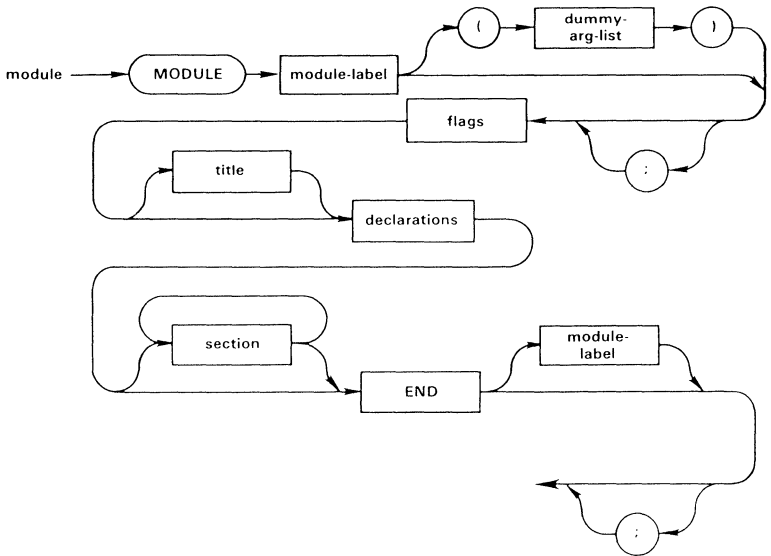
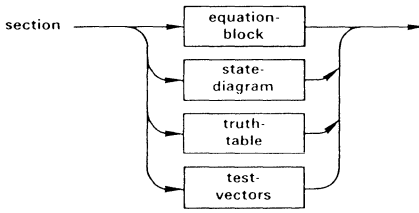


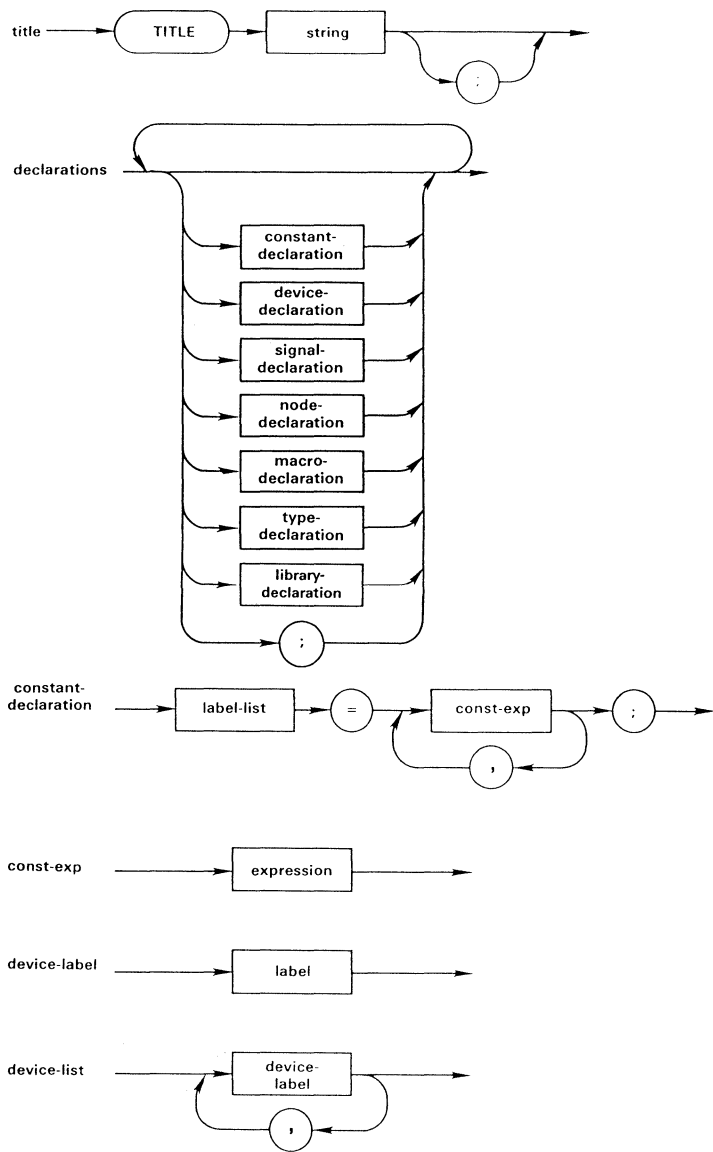




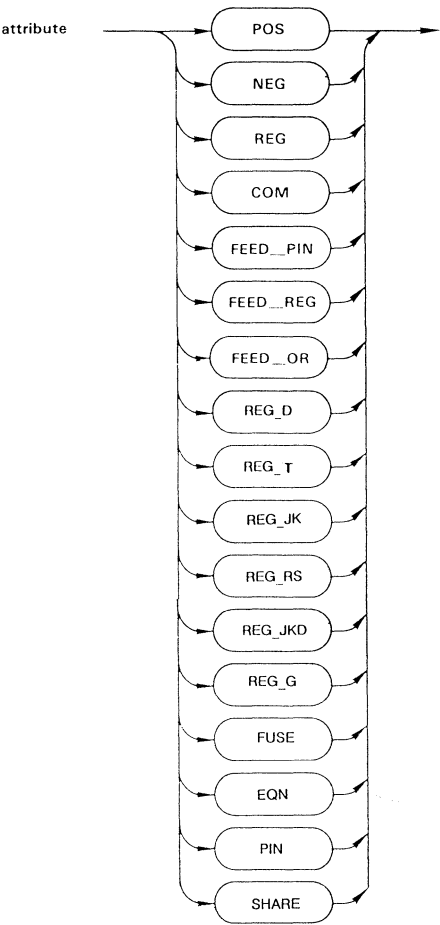


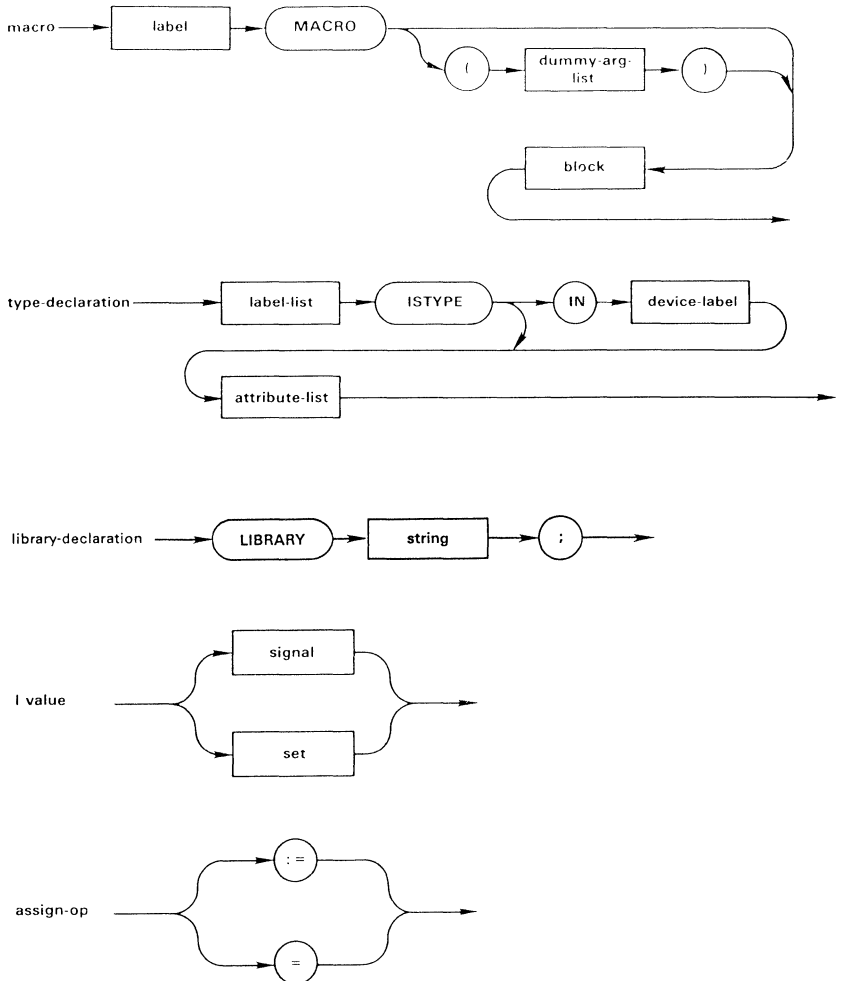


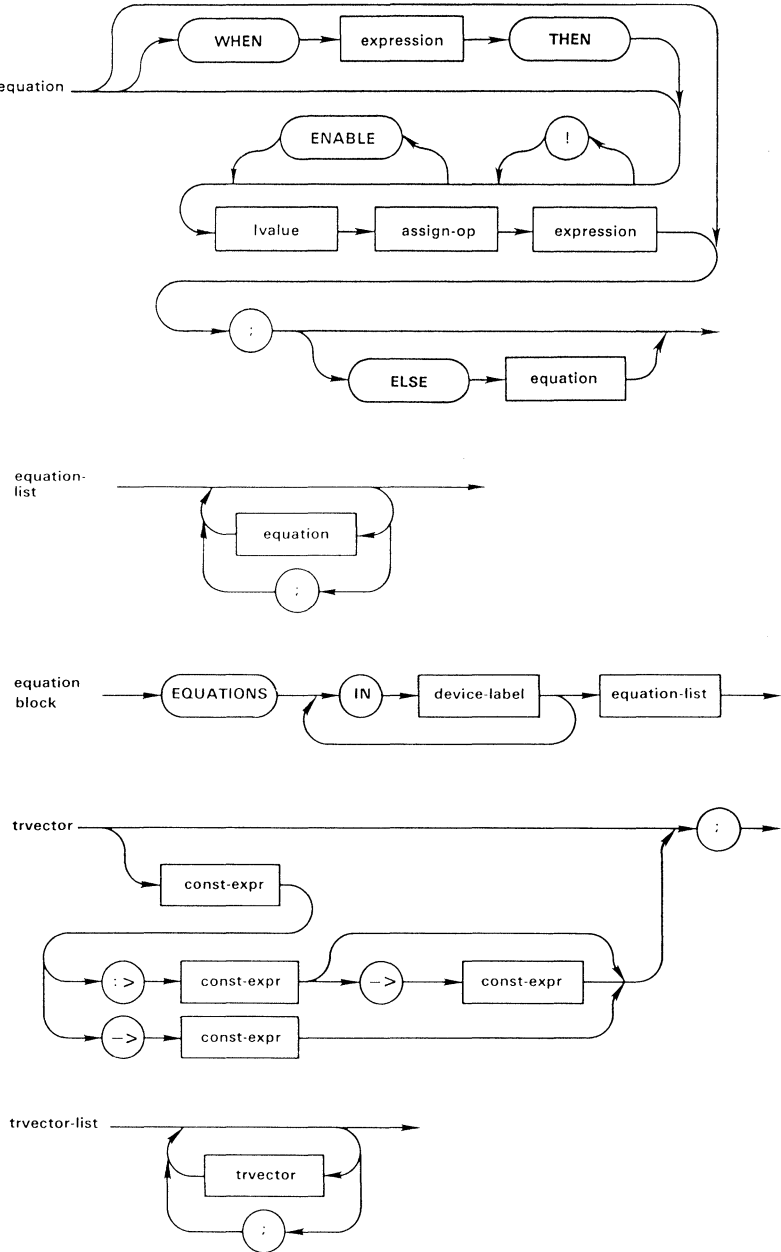


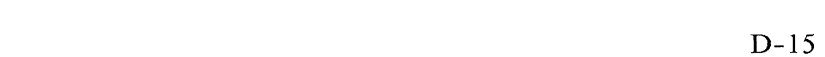


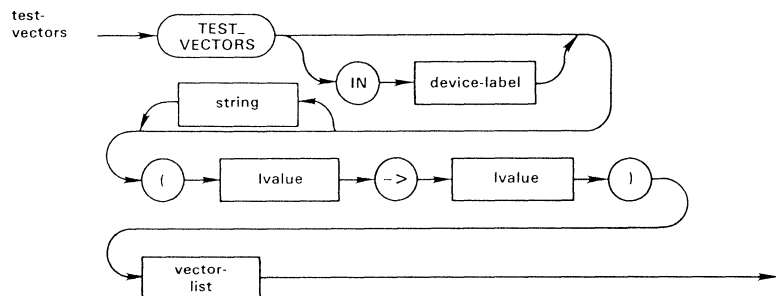
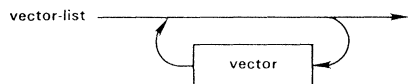
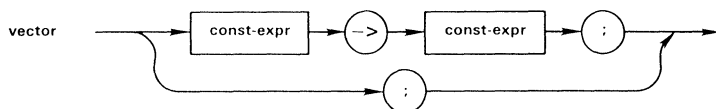
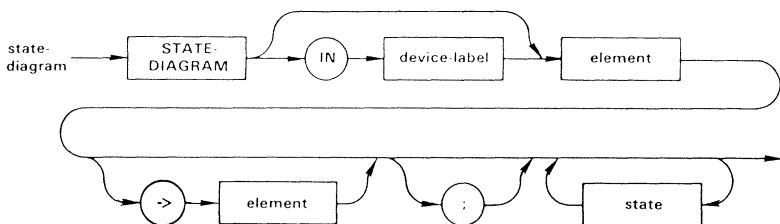
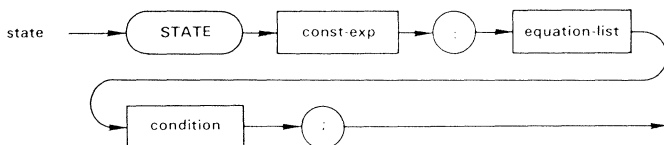
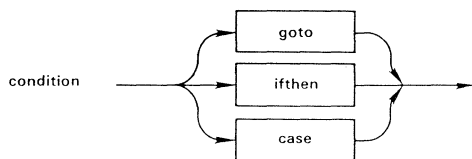


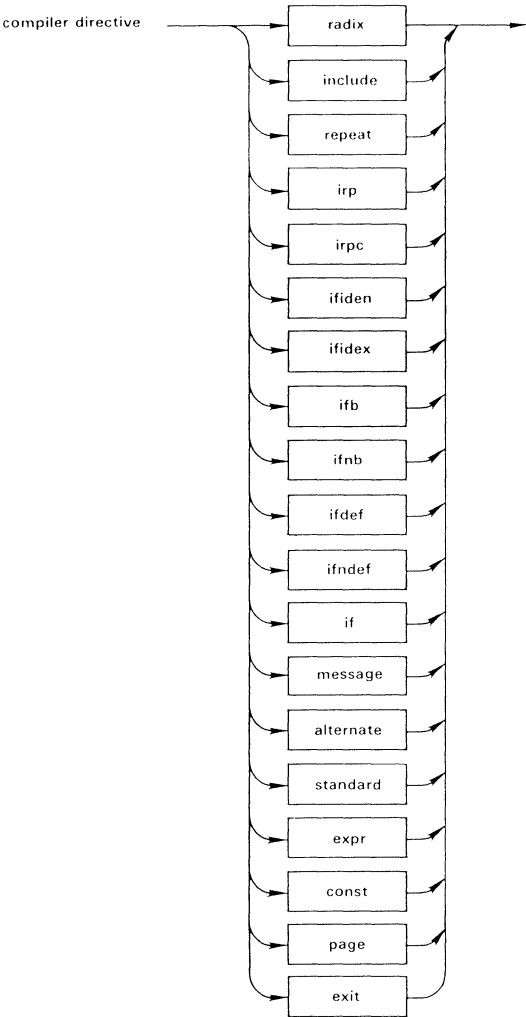




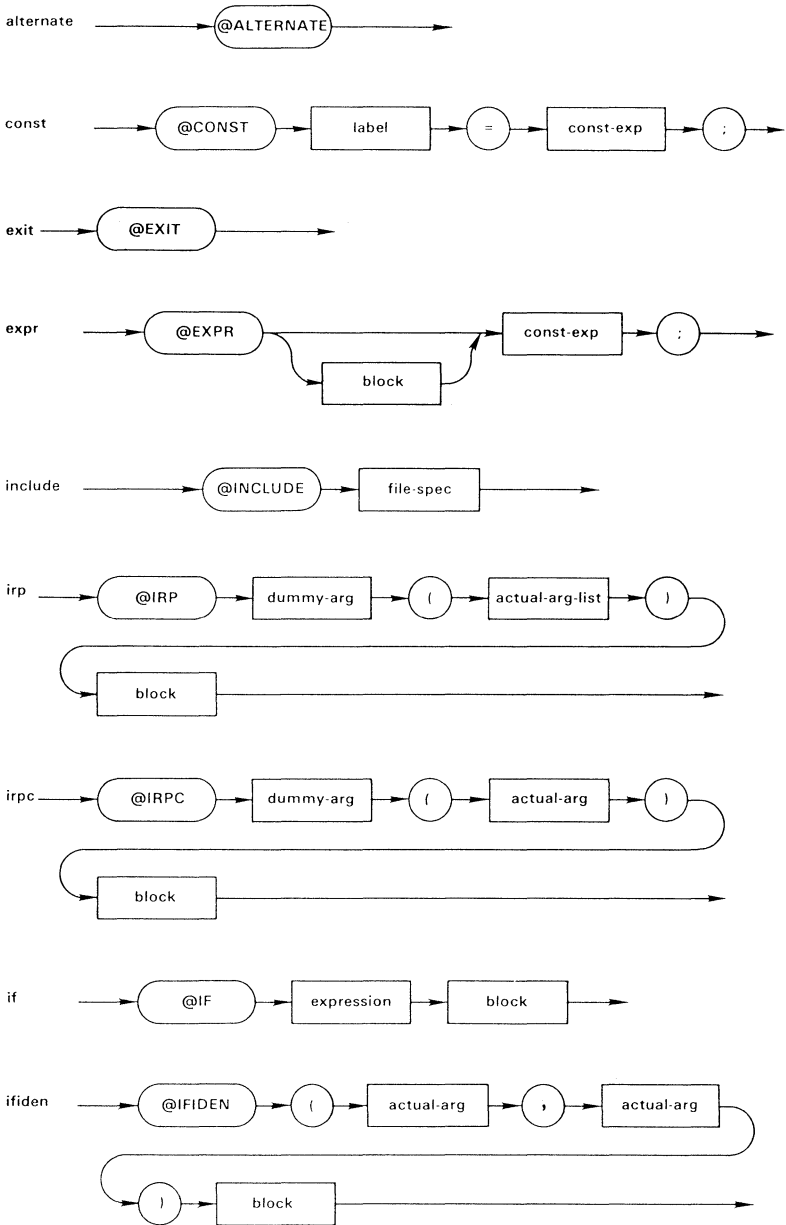


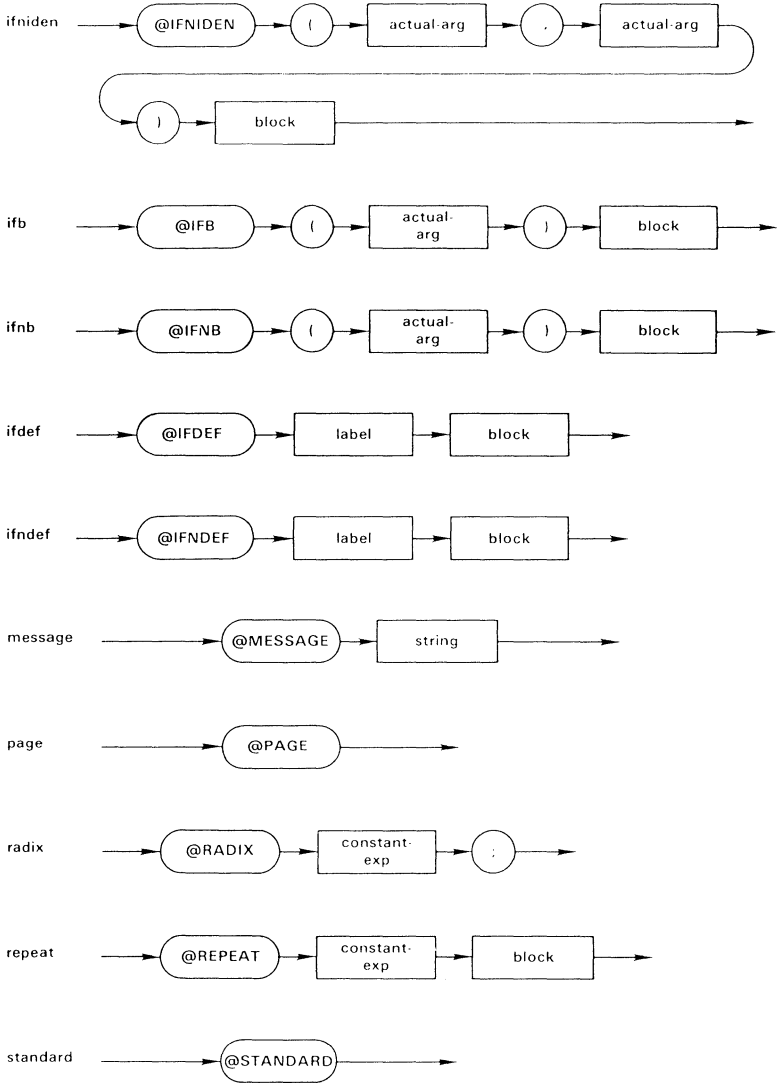






Appendices





Appendix E. ASCII Table

ASCII in Decimal and Hexadecimal

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	/
7	07	BEL	50	32	2	93	5D]
8	08	BS	51	33	3	94	5E	^
9	09	HT	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	SP	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	\$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	
39	27	'	82	52	R	125	7D	}
40	28	(83	53	S	126	7E	~
41	29)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

Appendix F. Set Operations

Set operations are applied according to normal rules of Boolean algebra, as described below. In the rules, upper-case letters are used to denote set names, and lower-case 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 (upper-case 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 .

$$!A_k ::= [!a_k, !a_{k-1}, \dots, !a_1]$$

$$-A_k ::= !A_k + 1$$

$$\text{ENABLE } A_k ::= [\text{ENABLE } a_k, \text{ENABLE } a_{k-1}, \dots, \text{ENABLE } a_1]$$

$$A_k \& B_k ::= [a_k \& b_k, a_{k-1} \& b_{k-1}, \dots, a_1 \& b_1]$$

$$A_k \# B_k ::= [a_k \# b_k, a_{k-1} \# b_{k-1}, \dots, a_1 \# b_1]$$

$$A_k \$ B_k ::= [a_k \$ b_k, a_{k-1} \$ b_{k-1}, \dots, a_1 \$ b_1]$$

$$A_k !\$ B_k ::= [a_k !\$ b_k, a_{k-1} !\$ b_{k-1}, \dots, a_1 !\$ b_1]$$

$$A_k == B_k ::= (a_k == b_k) \& (a_{k-1} == b_{k-1}) \& \dots \& (a_1 == b_1)$$

$$A_k != B_k ::= (a_k != b_k) \# (a_{k-1} != b_{k-1}) \# \dots \# (a_1 != b_1)$$

$$A_k + B_k ::= D_k$$

where:

$$d_n ::= a_n \$ b_n \$ c_{n-1}$$

$$c_n ::= (a_n \& b_n) \# (a_n \& c_{n-1}) \# (b_n \& c_{n-1})$$

$$c_0 ::= 0$$

$$A_k - B_k ::= A_k + (-B_k)$$

$$A_k < B_k ::= c_k$$

where:

$$c_n ::= (!a_n \& (b_n \# c_{n-1}) \# a_n \& b_n \& c_{n-1}) != 0$$

$$c_0 ::= 0$$

$$A_k <= B_k ::= !(B_k < A_k)$$

