

# MicroSim PLSyn

---

PLD/CPLD Design Software

## User's Guide

  
**MicroSim®**  
MicroSim Corporation  
20 Fairbanks  
Irvine, California 92618  
(714) 770-3022

Version 8.0, June, 1997.

Copyright 1997, MicroSim Corporation. All rights reserved.  
Printed in the United States of America.

## TradeMarks

Referenced herein are the trademarks used by MicroSim Corporation to identify its products. MicroSim Corporation is the exclusive owners of "MicroSim," "PSpice," "PLogic," "PLSyn."

Additional marks of MicroSim include: "StmEd," "Stimulus Editor," "Probe," "Parts," "Monte Carlo," "Analog Behavioral Modeling," "Device Equations," "Digital Simulation," "Digital Files," "Filter Designer," "Schematics," "PLogic," "PCBoards," "PSpice Optimizer," and "PLSyn" and variations thereon (collectively the "Trademarks") are used in connection with computer programs. MicroSim owns various trademark registrations for these marks in the United States and other countries.

SPECCTRA is a registered trademark of Cooper & Chyan Technology, Inc.

Microsoft, MS-DOS, Windows, Windows NT and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Exchange and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

EENET is a trademark of Eckert Enterprises.

*All other company/product names are trademarks/registered trademarks of their respective holders.*

## Copyright Notice

Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of MicroSim Corporation.

As described in the license agreement, you are permitted to run one copy of the MicroSim software on one computer at a time. Unauthorized duplication of the software or documentation is prohibited by law. Corporate Program Licensing and multiple copy discounts are available.

## Technical Support

Internet	Tech.Support@microsim.com
Phone	(714) 837-0790
FAX	(714) 455-0554
WWW	<a href="http://www.microsim.com">http://www.microsim.com</a>

## Customer Service

Internet	Sales@MicroSim.com
Phone	(714) 770-3022

---

# Contents

---

## Before You Begin

Welcome to MicroSim . . . . .	xvii
MicroSim PLSyn Overview . . . . .	xviii
How to Use this Guide . . . . .	xix
Typographical Conventions . . . . .	xx
Related Documentation . . . . .	xxi
Online Help . . . . .	xxii
The PLSyn Features In Your Configuration . . . . .	xxiii

## Chapter 1 The Programmable Logic Design Process—An Overview

Chapter Overview . . . . .	1-1
Steps for Designing Systems with Programmable Logic . . . . .	1-2
Design . . . . .	1-3
Simulate . . . . .	1-3
Set Constraints and Priorities . . . . .	1-4
Fit and Partition . . . . .	1-4
Select Device . . . . .	1-5
Simulate with Timing . . . . .	1-5
Program Device . . . . .	1-5

## Chapter 2 Primer: How to Define Programmable Logic

Chapter Overview . . . . .	2-1
Implementing a 3-to-8 Decoder with Programmable Logic . . . . .	2-2
Design Phase: Defining Programmable Logic using Schematic Symbols . . . . .	2-3
Before you begin . . . . .	2-3
Loading and simulating the design . . . . .	2-3
Converting 74LS Symbols to Programmable Logic . . . . .	2-4
Verifying Functionality using Simulation . . . . .	2-5

- Implementation Phase: Fitting and Partitioning the Design . . . . . 2-5
  - Setting Constraints . . . . . 2-6
  - Setting Priorities . . . . . 2-7
  - Partitioning and Fitting . . . . . 2-7
  - Verifying Timing Behavior using Simulation . . . . . 2-8
  - Creating Device Programming Files . . . . . 2-9
  - Back Annotating the Schematic . . . . . 2-9
- Using a DSL Block to Define the Programmable Logic . . . . . 2-10
  - Before You Begin . . . . . 2-10
  - Loading the Design . . . . . 2-10
  - Adding a DSL Block . . . . . 2-11
  - Defining DSL Source Code . . . . . 2-11
  - Equivalent Ways to Define the Decoder with DSL . . . . . 2-13

**Chapter 3 Designing with Programmable Logic**

- Chapter Overview . . . . . 3-1
- The Different Ways to Specify Programmable Logic in Schematics . . . . . 3-2
- Using Programmable Logic Symbols . . . . . 3-2
  - Generic Logic Symbols . . . . . 3-2
  - 74xx Series Logic Symbols . . . . . 3-3
- Using DSL Blocks . . . . . 3-4
  - What Are DSL Blocks? . . . . . 3-4
  - What Are DSL Procedures? . . . . . 3-5
  - Creating a DSL Block in Your Schematic . . . . . 3-6
  - Using the MicroSim Text Editor to Define DSL Procedures . . . . . 3-7
  - Changing the DSL Block Interface . . . . . 3-8
  - Using Existing DSL Source Code . . . . . 3-9
  - Structuring DSL Source Files . . . . . 3-10
  - Calling DSL Procedures and Functions from within a Procedure . . . . . 3-12
- Understanding Programmable Logic Nodes . . . . . 3-13
  - Labeling Nodes . . . . . 3-13
    - Node naming restrictions . . . . . 3-14
    - Labeling interface nodes . . . . . 3-14
  - Creating Active-Low Interface Nodes . . . . . 3-15
  - Converting Internal Nodes to Interface Nodes . . . . . 3-15
  - Creating Physical Nodes . . . . . 3-15
  - Assigning a Logic 0 or 1 to an Input . . . . . 3-16
- Guidelines for Entering Programmable Logic . . . . . 3-16

## Chapter 4 Simulating Programmable Logic Designs

Chapter Overview . . . . .	4-1
Introduction to Simulating with PLogic or PSpice A/D . . . . .	4-2
Setting Up Simulations . . . . .	4-3
Displaying the Dialog Box for Simulation Setup . . . . .	4-3
If you have PLogic . . . . .	4-3
If you have PSpice A/D . . . . .	4-3
Defining Simulation Setup Options for Programmable Logic . . . . .	4-4
Starting Simulations . . . . .	4-4
How the Simulator Uses Programmable Logic I/O Models . . . . .	4-5
Simulating with Timing . . . . .	4-6
Generating Test Vectors . . . . .	4-6
Enabling Test Vector Generation . . . . .	4-7
If you have PLogic . . . . .	4-7
If you have PSpice A/D . . . . .	4-7
How the Simulator Responds . . . . .	4-7
Using the “Sample Window” Control . . . . .	4-8
Example: How the Simulator Creates Test Vectors . . . . .	4-8
Troubleshooting Test Vector Differences . . . . .	4-10
Using Probe Markers . . . . .	4-11
A caution about collapsed nodes . . . . .	4-11

## Chapter 5 Creating the Physical Implementation

Chapter Overview . . . . .	5-1
Overview of the Physical Implementation Process . . . . .	5-3
If You Want More Control . . . . .	5-4
Where to Find Status and Design Information . . . . .	5-4
Activating and Loading PLSyn . . . . .	5-5
Activating PLSyn . . . . .	5-5
From Schematics . . . . .	5-5
From the Windows Program Manager . . . . .	5-5
Loading a Different Design . . . . .	5-6
The PLSyn Main Window . . . . .	5-6
Compiling the Logic . . . . .	5-7
Manually Compiling Logic . . . . .	5-7
Compiling DSL Libraries . . . . .	5-8
Responding to Compile-Time Status and Errors . . . . .	5-8
Controlling Node Generation During Compilation . . . . .	5-9
Resolving “Out of Memory” Conditions . . . . .	5-9

Optimizing the Logic Equations . . . . .	5-10
How the PLSyn Optimizer Synthesizes Logic Equations . . . . .	5-11
Choosing the Optimization Method . . . . .	5-13
Overview of Fitting and Partitioning Logic . . . . .	5-14
If You Don't Have the Partitioning Option . . . . .	5-14
How the PLSyn Fitter Works . . . . .	5-15
Limiting the PLD Parts Available for Search . . . . .	5-16
Constraining Devices . . . . .	5-18
Setting Up User-Defined Constraints . . . . .	5-20
How PLSyn Calculates Maximum Propagation Delay . . . . .	5-22
The Default Constraints File . . . . .	5-22
Prioritizing the Solutions . . . . .	5-23
Using Constraints and Priorities Together . . . . .	5-25
Running the PLSyn Fitter and Partitioner . . . . .	5-25
Selecting Devices . . . . .	5-26
Creating Fuse Maps . . . . .	5-27
Including Test Vectors . . . . .	5-27
The Implementation-Specific Physical Information File (.npi) . . . . .	5-28
Updating the Schematic . . . . .	5-28
Creating PCB Netlists . . . . .	5-29
When You Change the Design . . . . .	5-30

## **Chapter 6 Controlling the Fitting Process Using the .pi File**

Chapter Overview . . . . .	6-1
Introduction to the .pi File . . . . .	6-2
Why Use the .pi File? . . . . .	6-2
Using the Default .pi File . . . . .	6-3
Referring to Nodes in Your Design . . . . .	6-3
Interface nodes . . . . .	6-3
Internal nodes . . . . .	6-3
Controlling PLD Utilization . . . . .	6-5
Fitting a Node as an OUTPUT or NODE . . . . .	6-6
Controlling How Signals Are Fit Together . . . . .	6-6
Disabling Outputs for Test . . . . .	6-8
Controlling Synthesis . . . . .	6-9
Cautions when using the DEMORGAN_SYNTH property . . . . .	6-9
Controlling the Size of Equations . . . . .	6-10
Specifying DEVICES without Specifying Signals . . . . .	6-11
Specifying JEDEC File Names . . . . .	6-12

More Examples Using the .pi File . . . . .	6-13
Forcing Signals to be Fit Together in the Same Device . . . . .	6-13
Using Specific Devices . . . . .	6-14
Maintaining Pin Assignments . . . . .	6-15
Fitting the Design into One Device . . . . .	6-16
Fitting the Design into Multiple Devices . . . . .	6-17
Mixing Automatic and Directed Partitioning . . . . .	6-17
Refitting a Design into the Same Footprint . . . . .	6-18

## **Chapter 7 PLD Device-Specific Fitting**

Chapter Overview . . . . .	7-1
Accessing Internal Points in a PLD Device . . . . .	7-2
The Kinds of Nodes . . . . .	7-2
Hidden nodes . . . . .	7-2
Unary nodes . . . . .	7-4
Unary Nodes in the P330 and P331 . . . . .	7-8
Fitting Specific Device Architectures . . . . .	7-11
22V10, 750, and 2500: Handling Synchronous Preset . . . . .	7-11
Using set and preset for the 22V10 and 750 . . . . .	7-11
Using set and preset for the 2500 . . . . .	7-12
P22V10I: Assigning Combinatorial Output During Feedback . . . . .	7-13
P750B AND P2500B: Controlling Clock Source . . . . .	7-14
P1800: Controlling Quadrant-Based Architectures . . . . .	7-16
Assigning pins and nodes . . . . .	7-16
Subgroups: Targeting quadrants . . . . .	7-17
P16V8HD, P22VP10, and P16VP10: Accessing the Open-Drain Output . . . . .	7-17

## **Chapter 8 MACH 1-4 Device-Specific Fitting**

Chapter Overview . . . . .	8-1
Designing with MACH Devices . . . . .	8-2
When You Have Fitting Problems . . . . .	8-2
Using the log file . . . . .	8-2
Using the report file . . . . .	8-2
Summary of MACH Devices . . . . .	8-3
Output Enable Functions . . . . .	8-3
Register Reset/Preset Functions . . . . .	8-4
Packaging . . . . .	8-4

Using Standard Clock Functions . . . . .	8-5
MACH 1xx, MACH 2xx: Synchronous Clock Functions . . . . .	8-5
MACH 215, 3xx, 4xx: Asynchronous Clock Functions . . . . .	8-5
Using Complex Clock Functions . . . . .	8-6
Clock Limitations . . . . .	8-7
Implementing Hazard-Free Combinatorial Latches . . . . .	8-8
Basic Latch Circuit . . . . .	8-8
Creating a Hazard-Free Latch . . . . .	8-8
Specifying Reserve Capacity . . . . .	8-9
Targeting PAL Blocks . . . . .	8-10
Using Signal Groups . . . . .	8-10
Using Device Sections . . . . .	8-11
Constraining the Size of Combinatorial Nodes . . . . .	8-13
Making Adjustments . . . . .	8-13
Optimizing MACH 4xx Devices Using MAX_XOR_PTERMS . . . . .	8-15
A Few Considerations . . . . .	8-15
Other Optimizing Parameters . . . . .	8-16
Understanding Pin Naming and Numbering . . . . .	8-17
Using the MACROCELL_X## notation . . . . .	8-18
Using the IN_REG_X## notation . . . . .	8-18
Achieving Satisfactory Pinouts . . . . .	8-19
MACH 2xx, 4xx: Using Input Registers . . . . .	8-23
Understanding Input Register Pin Names . . . . .	8-23
MACH 2xx and 4xx Compared . . . . .	8-24
Input Registration . . . . .	8-24
Finding Signals Fit as Unary . . . . .	8-25
Forcing a Function to be Fit as Unary . . . . .	8-26
Preventing a Function from Being Fit as Unary . . . . .	8-26
Preserving Pinouts when Refitting . . . . .	8-27
Plan for Refitting . . . . .	8-27
Method 1: Creating a Two-Level .pi File . . . . .	8-28
Method 2: Floating Nodes . . . . .	8-34
When Fitting into One Device Fails . . . . .	8-35
Using the “Default” Signal Reference . . . . .	8-35
What you can find out in the log file . . . . .	8-36
What you can find out in the report file . . . . .	8-36
Using a Second Device . . . . .	8-37
Accessing the MACH Internal Feedback Path . . . . .	8-38
MACH 215, 4xx: Fitting Asynchronous Functions . . . . .	8-40
PTERM Clock and RESET and PRESET . . . . .	8-40
More Than One RESET/PRESET Pair per PAL Block . . . . .	8-40
MACH 4xx: Using XOR T-Equations . . . . .	8-41



MACH 4xx: Controlling Asynchronous Mode . . . . .	8-42
MACH 4xx: Controlling T-Flop Synthesis . . . . .	8-43
Normal Operation . . . . .	8-43
DFF-Only Fitting . . . . .	8-43
Using the T-Equation . . . . .	8-44
MACH 4xx: Controlling Power-On Reset . . . . .	8-44
What Is a Logical Reset? . . . . .	8-44
The Nominal Case . . . . .	8-45
Exception Cases . . . . .	8-45
MACH 230 and 435: Possible Pin Incompatibility Between . . . . .	8-46
MACH 445 and 465: Configuring for Zero-Hold Time . . . . .	8-47
MACH 445 and 465: Accessing Signature Bits . . . . .	8-48
MACH 1xx and 2xx: Driving or Floating Unused Outputs . . . . .	8-49
Forcing Outputs Driven . . . . .	8-49
Forcing Outputs Floating . . . . .	8-50
The MACH Report File . . . . .	8-52
Obtaining a Report File . . . . .	8-52
Contents of the Report File . . . . .	8-53
Failure Disclaimers . . . . .	8-54
Summary Statistics . . . . .	8-56
Device Resource Utilization . . . . .	8-58
Partitioner Report . . . . .	8-60
Clock Assignments . . . . .	8-60
Signal Directory . . . . .	8-61
Resource Assignment Map . . . . .	8-63
PTERM steering of clusters . . . . .	8-66

## Chapter 9 MACH 5 Device-Specific Fitting

Chapter Overview . . . . .	9-1
Comparing the MACH 5 to Other MACH Architectures . . . . .	9-2
MACH1xx/2xx/3xx/4xx . . . . .	9-3
MACH5xx . . . . .	9-4
Using the .pi File to Control MACH 5 Fitting . . . . .	9-5
Routing in a Segment and Block . . . . .	9-6
Assigning Pins and Nodes . . . . .	9-7
Placing a Signal on an Input Register or Latch . . . . .	9-9
Using Dual Feedback . . . . .	9-10
Forcing the Feedback Path to be Local . . . . .	9-11
Specifying Fanout . . . . .	9-12
Implementing Toggle Register Feedback . . . . .	9-14
Implementing Dual-Edge Clocking . . . . .	9-15

Specifying Reserve Capacity . . . . .	9-16
Constraining the Size of Combinatorial Nodes . . . . .	9-17
Making Adjustments . . . . .	9-17
A Few Considerations . . . . .	9-18
Other Optimizing Parameters . . . . .	9-19
Controlling Power Levels . . . . .	9-19
Controlling Slew Rates . . . . .	9-20
The Document File . . . . .	9-21
The Report File . . . . .	9-22
Heading . . . . .	9-22
Summary Statistics . . . . .	9-23
Power Resource Utilization . . . . .	9-24
Device Resource Utilization . . . . .	9-24
Partition Groups . . . . .	9-27
Signal Directory . . . . .	9-28
Fanout Table . . . . .	9-30
Power Table . . . . .	9-32
Block Configuration Tables . . . . .	9-32

## **Chapter 10 ATV5000 Device-Specific Fitting**

Chapter Overview . . . . .	10-1
Designing with the ATV5000 . . . . .	10-2
Constraining the Size of Combinatorial Nodes . . . . .	10-2
The Effect of MAX_PTERMS . . . . .	10-3
The Effect of MAX_SYMBOLS . . . . .	10-4
Specifying Device Utilization . . . . .	10-5
Using the Flip-Flop Clock Option . . . . .	10-5
Enabling Clocking . . . . .	10-6
Controlling the Clock Source . . . . .	10-6
Using the I/O Pin Latches . . . . .	10-8
Identifying Pins and Nodes . . . . .	10-8
Targeting Quadrants in the ATV5000 . . . . .	10-10
Using the GROUP Construct . . . . .	10-10
Using the SECTION Construct . . . . .	10-11
Placing Node Signals on Buried Logic Cells . . . . .	10-13
Understanding RU Conversion . . . . .	10-14

Understanding Regionalization . . . . .	10-14
Universal and regional PTERMs . . . . .	10-14
Regionalization, sum-term combining, and fitting PTERMs . . . . .	10-15
How PLSyn Does Regionalization . . . . .	10-16
Signal Regionalization . . . . .	10-16
Using input pins . . . . .	10-16
Using feedback paths (UR conversion) . . . . .	10-17
PTERM Regionalization . . . . .	10-17
The Report File . . . . .	10-18
Obtaining Report File . . . . .	10-18
Heading . . . . .	10-19
Failure-to-Partition Disclaimer . . . . .	10-20
Partitioner Report . . . . .	10-20
Signal Directory . . . . .	10-20
Signals Universalized on Sum Term B . . . . .	10-22
Signals Regionalized on Input Pins . . . . .	10-22
Function Placement Report . . . . .	10-22
Quadrant sections . . . . .	10-22
Fit attempt sections . . . . .	10-23
UR conversion report . . . . .	10-23
Pterm regionalization report . . . . .	10-23
Output/node signal placement report . . . . .	10-23
Input Signal Placement Report . . . . .	10-24
Failure-to-Fit Disclaimer . . . . .	10-24

## Appendix A The Documentation File

Appendix Overview . . . . .	A-1
Summary of Documentation File Contents . . . . .	A-2
Reduced Design Equations . . . . .	A-3
Equation Extensions Used in the .doc File . . . . .	A-3
DeMorgan Equations . . . . .	A-4
Equation Display . . . . .	A-5
Partitioning Criteria . . . . .	A-6
Solutions List . . . . .	A-6
Fuse Map Files . . . . .	A-6
Pinout Diagrams . . . . .	A-7
Possible Devices List . . . . .	A-7
Wire List . . . . .	A-7

Appendix BSummary of Files

Appendix Overview . . . . .	B-1
Files Used by PLSyn . . . . .	B-2

Appendix CAMD MACH Device Tables

Appendix Overview . . . . .	C-1
Pin Name Tables . . . . .	C-2
MACH 110 . . . . .	C-2
MACH 111, 111SP . . . . .	C-2
MACH 120, 121 . . . . .	C-3
MACH 130, 131, 131SP . . . . .	C-3
MACH 210, 211, 211SP . . . . .	C-4
MACH 215 . . . . .	C-5
MACH 220, 221, 221SP . . . . .	C-6
MACH 230, 231 . . . . .	C-7
MACH 435, 436 . . . . .	C-8
MACH 445, 446 . . . . .	C-9
MACH 465, 466 . . . . .	C-10
MACH 1xx and 2xx: Fuse Commands for Driving Outputs . . . . .	C-12
MACH 110 . . . . .	C-12
MACH 120 . . . . .	C-13
MACH 130 . . . . .	C-14
MACH 210 . . . . .	C-16
MACH 215 . . . . .	C-17
MACH 220 . . . . .	C-18
MACH 230 . . . . .	C-20

---

# Figures

---

Figure 1-1	PLD Synthesis Design Flow . . . . .	1-2
Figure 2-1	3-to-8 Decoder Schematic . . . . .	2-2
Figure 2-2	Results of Decoder Simulation . . . . .	2-4
Figure 2-3	Back-Annotated Schematic Page . . . . .	2-9
Figure 2-4	The decoder1.sch Example Schematic . . . . .	2-10
Figure 2-5	Connecting the DSL Block . . . . .	2-11
Figure 2-6	Finished DSL Block . . . . .	2-11
Figure 3-1	7400 Symbol as Programmable Logic . . . . .	3-3
Figure 3-2	Relation of PLMODEL Attribute and DSL Procedure Name . . . . .	3-5
Figure 3-3	The DSL Procedure Template . . . . .	3-8
Figure 3-4	A Source Code File for Each DSL Block . . . . .	3-10
Figure 3-5	Single DSL Source Code File with More Than One Procedure . . . . .	3-11
Figure 3-6	Programmable Logic Interface Node Labeled with a Global Port . . . . .	3-14
Figure 5-1	Main PLSyn Window . . . . .	5-6
Figure 5-2	PLSyn Functional Architecture . . . . .	5-15
Figure 7-1	Hidden Node . . . . .	7-2
Figure 7-2	Shadow Node . . . . .	7-3
Figure 7-3	Input Unary . . . . .	7-4
Figure 7-4	Feedback Unary . . . . .	7-4
Figure 7-5	P33x Local Unary . . . . .	7-9
Figure 7-6	P33x Shared Unary . . . . .	7-9
Figure 9-1	Simplified MACH 1xx/2xx/3xx/4xx Block Diagrams . . . . .	9-3
Figure 9-2	Simplified 5xx Block Diagram . . . . .	9-5
Figure 9-3	Mach 5 Architecture . . . . .	9-7

---

# Tables

---

Table 4-1	PLSyn I/O Models . . . . .	4-5
Table 4-2	Test Vectors for Case 1 . . . . .	4-8
Table 4-3	Test Vectors for Case 2 . . . . .	4-9
Table 4-4	Test Vectors for Corrected Case 2 . . . . .	4-10
Table 5-2	Temperature Rating Abbreviations . . . . .	5-19
Table 5-1	Device Selection Constraints Dialog Box Controls . . . . .	5-19
Table 5-3	Solution Priorities Dialog Box Controls . . . . .	5-23
Table 6-1	PLD Utilization Properties . . . . .	6-5
Table 6-2	Synthesis Control Properties . . . . .	6-10
Table 7-1	Node Descriptions and Labels by Device Architecture . . . . .	7-6
Table 7-2	Node Descriptions and Labels for P330 and P331 . . . . .	7-10
Table 7-3	Node Descriptions and Labels for P1800 . . . . .	7-16
Table 8-1	MACH Device Properties . . . . .	8-3
Table 8-2	MACH PAL Block Names . . . . .	8-11
Table 8-3	Minimum and Maximum Number of PTERMS . . . . .	8-14
Table 9-1	MACH 5 Node Names and Pin Numbers . . . . .	9-8
Table 10-1	Equation Extensions Used in the .doc File . . . . .	A-3
Table 10-2	MACH 110 OE Fuse Commands . . . . .	C-12
Table 10-3	MACH 120 OE Fuse Commands . . . . .	C-13
Table 10-4	MACH 130 OE Fuse Commands . . . . .	C-14
Table 10-5	MACH 210 OE Fuse Commands . . . . .	C-16
Table 10-6	MACH 215 OE Fuse Commands . . . . .	C-17
Table 10-7	MACH 220 OE Fuse Commands . . . . .	C-18
Table 10-8	MACH 230 OE Fuse Commands . . . . .	C-20

---

# Before You Begin

---

## Welcome to MicroSim

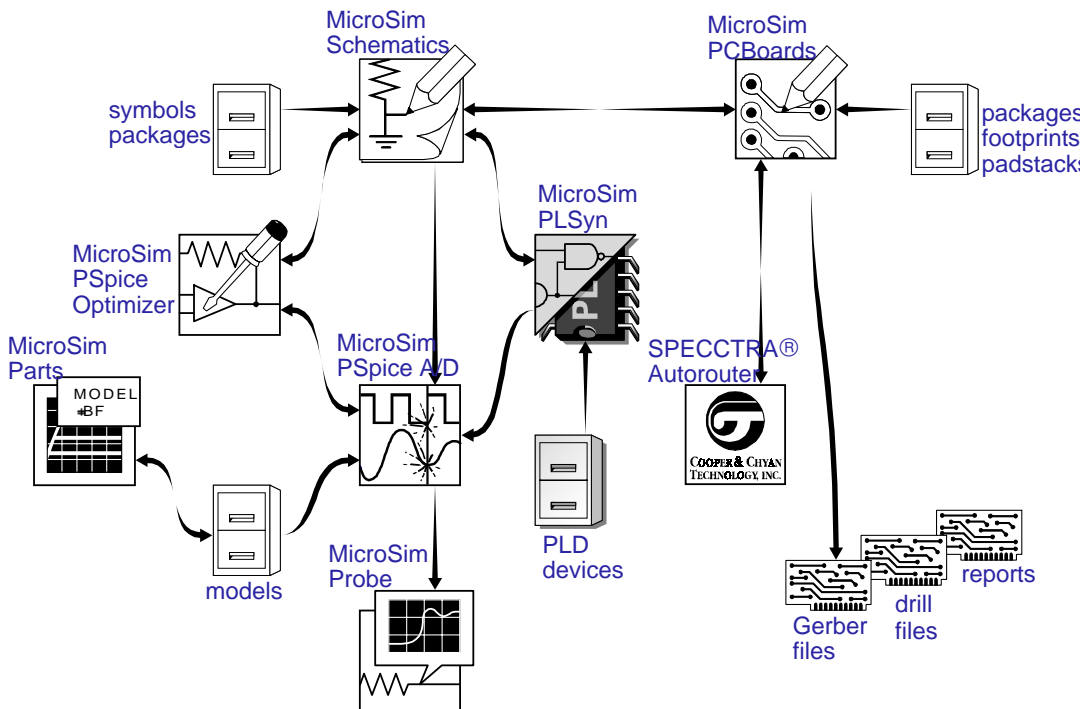
Welcome to the MicroSim family of products. Whichever programs you have purchased, we are confident that you will find that they meet your circuit design needs. They provide an easy-to-use, integrated environment for creating, simulating, and analyzing your circuit designs from start to finish.

# MicroSim PLSyn Overview

MicroSim PLSyn is a programmable logic synthesis program that allows you to synthesize all or any portion of your design into PLD and/or CPLD parts.

PLSyn is fully integrated with other MicroSim programs. This means you can do all of the following within the same environment:

- Design your circuit with MicroSim Schematics.
- Synthesize programmable logic with MicroSim PLSyn.
- Simulate with MicroSim PSpice A/D (for mixed digital and analog simulation) or MicroSim PLogic (for digital logic and timing simulation).
- Analyze simulation results with MicroSim Probe.





# How to Use this Guide

This guide is designed so you can quickly find the information you need to use PLSyn, including:









- how to create and edit designs which use PLDs (schematic and language-based), and
- how to optimize, partition, and fit devices.

This guide assumes that you are familiar with Microsoft Windows (NT or 95), including how to use icons, menus, and dialog boxes. It also assumes you have a basic understanding about how Windows manages applications and files to perform routine tasks, such as starting applications and opening, and saving your work. If you are new to Windows, please review your *Microsoft Windows User's Guide*.

# Typographical Conventions

Before using PLSyn, you need to understand the terms and typographical conventions used in this documentation.

This guide generally follows the conventions used in the *Microsoft Windows User's Guide*. Procedures for performing an operation are generally numbered with the following typographical conventions.

Notation	Examples	Description
	Press 	A specific key or key stroke on the keyboard.
monospace font	Type VAC... or dig_prim.slb	Commands/text entered from the keyboard, or file names.
	 Partitioning Option Required	Feature available in systems with the partitioning option only
	 To improve accuracy...	Tip providing advice or different ways to do things.
	 Be careful...	Cautionary message.

# Related Documentation

Documentation for MicroSim products is available in both hard copy and online. To access an online manual instantly, you can select it from the Help menu in its respective program (for example, access the Schematics User’s Guide from the Help menu in Schematics).

**Note**    *The documentation you receive depends on the software configuration you have purchased.*

The following table provides a brief description of those manuals available in both hard copy and online.

This manual...	Provides information about how to use...
MicroSim Schematics User’s Guide	MicroSim Schematics, which is a schematic capture front-end program with a direct interface to other MicroSim programs and options.
MicroSim PCBoards User’s Guide	MicroSim PCBoards, which is a PCB layout editor that lets you specify printed circuit board structure, as well as the components, metal, and graphics required for fabrication.
MicroSim PSpice A/D & Basics+ User’s Guide	PSpice A/D, Probe, the Stimulus Editor, and the Parts utility, which are circuit analysis programs that let you create, simulate, and test analog and digital circuit designs. It provides examples on how to specify simulation parameters, analyze simulation results, edit input signals, and create models.
MicroSim PSpice & Basics User’s Guide	MicroSim PSpice & MicroSim PSpice Basics, which are circuit analysis programs that let you create, simulate, and test analog-only circuit designs.
MicroSim PSpice Optimizer User’s Guide	MicroSim PSpice Optimizer, which is an analog performance optimization program that lets you fine tune your analog circuit designs.
MicroSim FPGA User’s Guide	MicroSim FPGA—the interface between MicroSim Schematics and XACTstep—with MicroSim PSpice A/D to enter designs that include Xilinx field programmable gate array devices.
MicroSim Filter Designer User’s Guide	MicroSim Filter Designer, which is a filter synthesis program that lets you design electronic frequency selective filters.

The following table provides a brief description of those manuals available online *only*.

This online manual...	Provides this...
MicroSim PSpice A/D Online Reference Manual	Reference material for PSpice A/D. Also included: detailed descriptions of the simulation controls and analysis specifications, start-up option definitions, and a list of device types in the analog and digital model libraries. User interface commands are provided to instruct you on each of the screen commands.
MicroSim Application Notes Online Manual	A variety of articles that show you how a particular task can be accomplished using MicroSim's products, and examples that demonstrate a new or different approach to solving an engineering problem.
Online Library List	A complete list of the analog and digital parts in the model and symbol libraries.
MicroSim PCBoards Online Reference Manual	Reference information for MicroSim PCBoards, such as: file name extensions, padstack naming conventions and standards, footprint naming conventions, the netlist file format, the layout file format, and library expansion and compression utilities.
MicroSim PCBoards Autorouter Online User's Guide	Information on the integrated interface to Cooper & Chyan Technology's (CCT) SPECCTRA autorouter in MicroSim PCBoards.

## Online Help

Selecting Search for Help On from the Help menu brings up an extensive online help system.

The online help includes:

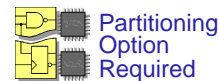
- step-by-step instructions on how to use PLSyn features
- DSL language reference
- PIL language reference
- device lists (by manufacturer and by template)
- Technical Support information

Every dialog box also includes a help button which, when selected, displays a description of the dialog box and each control.

# The PLSyn Features In Your Configuration

PLSyn, running with other MicroSim programs, provides the following features:

- Multiple design entry modes.
- Schematic entry with support for hierarchical design.
- Design Synthesis Language (DSL) support of arithmetic operators and arrays, procedure and function library linking.
- Device-independent design entry.
- Integrated simulation at the system level to detect problem areas in the design; you can simulate the functionality of your design while it is still in the design phase.
- Compilation, optimization, and device selection.
- Logic consolidation; optimization and reduction of your design to the smallest set of gates using industry-standard methods.
- Multiple equation reduction levels; automatic DeMorganization, automatic flip-flop synthesis, XOR synthesis, *don't care* generation, and node collapsing.
- Automatic or manual placement of input and output signals in the selected programmable logic devices.
- Automatic partitioning of the design across as many as 20 devices.
- Libraries with up to 3,500 PLDs from twelve manufacturers and 100+ architectures.
- Ability to test programmable devices by automatically generating test vectors from the functional simulation results and downloading them to the programmer with the fuse map file.
- On-line reference to the complete list of devices supported by PLSyn.



Your configuration depends on which of the design modules you purchased: PLDs, AMD MACH, and/or Atmel V-Series.

---

# The Programmable Logic Design Process—An Overview

---

# 1

## Chapter Overview

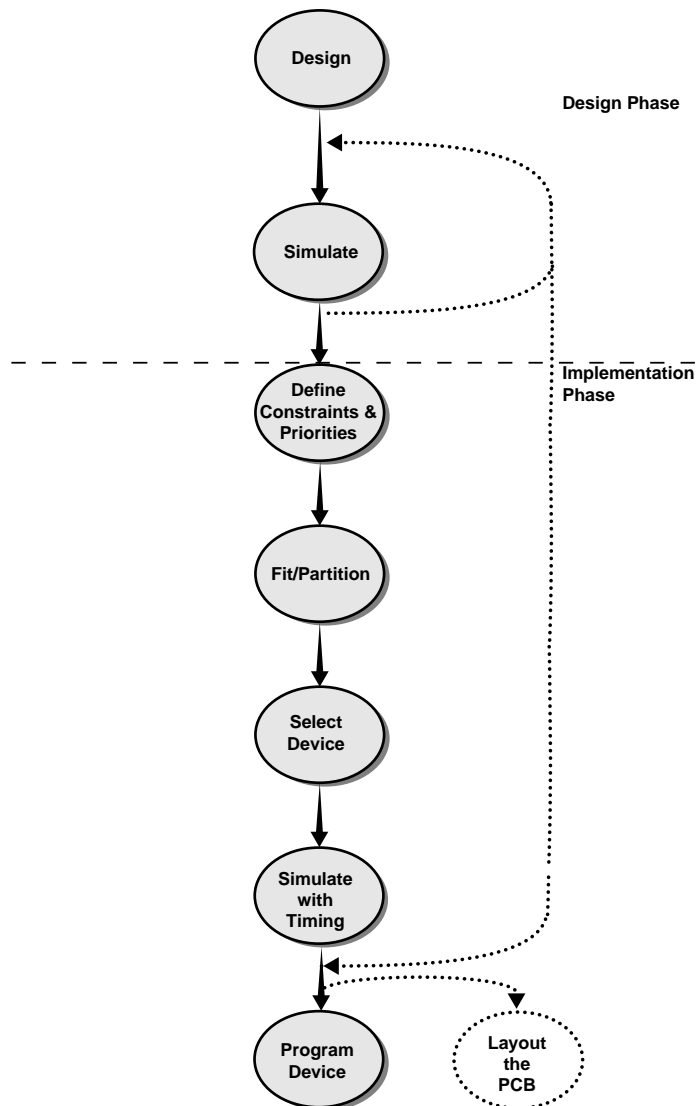
This chapter introduces the programmable logic design process, and terms and concepts used throughout this manual. Topics include:

- Design entry, *page* [1-3](#)
- Functional simulation, *page* [1-3](#)
- Constraints and priorities definition, *page* [1-4](#)
- Fitting and partitioning process, *page* [1-4](#)
- Device selection, *page* [1-5](#)
- Timing simulation, *page* [1-5](#)
- Device programming, *page* [1-5](#)

# Steps for Designing Systems with Programmable Logic

Because the design phase is separate from the implementation phase, you can design and simulate your system *before* choosing which PLD part(s) you want to use.

Figure 1-1 illustrates the typical design flow for synthesizing programmable logic.



**Figure 1-1** PLD Synthesis Design Flow

## Design

You can program all or any part of your design into PLD parts. To start, this means you need to define the functionality targeted for PLDs as *programmable logic* in your schematic.

Programmable logic takes the form of either:

- programmable logic *symbols*, such as gates, flip-flops, shift registers, and counters, or
- Design Synthesis Language (DSL) blocks, which describe programmable logic in a hardware description language

Your schematic can also include logic that is not targeted for PLDs. This is called *non-programmable logic* which takes the usual form of discrete parts.

Your schematic can contain any combination of programmable logic symbols, DSL blocks, non-programmable logic, and even analog parts.



### Schematics

Example: Your design might be a large system which contains discrete PCB-level parts and one or more PLDs. Or, it might be a design of a reusable system which you want to implement entirely in a PLD.

## Simulate

You can simulate your design before you know which PLD architectures (part types) you want to use. Before running the simulation, PLSyn automatically *compiles* all of your programmable logic into logic equations which are then used by the simulator.

Because simulations at this stage are before implementation, they do not include timing information. However, functional simulations can save a lot of time early in the design process, because the more time-consuming steps of optimization and fitting are not required until your design is finished.



### PSpice A/D



### PLogic



### Probe





See *The PLSyn Features In Your Configuration* on page xxiii for more information.

Example: You can place more importance on lower power than total price

## Set Constraints and Priorities

By default, the PLSyn fitter considers every device in the library. The number of devices you have available depends on the design module options you have purchased.

Before you begin the fitting/partitioning process, you can *constrain* the parts that the PLSyn fitter considers by device properties such as architecture, logic family, package type, speed, etc. This helps narrow the architecture-set from which PLSyn can choose, which results in faster completion of the fitting/partitioning process.

You can also have PLSyn rank the solutions by defining the relative merit of device properties like price, number of pins, size, propagation delay, and frequency, before running the fitter. These are your solution *priorities*.



**Note** You must have the *partitioning feature* to fit a design into multiple devices.

## Fit and Partition

After you have completed the functional design and set the fitting constraints and priorities, you are ready to *fit* your programmable logic into PLD parts. Fitting is the process of mapping a logic design into physical devices.

PLSyn finds and displays a list of up to ten *solutions* which implement your design's programmable logic while abiding to your constraints. PLSyn lists the best solutions, ranked by your assigned solution *priorities*.

For each solution it finds, PLSyn displays the generic *architecture*, or *template*, for the device, along with its cost, speed, and power consumption.



If your PLSyn package includes the partitioning feature, PLSyn automatically allocates or *partitions* logic into two or more devices (up to a maximum of twenty). PLSyn can also partition logic between devices with different architectures. If so, PLSyn shows each architecture in the solution list.

PLSyn's fitting and partitioning process works automatically. You can also direct the process by using the *physical information* file which contains statements in the Physical

Information Language (PIL). Using PIL, you can specify exact part numbers, put groups of logic into specific devices, and specify device pinouts.

## Select Device

After the PLSyn fitter has found the solutions that implement your design, the next step is to choose one of the architectures and the corresponding physical part(s) you want to use.

When you select an architecture in the solution list, PLSyn displays a list of all part numbers meeting the constraints you have specified. These appear in the Solution Detail at the bottom of the PLSyn window. All you have to do is select which one(s) to use.



The screenshot shows the PLSyn (Decoder.sch) window. It has a menu bar with File, Edit, Layout, Synthesize, Help, and a toolbar with icons for File, Edit, Layout, Synthesize, and Help. The main area is divided into two sections: 'Solutions' and 'Solution Detail'.

**Solutions**

	Part	115ms	7ns	\$7.90
1.	P16VBA			
2.	P22V10	190ms	5ns	\$12.74

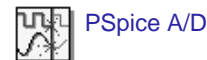
**Solution Detail**

Device Name	Mfg	Fam.	Pkg	Temp	Loc	Trin	Price	U1	U2
GAL16VDCSLP	LAT	CMOS	DIP	COM	115ms	7ns	\$7.90	0	0

At the bottom right of the Solution Detail table is a 'Browse...' button.

## Simulate with Timing

Any simulations that you perform after you have selected actual PLD part numbers, include timing information specific to those parts. This allows you to check the device's timing within your system and find potential problems such as setup/hold time violations or worst-case timing hazards which involve the PLD device.



## Program Device

As the final step, you need to generate the fuse maps that your device programmer needs to program the PLDs. PLSyn generates these as JEDEC files, one for each PLD device in your implementation.



---

# Primer: How to Define Programmable Logic

---

## 2

## Chapter Overview

This chapter guides you through the steps needed to synthesize a PLD device for a simple 3-to-8 decoder.

[Implementing a 3-to-8 Decoder with Programmable Logic on page 2-2](#) describes the sample circuit.

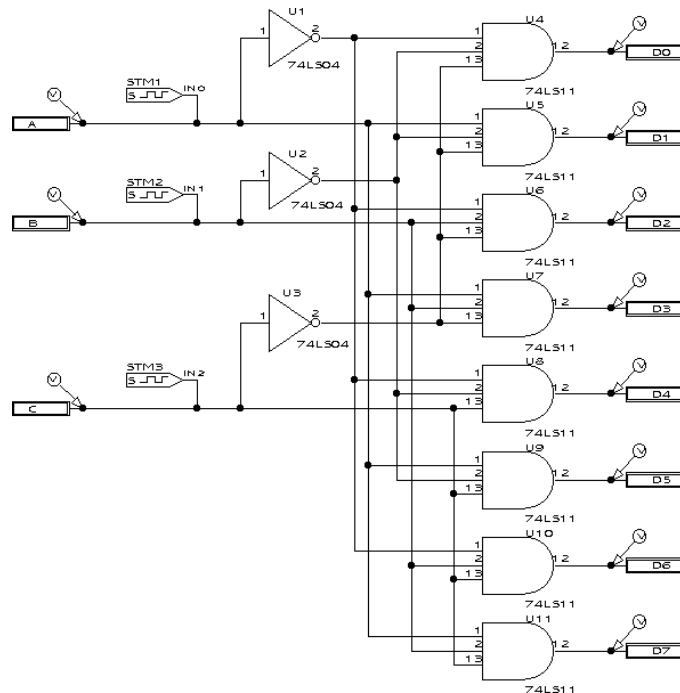
[Design Phase: Defining Programmable Logic using Schematic Symbols on page 2-3](#) walks you through the steps needed to convert existing schematic symbols to programmable logic.

[Implementation Phase: Fitting and Partitioning the Design on page 2-5](#) walks you through the steps needed to fit, select and program a PLD device subject to the constraints and priorities you define.

[Using a DSL Block to Define the Programmable Logic on page 2-10](#) presents a way of defining the programmable logic that is equivalent to that using schematic symbols.

# Implementing a 3-to-8 Decoder with Programmable Logic

Figure 2-1 illustrates a simple 3-to-8 decoder, consisting of three 74LS04 inverters and eight 74LS11 3-input AND gates.



**Figure 2-1** 3-to-8 Decoder Schematic

Assume that you want to target all of the decoder for PLD implementation. You have two alternative but equivalent methods from which to choose:

**Note** You can mix both programmable logic symbols and DSL blocks on your schematic.

- Convert discrete components to programmable logic using schematic symbols.
- Create Design Synthesis Language (DSL) blocks to define functionality using a hardware description language.

In the remainder of this chapter, you will see how to use both of these methods. You will also learn how to set up and run the physical implementation process, which is the same regardless of how you specify the programmable logic.

# Design Phase: Defining Programmable Logic using Schematic Symbols

## Before you begin

Copy the following files from the *\MicroSim root directory\examples\plsyn\decoder* directory to your working directory:

<code>decoder.sch</code>	schematic file
<code>decoder.stl</code>	stimulus library file

## Loading and simulating the design

### To load the schematic

- 1 In the MicroSim program group, double-click the Schematics icon to start Schematics.
- 2 From the File menu, select Open .
- 3 Move to the directory containing `decoder.sch`.
- 4 In the File Name list box, select the schematic file that you are interested in.

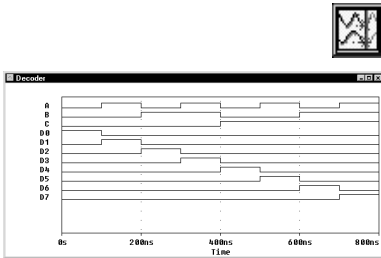


Once the circuit is loaded, you should run a simulation to ensure that the circuit is working properly before you fit it into a PLD. The schematic is already configured to perform an 800 nsec simulation.

### To simulate

- 1 From the Analysis menu, select Simulate.

Next you can view the results in Probe. This schematic has been set up to start Probe automatically and to display the signals which have markers attached. The resulting signals shown in Figure 2-2 indicate that the decoder is in fact working correctly.



**Figure 2-2** *Results of Decoder Simulation*


## Converting 74LS Symbols to Programmable Logic

There are two ways to enter programmable logic symbols, using either:

- pre-defined programmable logic symbols found in the `dig_prim.slb` symbol library, or
- 74xx symbols and then setting their `IMPL` attributes to `PLSYN`.

Since the decoder is already defined with discrete logic, the second method (defining the `IMPL` attribute) is the most convenient way to turn the existing design into a PLD design.

### To include devices in the programmable logic

- 1 Select all 74LS symbols on the schematic. Either:
  - draw a box around each symbol, or
  - +click on each 74LS part.
- 2 From the Edit menu, select Attributes.
- 3 Click Yes to the prompt: Globally edit attributes of all selected items?



- 4 In the Attribute Name text box, type `IMPL`; in the Value text box, type `PLSYN`.

This sets the value of the `IMPL` attribute to `PLSYN` for all selected parts that have an `IMPL` attribute (in this case, all parts).

- 5 Click OK.

Notice that the reference designator for each logic device changes to `PLSYN_U1`, `PLSYN_U2`, ..., and the color changes from green to blue, by default.

As an alternative, you can change each part individually (rather than globally) by double-clicking each 74LS device and setting the value of each `IMPL` attribute to `PLSYN`.

**Note** *If you changed the default color settings, your colors may differ from this example.*

## Verifying Functionality using Simulation

At this point, you can re-run the simulation to verify that the programmable logic representation matches the discrete device representation. The programmable logic is compiled for you automatically before the simulation starts.

Although you have just made the entire decoder design programmable logic, you can also specify only a portion of a design as programmable logic.

It's also easy to change a symbol back into a non-programmable logic symbol. Just edit the symbol's `IMPL` attribute and clear its value so that it is blank.

# Implementation Phase: Fitting and Partitioning the Design

You are now ready to create the physical implementation. To do this, you must run `PLSyn`.

### To activate `PLSyn`

- 1 From the Tools menu, select Run `PLSyn`.

`PLSyn` starts with the current design file loaded.

## Setting Constraints

Constraints allow you to choose the types of devices into which PLSyn must fit the design. You can narrow the search for solutions by selecting criteria such as device template (architecture), logic family, manufacturer, package type, power, speed, and temperature.

By default, PLSyn considers all devices. Suppose you want to narrow the solution search by selecting specific *device templates*: P16V8A and P22V10.

### To constrain the solution to the P16V8A and P22V10 device templates

- 1 From the Edit menu, select Constraints.

PLSyn displays a list of constraints that you can enable. Some constraints, such as Device Template, also require that you select from a list of values.

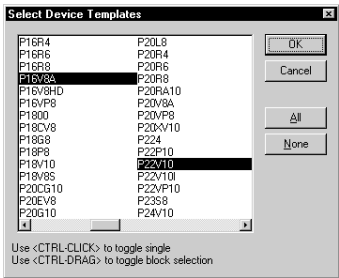
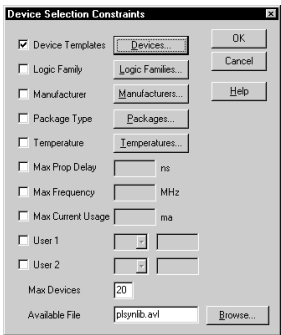
- 2 Click Devices.
- 3 Click None to deselect all items.
- 4 Scroll until P16V8A is visible and click on it.
- 5 Scroll and find P22V10.
- 6 Hold down the **[Ctrl]** key and click P22V10.
- 7 Click OK.

You can also constrain the device search by telling PLSyn to look only for one logic family of devices. By default, all three logic families—CMOS, ECL, and TTL—are included in the search.

Suppose that you don't want to use ECL.

### To exclude the ECL logic family from the solution

- 1 Click Logic Families.
- 2 Hold down the **[Ctrl]** key and click ECL. This leaves CMOS, OBS, and TTL highlighted.





- 3 Click OK to return to the constraints selection dialog box.
- 4 Click OK to exit constraints specification.

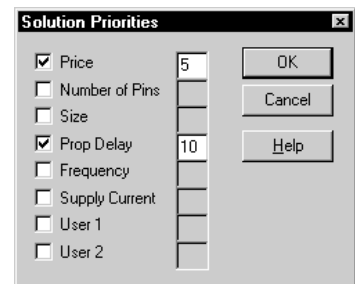
## Setting Priorities

Solution priorities allow you to determine the ranking of the solutions found during the fitting and partitioning process according to factors such as price, speed, power consumption, and pin count. They also determine the ordering of alternate devices for a given solution.

By default, price has the highest priority.

### To indicate a preference for faster parts

- 1 From the Edit menu, select Priorities.
- 2 In the Prop Delay text box, type 10.
- 3 In the Price text box, type 5.
- 4 Click OK.



## Partitioning and Fitting

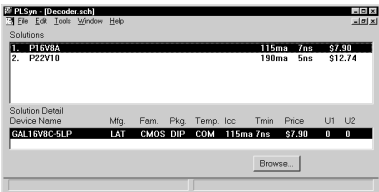
You are now ready to start the fitting and partitioning process. During the fitting process, PLSyn finds and displays a list of up to 10 solutions which implement the programmable logic according to your constraints. PLSyn lists the best solutions, ranked according to solution priorities that you just assigned.

### To begin the fitting process

- 1 From the Tools menu, select Fitter/Partitioner.

PLSyn first checks the netlist to make sure that the design has not changed. Then PLSyn automatically compiles the design (if not already compiled), optimizes the design, and starts the fitting process.

PLSyn scans the available file to find devices which match your constraints. PLSyn then searches for the devices which actually fit your design’s programmable logic. When this process is complete, PLSyn displays the solutions in the solution list at the top of the PLSyn window.

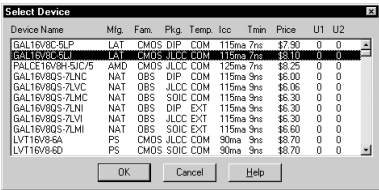


This design fits into either of the two templates which you selected earlier: P16V8A and P22V10. P16V8A is listed first because it is the *best* device meeting the specified priorities. Further, the *best* P16V8A is a GAL16V8C-5LP, shown in the solution detail list.

To select a different part number

For example, suppose you want to use a leadless chip carrier,

- 1 Click Browse to view the list of alternate parts.
- 2 Select the PALCE16V8H-5JC/5.
- 3 Click OK to keep the selection.



The PALCE16V8H-5JC/5 is now the physical device which implements the decoder (although a rather expensive implementation!).

Verifying Timing Behavior using Simulation

If you now simulate the design, the simulator includes the timing specifications for the PALCE16V8H-5JC/5. This allows you to check the timing behavior for both:

- the device itself, and
- the device operating within your entire system.

# Creating Device Programming Files

You are now ready to run the Fuse Map Generator to create a device programming file in JEDEC format.

The JEDEC file is the input to your device programmer.

### To generate fuse maps

- 1 From the Tools menu, select Fuse Map Generator.  
PLSyn displays a warning message that no test vectors will be included in the fuse map file at this time. For now, this is fine.
- 2 Click Yes when prompted to continue.

This creates a file named `decoder.j1`.

Alternatively, you could go back to the schematic and set the switch to generate test vectors in the PLSyn Setup dialog box (see page 4-3), then re-simulate to include the test vector in the JEDEC file.

### To view the JEDEC file name and other useful information

- 1 Select Examine Doc File in the File menu.

## Back Annotating the Schematic

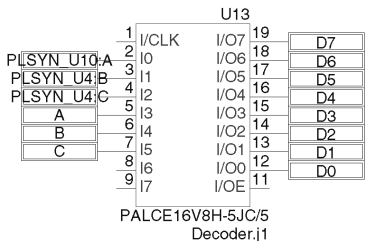
You can now back annotate the schematic to include the physical device(s) that you selected.

### To back-annotate the schematic

- 1 In PLSyn, from the Tools menu, select Update Schematic.  
Schematics places the selected PLD(s) on a new schematic page along with the appropriate input/output ports.

### To view the PLD part as shown in Figure 2-3

- 1 In Schematics, from the Navigate menu, select Next Page.
- 2 Click YES to the prompt: Save changes to current page?



**Figure 2-3** Back-Annotated Schematic Page

# Using a DSL Block to Define the Programmable Logic

Physical implementation is the same no matter how you set up the programmable logic in your schematic. If, after having defined the DSL block, you want to implement the design, follow the instructions in [Implementing a 3-to-8 Decoder with Programmable Logic on page 2-2](#).

The following steps describe how to implement the 3-to-8 decoder with a DSL procedure which is equivalent to the programmable logic symbols you used in the previous example.

## Before You Begin

Copy the following files from the `\MicroSim root directory\examples\plsyn\decoder1` directory to your working directory:

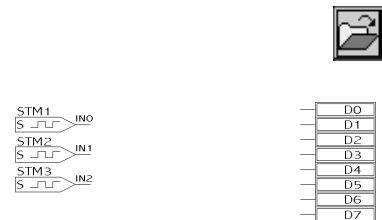
<code>decoder1.sch</code>	schematic file
<code>decoder1.stl</code>	stimulus library file

## Loading the Design

The schematic file, `decoder1.sch`, contains only digital stimulus and global output ports. The analysis setup is also pre-configured to perform an 800 nsec simulation.

### To load the schematic

- 1 From the File menu, select Open.
- 2 Move to the directory containing `decoder1.sch`.
- 3 In the File Name list, select the schematic file that you are interested in.



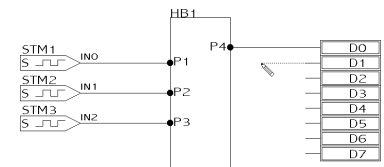
**Figure 2-4**    *The decoder1.sch Example Schematic*

## Adding a DSL Block

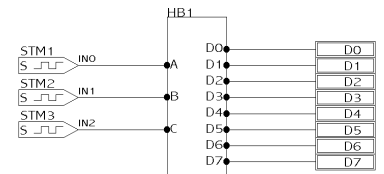
DSL blocks are simply hierarchical blocks which reference DSL source code files instead of schematic files.

### To add a DSL block

- 1 From the Draw menu, select Block.
- 2 Place one block on the schematic page between the inputs and the outputs, as shown in Figure 2-3.
- 3 From the Draw menu, select Wire and connect each stimulus input directly to the block.
- 4 Repeat step 3 for each global output port, as shown in Figure 2-3. Each connection to the block creates a pin.
  - a Rename the DSL block's pins as shown in Figure 2-3: double-click the pin name (for example, P1) to bring up the Change Pin dialog box.
  - b Enter a new pin name.
  - c Click OK.
  - d Repeat steps a-c for each of the input and output pins.



**Figure 2-5** Connecting the DSL Block



**Figure 2-6** Finished DSL Block

## Defining DSL Source Code

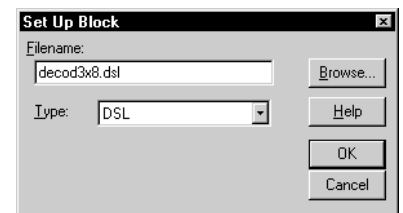
You are now ready to enter the DSL source code for the block.

### To define DSL source code

- 1 Double-click the block to *push* into it.
- 2 Enter the DSL source code file, `decod3x8.dsl`, in the Setup Block dialog box, then click OK.

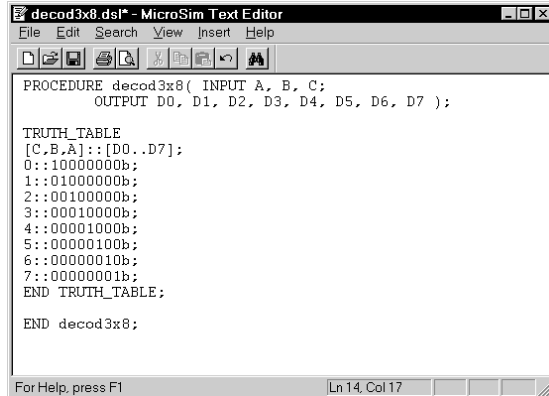
Schematics displays the MicroSim Text Editor. Because you are defining a new block, the `PROCEDURE` header and `END` statements are defined for you as follows.

```
PROCEDURE decod3x8( INPUT A, B, C;
    OUTPUT D0, D1, D2, D3, D4, D5, D6, D7 );
END decod3x8;
```



Notice that the INPUT and OUTPUT nodes in the procedure header correspond to the pin names of the DSL block.

- 3 Type the entire TRUTH\_TABLE statement *between* the PROCEDURE header and END statement as shown:



```
decod3x8.dsl - MicroSim Text Editor
File Edit Search View Insert Help
[Icons]
PROCEDURE decod3x8( INPUT A, B, C;
                   OUTPUT D0, D1, D2, D3, D4, D5, D6, D7 );

TRUTH_TABLE
[C,B,A]::[D0..D7];
0::10000000b;
1::01000000b;
2::00100000b;
3::00010000b;
4::00001000b;
5::00000100b;
6::00000010b;
7::00000001b;
END TRUTH_TABLE;

END decod3x8;

For Help, press F1                               Ln 14, Col 17
```

This simple construct sets a single bit in the D7..D0 output based on the three inputs' integer value.



- 4 From the File menu, select Save.
- 5 From the File menu, select Close to exit the MicroSim Text Editor.

**To verify that the DSL version of the decoder performs exactly as the logic symbol version**



- 1 From the Analysis menu, select Simulate.

## Equivalent Ways to Define the Decoder with DSL

Try experimenting with the different features of DSL. For example, you could also implement the decoder using the following CASE statement:

```
CASE [C,B,A]
WHEN 0 => [D7..D0] = 00000001b;
WHEN 1 => [D7..D0] = 00000010b;
WHEN 2 => [D7..D0] = 00000100b;
```

```
WHEN 3 => [D7..D0] = 00001000b;  
WHEN 4 => [D7..D0] = 00010000b;  
WHEN 5 => [D7..D0] = 00100000b;  
WHEN 6 => [D7..D0] = 01000000b;  
WHEN 7 => [D7..D0] = 10000000b;  
END CASE;
```

Or, you could use the following (somewhat crude, but still effective) set of equations:

```
D0 = /(A + B + C);  
D1 = A * /(B + C);  
D2 = B * /(A + C);  
D3 = A * B * /C;  
D4 = /(A + B) * C;  
D5 = A * /B * C;  
D6 = /A * B * C;  
D7 = A * B * C;
```

With a little experimentation, you should find that DSL is both easy-to-learn and powerful enough to describe complex blocks of logic.

---

# Designing with Programmable Logic

---

## 3

## Chapter Overview

This chapter describes in detail how to specify programmable logic using Schematics.

[The Different Ways to Specify Programmable Logic in Schematics on page 3-2](#) introduces the two equivalent mechanisms you can use to define programmable logic.

[Using Programmable Logic Symbols on page 3-2](#) describes where to find programmable logic symbols and how to convert discrete logic symbols to programmable logic.

[Using DSL Blocks on page 3-4](#) explains how to place and define functional blocks describing programmable logic using a hardware description language.

[Understanding Programmable Logic Nodes on page 3-13](#) explains how to define the internal and interface nodes connecting to programmable logic.

[Guidelines for Entering Programmable Logic on page 3-16](#), lists the do's and don'ts that you should follow to avoid problems during the physical implementation phase.

The discussion in this chapter assumes that you are familiar with Schematics, including the use of hierarchical blocks. Refer to your *MicroSim Schematics User's Guide* for details on using Schematics.



# The Different Ways to Specify Programmable Logic in Schematics

You can define programmable logic in two ways using:

- logic symbols (such as gates and flip-flops)
- DSL (Design Synthesis Language) blocks

You can place programmable logic symbols and DSL blocks anywhere on your schematic—that means on any page and at any level of the hierarchy.

## Using Programmable Logic Symbols

**Note** *IMPL is short for “implementation.”*

Logic symbols used as programmable logic have their IMPL attribute set to the value PLSYN. The available logic symbols fall into two classes:

- Generic logic symbols  
Example: NAND4, JKFF
- 74xx series symbols  
Example: 74LS04 or 74HC107

## Generic Logic Symbols

For a complete list of symbols, refer to the *Programmable Logic Symbol Reference* in PLSyn online help.

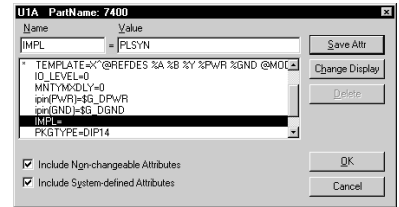
The `dig_prim.slb` symbol library contains ready-to-use programmable logic symbols, including gates, enabled gates, flip-flops, and latches. Each symbol already has its IMPL attribute set to PLSYN.

## 74xx Series Logic Symbols

You can also convert the common 74xx series logic symbols found in the 74xx.s1b symbol libraries to programmable logic.

### To convert one 74xx series logic symbol to programmable logic

- 1 Double-click the symbol.
- 2 Click the IMPL= entry.
- 3 In the Value text box, type PLSYN.
- 4 Click Save Attr.
- 5 Click OK.



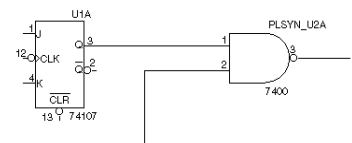
### To convert several 74xx series logic symbols to programmable logic all at once

- 1 Select the 74xx symbols.
- 2 From the Edit menu, select Attributes.
- 3 Click Yes to the prompt: Globally edit attributes of all selected items?
- 4 Click the IMPL= entry.
- 5 In the Value text box, type PLSYN.
- 6 Click Save Attr.
- 7 Click OK.



Schematics automatically updates the symbol's reference designator and changes its color to blue (by default), to show that it is programmable logic.

**Note** Some of the 74xx symbols cannot be converted to programmable logic. These symbols do not have the IMPL attribute. Adding an IMPL attribute will not work because PLSyn does not know the symbol's logic function.



**Figure 3-1** 7400 Symbol as Programmable Logic

You can also change programmable logic symbols back to discrete PCB devices.

#### To revert to *non-programmable* logic

- 1 Select the symbol(s) and bring up the Edit Attributes dialog box as described in the above two procedures.
- 2 Click the `IMPL=` entry.
- 3 Clear (set to blank) the Value text box.
- 4 Click Save Attr.
- 5 Click OK.

## Using DSL Blocks

This section describes how to define and edit DSL blocks within Schematics. For information on DSL language syntax, refer to the *PIL Reference* in PLSyn online help.

In addition to logic symbols, you can define programmable logic using DSL (Design Synthesis Language) blocks on your schematic.

### What Are DSL Blocks?

DSL blocks are hierarchical blocks which have a language-based definition instead of a symbolic definition. DSL logic expressions and constructs take the place of discrete logic symbols. So, instead of referencing a schematic file (`.sch`), DSL blocks reference a DSL source code file (`.dsl`).

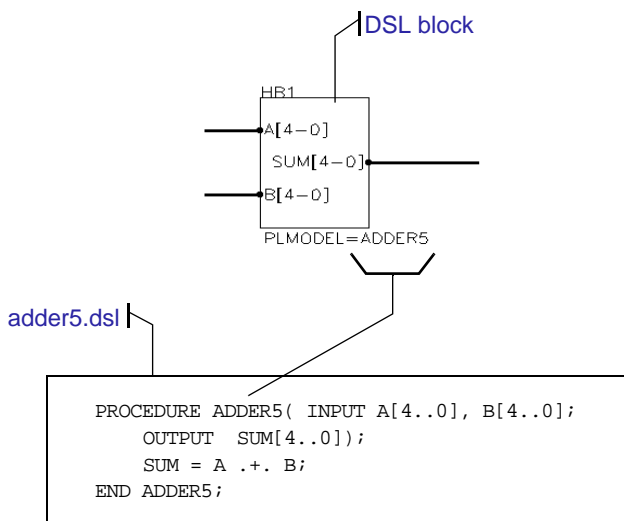
**Note** *DSL files must have the .dsl extension.*

## What Are DSL Procedures?

Each DSL block you place corresponds to a single *procedure* within the source code file. Procedures contain language constructs such as simple logic expressions, truth-tables, or state-machine definitions. The signals coming into the DSL block define the inputs to the procedure. Likewise, the outputs of the procedure define the output signals of the DSL block.

A DSL block has a PLMODEL attribute which defines the procedure name.

Example: The HB1 DSL block shown in Figure 3-2 references the adder5.dsl DSL source code file which contains the ADDER5 procedure referenced by the block's PLMODEL attribute.



**Figure 3-2** Relation of PLMODEL Attribute and DSL Procedure Name

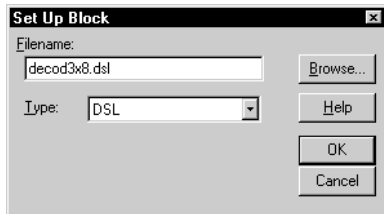
## Creating a DSL Block in Your Schematic



You can change the size of the block by selecting the block, then using right-click on one of the corners to drag it to the desired size.

**Note** Pin names must not be one of the DSL keywords, such as *INPUT* or *OUTPUT*. For the list of DSL keywords, refer to the *DSL Reference in PLSyn online help*.

The ERC attribute defines the electrical purpose of the pin.



**Note** The *.dsl* file cannot have the same name as the schematic file. (It is reserved for system use.) For example, a schematic named *decoder.sch* cannot reference a file named *decoder.dsl*.

### To create a DSL block

- 1 In Schematics, from the Draw menu, select Block.
- 2 Click to place the block on the schematic page.
- 3 Connect wires or buses directly to the block. Each connection automatically creates a pin at the junction.
- 4 Define the names and types of each pin.
  - a Double-click the pin name.
  - b Enter a new pin name.  
When naming a bus connection, use the Schematics bus label syntax, for example, A[4-0].
  - c If necessary, select the correct ERC value for the pin.  
By default, the pins on the left are given an ERC attribute of *input* and pins on the right are given an ERC of *output*. Do not set the ERC attribute to *DON'T CARE*; this is not allowed for a DSL block.
- 5 Push into the DSL block. Either:
  - double-click the block, or
  - from the Navigate menu, select Push.
 Because this is a new block, you are prompted for the name of the file containing the DSL source code.
- 6 Enter the name of the DSL source file (using the *.dsl* extension) that you want to create or reference.

- 7 If you have not yet created the DSL procedure for this block, then do one of the following:
- If the DSL file does not exist, Schematics activates the MicroSim Text Editor automatically. Specify the new DSL procedure and save the .dsl file.
  - If the DSL file does exist but you still need to specify the procedure, activate the MicroSim Text Editor from the MicroSim program group, open the .dsl file, specify the new DSL procedure, and save the file.

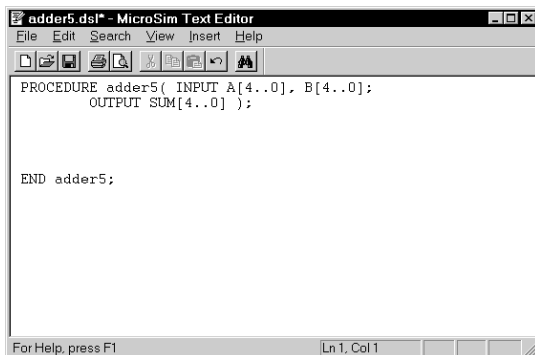
See the next section for information on defining DSL procedures.

## Using the MicroSim Text Editor to Define DSL Procedures

When given a file name with the .dsl extension, Schematics displays the MicroSim Text Editor which you can use to:

- Define the body of the DSL procedure.
- Add other procedures or functions.

For new DSL procedures, Schematics automatically creates a procedure template with input and output ports corresponding to the DSL block's pin names and attributes.



**Figure 3-3** *The DSL Procedure Template*

If the block's `PLMODEL` attribute is undefined, Schematics defines it for you using the DSL file name (excluding the `.dsl` extension).

Schematics also automatically translates the bus label format to the DSL array format.

Example: In the procedure header shown in Figure 3-3, the bus format `A[4-0]` is translated to the array format `A[4..0]`.

## Changing the DSL Block Interface

The pins on the DSL block must match the number, name, and signal direction of the port nodes used in the DSL procedure. This means that if you add or delete pins, or change the width of a bus on your DSL block, you must update the procedure's port nodes corresponding to the changed pins.

Example: If you change a port's direction from an output to an input, you must change the ERC value of the corresponding DSL block pin to `INPUT`.

## To change the pin properties in Schematics

- 1 In Schematics, double-click the pin name in the DSL block.
- 2 Change values in the Pin Name text box or Pin Attributes frame as needed.

## To change the pin properties in the MicroSim Text Editor

- 1 In Schematics, double click the DSL block.
- 2 Modify the procedure header to match the new interface.

# Using Existing DSL Source Code

You can create a DSL file ahead of time and then associate it with any DSL block you create thereafter.

## To associate an existing DSL file with a new DSL block

- 1 Check the port node names in the DSL procedure you plan to use with the new DSL block.
- 2 Place a block.
- 3 Add a pin for each of the port nodes in the DSL procedure.
- 4 For each pin, change its name and ERC (if needed) to match the corresponding port node in the DSL procedure.
- 5 Add a PLMODEL attribute to the block and assign the DSL procedure's name as its value.
  - a Select the DSL block.
  - b From the Edit menu, select Attributes.
  - c In the Name text box, type PLMODEL. In the Value text box, type the DSL procedure name.

For detailed instructions (menu options and mouse moves), see the following procedures:

- [To create a DSL block on page 3-6.](#)
- [To change the pin properties in Schematics on page 3-9.](#)





- d** Click Save Attr.
- e** Click OK.
- 6** Push into the block, and when prompted, enter the name of the existing DSL source code file.

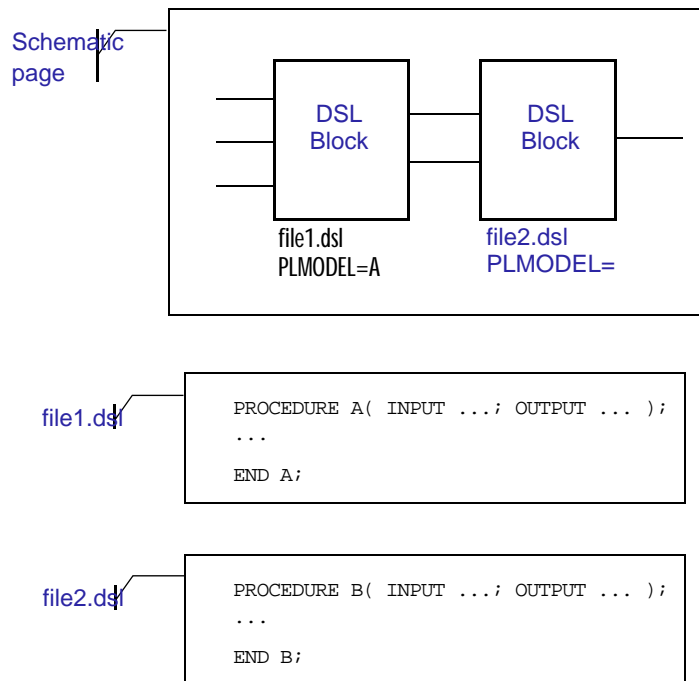
## Structuring DSL Source Files

When organizing your DSL procedures, you can have

- one procedure per file, or
- multiple procedures per file.

### Example: A single DSL procedure in each file

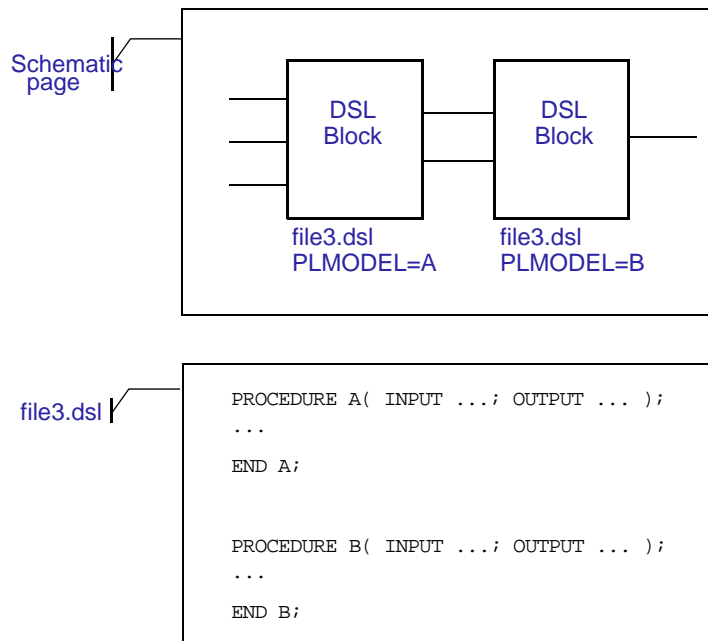
In Figure 3-4, if you were to make a change only to `file2.dsl`, `file1.dsl` is not recompiled.



**Figure 3-4** A Source Code File for Each DSL Block

## Example: More than one DSL procedure in a single file

From a maintenance point of view, this method is easier to manage because there are fewer files.



**Figure 3-5** *Single DSL Source Code File with More Than One Procedure*

## Calling DSL Procedures and Functions from within a Procedure

Like other programming languages, DSL allows a procedure to contain calls to other procedures and functions.

You must define called procedures and functions *before* they are called from the main DSL procedure. There are several ways to do this:

- Add the called procedure or function directly to the source code *before* the calling procedure.
- Include another DSL source file into your source before the calling procedure by using the INCLUDE statement.
- Reference a pre-compiled DSL file (for example, a library of commonly used DSL procedures) from your source by using the USE statement before the calling procedure.

For information on the use of the INCLUDE and USE statements, refer to the *DSL Reference* in PLSyn online help.

To create a pre-compiled DSL file, manually compile the file from PLSyn using Compile Library from the Tools menu.

# Understanding Programmable Logic Nodes

As you enter a programmable logic design, the nodes which connect to programmable logic symbols or DSL blocks are of two types.

**Internal nodes** These connect programmable logic to other programmable logic.

**Interface nodes** These are at the boundary of the programmable logic, and connect to all other schematic symbols, such as global ports, non-programmable logic, and analog devices.

After you have performed the physical implementation of your design, interface nodes correspond to physical pins on a PLD.

## Labeling Nodes

You are not required to label the programmable logic nodes in Schematics. Schematics automatically generates a unique name, such as NPL\_0013.

However, to reference a node in your design's Physical Information (.pi) file, you should label the node so that you'll know how to refer to it. Once labeled, PLSyn carries the name throughout the physical implementation process by PLSyn.

For more information on the .pi file, see Chapter 6, *Controlling the Fitting Process Using the .pi File* and refer to the *PIL Reference* in PLSyn online help.

### To label any node

- 1 Double-click the wire.
- 2 Enter a name.

## Node naming restrictions

Programmable logic node names must adhere to the following naming conventions:

- The first character must be alphabetic (a–z, or A–Z).
- Remaining characters can be any combination of alphabetic (a–z, A–Z), numeric (0–9), and underscore (\_) characters.
- Names cannot be any of the DSL keywords.

For a listing of DSL keywords, refer to the *DSL Reference* in PLSyn online help.

Node names are case-*insensitive* which means upper-case and lower-case letters are treated alike.

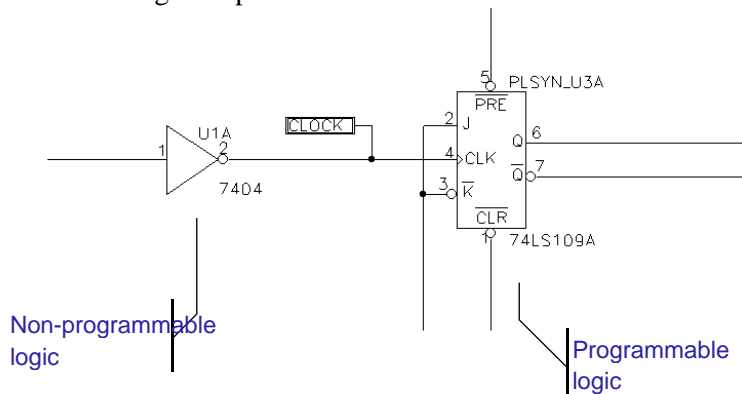
## Labeling interface nodes

For interface nodes, you can insure that the node label will persist with the PLD implementation.

## To force the label to appear in the back-annotated PLD symbol

For more information, see *Back Annotating the Schematic* on page 2-9.

- 1 Attach a global port to the interface node, as shown in Figure 3-6.
- 2 Label the global port.



**Figure 3-6** Programmable Logic Interface Node Labeled with a Global Port

## Creating Active-Low Interface Nodes

### To create active-low inputs or outputs to your programmable logic

Place a LOW\_TRUE port (instead of a global port).

**Note** *The LOW\_TRUE port creates interface nodes in the same manner as the global port. Therefore, you cannot use the LOW\_TRUE port to create active-low internal nodes.*

The LOW\_TRUE port symbol is contained in the `dig_prim.slb` symbol library.

## Converting Internal Nodes to Interface Nodes

When PLSyn runs an optimization, internal programmable logic nodes are automatic candidates for removal known as node collapsing. To avoid this, you can change an internal node to be an interface node, and have the node appear at a physical PLD pin.

You could use this method to make an internal node available for testing. See Figure 3-6 on page 3-14 for an example.

### To convert an internal node to an interface node

- 1 Attach a global port.
- 2 Assign a label.

## Creating Physical Nodes

### To create a physical node at the schematic level

Place the PHYNODE/PL symbol and connect it to a wire.

For more information on physical nodes, refer to the *DSL Reference* in PLSyn online help.

## Assigning a Logic 0 or 1 to an Input

The LO and HI symbols are contained in the `port.slb` symbol library.

You can assign constant 0 or 1 to a programmable logic symbol by using the LO and HI symbols in one of the ways described below.

**Alone** If you attach a LO or HI symbol directly to an input pin of a programmable logic symbol (or to an unlabeled wire connected to an input pin), PLSyn treats that input as a logic constant 0 or 1.

Example: Use the HI symbol to tie an unused input on an AND gate high, or to tie the J and K inputs of a flip-flop high to create a T flip-flop.

**Attached to an interface node** If a LO or HI symbol is attached to an interface node, the LO or HI behaves like a stimulus during simulation. PLSyn still creates a physical device pin.

## Guidelines for Entering Programmable Logic



### Do this

- Always begin the names of the following objects with an alphabetic character (a–z or A–Z):
  - Schematic (.sch) and DSL source (.dsl) file names
  - Programmable logic interface and internal nodes
  - DSL block pins

The remainder of the name can contain numbers (0–9) or the underscore (\_).

**Note** *Do not use any other punctuation characters in the name.*

- Make sure that each independent collection of programmable logic has at least one input interface and one output interface node. That is, at least one input and one output signal must connect either to a global port or to non-programmable logic.

### Don't do this

- Label any programmable logic node (interface or internal) the same as any of the DSL keywords. For example, you can use OUT, but not OUTPUT.
- Tie output interface nodes together. That is, the same node may not be driven by two or more programmable logic output pins.
- Connect the analog ground node (node 0) to any programmable logic interface. Use the digital constant sources LO and HI instead.
- Make a port label an integer.
- Use punctuation marks (except for underscore characters) in names. See *Do this* above for naming conventions.
- Name the DSL file or procedure the same name as any of the programmable logic symbols contained in digprim.slb.



For a listing of DSL keywords, refer to the *DSL Reference* in PLSyn online help.



---

# Simulating Programmable Logic Designs

---

# 4

## Chapter Overview

This chapter describes how to simulate your programmable logic design both before and after PLD implementation. Topics include:

[Introduction to Simulating with PLogic or PSpice A/D on page 4-2](#)

[Setting Up Simulations on page 4-3](#)

[Starting Simulations on page 4-4](#)

[How the Simulator Uses Programmable Logic I/O Models on page 4-5](#)

[Simulating with Timing on page 4-6](#)

[Generating Test Vectors on page 4-6](#)

[Using Probe Markers on page 4-10](#)

For more information on PSpice A/D, refer to your *MicroSim PSpice A/D User's Guide*.

# Introduction to Simulating with PLogic or PSpice A/D

Once you have entered a design which includes programmable logic, you can simulate both before and after you have chosen a physical implementation. The purpose of the simulation depends on the development stage of your design.

**Verify function before implementation** At this stage, simulations do not include timing. Instead, this is a good time to verify that your design is behaving as you expect it to operate.

**Verify timing after implementation** At this stage, after having selected the PLD devices, simulations automatically include timing information for the devices such as propagation delays and setup times. You can verify not only the timing of each PLD, but also the timing of the entire circuit including the PLD(s).

# Setting Up Simulations

Simulation setup for circuits containing programmable logic is similar to that for any other circuit. The way you navigate to the setup options depends on which simulator you have: PLogic or PSpice A/D.

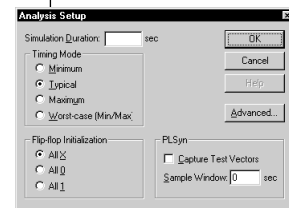
## Displaying the Dialog Box for Simulation Setup

### If you have PLogic

#### To display the Analysis Setup dialog box

- 1 In Schematics, from the Analysis menu, select Setup.

PLogic simulation setup

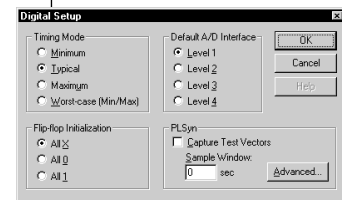


### If you have PSpice A/D

#### To display the Digital Setup dialog box

- 1 In Schematics, from the Analysis menu, select Setup.
- 2 Click Digital Setup.

PSpice A/D simulation setup



## Defining Simulation Setup Options for Programmable Logic

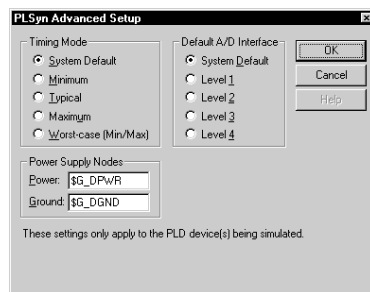
Refer to your *MicroSim PSpice A/D User's Guide* for detailed information on how to specify the delay, A/D interface level (PSpice A/D only), and flip-flop initialization for your design as a whole.

**Note** *The power and ground nodes and the A/D interface settings only apply to mixed-signal simulations with PSpice A/D. These options have no effect on digital-only simulations.*

Besides the usual simulation setup options, you can also specify simulation setup *specific to* the programmable logic part of your design. This includes options for delay, A/D interface level, and power supplies.

### To display the dialog box for programmable logic settings

- 1 From within the simulation setup dialog box, click Advanced.



In addition to the usual settings for the simulator, you can enable the capture of test vectors for the fuse map (JEDEC) file. See [Generating Test Vectors on page 4-6](#) for more information.

## Starting Simulations

There are two ways to start a simulation, either from Schematics, or from the simulator (PSpice A/D or PLogic).



Using Schematics, the netlist generates a netlist and compiles the programmable logic. Although you can run simulations on circuit files previously generated by Schematics, they might not reflect the current state of your design. To insure that what you are simulating is always in sync with your schematic design, we recommend that you always start your simulations from Schematics.

To start a simulation from within Schematics

Select Simulate in the Analysis menu.

# How the Simulator Uses Programmable Logic I/O Models

I/O models define the digital and analog characteristics of digital input and output pins. As with all other digital devices, your programmable logic also uses I/O models. If a digital pin is connected to other digital devices, the simulator refers to the I/O model to obtain the pin’s output resistance, as well as its input or output capacitance.

If your package includes PSpice A/D, you can simulate analog devices along with your programmable logic. If a digital pin is connected to analog devices, PSpice A/D refers to the I/O model to obtain the name of an interface subcircuit (either AtoD or DtoA) to insert between the devices.

When you simulate programmable logic, the simulator attempts to use the I/O model appropriate for the technology. Table 4-1 lists the I/O models used by PSpice A/D and PLogic for programmable logic.

The simulator determines the correct technology if:

- You have constrained the physical implementation to one technology.
- The fitting process is complete and you have selected PLD part numbers.

If the simulator cannot determine the technology of the programmable logic (for example, you have selected two or more technologies in your device constraints), the simulator uses IO\_DEFAULT\_PLSYN, which has 74LS characteristics.

Table 4-1 PLSyn I/O Models

I/O Model Name*
PLSYN_IO_DEFAULT
PLSYN_IO_TTL
PLSYN_IO_CMOS
PLSYN_INT_IO_ECL
PLSYN_EXT_IO_ECL

\* These models are located in the dig\_io.lib symbol library.

**Note** *If your design is partitioned into two or more devices, the simulator automatically uses the appropriate I/O model at the logical boundaries of each device.*

## Simulating with Timing

After you have performed the physical implementation and selected PLD devices, any simulations that you run will include timing information for those devices. This timing information is obtained from PLSyn's device library. The simulator uses these timing values:

$t_{PD}$	combinatorial propagation delay
$t_{CO}$	clock-to-output propagation delay
$t_S$	setup time

For more information on how PSpice A/D treats unspecified propagation delays, refer to your *MicroSim PSpice A/D User's Guide*.

The device library contains maximum-rated values for  $t_{PD}$  and  $t_{CO}$ , and minimum values for  $t_S$ . The simulator calculates minimum and typical values from the maximum propagation delay values.

## Generating Test Vectors

In the PLSyn context, the term *test vectors* refers to the section of the JEDEC file used by device programmers to validate the device after it has been programmed. Each line of the JEDEC file's test vector section contains input signals (which stimulate the programmable logic) and the expected output signals. Device programmers apply the input signals and compare the results to the expected outputs specified in the JEDEC file.

## Enabling Test Vector Generation

If you enable test vector generation during the simulation, PLSyn collates and formats the input and output signals for each PLD in the solution into test vectors. PLSyn adds these vectors to the JEDEC file(s) when the fuse map is created *after* you have fitted and performed device selection.

### If you have PLogic

#### To enable test vector generation

- 1 In Schematics, from the Analysis menu, select Setup.
- 2 Select (✓) the Capture Test Vectors check box.



### If you have PSpice A/D

#### To enable test vector generation

- 1 In Schematics, from the Analysis menu, select Setup.
- 2 Click Digital Setup.
- 3 Select (✓) the Capture Test Vectors check box.



## How the Simulator Responds

With test vector generation enabled, the programmable logic portion of the design runs in unit delay mode. This avoids test vector mismatches during device programming.

**Note** *Any time you re-fit or select a different solution, you must re-simulate in order to generate the test vectors for the new device(s).*

Unit delay mode effectively turns off the simulator's inertial delay behavior which causes short pulses to be swallowed. Because device programmers do not support inertial behavior, this helps avoid test vector mismatches.

# Using the “Sample Window” Control

See page 4-3 for information on how to get to the setup dialog box for your simulator.



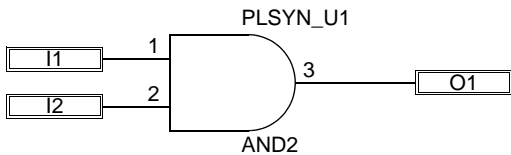
The Sample Window value (specified in the simulation setup dialog box) defines the interval during which the simulator considers input changes to occur at the same time.

Set this value when signals, considered part of the same input vector, arrive at the boundary of the programmable logic at slightly different times. This is useful, for example, in mixed analog/digital designs.

## Example: How the Simulator Creates Test Vectors

### Case 1

Consider the following case,

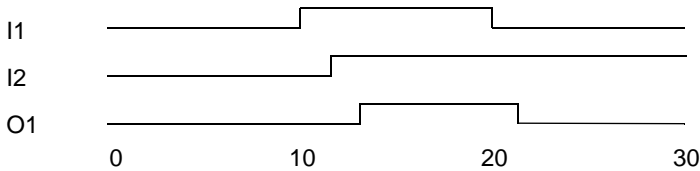


with inputs and outputs as follows.

**Table 4-2** Test Vectors for Case 1

Time Sample Taken	I1	I2	O1
10	0	0	L
11	1	0	L
20	1	1	H
30	0	1	L

**Note** In JEDEC files, 0, and 1 are input values; L and H are output values.



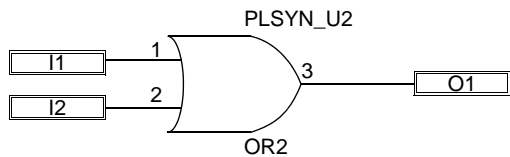
With Sample Window set to zero, the simulator creates test vectors by recording the value of all inputs and outputs whenever any input changes. The vector consists of all prior input values, along with the current output value. In other words, the simulator assumes that any input change propagates to the output by the time the next input change occurs. Table 4-2 shows the test vectors that the simulator creates using this logic.



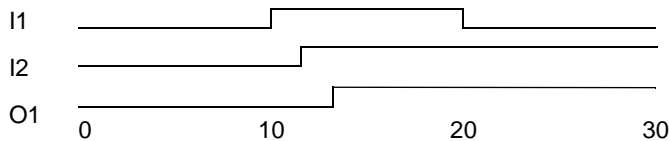
The simulator takes one final sample at the end of the simulation. As you can see, these test vectors are correct, even though the input changes do not arrive simultaneously.

Case 2

Unfortunately, this approach produces the wrong results in the following case,



with inputs and outputs shown below.



With Sample Window set to zero, the simulator produces the vectors shown in [Table 4-2](#). At time 11, the output value of L is incorrect. The result of I1 changing to 1 had not yet propagated to the output.

Corrected Case 2

The sampling window allows you to treat staggered inputs as if they had arrived simultaneously. A sampling window begins when any input changes and ends after the sample time expires. The inputs in the test vector consist of the input values at the end of the sampling window.

In case 2, a sampling window at least 1 unit in the duration, corrects the problem.

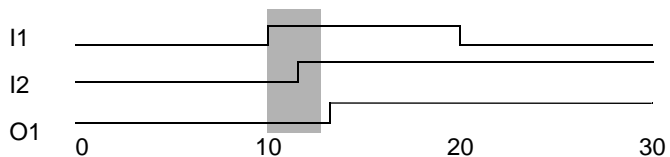


Table 4-3 Test Vectors for Case 2

Time Sample Taken	I1	I2	O1
10	0	0	L
11	1	0	L
20	1	1	H
30	0	1	H

Table 4-4 Test Vectors for Corrected Case 2

Time Sample Taken	I1	I2	O1
10	0	0	L
20	1	1	H
30	0	1	H

**Table 4-2** shows the test vectors.



### Troubleshooting Test Vector Differences

Sometimes, when the device programmer tests the device, the results produced by the part are different from the expected output results produced by the simulator. If this occurs, the following hints can help you solve the problem.

- Try specifying a non-zero sampling window as described in the previous section. Use a value greater than the propagation delay ( $t_{\min}$ ) of the device.
- Make sure that the initial value of the clock stimulus is inactive for the type of flip-flop you are using. Why? In the simulator, the flip-flop primitive requires the entire clock edge (for example,  $0 \rightarrow 1$ ) to register the data. However, in the programmer, the flip-flop registers its input if the first vector contains a value of 1 for the clock.

## Using Probe Markers

To view logic levels, you can place markers on both programmable logic interface nodes and nodes internal to the programmable logic in the schematic. If you've configured your system to automatically run Probe after simulation, waveform results immediately display in the Probe window.



Collapsed nodes happen when PLSyn removes an internal signal node by substituting the node's equation into any equation that references the node.

### A caution about collapsed nodes

When optimizing a design, PLSyn reduces the logic equations which can result in collapsed nodes. Collapsed nodes are not available in Probe, even if you have placed markers on them.

**Note** *Results at interface nodes are still available.*

---

# Creating the Physical Implementation

---

# 5

## Chapter Overview

This chapter describes how to create the physical implementation of your programmable logic using PLSyn.

[Overview of the Physical Implementation Process on page 5-3](#) reviews the steps you must follow to implement the programmable logic.

[Where to Find Status and Design Information on page 5-4](#) talks about the log and document files that PLSyn generates.

[Activating and Loading PLSyn on page 5-5](#) explains how to activate PLSyn.

[Compiling the Logic on page 5-7](#) explains how PLSyn converts the programmable logic symbols and DSL blocks to logic equations.

[Optimizing the Logic Equations on page 5-10](#) describes the kinds of algorithms PLSyn uses to reduce the logic equations.

[Overview of Fitting and Partitioning Logic on page 5-14](#) explains how to run the PLSyn fitter and how it works.

[Limiting the PLD Parts Available for Search on page 5-16](#) explains how to use the *available* file to specify a preferred device set.

[Constraining Devices on page 5-18](#) explains how to narrow the search by manufacturer, logic family, speed, and/or part type.

[Prioritizing the Solutions on page 5-23](#) explains how to use PLSyn to rank the solution set by speed, cost, power consumption, and pin count preferences.

[Running the PLSyn Fitter and Partitioner on page 5-25](#) explains how to start the PLSyn fitter.

[Selecting Devices on page 5-26](#) explains how you can select a different device from the solution list.

[Creating Fuse Maps on page 5-27](#) explains how to generate fuse maps to program the devices.

[Updating the Schematic on page 5-28](#) explains how to back-annotate the schematic with the selected PLD implementation.

[Creating PCB Netlists on page 5-29](#) provides tips when preparing to generate a netlist for board layout.

[When You Change the Design on page 5-30](#) provides tips when trying iterative what-if implementations.

# Overview of the Physical Implementation Process

After you have described your design in Schematics, you are ready to create the physical implementation of your programmable logic using PLSyn.

## To have PLSyn determine solutions for the physical implementation automatically

- 1** If needed, customize the available file (.avl) with user-defined properties that you want to constrain.
- 2** Define the selection constraints using Constraints in the Edit menu.
- 3** Define the solution priorities using Priorities in the Edit menu.
- 4** Run the PLSyn fitter using Fitter/Partitioner from the Tools menu. PLSyn automatically compiles and optimizes the programmable logic in your design.
- 5** Select the PLD device(s) you want to use.
- 6** Create the fuse maps using Fuse Map Generator in the Tools menu.
- 7** Back-annotate the schematic with the PLD device(s) using Update Schematic in the Tools menu.

See [Compiling the Logic on page 5-7](#) and [Optimizing the Logic Equations on page 5-10](#) for more information on what PLSyn does when compiling and optimizing your design.

For more information on the using the .pi file, refer to [Chapter 6, Controlling the Fitting Process Using the .pi File](#) and the *PIL Reference* in PLSyn online help.

For MACH devices, PLSyn also produces a report file. For more information, see [The MACH Report File on page 8-52](#).

For a detailed description of documentation file contents, see [Appendix A, The Documentation File](#).

## If You Want More Control

As described above, you can leave the synthesis details to PLSyn. But if you want more control, you can:

- Manually run the PLSyn compiler.
- Manually run the PLSyn optimizer.
- Direct the fitting/partitioning process by specifying controls in the physical implementation file (.pi).

## Where to Find Status and Design Information

To document your design, or, if your design fails to fit, PLSyn furnishes tools that can help you solve any problems:

**Message Viewer** The Message Viewer displays warnings and error messages that occur when PLSyn (or another MicroSim program) encounters a problem. You can access help text that relates directly to each message. For some messages, you can also jump to the point in your design where the problem was detected.

**Log file** The log file (*design\_name.log*) contains status and error messages from each of the implementation processes.

**Documentation file** The documentation file (*design\_name.doc*) contains detailed information about your design, such as the logic equations and device pinouts. PLSyn automatically creates this file after the optimization phase, and updates this file after you generate the fuse map file(s).

# Activating and Loading PLSyn

This section describes how to:

- Start PLSyn.
- Load a design.
- Interpret the PLSyn window.

## Activating PLSyn

Start the PLSyn program either from:

- Schematics, or
- the PLSyn program icon in Windows.

### From Schematics



#### To activate PLSyn from Schematics

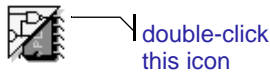
- 1 In the Schematic Editor, from the Tools menu, select Run PLSyn.

If your design is already open in Schematics, then you can start the physical implementation phase of your design once the PLSyn main window displays. If not, you must load a design directly into PLSyn as described in [Loading a Different Design on page 5-6](#).

### From the Windows Program Manager

In the Windows program manager, there is a program group that contains Windows icons for all installed MicroSim programs, including PLSyn.

### To activate PLSyn from the Windows Program Manager



- 1 In the MicroSim program group, double-click the PLSyn icon.

PLSyn activates without a design. See [Loading a Different Design](#) for further instructions.

### Loading a Different Design

Once you have activated PLSyn, you can change to a different design at any time.

#### To load an existing design



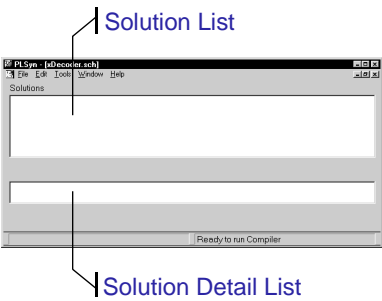
- 1 From the File menu, select Open.
- 2 Select a schematic (.sch) or DSL source (.dsl) file.

### The PLSyn Main Window

Once loaded, PLSyn's main window appears as shown in Figure 5-1.

The top area, called the *solution list*, displays architectures chosen by the PLSyn fitter. The bottom area, called the *solution detail list*, displays a list of alternative part numbers available for the architecture you selected in the solution list above.

During fitting, this window contains a list of the device templates PLSyn is considering.



**Figure 5-1** Main PLSyn Window



# Compiling the Logic

The PLSyn compiler converts all of your design's programmable logic (logic symbols and DSL blocks) into equivalent logic equations. PLSyn writes the compiled logic to an internal file named `design_name.afb` which the simulator and the PLSyn optimizer use later on.

You can compile programmable logic at different stages of design development:

- Automatically during the fit and partition process.
- Manually to verify the syntax of the programmable logic.
- Manually to compile DSL files that include the USE statement to reference other DSL files—also known as a library compile.

**Note** *You must manually compile DSL files that contain the USE statement before running a general compilation of your design, or before starting the fit and partition process. This is explained in more detail in the following two sections.*

For more information on fitting and partitioning, see [Overview of Fitting and Partitioning Logic on page 5-14](#) and the sections that follow.

## Manually Compiling Logic

### To manually compile all programmable logic in your design

- 1 Pre-compile any DSL files that include the USE statement (see [Compiling DSL Libraries on page 5-8](#)).
- 2 From the Tools menu, select Compiler.

To ensure that the schematic matches the PLSyn database, PLSyn first checks to see if any of the programmable logic, or its interfaces, has changed since the last generated netlist. If the programmable logic portion of your design has changed, PLSyn automatically regenerates the netlist.

Another way to automatically compile the programmable logic is as follows:

- 1 In Schematics, from the Analysis menu, select Create Netlist.

## Compiling DSL Libraries

Whenever your design includes DSL blocks that include the USE statement, you must load each DSL file and run a library compile before any other manual or automatic compilations take place.

### To compile DSL blocks that include the USE statement



- 1 From the File menu, select Open, and then select the name of the DSL file you want to compile.
- 2 From the Tools menu, select Compile Library.

## Responding to Compile-Time Status and Errors

During compilation, PLSyn displays a status window which shows the compiler's progress. You can abort the compilation at any time.

### To abort the compilation

- 1 In the status window, click Cancel.

If there are compile-time errors, PLSyn displays the messages in the Message Viewer. In addition, PLSyn keeps a written log.

### To control whether PLSyn writes compiler errors to the log file

Do the following before starting the compile:

- 1 From the Tools menu, select Options.
- 2 Select (✓) the Output Warnings check box.
- 3 Click OK.

After you have corrected any errors in your DSL blocks, you must restart the compiler.

## Controlling Node Generation During Compilation

You can control whether the PLSyn compiler creates internal nodes for carry bits for arithmetic and relational operators.

### To allow PLSyn to create internal nodes

Do the following before starting the compile:

- 1 From the Tools menu, select Options.
- 2 Select (✓) the Create Nodes check box.
- 3 Click OK.

## Resolving “Out of Memory” Conditions

If you encounter an “Out of Memory” message, you can reduce memory requirements by setting the maximum number of product terms any equation form can have.

### To limit the number of product terms

Do the following before starting the compile:

- 1 From the Tools menu, select Options.
- 2 In the Product Term text box, type an integer value for the maximum number of allowed product terms ranging from 64 to 5012. The default is 1024.
- 3 Click OK.

As a rule of thumb, lower the Product Term value by a factor of two. If you continue to get an “Out of Memory” message, lower the value again.



# Optimizing the Logic Equations

**Note** *Optimization is only needed prior to fitting and partitioning, not prior to running simulations.*

For more information on fitting and partitioning, see [Overview of Fitting and Partitioning Logic on page 5-14](#) and the sections that follow.

If you experience an “Out of Memory” message, try limiting the maximum number of product terms allowed in an equation form and restart the optimization. For more information, see [Resolving “Out of Memory” Conditions on page 5-9](#).

PLSyn performs optimization prior to fitting to compact your design’s programmable logic into as few equations and nodes as possible. This allows your design to fit into the fewest and smallest possible devices.

PLSyn writes the optimizer output to a file named `design_name.fdb` which the fitter and partitioner use later on.

Though you would usually have PLSyn run the optimizer automatically in the fit and partition process, you can also manually run the optimizer.

## To manually optimize all programmable logic in your design

- 1 Make sure your design is compiled (see [Compiling the Logic on page 5-7](#)).
- 2 From the Tools menu, select Optimizer.

During optimization, PLSyn displays a status window which shows the optimizer’s progress. You can abort the optimization at any time.

## To abort the optimization

- 1 In the status window, click Cancel.

## How the PLSyn Optimizer Synthesizes Logic Equations

In addition to compacting the logic, the optimizer also produces multiple, functionally-equivalent equation sets to accommodate the wide variety of device architectures available on the market. This means that for each potential device solution, the PLSyn fitter is able to select the set of equations that best uses the characteristics of that particular architecture.

The optimizer employs several techniques to synthesize the equations.

**DeMorganization** DeMorganization allows the PLSyn fitter to invert signals internal to a device while maintaining the signal polarity and functionality as described by the logic design. The ability to tailor equations internally to the device lets you create a functional design that is independent of the signal polarity capabilities of a particular device. It also gives maximum flexibility to the fitter so that PLSyn can place larger, more complex designs into fewer devices.

**Register synthesis** The optimizer synthesizes flip-flop types to optimize equation placement within a device. For example, you can describe logic in terms of J-K flip-flops. The optimizer also synthesizes the D equation so that the PLSyn fitter can place the equation in a device with D flip-flops.

**Don't care generation** You can express *Don't Care* conditions using the DSL If/Then/Else, Case, Truth Table, and State Machine statements. You can also assign *Don't Care* conditions to signals within procedures and functions. When output values are unspecified, the optimizer assumes a *Don't Care* condition. This allows the optimizer to assign either a zero or one value, depending upon which value generates the most optimal equation.

**Exclusive-OR (XOR) synthesis** Whenever possible, the optimizer maintains exclusive-OR representations of all

equations. The partitioner can then use the exclusive-OR representation in devices with that capability. In devices without exclusive-OR capability, the partitioner uses the sum-of-products representation.

**Node collapsing** The optimizer minimizes the use of intermediate nodes. The optimizer removes nodes by collapsing their equations into any equations that reference them.

**Logic minimization** There are three final reduction algorithms available: Espresso, Espresso/Exact, and Quine-McCluskey. You can set the method from the Tools/Options dialog. The Espresso algorithm is the fastest method and usually produces results as good as the other two algorithms. The Espresso/Exact and Quine-McCluskey methods are slower and use more dynamic memory but may result in smaller equations. Due to the speed and memory use issues, these optional reduction techniques should be restricted to designs with relatively small equations where optimal equation minimization is critical. The default reduction technique is Espresso.

## Choosing the Optimization Method

You can control the optimization algorithm PLSyn uses to reduce the logic equations, choosing from:

- Espresso
- Espresso/Exact
- Quine-McCluskey

Espresso is the default reduction technique.

### To change the optimization algorithm

- 1 From the Tools menu, select Options.
- 2 From the Optimization Method list, select the algorithm name.
- 3 Click OK.

The Espresso technique is fast and generally produces very good equations. The Espresso/Exact and Quine-McCluskey methods are slower and use more dynamic memory but may result in smaller equations. Due to the speed and memory use issues, you should restrict using these optional reduction techniques to designs with relatively small equations where optimal equation minimization is critical.

You can also control optimization using the `.pi` file. See Chapter 6, *Controlling the Fitting Process Using the .pi File* and the *PIL Reference* in PLSyn online help.



## Overview of Fitting and Partitioning Logic

During the fitting and partitioning process, PLSyn searches its library of parts for the PLD device architecture(s) which can implement, or *fit*, the programmable logic in your design.

There are two ways you can proceed with the fitting/partitioning process:

- Completely automatic, letting PLSyn determine how to best fit the logic.
- Using the `.pi` file to control how logic is fit; for example, into specific part numbers, with specific grouping of the logic into specific devices, and with specific pinouts for individual nodes.

For information on using the `.pi` file see Chapter 6, *Controlling the Fitting Process Using the .pi File* and refer to the *PIL Reference* in PLSyn online help.

The rest of this chapter describes how to use PLSyn to automatically fit and partition the programmable logic.

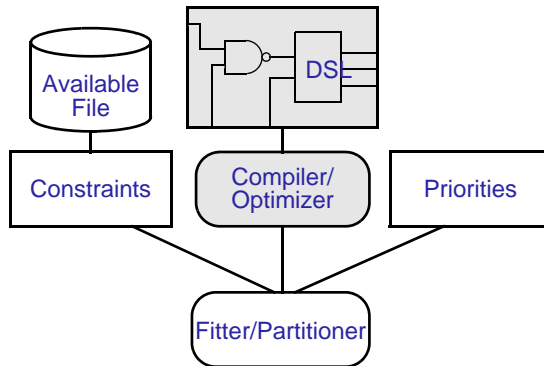
## If You Don't Have the Partitioning Option

Systems with the partitioning option allow PLSyn to fit your design into multiple PLD devices. If you do not have the partitioning feature, all of your programmable logic must fit into a single device. The discussions which mention more than one device in a solution do not apply to you.



## How the PLSyn Fitter Works

Figure 5-2 shows how the PLSyn fitter and partitioner relate to other PLSyn functions, data, and programmable logic. The shaded objects indicate functions that must occur before PLSyn can fit and partition the design: define, compile, and optimize the programmable logic.



**Figure 5-2** *PLSyn Functional Architecture*

If needed, PLSyn automatically compiles and optimizes the programmable logic before starting the fitting process.

There are two ways you can narrow the device search to the devices that interest you, and thereby speed up the fitting process:

- Edit the available parts file (.avl).
- Define constraints, such as device architecture, manufacturer, technology, speed, power consumption, and temperature.

PLSyn begins the search by scanning the list of available parts contained in the available parts file (.avl), then filtering the search further by the constraints you have defined. When the search is complete, PLSyn displays a list of up to ten solutions (architectures), ranked by the priorities which you defined earlier. For a given architecture, you can then select the exact part numbers that you want to use.

**Note** If any of the DSL blocks contain *USE* statements to refer to other DSL blocks, you must first manually compile these DSL files using the *Compile Libraries* option in the *Tools* menu. For more information, see [Compiling DSL Libraries on page 5-8](#).

Solutions, or architectures, are sometimes referred to as *templates*. For example, in the dialog box that PLSyn displays when you select *Constraints* in the *Edit* menu, you'll see a constraint named *Device Templates*.

Partitioning  
Option  
Required



If your system includes the partitioning option, and if needed, PLSyn automatically fits your programmable logic into more than one device, up to a maximum of twenty. PLSyn can also partition into different architectures. If so, PLSyn displays each architecture in the Solution Detail list.

## Limiting the PLD Parts Available for Search

The available file (`.avl`) contains the list of only those devices that you want PLSyn to consider as potential solutions. This file contains information on:

- available device types
- device manufacturer names
- logic families
- package types
- temperatures
- prices

The default available file, `plsynlib.avl`, resides in the `bin` subdirectory under your MicroSim root directory. When shipped, this file contains every part in the master device library.



Because limiting architectures, not devices, is what speeds up the fitting process, we recommend that you avoid editing the `.avl` file unless you plan to constrain the device search using user-defined properties. In this case, we recommend that you create a new `.avl` file using `plsynlib.avl` as a starting point.

## To create and use a custom available file

- 1 Copy `plsynlib.avl` to a different file name with the `.avl` extension.
- 2 Using any text editor, open the file, make modifications according to the restrictions explained after this procedure, and save it.
- 3 In PLSyn, from the Edit menu, select Constraints.
- 4 In the Available File text box, enter the file name.

Each line in the file is a complete record of a device. The only changes you should make to the available file are to:

- Delete the entire line containing a device to remove that device from consideration.
- Update the last three fields on a line, which are (listed in order of appearance):
  - part price (in cents),
  - a user-defined numeric property
  - a second user-defined numeric property

**Note** *Do not change any other fields or the format of the available file.*



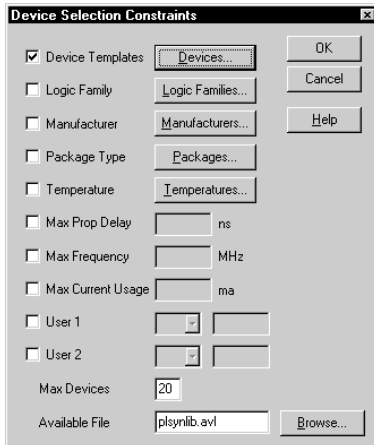
You can enforce constraint-checking on the user-defined fields by defining the User1 and User2 constraint controls. See [Constraining Devices on page 5-18](#) and [Setting Up User-Defined Constraints on page 5-20](#).

# Constraining Devices

Constraints allow you to narrow the list of devices that the PLSyn fitter considers when searching for solutions. The fitter compares your constraints against each part in the available file, and only those matching the specified constraints are considered for fitting.

**Note** *If your constraints are too narrow, the PLSyn fitter and partitioner may not be able to implement your design.*

## To edit the constraints for your current design



- 1 From the Edit menu, select Constraints.
- 2 Enable the constraints you want PLSyn to consider as follows:
  - For Device Templates, Logic Family, Manufacturer, Package Type, or Temperature, click the button to the right of the constraint and select the items you want considered. See [Table 5-1](#) for a description of each of these.
  - For all other constraints, type an appropriate value into the corresponding text box as described in [Table 5-1](#).
- 3 Clear any constraints you don't want PLSyn to consider (✓removed).
- 4 Click OK.

**Table 5-1** *Device Selection Constraints Dialog Box Controls*

Control Name	Meaning
Device Templates	List of the architectures that are available in your package. PLSyn considers only the selected architectures. For more information, refer to the the <i>Device Lists</i> in PLSyn online help.
Logic Family	List of available logic families. PLSyn considers only the selected logic families. For more information, refer to the the <i>Device Lists</i> in PLSyn online help.
Manufacturer	List of available manufacturers. PLSyn considers only the selected logic families. For more information, refer to the the <i>Device Lists</i> in PLSyn online help.
Package Type	List of footprints or package types available for partitioning. PLSyn considers only the selected footprints. For more information, refer to the the <i>Device Lists</i> in PLSyn online help.
Temperature	List of available temperature ratings. PLSyn considers only the selected temperature ratings. <a href="#">Table 5-2</a> lists the valid temperature rating abbreviations.
Max Prop Delay	Highest allowable value for propagation delay in nanoseconds. See <a href="#">How PLSyn Calculates Maximum Propagation Delay on page 5-22</a> for more information.
Max Frequency	Highest allowable frequency value in MHz. Default is 10 MHz.
Max Current Usage	Highest allowable value for power supply current in mAmps. Default is 10 mA.

**Table 5-2** *Temperature Rating Abbreviations*

Temperature Abbreviation	Meaning
883B	MIL-STD-883B
COM	0 to +75 °C
EXT	-40 to 85 °C
MIL	-55 to 125 °C

**Table 5-1**    *Device Selection Constraints Dialog Box Controls*  
(continued)



Control Name	Meaning
User 1	Comparison criteria used on the first user-defined property in each device statement in the available file. Defined as a pair of a values: <ul style="list-style-type: none"><li>relational operator (e.g., &lt;, &lt;=, =, etc.)</li><li>target number between 0 and 255</li></ul> See <a href="#">Setting Up User-Defined Constraints on page 5-20</a>
User 2	Comparison criteria used on the second user-defined property in each device statement in the available file. See User1 above and <a href="#">Setting Up User-Defined Constraints on page 5-20</a>
Max Devices	Highest allowable number of devices into which the partitioner can allocate programmable logic, ranging from 1 to 20.
Available File	The name of the available file. Default is <code>plsynlib.avl</code> .

## Setting Up User-Defined Constraints

Enabling user-defined constraints requires:

- Associating a property and value for each device listed in your available file.
- Defining the comparison that must be satisfied to include that device in the fitting/partitioning process.

## To set up user-defined constraints

- 1 In your `.avl` file, use a standard text editor to enter the value of a numeric property in each device line either after the price property (referred to as the User1 property) or after the first user-defined property (referred to as the User2 property).  
You can have *at most* two user-defined properties.
- 2 In PLSyn, when defining constraints (by selecting Constraints in the Edit menu), select (✓) the corresponding User1 or User2 check box.
- 3 Select a relational operator from the drop-down list to the right of the User1 or User2 constraint that you selected.
- 4 Type the target value for comparison in the text box to the right of the relational operator you just selected.

For more information on the available file format, see [Limiting the PLD Parts Available for Search on page 5-16](#).

## Example

A common application for the user-defined fields is device defect rate. If your production group has failure statistics on devices that range from 0 to 100, then you can enter those values into your available file.

Suppose that each device statement in your `.avl` file contains device defect rate values in the field after price—the User1 field. Then you can enforce device selections with a failure rate of less than 10% by:

- 1 In PLSyn, from the Edit menu, selecting Constraints.
- 2 Selecting (✓) the User1 check box.
- 3 Selecting `<` for the relational operator.
- 4 Typing 10 in the text box.

## How PLSyn Calculates Maximum Propagation Delay

**Combinatorial (non-registered) devices** The maximum propagation delay is the worst case  $t_{PD}$ , as published by the manufacturer.

**Registered devices** The maximum propagation delay is the sum of the  $t_S$  and  $t_{CO}$  (setup time and clock-to-output delay).

**Devices with both combinatorial and registered outputs** The maximum propagation delay is the larger of the two cases described above.

## The Default Constraints File

When fitting a new design, PLSyn initializes the constraints to values contained in the default constraints file, `default.cst`. This file resides in the `bin` subdirectory under your MicroSim root directory.

### To customize the set of default constraints

- 1 Choose a constraints file created for an existing design (residing in your working directory).
- 2 Make a copy of that file and save it to the `bin` subdirectory with the name `default.cst`.



# Prioritizing the Solutions

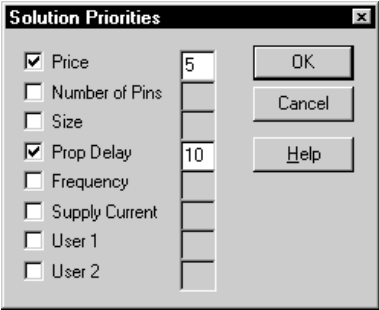
When PLSyn finds a solution which fits your programmable logic, it ranks the solution to determine whether it is *better* or *worse* than other solutions it has found. If the solution is within the ten best, PLSyn positions it in the solutions list according to its relative merit.

The ranking is based on priorities that you define.

## To define ranking priorities

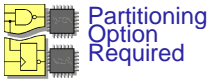
- 1 From the Edit menu, select Priorities.
- 2 Select (✓) the check boxes for the priorities you want to use to rank the solution. See Table 5-1 for a description of each one.
- 3 Enter weighting factors from 1 to 10 for the criteria that you enabled where 10 indicates most important.

**Note** *Disabled criteria are not considered in the ranking of solutions.*



**Table 5-3**    *Solution Priorities Dialog Box Controls*

Control Name	Meaning
Price	<p>Minimize the total price of the solution. Use a high priority to indicate a preference for lower-cost solutions. In multiple-template solutions, PLSyn considers the total price of all devices in the solution.</p> <p>If you have the partitioning option, then PLSyn can opt for cheaper, multiple-device solutions instead of more costly, single-device solutions.</p>
Number of Pins	<p>Minimize the total pin count. Use a high priority to indicate a preference for a lower pin count. In multiple-template solutions, PLSyn considers the pin count of all devices.</p>



**Table 5-3**    *Solution Priorities Dialog Box Controls*

Control Name	Meaning
Size	Minimize total size. Use a high priority to indicate a preference for physically smaller parts. In multiple-template solutions, PLSyn considers total size.
Prop Delay	Maximize speed. Use a high priority to indicate a preference for faster parts. In multiple-template solutions, PLSyn considers the device with the longest propagation delay.
Frequency	Maximize clock speeds. Use a high priority to indicate a preference for parts with higher maximum clock speeds. In multiple-template solutions, PLSyn considers the device with the lowest frequency rating.
Supply Current	Minimize power consumption. Use a high priority to indicate a preference for parts with lower power supply consumption. In multiple-template solutions, PLSyn uses the sum of the individual lcc values.
User 1	Use in conjunction with a USER1 constraint as follows: <ul style="list-style-type: none"><li>• If <math>USER1 &gt; 0</math> is the constraint, then PLSyn considers a solution to be better that has a USER1 value that is higher than another USER1 value. Example: 99 is better than 4.</li><li>• If <math>USER1 &lt; 1</math> is the constraint, then PLSyn considers a solution to be better that has a USER1 value that is lower than another USER1 value. Example: 4 is better than 99.</li></ul>
User 2	See User1

In multiple-device solutions, PLSyn uses all of the criteria where appropriate.

Example: The price given for a particular solution is the sum of the prices of all parts in the solution.

## Using Constraints and Priorities Together

While constraints eliminate devices, priorities eliminate solutions. Used together, you can effectively focus the fitting/partitioning process to find the devices that best meet your needs.

Example: You can enable a constraint to eliminate devices with propagation delays greater than 50 nsec, and then specify a priority that indicates a *preference*, but not a requirement, for low-power devices.

## Running the PLSyn Fitter and Partitioner

### To start the fitting/partitioning process

- 1 From the Tools menu, select Fitter/Partitioner.

As PLSyn tries solutions, it updates the number of attempted and found solutions on the status line. If a successful solution ranks within the top ten, PLSyn places it in the solution list.

Depending on the amount of programmable logic in your design, and the number of architectures PLSyn has to consider, the fitting/partitioning process can take from less than a minute to several minutes or even hours. Therefore, select your constraints carefully.

The fitting/partitioning process can fail because:

- PLSyn can't find any parts in the available file which meet your constraints.
- PLSyn can't fit your logic into the architectures which did meet your constraints.



If this happens to you, try relaxing the constraints, thereby allowing PLSyn to consider additional device architectures.

# Selecting Devices

After PLSyn has found the solution(s) which will implement your design, you can select part numbers for each architecture in the solution list.

PLSyn uses the solution and part numbers that you choose to:

- update the schematic.
- perform timing simulations, and
- generate fuse maps.

You can explore different implementations by changing your selection to different part(s) or even a different solution.

## To select the PLD implementation

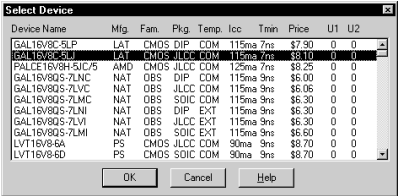
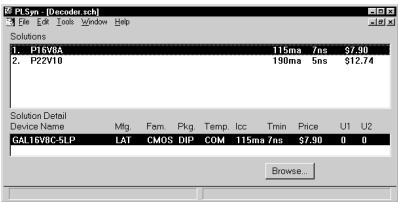
- 1    Select the solution (architecture) you want from the solution list.

A list of part numbers corresponding to the selected architecture appears in the solution detail list.

- 2    Select part numbers for the chosen solution. Either:

- double-click the device name in the list, or
- select the device and click Browse.

- 3    Select a different part and click OK.



# Creating Fuse Maps

After you have selected the PLD device(s) to implement your programmable logic, you can use PLSyn to create fuse maps. PLSyn creates one fuse map file, called a JEDEC file, for each device in the solution.

The JEDEC file has a special format used by your device programmer to determine which of the PLD fuses to blow.

## To create fuse maps

- 1 From the Tools menu, select Fuse Map Generator.

This command generates as many files as there are devices in the solution named *design\_name.jn*, where *n* is an integer from one to the number of devices.

After creating the fuse map file(s), PLSyn updates the documentation file with the names of the JEDEC files created for each device architecture. Each JEDEC file also contains the name of the device architecture in its header. The remainder of the device programming is handled by your device programmer.

## Including Test Vectors

The JEDEC file also includes any test vectors created during simulation, which the device programmer uses to validate the programming. If you have not run a simulation which generates test vectors, you will see a warning message, but you can continue creating the fuse map file without test vector information.

For more information, see *Generating Test Vectors* on page 4-6.

You will find this feature useful, for example, when you must change the functionality of your design after having laid out the printed circuit board.

For general information, see Chapter 6, *Controlling the Fitting Process Using the .pi File* and refer to the *PIL Reference* in PLSyn online help.

For more information on PCB layout, see [Creating PCB Netlists on page 5-29](#).

**Note** *If Schematics cannot find the PLD symbols, you need to add the PLD symbol libraries to your library configuration (using the Editor Configuration option in the Options menu). These library files are named `pld_xxx.slb`, where `xxx` is the three character manufacturer abbreviation.*

## The Implementation-Specific Physical Information File (.npi)

When you generate fuse maps, PLSyn creates a new physical information file named `design_name.npi`. This file contains the Physical Information Language (PIL) representation of your design's *current* implementation (target device(s) and pinout information) so that PLSyn can exactly duplicate the fitting and partitioning of your design in subsequent iterations.

### To recreate the implementation

- 1 In your design directory, copy the `design_name.npi` file to the `design_name.pi` file.
- 2 In PLSyn, from the Tools menu, select Fitter/Partitioner to refit the design.

**Note** *Groups and fixed groups to which the PLSyn fitter assigned a NAME property retain the given NAME in the .npi file.*

## Updating the Schematic

After you have selected a solution and PLD part number(s), you can back annotate your schematic with symbols for those parts. This is useful if when you want to create a PCB layout from your schematic.

### To update your schematic

- 1 From the Tools menu, select Update Schematic.

Schematics adds a page to your schematic, and places a symbol for each PLD in the selected implementation.

The PLD symbols use pin names which don't always match the pin-names found in the manufacturer's data books. However, the pin numbers and functionality are the same.

The used pins connect to either a:

- global port, or
- off-page port.

A global port connects a pin to its corresponding programmable logic interface node, which resides elsewhere on the schematic. If you have attached a global port to the programmable logic interface node, the PLD symbol's global port label is the same as the interface's global port label. Otherwise, the PLD symbol's global port label is in the form of *REFDES:pin name* corresponding to the programmable logic symbol or DSL block.

For an internal node in the programmable logic, the PLSyn fitter places an off-page port on the PLD pin. The fitter may do this for a variety of reasons; for example, to use a feedback path within a device, or to connect internal logic from one PLD to another.

**Note** *Each time you update your schematic from PLSyn, Schematics deletes then recreates the page containing the PLD symbols. Because of this, you should make few or no additions or changes to this page.*

For more information on interface nodes, see *Understanding Programmable Logic Nodes* on page 3-13.



## Creating PCB Netlists

After you have updated your schematic, you can create a PCB layout. You do not have to make any further changes to your schematic.

**Note** *Schematics only netlists non-programmable logic symbols including the back-annotated PLD symbols, and analog devices.*

For more information, see [Updating the Schematic on page 5-28](#).

For information on generating a PCB netlist, refer to your *MicroSim Schematics User's Guide*.

## When You Change the Design

When you make changes to your design, Schematics and PLSyn determine whether any changes have been made to the programmable logic or its interfaces. If changes have occurred, you must start the physical implementation process over at the compilation step. That means PLSyn will overwrite the original solution with the new solution.

For more information, see [The Implementation-Specific Physical Information File \(.npi\) on page 5-28](#) and *Specifying JEDEC File Names* on page 6-12.

You can *freeze* your latest implementation by copying the .npi file to the .pi file after having generated the fuse maps. On subsequent runs, PLSyn will create a physical implementation, including device numbers and pin-outs, exactly as specified in the .pi file.



---

# Controlling the Fitting Process Using the .pi File

---

## 6

## Chapter Overview

This chapter introduces the .pi file and ways of using it to control the fitting process. Topics include:

[Introduction to the .pi File on page 6-2](#)

[Controlling PLD Utilization on page 6-5](#)

[Fitting a Node as an OUTPUT or NODE on page 6-6](#)

[Controlling How Signals Are Fit Together on page 6-6](#)

[Disabling Outputs for Test on page 6-8](#)

[Controlling Synthesis on page 6-9](#)

[Controlling the Size of Equations on page 6-10](#)

[Specifying Devices without Specifying Signals on page 6-11](#)

[Specifying JEDEC File Names on page 6-12](#)

**Note** *This chapter does not describe the syntax of the Physical Information Language (PIL) statements used in the .pi file. Refer to the PIL Reference in PLSyn online help for this information.*

For more information on device-specific fitting, see:

- Chapter 7, *PLD Device-Specific Fitting*
- Chapter 8, *MACH 1-4 Device-Specific Fitting*
- Chapter 9, *MACH 5 Device-Specific Fitting*
- Chapter 10, *ATV5000 Device-Specific Fitting*

# Introduction to the .pi File

Though PLSyn can handle the physical implementation of your design automatically, you can also use the physical information file (.pi file), to exercise control over implementation during the optimization and fitting/partitioning process.

## Why Use the .pi File?

With PLSyn, programmable logic designs are completely device independent. This means, for example, that you don't need to make pin assignments with a DSL source file. However, you might need to control the mapping from design to device. For example, you might need to:

- Group signals together to make sure they are fit on the same device, while letting the PLSyn fitter and partitioner select devices and perform pin assignments automatically.
- Specify a device, letting the PLSyn fitter and partitioner perform pin assignments.
- Specify a device and some or all pin assignments.
- Control equation sizes.

The .pi file lets you do any of these and more.

## Using the Default .pi File

When you create a design for programmable logic synthesis, PLSyn copies the file `default.pi` (in the `bin` subdirectory under the MicroSim root directory) to a file named `design_name.pi` in your design directory. The default file contains the Physical Information Language (PIL) statements PLSyn needs to optimize and partition most designs automatically. You can add statements to this file or change it to suit your needs.

## Referring to Nodes in Your Design



Much of what goes into your `.pi` file controls the properties and placement of signals, which you identify by node name. In general, label the nodes that you plan to reference within the `.pi` file.

Here are a few considerations when working with the different node types.

For more information on nodes, see *Understanding Programmable Logic Nodes* on page 3-13

### Interface nodes

Use the associated label on your schematic directly in the `.pi` file.

### Internal nodes

PLSyn transforms node names which are internal to a group of programmable logic (including `NODES` statements inside of DSL blocks), into unique names by adding prefixes to the name.

The new name has the form

*proc\_name.instance\_name.local\_name*

where

<i>proc_name</i>	is the name of the procedure or function
<i>instance_name</i>	is the name assigned to this instance of the procedure or function
<i>local_name</i>	is the name of the INPUT or OUTPUT parameter or local NODE within the procedure or function.

If you are in doubt, refer to the equation section in the documentation file (.doc) for the list of actual node names PLSyn uses.

By default, the DSL compiler assigns 1 as the first *instance\_name*, 2 as the second *instance\_name*, and so on. You can also define the *instance\_name* in your DSL source.

To define the instance name of a DSL procedure or function

- 1 In your DSL source, specify the *instance\_name* in the procedure invocation statement as follows:

*instance\_name:procedure\_name(signal\_list)*

Example

Consider this DSL procedure:

```
PROCEDURE proc( INPUT d, clk; OUTPUT y );
  NODE dff CLOCKED_BY clk;
  dff = d;
  y = dff;
END proc;
```

The following table shows the nodes names the compiler generates given the DSL procedure invocation.

Procedure/Function Invocation	Generated Nodes
u1:proc(data3, clock3, q3);	proc.u1.d, proc.u1.clk, proc.u1.y, proc.u1.dff
proc(data1, clock1, q1);	proc.1.d, proc.1.clk, proc.1.y, proc.1.dff
proc(data2, clock2, q2);	proc.2.d, proc.2.clk, proc.2.y, proc.2.dff

# Controlling PLD Utilization

For some designs, you should reserve PLD resources for future logic expansion.

See also *Specifying Reserve Capacity* on page 8-9 for information on the MACH\_UTILIZATION property.

## To keep specific pins free

- 1 Use the NO\_CONNECT construct.

## To control the percentage of inputs, outputs, and product terms that can be used

- 1 Use the properties summarized in [Table 6-1](#).

**Table 6-1** PLD Utilization Properties

Property Syntax*	Meaning
{ PLD_INPUT_UTILIZATION % };	Sets the maximum percentage of array inputs on a device that may be used during fitting.
{ PLD_OUTPUT_UTILIZATION % };	Sets the maximum percentage of output pins or output macrocells on a device that may be used during fitting.
{ PLA_PTERM_UTILIZATION % };	Sets the maximum percentage of PLA and row-product terms used during PLA fitting. There is no equivalent control property for PALs.

\*. The default percentage for each of these properties is 100%, meaning that the device properties are fully utilized.

## Example

Suppose you are targeting a P22V10 architecture having defined the following utilization properties in your .pi file:

```
{PLD_INPUT_UTILIZATION 90};
{PLD_OUTPUT_UTILIZATION 80};
{PLA_PTERM_UTILIZATION 95};
```

PLSyn will use only 19 of the 22 available array inputs, and only 8 of the 10 available outputs.

If a PLA such as the S6001 is the target device, PLSyn will use only 60 of the 64 product terms.

# Fitting a Node as an OUTPUT or NODE

**To control whether a node is fit as an OUTPUT or as a NODE**

- 1 Use the FIT\_AS\_OUTPUT property.

FIT\_AS\_OUTPUT has no effect on output signals, which are already destined to be fit on a visible output pin of a device. For node signals, this property alerts the PLSyn fitter to place this node signal on an output pin.

# Controlling How Signals Are Fit Together

Early in the fitting process, the PLSyn decides which signals to fit together as one inseparable block of functionality. For PLDs, this means signals are fit in the same output macrocell. Signals can be fit together if a NODE is the only signal feeding another NODE or OUTPUT that has no register or latch equations.

**To control how signals are fit**

- 1 Use the NO\_COLLAPSE and FIT\_WITH properties.

**NO\_COLLAPSE** The NO\_COLLAPSE property tells PLSyn to fit this signal individually, separate from the fitting of any other signal.

**FIT\_WITH** The FIT\_WITH property lets you specify two signals to be fit together. The FIT\_WITH property is allowed on any .pi output, and takes one argument. For example, to say that signal node\_x should be fit with x, you would need to add the following statement to the .pi file:

```
node_x {FIT_WITH x};
```

## Example

### SOURCE FILE

```
INPUT d, e, clk, oe;
NODE d_node CLOCKED_BY clk;
NODE e_node CLOCKED_BY clk;
OUTPUT out, e_out, not_e_out ENABLED_BY oe;

d_node = d;
e_node = e;
out = d_node;
not_e_out = e_node;
e_out = e_node;
```

### PHYSICAL INFORMATION FILE

```
d_node {NO_COLLAPSE}
e_node {FIT WITH 'e_out'}
```

# Disabling Outputs for Test

In some cases, you might want to disable an output only during testing, but otherwise leave the output enabled during normal operation.

## To indicate that an output is disabled only during testing

1 Use the `DISABLED_ONLY_FOR_TEST` property.

When the output is *enabled*, PLSyn treats the input and output of a buffer as functionally different. When the output is *disabled* (using the `DISABLED_ONLY_FOR_TEST` property), PLSYN:

- Programs the enable equation.
- Treats the signal on the input of the tri-state buffer as equivalent to the signal on the output of the tri-state buffer (for feedback purposes).

## Example

The following PIL statement disables a single output:

```
out_x {DISABLED_ONLY_FOR_TEST};
```

If the output signal `out_x` has an enable, the PLSyn fitter programs the enable equation. If `out_x` is given only a single signal (e.g., `node_y`), then `out_x` and `node_y` are interchangeable (for feedback purposes).

## Example

The following PIL statement disables all outputs:

```
{DISABLED_ONLY_FOR_TEST};
```



# Controlling Synthesis

## To control DeMorgan synthesis of data equations in PLSyn:

Use the DEMORGAN\_SYNTH property where data equations are the D, JK, SR, T, XOR left and XOR right equations.

## Cautions when using the DEMORGAN\_SYNTH property

**Note** Because PLSyn automatically optimizes your design by default, there is generally little reason to use these properties.



When using DEMORGAN\_SYNTH, do not do the following:

- Control DeMorganization of control equations, such as ENABLE, CLOCK, RESET, or PRESET.
- Control DeMorganization of the J equation of a JK flip-flop with no corresponding DeMorganization of the K equation.

## To control flip-flop synthesis

- 1 Use the FF\_SYNTH property.

## To control XOR to Sum-of-Products synthesis

- 1 Use the XOR\_TO\_SOP\_SYNTH property.

**Table 6-2** summarizes the settings and meanings for all three of these properties.

**Table 6-2** *Synthesis Control Properties*

Property	Value	Action
DEMORGAN_SYNTH	AUTO (default)	The optimizer will automatically select the best DeMorganization choice.
	FORCE	Force the optimizer to DeMorganize the primary equation (use the offset).
	OFF	Prevent the optimizer from DeMorganizing the primary equation (use the onset).
FF_SYNTH	AUTO (default)	The optimizer will automatically do flip-flop synthesis to meet the needs of the target device.
	OFF	Require the target device to have the flip-flop type given in the design.
XOR_TO_SOP_SYNTH	AUTO (default)	The optimizer will automatically select between the XOR equation and the sum-of-products equation.
	FORCE	Force the optimizer to use the sum-of-products equation.
	OFF	Force the optimizer to use the XOR equation.

# Controlling the Size of Equations

Controlling the size of equations can have a major impact on the success of the PLSyn fitter and the number of solutions it generates.

## To control the size of equations

- 1 Use the MAX\_PTERMS and MAX\_SYMBOLS properties.

## Example

If you know that you want to use devices with macrocells that have eight or fewer PTERMs, then you want to keep the optimizer from collapsing nodes into equations with more than eight PTERMs using these PIL statements:

```
{MAX_PTERMS 8};
{MAX_SYMBOLS 16};
```

# Specifying Devices without Specifying Signals

## To specify the devices to use without specific pin information

- 1 Use the DEVICE property without a signal list.

## Example

The following PIL statements will fit a design into two MACH210 devices and a MACH130 device:

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH210-15JC';
END DEVICE;

DEVICE
  TARGET 'PART_NUMBER AMD MACH210-15JC';
END DEVICE;

DEVICE
  TARGET 'PART_NUMBER MACH130-15JC';
END DEVICE;
```

# Specifying JEDEC File Names

PLSyn automatically creates JEDEC files and saves them in your design directory, using names of the form *design\_name.jn* where *n* is a number from one up to the number of devices.

## To specify a name for each JEDEC file

- 1 Use the FUSEMAP\_FILE property within a DEVICE construct of the form:

```
{ FUSEMAP_FILE 'file name' } ;
```

## Example

```
DEVICE
{ FUSEMAP_FILE 'mypal.jedec' } ;
.
.
.
END DEVICE ;
```

# More Examples Using the .pi File

## Forcing Signals to be Fit Together in the Same Device

### Scenario

You have a design that implements a counter, and the output signals are heavily interdependent. For timing reasons you want them to be fit together in the same device, but want the automatic device selection and partitioning to determine the best device according to your priorities.

### Solution

```
GROUP
    q0..q5, carry;
END GROUP;
```

which tells PLSyn to fit the signals that are members of the GROUP, `q0..q5` and `carry` together in the same device. There are no limitations imposed by the GROUP on the device to use. In addition, other groups and ungrouped signals can fit in the same device with this group.

## Using Specific Devices

### Scenario

You are prototyping a small design, and have several reprogrammable P16V8As that you know that you want to use during the debugging stage.

### Solution

```
DEVICE
  TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
  o [0..6];
  carry;
END DEVICE;
```

Using the GROUP construct instead would specify that the signals o[0..6] and carry must fit together into the same device.

The DEVICE construct specifies that no other groups can be fit into the same device. This means that you can give device-specific information in fixed groups. One kind of device-specific information is the device to TARGET or fit this fixed group into. Here, the target device is named by its PART\_NUMBER.

# Maintaining Pin Assignments

## Scenario

You have an existing design in which you have changed some logic and you want to refit the design into the same device. The device is a P20V8 in a JLCC package, and you want to maintain the pin assignments.

## Solution

```
DEVICE
  TARGET 'TEMPLATE P20V8 JLCC-28-P28';
  INPUT clk:2, in1:3, in2:4, in3:5, in4:6;
  out1:18, out2:19, out3:20, out4:21;
  NO_CONNECT 7..13, 15, 22..27;
END DEVICE;
```

where, the target device is named by its TEMPLATE P20V8 and its footprint JLCC-28-P28.

A template is a device architecture and the footprint is a certain pinout configuration consisting of three things:

- The type of package (e.g., DIP, SOIC, or JLCC).
- The number of pins in the package.
- The mapping of physical pins to logical, or virtual, pins.

## Example

DIP-24-STD indicates a 24 pin DIP package with the standard pinout mapping (pin 12 as ground and pin 24 as VCC). Most parts use a standard pin mapping, abbreviated as STD. An example of a non-standard pin mapping is the 4.5ns P16L8 from AMD, which uses extra power and ground pins in a 28-pin DIP. The footprint for such a device is a DIP-28-A28.

Signals used as inputs to the device are marked with INPUT in the .pi file. The signals in the fixed group are assigned to pins by appending `:pin_name` to the signal name, such as `clk:2`. If device pins must be left free, use the NO\_CONNECT property. The pin names in the pin assignments and no-connect pins are the actual physical pin names for the targeted device.

## Fitting the Design into One Device

### Scenario

You want to fit your entire design into one AMD PAL16R6B4CJ.

### Solution

```
DEVICE
  TARGET 'PART_NUMBER AMD PAL16R6B4CJ' ;
  DEFAULT ;
END DEVICE ;
```

The DEVICE specification is marked as the DEFAULT group. The default group is the group that contains all the output signals that you have *not* mentioned elsewhere in the .pi file.

Specifying a default group is optional. Here, it provides a quick way to put all the signals in the design into the same device. You can also specify DEFAULT at the global level, outside of any group or DEVICE specification. This means PLSyn will automatically fit and partition all unmentioned signals.



## Fitting the Design into Multiple Devices

### Scenario

You have a design that will take two AMD parts.

### Solution

```
DEVICE
  TARGET 'PART_NUMBER AMD PAL16R6B4CJ';
  out1..out5;
END DEVICE;

DEVICE
  TARGET 'PART_NUMBER AMD PAL16R6B4CJ';
  out6..out10;
END DEVICE;
```

## Mixing Automatic and Directed Partitioning

### Scenario

Assume that your design is similar to the design of the last example. However, it has several critical functions that you want placed into fast PLDs.

### Solution

```
DEVICE
  TARGET 'PART_NUMBER AMD PAL16R6B4CJ';
  out1..out5;
END DEVICE;

DEVICE
  TARGET 'PART_NUMBER AMD PAL16R6B4CJ';
  out6..out10;
END DEVICE;
```

**Note** *The contents of this .pi file are the same as the previous example. In this case, nothing needs to be said in the .pi file about the critical functions. If you prioritize for speed during partitioning, PLSyn will automatically find the fastest device or combination of devices available that will fit the critical functions.*

# Refitting a Design into the Same Footprint

## Scenario

Your board is already in production, but a logic flaw indicates that you have changed the logic implemented in your PLD. This causes your design to outgrow the P20R8 you were using. You need to refit the design into another architecture, but must keep the pinout the same.

## Solution

```
DEVICE
  TARGET 'FOOTPRINT DIP-24-STD';
  INPUT clk:1, oe:13, in1:2, in2:3, in3:4, in4:5;
  INPUT in5:6, in6:7, in7:8, in8:9, in9:10, in10:11;
  INPUT in11:14, in12:23;
  out1:15, out2:16, out3:17, out4:18;
  out5:19, out6:20, out7:21, out8:22;
END DEVICE;
```

The fixed group is targeted to FOOTPRINT DIP-24-STD. Targeting a device to a footprint causes automatic device selection and fitting across devices that match the footprint. Depending on the form of the actual equations, there are up to 79 architectures that can potentially fit this example.

For this example, the P20R8 architecture and P312 architecture in a DIP package both have the same footprint, so you can use the P312 instead of the outgrown P20R8. The old pin assignments are enforced, regardless of which architecture you choose. This means that the board layout is preserved.

**Note** *You can narrow the search by setting constraints and priorities to optimize the fit for price, speed, or other factors.*

---

# PLD Device-Specific Fitting

---

# 7

## Chapter Overview

This chapter describes how to control the fitting process for specific PLD device architectures using the `.pi` file.

[Accessing Internal Points in a PLD Device on page 7-2](#) describes the different kinds of internal nodes and how to reference them in your `.pi` file.

[Fitting Specific Device Architectures on page 7-11](#) describes control mechanisms for the 22V10, 750, 2500, P22V10I, P750B, P2500B, P1800, P16V8HD, P22VP10, and P16VP10, including:

- Handling synchronous preset
- Assigning combinatorial output during feedback
- Controlling clock source
- Controlling quadrant-based architectures
- Accessing the open-drain output

# Accessing Internal Points in a PLD Device

## To reference signals internal to a PLD device

- 1 Use the node name convention corresponding to the kind of node: hidden/buried, shadow, unary.

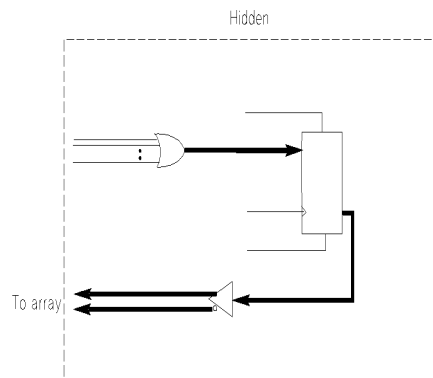
This section describes the node types and naming conventions. See [Table 7-1 on page 7-6](#) for a summary of the node naming conventions that apply to specific PLD device architectures.

## The Kinds of Nodes

### Hidden nodes

A hidden node is a node that does not terminate in a physical pin connection. Shadow and buried nodes are examples of hidden nodes, typically used to hold functions used only within a device.

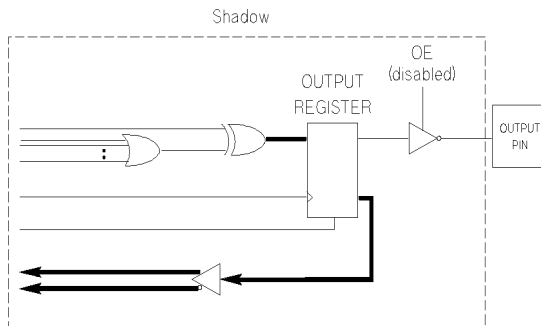
Node signals are signals that you place on hidden nodes. However, node signals are not restricted to hidden nodes; you can also place them on visible pins.



**Figure 7-1** *Hidden Node*

## Shadow nodes

You can create a shadow hidden node (known simply as a shadow node or shadow) by disabling the output buffer of a normal output macrocell. The shadow node terminates with the internal feedback to the array, and is therefore not visible outside the device as shown in Figure 7-2.



**Figure 7-2** *Shadow Node*

## Buried nodes

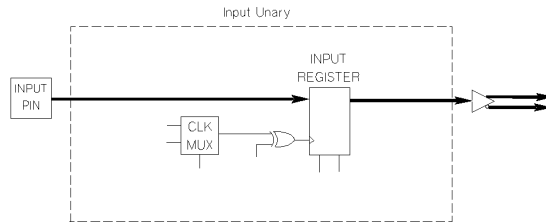
A buried node is a hidden node where some external pin number is associated.

## Unary nodes

Unary nodes are nodes with a single input. Usually the node is registered. There are two basic types of unaries. The most common is a registered input pin, also called an input unary. A second type is a clocked feedback path, called a feedback unary.

### Input unary

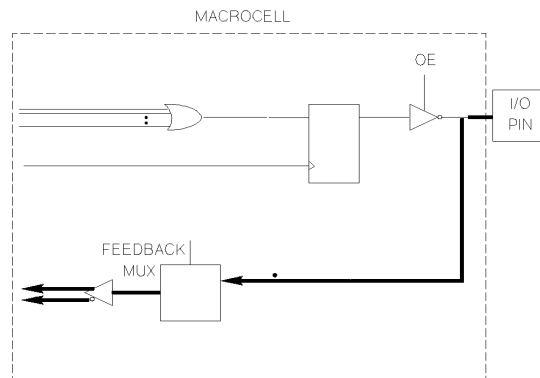
An input unary is a hidden unary in an input macrocell, i.e., a clocked input pin, as shown in Figure 7-3.



**Figure 7-3** *Input Unary*

### Feedback unary

A feedback unary is a hidden unary path through the feedback register of an output macrocell, as shown in Figure 7-4.



**Figure 7-4** *Feedback Unary*

## To select a node or unary path

- 1 In your .pi file, use the label associated with the node or unary using the following labeling convention:

hidden node	NODE##
unary node	UNARY_OF_##
buried node	BURIED_OF_##

where ## is the manufacturer-specified pin number in the primary package, usually DIP.

**Note** MACH devices use a different naming convention. For more information, see *Understanding Pin Naming and Numbering* on page 8-17.

## Example

There are a large number of devices that have general-purpose registers. This example shows how you can define DSL that allows the fitter to take advantage of these general-purpose registers.

The following DSL statements reflect a clocked input defined as a unary node:

```
INPUT i_unlocked, clk;
NODE i CLOCKED_BY clk;

i = i_unlocked;
```

This approach provides certain advantages over a standard clocked input.

- The design references both the clocked (i) and unlocked (i\_unlocked) versions of the signal.
- Reference the hidden node in the .pi file.
- Map this description into any device with a register.



**Table 7-1** *Node Descriptions and Labels by Device Architecture*

Architecture	Pin Description	Pin Label
P16V8HD	Input unaries	UNARY_OF_2...UNARY_OF_9
	Feedback unaries	UNARY_OF_13...UNARY_OF_16
		UNARY_OF_19...UNARY_OF_20
		UNARY_OF_22...UNARY_OF_23
P204R	Shadow nodes	SHADOW_OF_12...SHADOW_OF_19
P23S8	Buried nodes	BURIED_OF_13...BURIED_OF_18
P241R	Shadow nodes	SHADOW_OF_4...SHADOW_OF_9
		SHADOW_OF_14...SHADOW_OF_23
P2500	Shadow nodes	SHADOW_OF_4...SHADOW_OF_9
		SHADOW_OF_11... SHADOW_OF_16
		SHADOW_OF_24... SHADOW_OF_29
		SHADOW_OF_31... SHADOW_OF_36
	Buried nodes	BURIED_OF_4...BURIED_OF_9
		BURIED_OF_11...BURIED_OF_16
		BURIED_OF_24...BURIED_OF_29
		BURIED_OF_31...BURIED_OF_36
P29M16	Shadow nodes	SHADOW_OF_3, SHADOW_OF_4
		SHADOW_OF_9, SHADOW_OF_10
		SHADOW_OF_15, SHADOW_OF_16
		SHADOW_OF_21, SHADOW_OF_22
	Input unaries	UNARY_OF_3...UNARY_OF_10
		UNARY_OF_15...UNARY_OF_22
P29MA16	Shadow nodes	SHADOW_OF_3, SHADOW_OF_4
		SHADOW_OF_9, SHADOW_OF_10
		SHADOW_OF_15, SHADOW_OF_16
		SHADOW_OF_21, SHADOW_OF_22
	Input unaries	UNARY_OF_3...UNARY_OF_10
		UNARY_OF_15...UNARY_OF_22
P312	Input unaries	UNARY_OF_3...UNARY_OF_10
	Shadow nodes	SHADOW_OF_2, SHADOW_OF_11
		SHADOW_OF_14...SHADOW_OF_23

**Table 7-1 Node Descriptions and Labels by Device Architecture (continued)**

Architecture	Pin Description	Pin Label
P324	Shadow nodes	SHADOW_OF_4...SHADOW_OF_7 SHADOW_OF_9...SHADOW_OF_12 SHADOW_OF_14...SHADOW_OF_17 SHADOW_OF_24...SHADOW_OF_27 SHADOW_OF_29...SHADOW_OF_32 SHADOW_OF_34...SHADOW_OF_37
	Input unaries	UNARY_OF_2, UNARY_OF_3 UNARY_OF_18...UNARY_OF_20 UNARY_OF_22, UNARY_OF_23 UNARY_OF_38...UNARY_OF_40  UNARY_OF_2, UNARY_OF_3 UNARY_OF_18...UNARY_OF_20 UNARY_OF_22, UNARY_OF_23 UNARY_OF_38...UNARY_OF_40
P332	Input unaries	UNARY_OF_1...UNARY_OF_7 UNARY_OF_9...UNARY_OF_14
	Feedback unaries	UNARY_OF_15...UNARY_OF_20 UNARY_OF_23...UNARY_OF_28
P336/P337	Input unaries	UNARY_OF_1...UNARY_OF_6 UNARY_OF_9...UNARY_OF_14 UNARY_OF_9...UNARY_OF_14
P32VX10	Shadow nodes	SHADOW_OF_14...SHADOW_OF_23
P448	Shadow nodes	SHADOW_OF_13 SHADOW_OF_15...SHADOW_OF_17 SHADOW_OF_19 SHADOW_OF_21...SHADOW_OF_23
P750	Buried nodes	BURIED_OF_14...BURIED_OF_23
	Shadow nodes	SHADOW_OF_14...SHADOW_OF_23
S105	Hidden nodes	NODE29...NODE34
S167/S168	Hidden nodes	NODE25...NODE30

Table 7-1 Node Descriptions and Labels by Device Architecture (continued)

Architecture	Pin Description	Pin Label
S30S16	Input unaries	UNARY_OF_21...UNARY_OF_24
	Hidden nodes	NODE29...NODE32
	Shadow nodes	SHADOW_OF_8...SHADOW_OF_9
	Shadow nodes	SHADOW_OF_15...SHADOW_OF_20
S405/S415	Hidden nodes	NODE29...NODE36
S506	Hidden nodes	NODE25...NODE40
S507	Hidden nodes	NODE25...NODE32
S6001/S6002	Hidden nodes	NODE25...NODE32
	Shadow nodes	SHADOW_OF_14...SHADOW_OF_23
	Input unaries	UNARY_OF_2...UNARY_OF_11
	Feedback unaries	UNARY_OF_14...UNARY_OF_23

The architectures which have unary nodes are the P16V8HD, P29M16, P29MA16, P312, P324, P330, P331, P332, S30S16, S6001, ATV5000, and the MACH2xx parts.

For more information on the MACH2xx parts, see Chapter 8, *MACH 1-4 Device-Specific Fitting*.

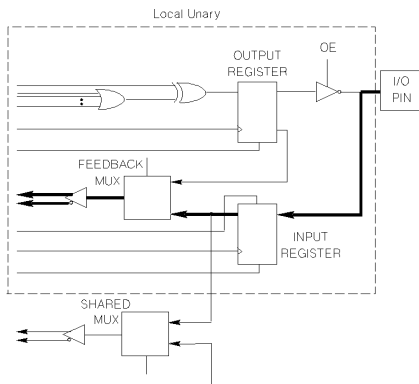
For more information on the ATV5000 parts, see Chapter 10, *ATV5000 Device-Specific Fitting*.

## Unary Nodes in the P330 and P331

The P330 and P331 architectures have unusual types of hidden unaries. In addition to the clocked input paths (input unaries), they have clocked feedback paths in the output macrocells. Neighboring macrocells can share the clocked feedback paths, which can result in a large number of hidden paths.

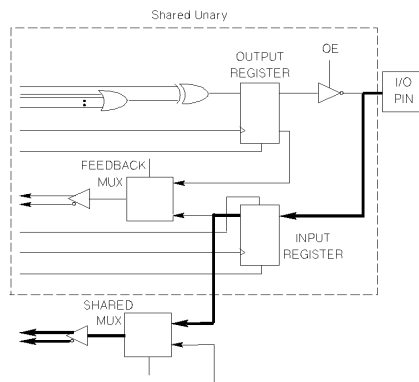
In the P330 and P331, you can build two types of unaries with these kinds of paths: local unary and shared unary

**Local unary** The local unary has a path through the feedback multiplexer, as shown in Figure 7-5.



**Figure 7-5** *P33x Local Unary*

**Shared unary** The shared unary has a path through a shared multiplexer, as shown in Figure 7-6.



**Figure 7-6** *P33x Shared Unary*

To select a node or unary path in a P330 or P331

1 In your .pi file, use the label associated with the node or unary according to the following labeling convention:

hidden node	NODE##
standard unary node	UNARY_OF_##
local unary node	LOCAL_OF_##
shared unary node	SHARED_OF_##
shadow node	SHADOW_OF_##

where ## is the manufacturer-specified pin number in the primary package, in this case, DIP.

Table 7-2 summarizes the node labels for the P330 and P331.

Table 7-2 Node Descriptions and Labels for P330 and P331

Architecture	Pin Description	Pin Label
P330	Hidden nodes	NODE29...NODE32
	Input unaries	UNARY_OF_3...UNARY_OF_7 UNARY_OF_9...UNARY_OF_14
	Local feedback unaries	LOCAL_OF_15...LOCAL_OF_20 LOCAL_OF_23...LOCAL_OF_28
	Shadow nodes	SHADOW_OF_15...SHADOW_OF_20 SHADOW_OF_23...SHADOW_OF_28
	Shared feedback unaries	SHARED_OF_15...SHARED_OF_20 SHARED_OF_23...SHARED_OF_28
P331	Shadow nodes	SHADOW_OF_15...SHADOW_OF_20 SHADOW_OF_23...SHADOW_OF_28
	Local feedback unaries	LOCAL_OF_15...LOCAL_OF_20 LOCAL_OF_23...LOCAL_OF_28
	Shared feedback unaries	SHARED_OF_15...SHARED_OF_20 SHARED_OF_23...SHARED_OF_28

# Fitting Specific Device Architectures

## 22V10, 750, and 2500: Handling Synchronous Preset

PLSyn supports several device architectures that have a synchronous reset. If PLSyn has DeMorganized the D equation on a device, then the asynchronous reset is now an asynchronous preset and the synchronous preset is a synchronous reset. Given this anomaly and the priority PLSyn places on insuring the same functionality for various implementations, PLSyn does not fit a preset equation onto any synchronous preset.

In some architectures, however, you can still use the *common set* (set or preset). A synchronous preset is like an extra AND row input to the OR, but available only when the output is registered.

### Using set and preset for the 22V10 and 750

For the 22V10 (which includes the P22V10, P22VP10, and P22V10I) and the 750 (which includes the P750B), the synchronous preset row is common to all macrocells in the device.

### To use set or preset in the 22V10 and 750 architectures

- 1 Use the COMMON\_SET\_PTERM property in your .pi file.

### Example

#### SOURCE FILE

```
INPUT clk, reset1, reset2;
OUTPUT a[10] CLOCKED_BY clk;

IF (reset1*reset2) THEN
    a = 0;
ELSE
    a = a .+. 1;
END IF;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE

    {COMMON_SET_PTERM 'reset1*reset2';};

TARGET 'TEMPLATE P22VP10 DIP-24-STD';

a;

END DEVICE;
```

The common set PTERM is `reset1*reset2`. This term sets the output low, so PLSyn automatically uses the DeMorgan to meet this common set PTERM requirement.

### Using set and preset for the 2500

The 2500 architecture (which includes the P2500) has eight synchronous preset rows shared by 2 or 4 macrocells.

#### To use set or preset in the 2500 architecture

- 1 Use the SET\_PTERM property in your .pi file.

If no pin assignments are given, PLSyn automatically determines macrocell pairing to meet the SET\_PTERM requirements.

## Example

### SOURCE FILE

```

INPUT clk, reset1, reset2;
OUTPUT a[10] CLOCKED_BY clk;

IF (reset1*reset2) THEN
    a = 0;
ELSE
    a = a .+. 1;
END IF;

```

### PHYSICAL INFORMATION FILE

```

DEVICE
    TARGET 'ATM ATV2500H-25DC';
    a {SET_PTERM' reset1* reset2 '};
    .
    .
    .
END DEVICE;

```

**Note** *If you specify the pin numbers for macrocells that share a synchronous preset term, all of the macrocells must have the same SET\_PTERM requirements.*

## P22V10I: Assigning Combinatorial Output During Feedback

Using the P22V10I device, you can assign a combinatorial output while feeding back a registered version of the signal.

### To assign combinatorial logic to the P22V10I architecture

- 1 Describe the DSL logic as follows:
  - Assign the logic to an internal combinatorial node.
  - Assign the internal combinatorial node to an internal registered node.



- Assign the internal combinatorial node to a combinatorial output. If needed, you can define this output to have an output enable.
- 2** In your .pi file, attach the COM\_OUT\_REG\_FB property to the output signal.

### Example

#### SOURCE FILE

```
INPUT clk, in1 in2;
OUTPUT out1, out2;
PHYSICAL NODE before_feedback;
NODE after_feedback CLOCKED_BY clk;

before_feedback = in1;
out1 = before_feedback;
after_feedback = before_feedback;
out2 = after_feedback * in2;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE
    TARGET 'TEMPLATE P22V10I DIP-24-STD';

    "Force the Combinatorial
    "Output/Registered Feedback mode
    "using out1 as the output and
    "after_feedback as the
    "registered feedback mode

    out1 {COMB_OUT_REG_FB after_feedback};
END DEVICE;
```

## P750B AND P2500B: Controlling Clock Source

The Atmel P750B and P2500B architectures can provide the clock for the registers from two locations:

- A dedicated clock pin.
- A row in the fuse array.

## To control the clock source for the P750B and P2500B architectures

- 1 Add a `CLOCKED_BY_XXX` property to the output or nodes that you wish to control, in your `.pi` file as follows:

<code>CLOCKED_BY_PIN</code>	The register must be clocked by the signal on the dedicated clock pin.
<code>CLOCKED_BY_ROW</code>	The register must be clocked by the internal clock product term.

**Note** *Do not use the `CLOCKED_BY_PIN` property when the signal is clocked by an equation (for example, `CLOCKED_BY (a*b)`).*

### Example

#### SOURCE FILE

```
INPUT clk, in1;
OUTPUT out1 CLOCKED_BY clk;
out1 = in1;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'TEMPLATE P750B DIP-24-STD';
  out1 {CLOCKED_BY_PIN}; "Force the
    "clock to come from the clock pin
END DEVICE;
```

### Example

#### SOURCE FILE

```
INPUT clk, in1;
OUTPUT out1 CLOCKED_BY clk;
out1 = in1;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'TEMPLATE P750B DIP-24-STD';
  out1 {CLOCKED_BY_ROW}; "Force the clock
    "to come from the product term
END DEVICE;
```

**Note** *If you do not specify CLOCKED\_BY\_PIN or CLOCKED\_BY\_ROW, the fitter will attempt to use CLOCKED\_BY\_PIN first, then will try to use CLOCK\_BY\_ROW.*

## P1800: Controlling Quadrant-Based Architectures

The P1800 device architecture is different from other PLDs because it has *quadrants*. Within a quadrant, local macrocells and the pre-enable feedback of global macrocells feed only the same quadrant and do not feed the other three quadrants (the input pins and the post-enable feedback of the global macrocells feed the entire device.)

### Assigning pins and nodes

You can assign a signal to a pin in much the same way as any other device using the .pi file. Unless it is in a SECTION, an OUTPUT in a P1800 DEVICE must have a pin assignment. An output signal without a pin assignment is ambiguous since the fitter needs to know (at a minimum) the quadrant you want.

**Table 7-2** lists the sixteen shadow nodes in the P1800 architecture which can accept node signal assignments.

**Table 7-3** *Node Descriptions and Labels for P1800*

Architect ure	Pin Descripti on	Pin Label
P1800	Shadow	SHADOW_OF_10...SHADOW_OF_13 SHADOW_OF_23...SHADOW_OF_26 SHADOW_OF_44...SHADOW_OF_47 SHADOW_OF_57...SHADOW_OF_60

As you make pin or node assignments, be aware of the requirements imposed by the quadrants of the P1800. For example, if signal x needs signal y, and signal x is assigned to a local macrocell output pin, then y must be fit in the same quadrant or the signal x must be brought in on a global input pin.

## Subgroups: Targeting quadrants

### To indicate a target quadrant in a P1800 device

- 1 Use the SECTION construct in your .pi file.

You can target the SECTION to any one of the four quadrants, labeled A, B, C, and D (the target string should contain the word quadrant followed by the quadrant letter (for example, TARGET 'Quadrant B';). You can include OUTPUTS without pin assignments in the SECTION construct.

### Example

```

DEVICE
  TARGET 'TEMPLATE P1800 JLCC-68-STD';
  SECTION
    TARGET 'Quadrant A';
    a:3;          " Place a on pin 3
                  " of quadrant A
  END SECTION;
  b:34, c:SHADOW_OF_44;
                  " b on pin 34,
                  " c on pin 44's shadow
  SECTION
    TARGET 'Quadrant D';
    d1,          " Place d1 ANYWHERE
                  " in quadrant D.
    d2:57;       " d2 goes on pin 57
                  " of quadrant D.
  END SECTION;
END DEVICE;

```

## P16V8HD, P22VP10, and P16VP10: Accessing the Open-Drain Output

The P16V8HD, P22VP10, and P16VP10 architectures support open-drain outputs. Unlike normal totem-pole outputs, an open-drain output only drives  $V_{ol}$ . Whereas  $V_{oh}$  is driven on a totem-pole output, nothing is driven from an open-drain output. The

voltage level of an open-drain output depends on external loading and pull-up circuitry.

In your .pi file, you can direct outputs to be open drain by attaching the OPEN\_DRAIN property to the output signals, provided those outputs support open drain.

To express this functionality, the enable equation of an output (in this case x) must be of the form:

```
/internal_name_for_x * enable_equation
```

This means that the output is enabled only if the data is low and the enable equation is true. The value *internal\_name\_for\_x* is any signal just prior to the enable buffer of the output on the device. The enable equation is independent of the open-drain functionality.

PLSyn provides a function that you can use to create open-drain output signals of the proper form:

```
FUNCTION open_drain(d, oe);  
  NODE out ENABLED_BY /d*oe;  
  out = d;  
  return out;  
END open_drain;
```

## Example

### SOURCE FILE

```
USE 'dfeature' ;  
LOW_TRUE INPUT oe;  
INPUT i, j, clk;  
NODE l_x CLOCKED_BY clk;  
OUTPUT x;  
  
i_x = i*j;  
x = open_drain(i_x, oe);
```

### PHYSICAL INFORMATION FILE

```
DEVICE  
TARGET 'PART_NUMBER amd PALCD16V8HD-15PC';  
x {OPEN_DRAIN};  
END DEVICE;
```

Once an output is in the proper form for an open-drain configuration, the simulator can simulate the functionality

correctly and test vectors sent to the device programmer are also be correct.

PLSyn generates two enable equations:

- open-drain capable devices
- all other devices

In the example given above, the enable equation for open-drain outputs is `oe`, and the enable equation for other outputs is `i_x*oe`. To maintain device independence, you can fit an output onto parts without the open-drain capability at the cost of increased enable equation complexity. Consider timing and parametric design issues independently of PLSyn's open-drain synthesis capability.

You can also use the open-drain function to aid in the design of buses.

## Example

### SOURCE FILE

```
USE 'dfeature';

" Declare the inputs
INPUT input_bus1[4];
INPUT input_bus2[4];
INPUT clk;

" Declare the two buses and the
" associated wired bus
NODE internal_bus1[4] CLOCKED_BY clk;
NODE internal_bus2[4] CLOCKED_BY clk;
OUTPUT bus1[4];
OUTPUT bus2[4];
WIRED_BUS combined_bus[4] : bus1, bus2;

" Declare an output that will refer to
" the wired bus
OUTPUT and_all;

" Make assignments to the two buses
internal_bus1 = input_bus1;
internal_bus2 = input_bus2;
```

```
" Declare each bus to have
" open-drain outputs
bus1[0] = open_drain (internal_bus1[0], 1);
bus1[1] = open_drain (internal_bus1[1], 1);
bus1[2] = open_drain (internal_bus1[2], 1);
bus1[3] = open_drain (internal_bus1[3], 1);
bus2[0] = open_drain (internal_bus2[0], 1);
bus2[1] = open_drain (internal_bus2[1], 1);
bus2[2] = open_drain (internal_bus2[2], 1);
bus2[3] = open_drain (internal_bus2[3], 1);

" Reference the wired bus and_all =
" combined_bus[0]*combined_bus[1]*
" combined_bus[2]*combined_bus[3];
```

### **PHYSICAL INFORMATION FILE**

```
bus2 {OPEN_DRAIN };
bus1 {OPEN_DRAIN };
```

---

# MACH 1-4 Device-Specific Fitting

---

## 8

### Chapter Overview

This chapter describes how to control the fitting process for specific MACH 1-4 device architectures. Topics include:

- When to design with MACH devices, *page [8-2](#)*
- Summary of MACH device properties, *page [8-3](#)*
- Tips and device details, *pages [8-10](#) through [8-49](#)*
- The report file, *page [8-52](#)*

See Appendix C, *AMD MACH Device Tables* for detailed information on:

- Device-specific pin names
- Fuse commands for forcing outputs to be driven



For additional device-specific information, refer to the *MACH Family Data Book* from AMD.

See Chapter 6, *Controlling the Fitting Process Using the .pi File* and the *PIL Reference* in PLSyn online help for more information on the .pi file.

# Designing with MACH Devices

MACH devices, summarized in [Table 8-1 on page 8-3](#), are handled like any other PLD with full support for automatic device selection and partitioning. As with PLDs, you can also control implementation using the .pi file.

## When You Have Fitting Problems

If your design fails to fit, there are several tools to help you find the problem(s). These include the:

- Log file
- Report file

### Using the log file

The PLSyn fitter generates the log file (.log) every time it runs. The log file is the *first* place to look when you have fitting problems.

If a fitting run fails, the log file contains information that explains why the design did not fit. If you are using group and pin assignments in the .pi file, the log file contains any messages regarding the validity of these assignments.

For more information, see [The MACH Report File on page 8-52](#).

### Using the report file

When you specify a MACH device in the .pi file, the PLSyn fitter generates a device-specific report file (.rpt)., whether the fitter succeeds in fitting or not. If the fitter fails, the report file contains valuable information that shows which resources presented the most problems in fitting. Use this information to help you decide how to change the design or the .pi file to make the design fit easier.

# Summary of MACH Devices

Table 8-1 summarizes the properties of MACH devices.

**Table 8-1** *MACH Device Properties\**

Device	Pins	Blocks	Array Inputs	Max Pterms	OMCs	BMCs	Input Regs	Inputs	Clocks
MACH 110	44	2	22	12	32	0	0	4	2
MACH 210	44	4	22	16	32	32	0	4	2
MACH 215	44	4	22	12	32	0	32	6	2+32
MACH 120	68	4	26	12	48	0	0	4	4
MACH 220	68	8	26	16	48	48	0	4	4
MACH 130	84	4	26	12	64	0	0	2	4
MACH 230	84	8	26	16	64	64	0	2	4
MACH 435	84	8	33	20	64	64	64	2	4+128
MACH 465	208	16	34	20	128	128	128	14	4+256

\*. For speed values, see the MACH Family Data Book from AMD.

## Output Enable Functions

**MACH 1xx** These devices have 12 or 16 outputs per block. There are two OE PTERMs for the top half of the block, and two OE PTERMs for the bottom half of the block. Each output selects its OE from either of the two available PTERMs or a constant: 1 or 0.

**MACH 2xx** These devices have 6 or 8 outputs per block. There are two OE PTERMs per PAL block. Each output selects its OE from either of the two available PTERMs or a constant: 1 or 0.

**MACH 215, MACH 4xx** These devices have one OE PTERM per output. You can program them independently as 1, 0, or any product of signals in the block.

## Register Reset/Preset Functions

**MACH 1xx, MACH 2xx** These devices have one reset and one preset in each block. The reset and preset apply to all registers in the block.

**Note** *Note, a registered function without a reset (or preset) is the same as RESET\_BY 0. This will not fit in the same block with other functions with non-zero reset expressions.*

**MACH 215** This device has a reset and preset PTERM for each output register. The input registers do not have reset capabilities.

**MACH 4xx** These devices have one reset and one preset in each block. These apply to the macrocells but not to the input registers. The macrocells have an asynchronous option which allows for a local reset *or* preset, but not both, for individual functions.

## Packaging

All like pin-count packages are pin compatible. For example, when a MACH 110 design exceeds the capacity of the device, you can generally substitute a MACH 210.

# Using Standard Clock Functions

## MACH 1xx, MACH 2xx: Synchronous Clock Functions

These devices support pin clock only.

## MACH 215, 3xx, 4xx: Asynchronous Clock Functions

Although both the MACH 215 and MACH 4xx support asynchronous functions, some functions or groups of functions fit only in the MACH 215. These are:

- Functions that are clocked by a PTERM and have a reset and preset.
- Groups of functions that have more than eight distinct pairs of reset and preset equations.

**MACH 215** This device supports pin clock or clock by PTERM.

You can clock the output macrocells by:

- pin 13, or
- a local PTERM, or
- the inverse of either of those signals.

You can clock the input registers by:

- pin 13, or
- pin 35, or
- the inverse of either of those signals.

**Note** For all MACH devices, the clock signals are also signal inputs to the switch matrix. You can route these to the blocks.

If your design needs a clock which is more complex, you can define a clock using a complex logic function. See [Using Complex Clock Functions on page 8-6](#).

**MACH 3xx and 4xx** These devices support clock by pin or clock by PTERM. You can set the pin clock mode from:

- any of four clock pins, or
- the inverse of those signals.

See device manufacturer literature for specifics.

Not all possible clock signal and inverse combinations are available in a given block.

## Using Complex Clock Functions

When a design requires a clock expression that can't be implemented directly in the clock resources of a MACH device, you can place the clock logic in a separate NODE or OUTPUT. The PLSyn fitter automatically wires the function to the clock resources of the device.

**MACH 1 and 2** You can use the complex clock output in the MACH 1 & 2 families either internally or externally as the clock. The only exception is if the MACH 215 clock pin is unavailable. Then PLSyn routes the clock signal to the PAL blocks where it is needed and connects it using the clock PTERM.

**MACH 3 and 4** You can use a function generated in the MACH 3 & 4 families either internally or externally as the clock. The PLSyn fitter defaults to using the clock signal internally to save the pins used in external routing. To prevent the clock from taking an I/O pin, you can declare the clock function to be a node.

If you need the faster timing provided by an external clock pin connection, simply place the clock signal on a clock pin in the .pi file.

## Example

The following source file can fit into any MACH device.

```
input i;  
input c1, c2;  
output ck;  
output a clocked_by ck;  
a = 1;
```

## Clock Limitations

- The synchronous MACH parts (MACH 1x0 and MACH 2x0) can only be clocked by pin.
- The synchronous MACH parts (MACH 215 and MACH 3 & 4 families) can clock by a single PTERM, and can invert clock signals in most cases.

In either case, the PLSyn fitter allows you to generate and use a more complex clock than the part supports directly. For example, you can use the sum of two or more PTERMs, or a single PTERM on an asynchronous part.

**Note** *This costs some extra delay.*

### To use a more complex clock than the part supports

- 1 Create an output node with a data equation that is the clock function you want generated.
- 2 Use the output node as the clock signal.

# Implementing Hazard-Free Combinatorial Latches

You may need to implement combinatorial latches in MACH devices. A combinatorial latch is a simple combinatorial function in which the output is derived from inputs and feedbacks. A seemingly correct latch design can be subject to hazard conditions that might cause latch failure. By inserting redundancy into the latch equation, you can protect against hazard conditions.

## Basic Latch Circuit

The basic transparent DLatch expression looks like the following:

```
INPUT Data;
INPUT LatchEnable;
NODE DLatch;
DLatch = LatchEnable * Data
        + /LatchEnable * DLatch;
```

## Creating a Hazard-Free Latch

A Karnaugh map reveals that it is possible to lose data when the LatchEnable goes from 1 to 0 while asserting Data.

### To create a latch that protects against losing data

- 1 Add a Cover Term to the DLATCH equation by encapsulating the combinatorial latch function in a DSL procedure.
- 2 Add the NO\_REDUCE option to the output to prevent PLSyn from reducing out the Cover Term.

The procedure to do this is as follows:

```
PROCEDURE DLatch(INPUT Data, LatchEnable;
  OUTPUT DLatchOut NO_REDUCE);
  DLatchOut = LatchEnable * Data
  + /LatchEnable * DLatchOut
  + Data * DLatchOut; "Cover Term
END DLatch;
```

## Specifying Reserve Capacity

There are two reasons to reserve resources in a device:

- Allow for expansion of logic.
- Simplify and speed up the fitting process. Simply put, it is easier to place and route a solution at 80% utilization than at 100% utilization. If design iteration speed is more important than density (for example, you're early in the design cycle), set the utilization factor to a lower value.

### To specify the amount of reserve capacity to leave available in a device

- 1 Use the MACH\_UTILIZATION property in your .pi file using the syntax

```
{MACH_UTILIZATION percent};
```

where % is the percentage of device resources to be used.  
The range of values is 0 to 100.

This affects the use of pins, PTERMs, and macrocells. PLSyn distributes the unused resources throughout the device.



# Targeting PAL Blocks

Although in MACH devices there is no timing advantage to placing signals in the same PAL block, doing so may make PCB layout easier by keeping related signals together.

You can specify which nodes, outputs and biputs you want placed together in the same PAL block. There are two ways to do this:

- Signal groups
- Device sections

## Using Signal Groups

Use this method if you don't care whether PLSyn fits one signal group into the *same* PAL block as another signal group.

### To group signals into a PAL block.

In your .pi file, use the GROUP property inside of a DEVICE construct.

### Example

#### SOURCE FILE

```
INPUT i[8];
OUTPUT ogroup1[8];
OUTPUT ogroup2[8];
ogroup1 = i;
ogroup2 = i;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH110-15JC';

GROUP
  ogroup1;      "all ogroup1 signals will
                "go into the
END GROUP;      "same PAL block
```

```

GROUP
    ogroup2;      "all ogroup2 signals
                  "may or may not
END GROUP;      "also go into ogroup1's
                  "PAL block
END DEVICE;
```

## Using Device Sections

Use this method if you want PLSyn to:

- restrict the set of signals in each device section to a *different* PAL block, and/or
- target the signals in a device section to a specific PAL block.

### To fit all signals in a device section into one PAL block.

In your .pi file, use the SECTION property inside of a DEVICE construct.

### To target the signals in a section to a specific PAL block

Use the TARGET property in your .pi file of the form

```

TARGET 'pal_block_name';
```

**Table 8-2** lists the names of the PAL blocks for the MACH family.

**Table 8-2** MACH PAL Block

Architecture	PAL Block Name
MACH110	A..B
MACH120	A..D
MACH130	A..D
MACH210	A..D
MACH211	A..D
MACH211sp	A..D
MACH215	A..D
MACH220/221	A..H
MACH230/231sp	A..H
MACH435	A..H
MACH465	A..P
MACH5xx	A..D

#### Names

**Note** MACH 5 devices have 4 PAL blocks (A-D) for each segment. The number of segments varies with specific devices in the M5 family.

### Example

#### SOURCE FILE

```
INPUT i[8];
OUTPUT ogroup1[8];
OUTPUT ogroup2[8];
ogroup1 = i;
ogroup2 = i;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH110-15JC';
  SECTION
    TARGET 'A';
    ogroup1;    "all ogroup1 signals
                "will go into PAL
                "block A
  END SECTION;
  SECTION
    TARGET 'B';
    ogroup2;    "all ogroup2 signals
                "will go into PAL
                "block B
  END SECTION;
END DEVICE;
```

# Constraining the Size of Combinatorial Nodes

You can constrain the size of combinatorial nodes PLSyn collapses during the optimization process, thereby affecting how the logic fits into MACH devices.

## To constrain the size of combinatorial nodes

Use the MAX\_PTERMS property in your .pi file using the syntax:

```
{MAX_PTERMS p};
```

where *p* is the maximum number of PTERMs to which the optimizer can collapse.

The PLSyn optimizer collapses combinatorial nodes up to a size specified by MAX\_PTERMS.

## Making Adjustments

If the value is low, then PLSyn typically implements the design as a larger number of smaller equations. This makes placement easier because smaller functions do not place demand on the PTERM allocation mechanism. However, more smaller functions can require more routing resources and can require more overall macrocell logic.

At the other end, fewer larger functions can ease the routing requirements, but be harder to place because the demand for PTERMs can cause conflicts when attempting to place functions together in a PAL block.

**Table 8-3** shows the minimum and maximum number of PTERMs along with a suggested value. For optimal fitting, you should try a number of values to determine the best value for a given design.

**Table 8-3** *Minimum and Maximum Number of PTERMS*

Family	Minimum Number of PTERMs per Output	Maximum Number of PTERMs per Output	Suggested Number of PTERMs per Output
MACH 1XX	4	12	8
MACH 2XX	4	16	8 or 12*
MACH 435	5	20	10 or 15

\*, Will vary with the design.

**Using higher MAX\_PTERMS generally results in**

- More node collapsing
- Larger functions
- Faster implementation
- May increase routing requirements

**Using lower MAX\_PTERMS generally results in**

- Less node collapsing
- Smaller functions
- Slower implementation
- May increase routing requirements

**To see the exact effect of changing the optimizing parameters**

- 1 Open the .doc file after optimizing and check the number of nodes. The number of nodes generally goes down as the MAX\_PTERMS parameter goes up.

**Note** You can use any optimization property (for example `MAX_PTERMS` or `MAX_SYMBOLS`) in `GROUPs`, `SECTIONs`, or with any individual signals.

For exact usage, see the *PIL Reference* in PLSyn online help.

## Optimizing MACH 4xx Devices Using `MAX_XOR_PTERMS`

In addition to the `MAX_PTERMS` property, you can adjust `MAX_XOR_PTERMS` for MACH 4xx devices. The `MAX_XOR_PTERMS` value is typically one less than the `MAX_PTERMS` value to allow for the single `PTERM` which is placed on the `XOR` row.

**Note** The MACH 1xx/2xx devices do not support `XOR`.

The following table shows suggested values for `MAX_XOR_PTERMS` and `MAX_PTERMS`.

	Larger ↔ Smaller			
Property	Faster	↔	Slower	
<code>MAX_XOR_PTERMS</code>	19	14	9	4
<code>MAX_PTERMS</code>	20	15	10	5

## A Few Considerations



- Either High or Low `MAX_PTERMS` can cause greater routing demand.
- Lower `MAX_PTERMS` can produce more internal nodes which PLSyn must route to the equations where they are used.
- Higher `MAX_PTERMS` allow PLSyn to collapse a node into multiple equations. This results in placing the signals needed to generate the node in multiple places. Furthermore, large equations can require PLSyn to route a large number of signals into the block where the equation is placed, producing a locally high routing demand.

For more information, refer to the *PIL Reference* in PLSyn online help.

## Other Optimizing Parameters

Other optimizing parameters suitable for MACH devices are listed below with suggested values.

### For the MACH 4xx

MAX_PTERMS	10
MAX_XOR_PTERMS	9
MACH_UTILIZATION	100
MAX_SYMBOLS	20
POLARITY_CONTROL	TRUE
XOR_POLARITY_CONTROL	TRUE

### For MACH 1xx/2xx devices

MAX_PTERMS	8
MACH_UTILIZATION	100
MAX_SYMBOLS	20
POLARITY_CONTROL	TRUE

**Note** *Within this range of suitable parameters there are trade-offs on equation size and speed.*

# Understanding Pin Naming and Numbering

In the MACH family, you can assign signals to pins and internal nodes:

<b>Physical pins</b>	input pins
	input-clock pins
	input/output pins
<b>Internal nodes</b>	shadow node
	buried node
	unary node

For more description of the internal node types, see *Accessing Internal Points in a PLD Device* on page 7-2. However, note that MACH devices reference internal nodes differently than other kinds of PLDs.

## To reference MACH device pins

- 1 Use the following notations:

MACROCELL_X##	physical pins
	shadow node
	buried node
IN_REG_X##	unary node

where X is the PAL block ID and ## is the macrocell number.

The macrocells and input registers are sequentially numbered through the device in the same order as the macrocell names (A00 - H15). Depending on the device and PAL block, these numbers are sequenced in either the same order as the neighboring physical pin numbers, or reverse order.

For a list of device-specific pin names and numbers, see Appendix C, *AMD MACH Device Tables*.



## Using the MACROCELL\_X## notation

**For physical pins** A physical pin (input, input-clock, or input/output) connects to either an input or biput macrocell. Reference physical pins by the block ID and pin number in the package diagram found in your data book.

**For buried nodes** A buried node is a macrocell within the device which cannot be connected to an I/O pin. In the MACH 2xx parts, these are the odd numbered macrocells.

Using a shadow node rather than a biput pin allows the physical pin and its pin feedback path to be used as an input.

**For shadow nodes** Shadow nodes are biput macrocells that, when disconnected from an I/O pin, are treated as a buried node with the pin as an input. In the MACH 1xx and 2xx parts, all I/O pins have corresponding shadow nodes.

For more information on using unary nodes in MACH devices, see [MACH 2xx, 4xx: Using Input Registers on page 8-23](#).

## Using the IN\_REG\_X## notation

The IN\_REG\_X## notation is reserved for unary nodes. Most often they are input registers. In the MACH 215 and MACH 4xx, input registers are available on all I/O pins.

# Achieving Satisfactory Pinouts

The general approach is to first fit the design unconstrained to prove that there is a solution; then mold that solution into a pinout that meets the board layout requirements.

## To achieve acceptable pinouts

- 1 Generate an unconstrained solution: run the PLSyn fitter and fuse map generator to produce an .npi file.
- 2 Copy the .npi file to the .pi file.
- 3 In the .pi file, strip the pin assignments.
- 4 Take out the NO\_CONNECT information.
- 5 Use the GROUP statement to control which sets of signals you want to fit together in localized or sequential pins.

**Note** *Leave the INPUT signals for later. Not every function must be in a group.*

The .pi file will look similar to this:

```

DEVICE
  TARGET 'PART_NUMBER AMD MACH130-15JC';

  INPUT B20M;
  INPUT NACKI0;
  INPUT NACKI1;
  ...

  TXC;

  GROUP
    COL1;
    CRS1;
  END GROUP;

  GROUP
    COL2;
    CRS2;
  END GROUP;

```

It may help to sort the .pi file first to get signals with like names together, since they often are grouped together.

See [Using Signal Groups on page 8-10](#) for more information.

```
GROUP
    NACKO0 ;
    NACKO1 ;
    NACKO2 ;
END GROUP ;
```

```
...
END DEVICE ;
```

- 6** Run the PLSyn fitter on the grouped .pi file to see which groups go best with other groups (for example, similar signal, OE, and RESET requirements).
- 7** If this fails to fit, check the log file to find the group which violates the constraints of a PAL block, and either:
  - dissolve the group, or
  - divide it into two groups.
- 8** When PLSyn successfully completes the fit, copy the new .npi file to a .pi file and make that your current .pi file.
- 9** If it is necessary to swap the contents between two PAL blocks, then target the PAL blocks.
  - a** Refer to a pinout table for the device and determine where the PAL block divisions occur.
  - b** Divide the current .pi file into PAL block groups using the SECTION construct with TARGET statements. (Save the inputs for later.)
  - c** Strip the pin numbers and reassign the groups as required.

See [Table 8-2 on page 8-11](#) for the names of the PAL blocks. See [Using Device Sections on page 8-11](#) for information on how to use the SECTION and TARGET properties.

Reminder: If you have outputs in different PAL blocks that must be adjacent, you can have them either:

- span the boundary of adjacent PAL blocks, or
- wrap-around between the last PAL block and the first.

The .pi file will look like this:

```

DEVICE
  TARGET 'PART_NUMBER AMD MACH130-15JC';

  INPUT B20M;
  INPUT NACKI0;
  INPUT NACKI1;
  ...
  TXC;

SECTION
  TARGET 'A';
  NACKO0;
  NACKO1;
  NACKO2;
END SECTION;

SECTION
  TARGET 'B';
  COL1;
  CRS1;
  COL2;
  CRS2;
END SECTION;

...
END DEVICE;

```

- d** Run the PLSyn fitter.
- e** If the fit fails, consult the log file and make adjustments as required. One thing you can try is to rotate the PAL block assignments (A to B, B to C, ... H to A).
- f** Repeat steps **d** and **e** until the PAL block assignments are satisfactory.

**10** Copy the .npi file to a new .pi file.

The intent here is to handle inputs last. Since inputs have only routing constraints, fitting them last leaves more possibilities for the programmable logic functions which have routing, PTERM allocation, and control function constraints.

- 11** Find suitable pin assignments within the PAL blocks.
  - a** Add comments to the `.pi` file to show where the PAL block's limits are.
  - b** Separate all of the inputs and strip off their pin numbers.

Be sure, however, to leave room for sequential assignments of input groups. You might find it helpful to leave binputs available adjacent to the dedicated input pins so that input groups can fit across dedicated inputs and onto the binputs. Remember that clock signals must go on clock/input pins.
  - c** Strip the pin numbers off of one PAL block.
  - d** Pick one group of signals and assign it the desired pin assignment.
  - e** Run the PLSyn fitter.
  - f** If it fails, be sure to check the log file. Try the following:
    - Shift the signals by one pin.
    - Try walking an unassigned pin through the group.
    - Try assigning the other pins, and see where the group ends up.
  - g** When you finish one PAL block, repeat steps c-f for the next PAL block.

# MACH 2xx, 4xx: Using Input Registers

The MACH 2xx and 4xx devices can register signals between the I/O pin and the switch matrix. The MACH 215 and MACH 4xx have a dedicated register for each I/O pin. The other MACH 2xx devices use the buried macrocell adjacent to the pin to perform the registration.

The PLSyn fitter attempts to use these input registers as often as possible because their use saves both routing resources and propagation delay.

## Understanding Input Register Pin Names

For a list of pin names, see [Appendix C, AMD MACH Device Tables](#).

The MACH 4xx and MACH 215 have dedicated hardware for the input register function. These are called unary pins because they support a function of exactly one signal. The naming convention for these pins is `IN_REG_X##` where `X` is the PAL block ID and `##` is the macrocell number.

To register the pin signal in the MACH 2x0 devices, the signal is routed through the adjacent buried register. This effectively takes one buried register macrocell and reduces the number of nodes which the part can fit internally.

The MACH 2x0 devices register I/O pin signals on nodes names `MACROCELL_X##` where `X` is the block ID and `##` is the macrocell number.

**Note** *Assigning a signal to that pin is not enough to force use of the input register mode. The assignment is ambiguous and PLSyn interprets it as an internal node assignment.*

## MACH 2xx and 4xx Compared

The MACH 4xx devices have separate input register resources. Because this simplifies the fitting of unary functions, these assignments are simple and direct. You can assign manually any unary function to `IN_REG_X##`, or let the PLSyn fitter do this automatically. The MACH 4xx is also able to automatically use these resources to register the feedback of an output function.

The MACH 215 does have separate hardware for input registers, but because of its general architecture, the PLSyn fitter handles it as it would MACH 1xx/2xx devices, sharing the same restrictions.

## Input Registration

With conventional routing, the input goes into the switch matrix and is brought to the PAL block array, then fit as any other node.

The input register configuration has several advantages over conventional routing:

- It saves one PAL block input and four PTERMs needed to generate the function in the standard configuration.
- It also saves propagation time of one pass through the array for the signal generated.

In PLSyn, there is *no* INPUT CLOCKED\_BY construct, so the fitter look for nodes that have a single signal as the D equation. These are *unary* functions because they are functions of one signal. Whenever possible, the PLSyn fitter automatically fits unaries in input registers.

If you are using the MACH 2xx, you might need to detect, force, or prevent use of input registers for any given signal.

### Example

The following source generates the unary-compatible function u.

```
INPUT i, ui, clk;
NODE u CLOCKED_BY clk;
OUTPUT o;
u = ui;
o = u * i;
```

## Finding Signals Fit as Unary

To detect signals which have been fit as unaries, you must inspect the Signal Directory section of report file. Check the number of clusters used for each function. A function with zero clusters has been fit as a unary.

### Example

Continuing with the example shown in the previous section, the function `u` is fit as a unary as shown in this excerpt from its report file:

Signal #	Name	Source Type	PalBlk Clusters	Pal Block Inputs
0	i	Input		A12
1	ui	Input		
2	clk	Input		
3	u	DFF Hidden	A 0	A18
4	o	Cmb Internal	A 1	



## Forcing a Function to be Fit as Unary

To force a function to be fit as unary, the function must meet all of the following conditions:

- Must be a NODE, not an OUTPUT
- Must have a single signal data equation
- Must be a DFF, TFF, or DLATCH equation
- Must conform to the reset and preset equation of the PAL block

### To force the function into the input register

- 1 In your .pi file, place the input signal on an I/O pin.
- 2 Place the function on the adjacent buried macrocell.

### Example

Continuing with the example shown in the previous section, the following PIL statements use the input register configuration to register the signal ui to form the function u which goes into the switch matrix:

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH210-12JC';
  INPUT ui :4;
  u :MACROCELL_A05;
END DEVICE;
```

## Preventing a Function from Being Fit as Unary

### To prevent a function from being fit as a unary

- 1 Fix either the input or the function signal to a pin.

The pin can be the same pin which PLSyn previously fit as a unary. Given that one but not both signals is fixed is sufficient to prevent the unary configuration.

# Preserving Pinouts when Refitting

This section describes how to refit your design by:

- Setting up your design with the intention of refitting before you ever start the physical implementation process.
- Using one of the following two methods to *fix* the pinouts when refitting your design:
  - Create a two-level `.pi` file from the `.npi` file by adding PAL block `SECTION`s within a `DEVICE` construct (page [8-28](#)).
  - Float nodes (page [8-34](#)).

## Plan for Refitting

Before you start the first fitting, follow these design guidelines to ensure the greatest success when refitting:

- Target a device using the `DEVICE` construct in the `.pi` file.
- Keep utilization low; below 70%.
- Keep pinout options open as long as possible.
- Don't release board layout after the first successful fit, since the design might change and changes may not refit the way the original design was fit.
- As much as possible, try to work with what the PLSyn fitter prefers to do, especially in terms of partitioning into PAL blocks, rather than forcing a specific pinout.

Before you apply this method, you must run a fit (which automatically generates a `.npi` file) and generate a fuse map.

## Method 1: Creating a Two-Level `.pi` File

This method preserves the PAL block partitioning of the programmable logic while giving the PLSyn fitter the freedom to move buried logic within a PAL block, but not from one PAL block to another. Outputs and inputs remain fixed to specific pins of the device.

### To create the two level `.pi` file

- 1** After completing the first fitting, copy: `design_name.npi` to `design_name.pi`
- 2** In the `.pi` file, move all inputs to the top (or bottom) of the file. Do not change or delete any of the pin assignments.
- 3** Set up two, four, or eight SECTIONS, depending on the device, within the DEVICE construct.
- 4** Segregate all outputs and nodes into sections according to which PAL block they were originally fit into.
- 5** Preserve pin assignments for different types of devices as follows:
  - For MACH 2xx parts, check the `.rpt` file (Signal Directory section) for nodes fit using zero clusters. Preserve these pin assignments; PLSyn fits these nodes with input registers.
  - For MACH 435, preserve pin assignments to `IN_REG_X##`; these are input register assignments.
- 6** Drop the pin assignment on nodes which have been fit on I/O pins and are not required on another device.

The `.doc` file lists all nodes, and also provides a wire list which shows which nodes are wired to another device.
- 7** Except as indicated in steps 4 and 5, drop all pin assignments for buried logic, and preserve all pin assignments for I/O pins.
- 8** Rerun the PLSyn fitter. If the design fits successfully, you have a repeatable solution.

## Example

Suppose you have fit a design into a MACH 230. The report file contains the following lines in the Signal Directory section showing that `df_reg[1]` and `df_reg[2]` are fit on input registers:

Signal # Name	Source Type	PalBlk Clusters	Pal Block Inputs
68 df_reg[2]	DFF Hidden	A 0	A13
69 df_reg[1]	DFF Hidden	A 0	A12

Notice that, for routing purposes, PLSyn placed node `df_reg[0]` on a pin since the signal is not needed outside of the device.

The `.npi` file looks like this:

```
----- .npi file -----
DEVICE
TARGET 'PART_NUMBER AMD MACH230-15JC';

dout[19]:3;
dout[6]:4;
dout[5]:5;
dout[2]:6;
INPUT dflags[1]:7;
INPUT dflags[2]:8;
dout[1]:9;
INPUT dflags[0]:12;
INPUT din[0]:13;
INPUT din[10]:14;
INPUT din[2]:15;
frame:16;
INPUT delay[4]:17;
INPUT rst:18;
INPUT new_con:19;
INPUT clk:20;
INPUT din[18]:23;
dout[9]:24;
dout[8]:25;
dout[4]:26;
dout[3]:27;
INPUT din[4]:29;
INPUT din[17]:33;
INPUT tx_en:34;
```

```
INPUT din[15]:35;
INPUT delay[0]:36;
INPUT din[16]:37;
INPUT din[11]:38;
INPUT ef0:39;
INPUT phase:40;
INPUT delay[5]:41;
dout[18]:45;
INPUT delay[2]:46;
INPUT din[9]:47;
INPUT delay[3]:48;
INPUT din[5]:49;
INPUT din[1]:50;
INPUT din[14]:51;
INPUT din[19]:52;
dout[17]:54;
INPUT delay[1]:55;
dout[14]:56;
dout[11]:57;
dout[7]:58;
INPUT din[3]:65;
dout[16]:66;
dout[15]:67;
INPUT din[12]:68;
INPUT din[8]:69;
dout[12]:70;
INPUT din[7]:71;
fifo_ren:72;
df_reg[0]:75;
INPUT ef1:76;
INPUT din[6]:77;
dout[13]:78;
dout[10]:79;
dout[0]:80;
INPUT din[13]:83;
df_reg[1]:MACROCELL_A13;
df_reg[2]:MACROCELL_A15;
s0:MACROCELL_B00;
s2:MACROCELL_B02;
dcnt[0]:MACROCELL_B04;
s1:MACROCELL_B10;
dval:MACROCELL_B12;
dcnt[2]:MACROCELL_D05;
dcnt[4]:MACROCELL_D08;
prep_done:MACROCELL_D10;
```

```

dcnt[5]:MACROCELL_D14;
dcnt[3]:MACROCELL_E11;
dcnt[1]:MACROCELL_G10;
dv_lv10:MACROCELL_H05;
dv_lv11:MACROCELL_H12;
END DEVICE;

```

The new .pi file (a modified version of the .npi file) looks like this:

```

----- .pi file -----
DEVICE
    TARGET 'PART_NUMBER AMD MACH230-15JC';

    INPUT dflags[1]:7;
    INPUT dflags[2]:8;
    INPUT dflags[0]:12;
    INPUT din[0]:13;
    INPUT din[10]:14;
    INPUT din[2]:15;
    INPUT delay[4]:17;
    INPUT rst:18;
    INPUT new_con:19;
    INPUT clk:20;
    INPUT din[18]:23;
    INPUT din[4]:29;
    INPUT din[17]:33;
    INPUT tx_en:34;
    INPUT din[15]:35;
    INPUT delay[0]:36;
    INPUT din[16]:37;
    INPUT din[11]:38;
    INPUT ef0:39;
    INPUT phase:40;
    INPUT delay[5]:41;
    INPUT delay[2]:46;
    INPUT din[9]:47;
    INPUT delay[3]:48;
    INPUT din[5]:49;
    INPUT din[1]:50;
    INPUT din[14]:51;
    INPUT din[19]:52;
    INPUT delay[1]:55;
    INPUT din[3]:65;
    INPUT din[12]:68;
    INPUT din[8]:69;

```

```
INPUT din[7]:71;
INPUT ef1:76;
INPUT din[6]:77;
INPUT din[13]:83;

SECTION
    dout[19]:3;
    dout[6]:4;
    dout[5]:5;
    dout[2]:6;
    dout[1]:9;
    df_reg[1]:MACROCELL_A13; "Part of input
                             "register assignment
    df_reg[2]:MACROCELL_A15; "Part of input
                             "register assignment
END SECTION;

SECTION
    frame:16;
    s0;          ":MACROCELL_B00;
    s2;          ":MACROCELL_B02;
    dcnt [0];    ":MACROCELL_B04;
    s1;          ":MACROCELL_B10;
    dval;        ":MACROCELL_B12;
END SECTION;

SECTION
    dout[9]:24;
    dout[8]:25;
    dout[4]:26;
    dout[3]:27;
END SECTION;

SECTION
    dcnt[2];     ":MACROCELL_D05;
    dcnt[4];     ":MACROCELL_D08;
    prep_done;   ":MACROCELL_D10;
    dcnt[5];     ":MACROCELL_D14;
END SECTION;

SECTION
    dout[18]:45;
    dcnt[3];     ":MACROCELL_E11;
END SECTION;

SECTION
    dout[17]:54;
    dout[14]:56;
    dout[11]:57;
```

```
    dout[7]:58;
END SECTION;

SECTION
    dout[16]:66;
    dout[15]:67;
    dout[12]:70;
    fifo_ren:72;
    dcnt[1];    ":MACROCELL_G10;
END SECTION;

SECTION
    df_reg[0]; ":75; This is a node
                "on a pin
    dout[13]:78;
    dout[10]:79;
    dout[0]:80;
    dv_lv10;    ":MACROCELL_H05;
    dv_lv11;    ":MACROCELL_H12;
END SECTION;
END DEVICE;
```

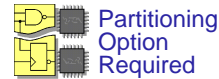




# When Fitting into One Device Fails

When your design fails to fit into a single MACH device, there are two ways to approach debugging the problem:

- Force the design into one device using the default signal reference in the .pi file.
- Partition the design between two devices and analyze the result.



## Using the “Default” Signal Reference

It would seem that setting the Max Devices constraint to one should force PLSyn to fit the design into one device. However, this actually tells the PLSyn fitter to *quit after one device is filled*. This means that when there is a failure, there is very little diagnostic information available in the log and report files.

For more information on Max Devices and setting constraints, see *Constraining Devices* on page 5-18.

### To force the entire design into one part and obtain a report file

- 1 Use the default signal reference in a DEVICE statement in your .pi file. The default reference is the same as naming all signals in the design not mentioned elsewhere in the .pi file.

### Example

The following PIL fits the design into a single MACH 210 device.

```
DEVICE
  TARGET 'part_number AMD MACH210-15JC';
  default;
END DEVICE;
```

When you do this, the design might fit the first time. If it does not, look at the log and report files for valuable information about why PLSyn could not fit the circuit.



### What you can find out in the log file

The log file (`.log`) can tell you things like:

- Your design exceeds device limits such as RESET/PRESET constraints. You might need to adjust the design to the limits of the device, or use another part or parts with greater resources.
- The PLSyn fitter did not find a suitable partition. You should check the report file for details.



### What you can find out in the report file

The report file (`.rpt`) can tell you things like:

- The best partition PLSyn could produce and why it is not valid for the device.
- The PLSyn partitioner succeeded assigning functions to PAL blocks, but the PLSyn fitter failed placing and routing the design.

Based on the progress and problems written to the report file, you need to use the `.pi` file to:

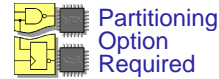
- adjust the design and/or
- adjust the implementation specification.

In general, look for resources which are in high utilization. If macrocells are in high demand, more node collapsing can relieve the problem. If PTERMs are in high demand, you might try extracting some common factors into a common node.

## Using a Second Device

Another approach to a difficult fitting problem is to allow the design to overflow into a second device, and then see which functions the PLSyn fitter leaves out of the first device.

If you generate fusemaps for the two-device solution, you can use the `.npi` file to work one or two functions back into the first device.



### To work functions back into the target device

- 1 Copy the `.npi` file to a new `.pi` file.
- 2 Move the functions assigned to the second device and include them in the `DEVICE` statement for the first device but without any pin assignments.

If that does not work, try the following:

- Node collapsing.
- Factoring to allow room for the left out functions.

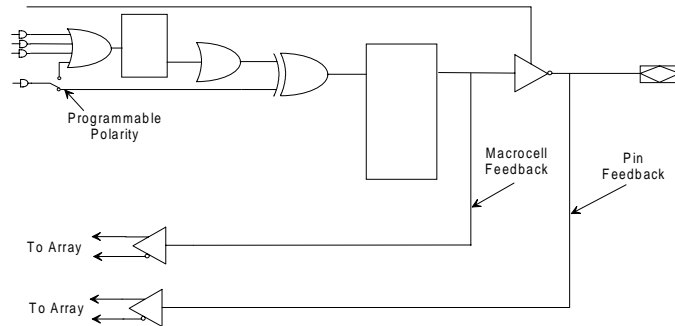
Considering a larger device.



# Accessing the MACH Internal Feedback Path

In MACH devices, outputs without an output enable can feed back into the device through two paths:

- Directly from the pin. This is called *pin feedback* and may or may not bond-out to a physical pin.
- Directly from the macrocell. This is called *macrocell feedback*.



These paths are functionally equivalent, but the pin feedback can be slower than the macrocell feedback. By default, PLSyn routes signals using the pin-feedback path.

## To use the macrocell-feedback path for one signal

- 1 In your `.pi` file, attach the `FORCE_INTERNAL_FB` property to the appropriate signal.

## To use the macrocell-feedback on all signals in the device

- 1 Include the `FORCE_INTERNAL_FB` property in the `DEVICE` specification.

## Example

This example shows the PIL that forces signal out1 to follow the macrocell (internal) feedback path instead of the pin feedback path.

### SOURCE FILE

```
INPUT a, b, c;
OUTPUT out1 CLOCKD_BY clk;
OUTPUT out2;

out1 = a * b;
out2 = out1 * c;
```

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH465-15KC';
  out1 {FORCE_INTERNAL_FB}; "Use the
    "macrocell feedback
  DEFAULT;
END DEVICE;
```

## Example

This example shows the PIL that specifies that all signals in the device should use the macrocell (internal) feedback path instead of the pin feedback path.

### SOURCE FILE

```
INPUT a, b, c;
OUTPUT out1 CLOCKD_BY clk;
OUTPUT out2;

out1 = a * b;;
out2 = out1 * c;
```

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH465-15KC';
  {FORCE_INTERNAL_FB}; "Use the macrocell
    "feedback for all signals in the
    "device
  DEFAULT;
END DEVICE;
```

## MACH 215, 4xx: Fitting Asynchronous Functions

Both the MACH 215 and MACH 4xx devices support asynchronous functions, but they have different capabilities. Some functions, or groups of functions, suitable for the MACH 215 will not fit in the MACH 4xx.

### PTERM Clock and RESET and PRESET

When the clock expression is a product term (PTERM), a device requires both RESET *and* PRESET in its equation. An equation such as this requires the device to run in asynchronous mode.

However, a MACH 4xx device can have either asynchronous RESET *or* PRESET, but not both. This means that functions of this type can only fit in the MACH 215 (which allows both asynchronous PRESET *and* RESET) using the following construct:

```
OUTPUT o1 CLOCKED_BY (clk1 * clk2) RESET_BY reset
PRESET_BY preset;
```

### More Than One RESET/PRESET Pair per PAL Block

In the MACH 4xx, any function which has both a RESET and PRESET expression must use the block resources for reset and preset. If a design has more than eight different pairs of RESET and PRESET equations, it cannot fit in one MACH 4xx, but may fit in one MACH 215.

The following set of functions can only fit in a MACH 215:

```

OUTPUT o1 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_1;
OUTPUT o2 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_2;
OUTPUT o3 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_3;
OUTPUT o4 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_4;
OUTPUT o5 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_5;
OUTPUT o6 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_6;
OUTPUT o7 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_7;
OUTPUT o8 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_8;
OUTPUT o9 CLOCKED_BY clk RESET_BY reset PRESET_BY pre_9;

```

## MACH 4xx: Using XOR T-Equations

If you are fitting an XOR T-equation that is greater than 20 PTERMs, you need to insert a node between the equation and the T-register. This rule also applies to a function that requires both a TFF register and an XOR equation because the PLSyn compiler expands the XOR equation into a T-equation which can be greater than 20 PTERMs.

### Example

This design will not fit due to equation o2.T expanding to 24 PTERMs.

```

INPUT clk;
INPUT i1, i2, i3, i4, i5;
INPUT j1, j2, j3, j4, j5;
T_FLOP OUTPUT o1 CLOCKED_BY clk;
T_FLOP OUTPUT o2 CLOCKED_BY clk;

o1.T = i1 (+) (i2 + j2 + j3 + j4 * j5);
o2.T = (i1*j1) (+) (i2*j2 + i3*j3 + i4*j4 + i5*j5);

```

If rewritten with a node for the T-equation, it fits because the combinatorial equation does not need to be expanded.

```

INPUT clk;
INPUT i1, i2, i3, i4, i5;
INPUT j1, j2, j3, j4, j5;
T_FLOP OUTPUT o1 CLOCKED_BY clk;
T_FLOP OUTPUT o2 CLOCKED_BY clk;
NODE n;

o1.T = i1 (+) (i2 + j2 + j3 + j4 * j5);

```



```
n = (i1*j1) (+) (i2*j2 + i3*j3 + i4*j4 + i5*j5);  
o2.T = n;
```

## MACH 4xx: Controlling Asynchronous Mode

You can manually control the implementation of functions with asynchronous clocking using the asynchronous macrocell features of the MACH 4xx.

Because asynchronous fitting can be a resource and timing cost, the PLSyn fitter opts for synchronous mode wherever possible. However, if by doing so PAL blocks are underutilized or the solution requires extra devices, PLSyn opts for asynchronous mode.

When using asynchronous mode, the PLSyn fitter selects the block reset and preset, and the block clock signals so as to minimize the number of macrocells that are fit.

Since the macrocell local-reset PTERM and the shared PAL block reset and preset PTERMs are generated in the PAL block array, there is no timing penalty for using the asynchronous mode reset. However, you might need more control over selecting the functions that use asynchronous clocking. The difference in timing between the pin clock and an array PTERM-generated clock signal can be of overriding importance.

### To control which functions are clocked asynchronously

- 1 Group and select the signals that you want placed on the clock pins.

# MACH 4xx: Controlling T-Flop Synthesis

For some equations, the T-flop might have a smaller equation, but slightly greater delay. For speed-sensitive circuits, you can use D-flops exclusively instead because the XOR in the MACH 4xx provides for relatively efficient implementation of T-equations using the D register.

## Normal Operation

Unless otherwise directed, PLSyn fits the smallest equation of D, T, or XOR, or their complements.

## DFF-Only Fitting

### To use DFF equations only

- 1 Design the circuit in terms of DFF equations. If you do not reference T\_FLOP or other register types, PLSyn will generate DFF equations by default.
- 2 To restrict the design to fitting only DFF equations, include the statement:

```
{ FF_SYNTH OFF }
```

in the .pi file.

Depending on where you place the statement, this option can apply to specific signals, or to the entire device or design.

## Using the T-Equation

If a given function is most easily expressed using an equation for toggle operation, then the D equation is the XOR of that equation and the register output.

If (T) defines the toggle equation of function F, then the direct TFF expression of that function in DSL is:

```
T_FLOP OUTPUT F CLOCKED_BY clk ... ;  
F = (T) ;
```

while the DFF equivalent function is:

```
OUTPUT F CLOCKED_BY clk ... ;  
F = (T) (+) F ;
```

## MACH 4xx: Controlling Power-On Reset

The MACH 4xx has a built-in power-on reset feature that sets all registers to a known state when power is applied to the part. This section discusses how you can determine the state of the registers, and the steps you can take to manage the power-on feature.

### What Is a Logical Reset?

DSL defines the term *reset* in a device-independent way. To *reset* a signal means to put the signal in the unasserted state. A HIGH\_TRUE signal goes to the low-voltage state when it is reset. If the signal is a LOW\_TRUE sense, then a reset causes the signal to go to the high voltage state. In both cases, the signal is in its unasserted condition. This is a *logical* reset.

## The Nominal Case

Most applications of the MACH 4xx perform a *logical* reset on power-up. Registered signals go to the unasserted state.

## Exception Cases

For each signal that violates the power-on logical reset, PLSyn flags the entry in the Signal Directory section of the report file with the string RS\_SWAP. These signals receive a logical preset at power-on.

A violation can be caused by one of two things:

- Macrocells in asynchronous mode that have a preset equation perform a power-on logical preset.
- A function performs a power-on logical preset if it is fit on a macrocell in a PAL block where its reset and preset are *out-of-phase* with the majority of functions in the PAL block. Out-of-phase means that a function's reset and preset equations are identical to the PAL block preset and reset equations, respectively.

PLSyn flags functions which are fit using an asynchronous macrocell with the string ASYNC in the Signal Directory section of the report file.

### To prevent the out-of-phase condition

- 1 Manually partition your design.

This allows PLSyn to fit a function with a preset equation fit in an asynchronous macrocell into a synchronous macrocell if the function is not inherently asynchronous (that is, if it does not have a clock which is a product of multiple signals).

## MACH 230 and 435: Possible Pin Incompatibility Between

In rare cases, designs that fit in a MACH 230 are not pin-compatible with the MACH 435. This only happens when you are using both registers and latches in the same PAL block using pins 20 and 22, or pins 62 and 65 for the clock and latch enable signals.

This is due to the change in latch implementation between the MACH 1 and 2 families and the MACH 3 and 4 families. In the MACH 1 and 2 case, latches are *transparent low* and *latched high*. In the MACH 435, this sense is reversed to provide the more common functionality of *transparent high*, *latched low*.

This is seldom a problem in the MACH 435 since it can select clock polarity. Not all combinations of clock polarities for all clock pins are available within a single PAL block. This means that a problem can arise when porting a design with clocks and registers in the same block using clock pins from the same clock pair.

The clock pins are paired internally as CLK0 (pin 20) and CLK1 (pin 22), and as CLK2 (pin 62) and CLK3 (pin 65). Within each PAL block, the MACH 435 can select a clock polarity configuration (from each pair) that allows:

- both clocks TRUE,
- both clocks inverted, or
- both phases of one of the clock pair.

A given PAL block cannot select the true sense of one clock of the pair and the inverted sense of the other.

### Example

Consider a MACH 230 design with a register and latch in the same PAL block. Assume that the register is clocked by one clock pin of a pair and the latch is enabled by the other pin of the pair. Differences between the latches of the MACH 230 and the

MACH 435 mean that the MACH 435 must invert the latch enable to achieve the same functionality. This also means that the PAL block needs exactly the same clock polarity. It can't have true sense of one pair member and inverted sense of the other.

If one of the functions is a node, you can move it to another block. You can also force one of the clocks to be asynchronous (clocking by PTERM row) by using an internal node to produce the clock signal.



## MACH 445 and 465: Configuring for Zero-Hold Time

The MACH 445 and MACH 465 have an option to insert a delay between the I/O pins and the input registers in the device. This increases the setup time for the input registers and reduces the hold time for these registers to zero.

### To set the hold time on the input registers

- 1 Use the MACH\_ZERO\_HOLD\_INPUT property in the DEVICE construct of your .pi file.

### Example

#### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH465-15KC';
  {MACH_ZERO_HOLD_INPUT}; "Set all input
    registers to zero hold time

  DEFAULT;
END DEVICE;
```

Assigning the MACH\_ZERO\_HOLD\_INPUT property to a device configures all of the input registers for zero-hold time.

# MACH 445 and 465: Accessing Signature Bits

The MACH 445 and MACH 465 devices have a 32-bit field that you can use to hold user data. This field is called the Signature Bits, or USERCODE, field.

## To place data in the USERCODE field

- 1 In your .pi file, use the SIGNATURE property in the DEVICE section with the syntax:

```
{SIGNATURE data};
```

where *data* is a string of up to four characters (enclosed in single quotes) or a 32-bit signed integer.

## Example

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH465-15KC';
  {SIGNATURE 'test'};

  DEFAULT;
END DEVICE;
```

# MACH 1xx and 2xx: Driving or Floating Unused Outputs

For MACH 1xx and 2xx devices, you can drive or float I/O pins that do not have input or output signals attached, depending on whether the associated macrocell (shadow pin) is in use. If you place a hidden function in the macrocell, the pin goes to the high impedance or *floating* state. If you do not use the macrocell, the pin goes to a driven state with a constant value.

**Note** *This does not apply to the MACH 435 because these outputs have built-in pull-ups on the outputs, providing a default input when left unconnected.*

## Forcing Outputs Driven

### To force an output to be driven

- 1 Assign all outputs to pins so that the unused pins are known.
- 2 In your .pi file, place fuse statements with the syntax

```
INTACT fuse #
BLOWN fuse #
```

to modify the implementation.

After placing a node on the corresponding shadow pin, its signal is present on the pin. Otherwise, the pin asserts either high or low depending on how other unused internal resources are dispensed.

For a list of fuse assignment statements that assert the tri-state enable for unused pins in all MACH devices, see [Appendix C, AMD MACH Device Tables](#).



## Example

An example `.pi` file looks like this with outputs on pins 2-9 and intent to assert the OE on pins 14 to 21.

```
DEVICE TARGET 'PART_NUMBER AMD MACH110-15JC';
  o1:2; o2:3; o3:4; o4:5;
  o5:6; o6:7; o7:8; o8:9;

  "Assert OE on remaining outputs
  INTACT 6230 ; BLOWN 6231 ; " Pin 14:
  INTACT 6238 ; BLOWN 6239 ; " Pin 15:
  INTACT 6246 ; BLOWN 6247 ; " Pin 16:
  INTACT 6254 ; BLOWN 6255 ; " Pin 17:
  INTACT 6262 ; BLOWN 6263 ; " Pin 18:
  INTACT 6270 ; BLOWN 6271 ; " Pin 19:
  INTACT 6278 ; BLOWN 6279 ; " Pin 20:
  INTACT 6286 ; BLOWN 6287 ; " Pin 21:

END DEVICE;
```

Use the fuse statements in Appendix C, *AMD MACH Device Tables* to configure floating outputs. Just replace the BLOWN keyword with INTACT.

## Forcing Outputs Floating

### To force an output to float

- 1 Assign all outputs to pins so that the unused pins are known.
- 2 In your `.pi` file, place fuse statements with the syntax:

```
INTACT fuse #
INTACT fuse #
```

to modify the implementation.

When placing a node on the corresponding shadow pin, its signal is present on the pin. Otherwise, the pin asserts either high or low depending on how other unused internal resources are dispensed.

## Example

An example .pi file would look like this with outputs on pins 2-9 and intent to float the OE on pins 14 to 21.

```
DEVICE TARGET 'PART_NUMBER AMD MACH110-15JC';
  o1:2; o2:3; o3:4; o4:5;
  o5:6; o6:7; o7:8; o8:9;

  "Float OE on remaining outputs
  INTACT 6230 ; INTACT 6231 ; " Pin 14:
  INTACT 6238 ; INTACT 6239 ; " Pin 15:
  INTACT 6246 ; INTACT 6247 ; " Pin 16:
  INTACT 6254 ; INTACT 6255 ; " Pin 17:
  INTACT 6262 ; INTACT 6263 ; " Pin 18:
  INTACT 6270 ; INTACT 6271 ; " Pin 19:
  INTACT 6278 ; INTACT 6279 ; " Pin 20:
  INTACT 6286 ; INTACT 6287 ; " Pin 21:

END DEVICE;
```

# The MACH Report File

The PLSyn fitter writes a complete description of a fitted MACH device showing:

- Resource utilization
- All signal and routing information
- Full placement details including internal nodes

## Obtaining a Report File

PLSyn creates a report file when fitting for targeted MACH devices; not during automatic device selection and partitioning.

### To obtain a MACH report file on the first fitting

Either:

- Use the DEVICE and TARGET properties in your .pi file using the syntax:

```
DEVICE TARGET 'part_number amd part #';  
END DEVICE;
```

in the simplest case.

- Put empty DEVICE constructs into your .pi file. This forces a report file while allowing the program the complete freedom to partition the design.

### Example

The following PIL partitions a design into two MACH110's and produces a report for each device.

```
DEVICE TARGET 'PART_NUMBER AMD MACH110-15JC';  
END DEVICE ;  
  
DEVICE TARGET 'PART_NUMBER AMD MACH110-15JC';  
END DEVICE ;
```

## To obtain a MACH report file when using automatic partitioning

- 1 Run the PLSyn fitter the first time.
- 2 Copy the `.npi` file to a new `.pi` file.
- 3 Run the PLSyn fitter the second time using the new `.pi` file.

## Contents of the Report File

The report file contains device-specific fitting information about the internal resources of the MACH device. It shows exactly which macrocells and routing paths each signal uses.

The report file is not a replacement for the documentation (`.doc`) file. It does not list the equations for any given function, or give a simple pinout diagram. It gives in depth information that the documentation file does not provide.

The report file serves two purposes:

- When the design fits, it describes the specific placement and routing of the solution.
- If a design fails to fit, it provides information to help you understand why the fit attempt failed, how far the fitting proceeded, and what aspect of the fitting caused problems.

For MACH 1xx and 2xx devices, the report file format is slightly different from that for the MACH 3xx and 4xx devices. However, the report file has the same sections of information as summarized here and described in greater detail in the remainder of this chapter.

**Failure Disclaimers** If the design fails when partitioning or during place-and-route, PLSyn writes a disclaimer immediately following the heading. This alerts you that the design did not fit successfully and to the possibility that information might be missing or inconsistent.

**Summary Statistics** Summarizes the number of inputs, nodes, and outputs for your design by PAL block.

**Device Resource Utilization** Reports utilization statistics for the different resource types for each device and its PAL blocks.

**Partitioner Report** Shows how the design is partitioned into PAL blocks.

**Clock Assignments** Shows which pin clocks are used in which PAL blocks for MACH 3xx and 4xx devices.

**Signal Directory** Lists all inputs, outputs, and nodes on the device with specific assignment information for each signal.

**Resource Assignment Map** Shows device details (in physical order by pin and macrocell) and which signals use which resources.

## Failure Disclaimers

If the design fails when partitioning or during place-and-route, PLSyn writes a disclaimer immediately following the heading. This alerts you that the design did not fit successfully and to the possibility that information might be missing or inconsistent.

There are different disclaimers depending on where the fitting failed and the device type that PLSyn attempted to fit.

### If a MACH 435 or MACH 1xx or 2xx device design fails when partitioning

The following disclaimer is printed:

```
FAILURE-TO-PARTITION DISCLAIMER:
```

```
The following partitioner reports show the  
last failed attempts to partition the
```

design. Partitions which violate device limits are indicated. Also, if there are more Block partitions than blocks in the device, the partition will fail.

Because of different fitting algorithms for the two MACH families, MACH 1 and 2 family devices have a different fit disclaimer from MACH 3 and 4 family devices.

### **If a MACH 1xx or 2xx family device fails when fitting**

The following disclaimer is printed:

FAILURE-TO-FIT DISCLAIMER:

The following report represents the final status of a failed fit attempt. The report is accurate but incomplete. It indicates which signals were not placed or routed. In the 'SIGNAL DIRECTORY' signal lines preceded by '-' represent signals which could not be placed. Founts ending in '--' represent signals which could not be routed.

The Signal Directory information indicates how far the fitting process proceeded before PLSyn gave up. The un-routed and/or unplaced signals should point to the cause of the fitting problems. To achieve a fit, try modifying the design or manually direct the partitioner.



### If a MACH 4xx design fails when fitting

The following disclaimer is printed:

```
FAILURE-TO-FIT DISCLAIMER:

The following report represents the final
status of a failed fit attempt. The
'SUMMARY STATISTICS', 'RESOURCE
UTILIZATION', and 'CLOCK ASSIGNMENTS'
sections are accurate. The 'SIGNAL
DIRECTORY' is accurate except for pin and
macrocell designations. The RESOURCE
ASSIGNMENT MAP may have missing or
redundant signals and conflicting resource
assignments.
```



This disclaimer includes statistics showing which resource proved most troublesome during the fit operation. Use this information to decide how to modify your design before attempting another fit.

### Example

The relative conflict levels for each resource type listed here, indicate the reason for failure when fitting:

Pins	3
Input Regs	0
Macrocells	0
PTERMs	352
Feedbacks	0
Fanouts	0

The PTERMs value of 352 indicates that the PLSyn fitter had trouble assigning product terms.

## Summary Statistics

This section summarizes the number of inputs, nodes, and outputs for your design by PAL block and how many functions per block. Because the MACH 3 and 4 families have more ways to fit a function, the PLSyn fitter provides more statistics for these designs.

Sample: MACH 1xx and 2xx statistics

```
5 Inputs
0 Registered/Latched Inputs
11 Outputs
0 Tri-states
0 Nodes

Functions by block ( 8, 3, 0, 0 )
```

Sample: MACH 3xx and 4xx statistics

```
4 Inputs
0 Outputs
32 Tri-states
0 Nodes

Functions by block      ( 4, 4, 4, 4, 4, 4, 4, 4 )
D Register Macrocells   2
T Register Macrocells  26
D Latch Macrocells      2
Combinatorial Macrocells 2
D Input Registers       0
D Input Latches         0

Xor Equations           0
Asynchronous Equations  0
Single-PTERM Equations  32
Total PTERMs Required   32
```

The sum total of Outputs, Tri-states, and Nodes should equal the total Functions by block and the total of the Macrocells and Input Registers/Latch statistics. The numbers for Xor Equations on down are not mutually exclusive nor should they match the total number of functions.



## Device Resource Utilization

This section provides utilization statistics for the resource types of each device and its PAL blocks. The section is broken into two parts:

- Global resource utilization statistics.
- Resource statistics for each PAL block.

The statistics for the MACH 1 and 2 families are slightly different to those for the MACH 3 and 4 families. These examples show the global statistics and one PAL block statistic set for each device family.

### Sample: MACH 1xx and 2xx resource statistics

Resource	Available	Used	Remaining	%
Clocks:	2	1	1	50
Pins:	38	35	3	92
Input Lines:	88	72	16	81
I/O Macro:	32	16	16	50
Total Macro:	64	48	16	75
PTERMS:	256	48	64	75
PAL_BLOCK A				
Input Lines:	22	18	4	81
I/O Macro:	8	4	4	50
Total Macro:	16	12	4	75
PTERMS:	64	12	16	75

**Sample: MACH 3xx and 4xx resource statistics**

Resource	Available	Used	Remaining	%
Clocks:	4	1	3	25
Pins:	70	67	3	95
Input Regs:	64	0	64	0
Macrocells:	128	96	32	0
PTERMs:	640	314	326	49
Feedbacks:	192	125	67	65
Fanouts:	264	161	103	60
PAL_BLOCK A				
Blk Clocks:	4	1	3	25
I/O Pins:	8	8	0	100
Input Regs:	8	0	8	0
Macrocells:	16	12	4	75
PTERMs:	80	42	38	52
Feedbacks:	24	16	8	66
Fanouts:	33	18	15	54

The resource utilization statistics are defined as follows:

<b>Clocks</b>	Clock pins used for clock signals
<b>Pins</b>	Input and I/O pins used in any capacity
<b>Input Lines</b>	Array inputs
<b>I/O Macro</b>	Output macrocells
<b>Total Macro</b>	Output and buried macrocells
<b>I/O Pins</b>	Number of bonded-out pin feedbacks
<b>Input Regs</b>	Macrocells used as input registers
<b>Macrocells</b>	Macrocells without output/buried distinction
<b>Pterms</b>	AND array rows used in equation generation
<b>Feedbacks</b>	Inputs to the Switch Matrix
<b>Fanouts</b>	Inputs to the AND Arrays
<b>Blk Clocks</b>	Number of selectable clock lines for each block

## Partitioner Report

This section shows how the functions (outputs and nodes) are partitioned into PAL blocks including:

- Which signals must be routed to the PAL block to generate the functions assigned to the block.
- How many unique clocks, enables, and register set/reset equations the assigned functions require.

## Clock Assignments

**Note** *In the MACH 3 and 4 families, the clock signals can vary from one PAL block to another.*

The Clock Assignments section is specific to the MACH 3 and 4 families, and shows:

- which clocks are required in which PAL blocks, and
- which phase (true or inverted) is needed.

The Clock Assignments section can have zero to four clock pins listed depending on how many clocks the design uses.

### Example

This report describes two clocks where:

- CLK0 is on pin 62 and is used in its true sense in all eight PAL blocks.
- CLK1 is on pin 23 and is used in its inverted sense in PAL block D.

CLOCK ASSIGNMENTS:

Notes: block usage 'H' indicates used in TRUE sense.  
block usage 'L' indicates used in INVERSEsense.

```
clock signal [ 35] CLK1
pin          23
block usage  , , , L, , , ,
```

```
clock signal [ 34] CLK0
pin          62
block usage  H , H , H , H , H , H , H , H
```

# Signal Directory

The Signal Directory section lists all inputs, outputs, and nodes on the device with specific assignment information for each signal. The format of this section for the MACH 1 and 2 families is different from that for the MACH 3 and 4 families.

## Sample: MACH 1xx and 2xx Signal Directory section

SIGNAL DIRECTORY:

Notes:   Leading '-' indicates signal not assigned.  
          Trailing '+' indicates feedback path is from pin.  
          Functions with '0' Clusters are input registered.

Signal # Name	Source Type	PalBlk Clusters	Pal Block Inputs	
0 A_10__p2	Cmb Output	D 1	D11	
1 A_11__p3	Cmb Output	D 1	D10	
2 A_8__p4	Cmb Output	D 1	D09	
3 A_9__p5	Cmb Output	D 1	D15	
4 A_19__p9	Input		A01	+
5 RESET__p10	Input		A05 B05 C05 D05	+
6 A_20__p11	Input		D21	+

Every input, output, and node is listed in this directory. The data columns are defined as follows:

Signal #	The index number used to reference the signal
Signal Name	The user identifier for the signal
Source Type	{Input   Hidden   Output   Biput   Internal} with register type qualifiers
PalBlk	Pal Block where output or node is assigned
Clusters: Used	Number of Pterm Clusters used to generate function
Clusters: Unused PTs	Unused Pterms left in used clusters
Pal Block Inputs	Array input lines for Signal Fanouts

## Sample: MACH 3xx and 4xx Signal Directory

SIGNAL DIRECTORY:

Notes: Register type suffix '\_X' indicates XOR used;  
Register type suffix '\_A' indicates Asynchronous mode used;  
Register type suffix '\_LT' indicates function is LOW\_TRUE.  
'RS\_SWAP' flags functions which are preset at power-on.  
'OE' flags tri-state functions.

[ 0] Output: SAO\_8\_  
Pin 72 (I/O) Block G Macrocell\_G14 1 PTERM COMB

[ 1] Output: SAO\_7\_  
Pin 48 (I/O) Block E Macrocell\_E10 1 PTERM COMB

[ 2] Output: SAO\_6\_  
Pin 45 (I/O) Block E Macrocell\_E00 1 PTERM COMB

...

[ 32] Reg. Input: NBDIR  
Pin 3 (I/O) Block A Unary\_of\_3 1 PTERM LATCH

[ 33] Reg. Input: NCDIR  
Pin 78 (I/O) Block H Unary\_of\_78 1 PTERM LATCH

[ 34] Node: ST4  
Block D Macrocell\_D03 13 PTERM DFF\_A

[ 35] Node: ST3  
Block H Macrocell\_H09 15 PTERM DFF\_A

...

[ 44] Input: ADIR  
Pin 5 (I/O) Block A

[ 45] Input: BDIR  
Pin 3 (I/O) Block A

Each of the entries has two lines:

- The first line contains the signal index (in brackets), signal type, and signal name. The Resource Assignment Map uses the signal index shown here since there is not always enough room for the full signal name. The Signal type is one of the following: Input, Reg. Input, Reg., Feedback, Node, Tri-state, or Output.
- The second line contains the assignment information for the signal. If the signal appears on a pin, PLSyn reports the pin number and type. Function and Inputs on I/O pins provide the block number of the pin and/or macrocell assignments. Functions provide macrocell assignment information along with specifics on how PLSyn fit the function. This includes the number of PTERMs the function requires, and the register type used to implement the function. These are noted in the Notes section at the top.

## Resource Assignment Map

This section follows the physical layout of the device and shows signal assignments. As with the Signal Directory, the format of this section is different for the MACH 1 and 2 and the MACH 3 and 4 families.

The MACH 1 and 2 families are simpler to represent since there is a one-to-one relationship between pins, macrocells, and array inputs.

Sample: MACH 1xx and 2xx Resource Assignment Map

RESOURCE ASSIGNMENT MAP:

MINC Node#	Node Type	Pin/Macro ID	Signal ###	Name
1	Vcc/Gnd	PWR		
2	I/O	IO-00	( 34)	A_13
45	Shadow	A00	( 34)	A_13
46	Buried	A01	( 64)	B_16
3	I/O	IO-01	( 24)	A_17
47	Shadow	A02	( 61)	C_13
48	Buried	A03		
...				
8	I/O	IO-06	( 32)	A_15
57	Shadow	A12	( 32)	A_15
58	Buried	A13		
9	I/O	IO-07	( 31)	A_20
59	Shadow	A14	( 65)	B_17
60	Buried	A15	( 63)	C_15
10	Input	IO	( 6)	A_24
11	Input	I1	( 30)	A_21
12	Vcc/Gnd	PWR		
13	In/Clk	I2/C0	( 8)	CLK2
14	I/O	IO-08	( 21)	A_30
75	Shadow	B14		
76	Buried	B15	( 53)	B_25

Every input, output, and node is listed in this directory. The data columns are defined as follows:

MINC Node #	The physical pin number or internal node number
Node Type	{ Vcc/Gnd   Shadow   Buried   I/O   Input   In/Clk }
Pin/Macro ID	Pin or macrocell identifier
Signal #	Signal index (see SIGNAL DIRECTORY)
Signal Name	Signal name

If the same signal is assigned to a shadow node and the adjacent I/O pin, the signal is an output. If these two are different, the signal on the shadow pin is a node, and the signal on the I/O pin is an input.

The MACH 3 and 4 families are more complex to represent since the paths between pins, macrocells, and array inputs are programmable.

### Sample: MACH 3xx and 4xx Resource Assignment Map

Resource Assignment Map

Notes:Signal index '[###]' refers to SIGNAL DIRECTORY entry ###.  
Signal index '[N/C]' is specified 'NO\_CONNECT' in the .pi file.  
Signal index '[- - -]' indicates no signal present.  
Resource 'IR' is input register; 'MC' is macrocell.  
PTERM Cluster 'E' is equation cluster (2 PTERMs).  
PTERM Cluster 'A' is async cluster (2 PTERMs).  
PTERM Cluster 'S' is single cluster (1 PTERM).  
Cluster Steering 'd': down one macrocell (by macrocell number).  
Cluster Steering 'u': up one macrocell.  
Cluster Steering 'U': up two macrocells.  
Cluster Steering '=': to adjacent macrocell.  
Cluster Steering '-': cluster not used.

--PINOUT--		-----PLACEMENT-----						ROUTING-----							
Pin	[Sig]	InReg/	[Sig]	PTERMs	Feedback	-----Fanout-----									
_____	_____	MCell_	_____	EAS	ID_	[Sig]	Src	Block	and	Input	Line	_____			
1	PWR														
2	PWR														
3	[45]	IR	0	[32]		A00	[32]	IR	A08	B08	C08	D11	E08	G08	H19
		MC	A00	[24]	===	A01	[---]	-							
		MC	A01	[---]	ddd	A02	[---]	-							
4	[---]	IR	1	[---]		A03	[---]	-							
		MC	A02	[ 42]	===	A04	[ 42]	MC							H00
		MC	A03	[---]	uuu	A05	[---]	-							
5	[ 44]	IR	2	[ 31]		A06	[ 31]	IR	A03	B03	C03	D03	E03	G03	H03
		MC	A04	[---]	UUU	A07	[---]	-							
		MC	A05	[---]	---	A08	[---]	-							



Every input, output, and node is listed in this directory. The data columns are defined as follows:

<b>PINOUT</b>	Signals on physical pins
<b>Pin</b>	Physical pin number
<b>[Sig]</b>	Signal index of pin signal
<b>PLACEMENT</b>	Resources used to generate nodes and outputs
<b>InReg/Mcell</b>	Input Register (IR) or Macrocell (MC) identifier
<b>[Sig]</b>	Signal index of node or output
<b>PTERMs EAS</b>	PTERM steering (See below)
<b>ROUTING</b>	Signals into and out of switch matrix
<b>Feedback ID</b>	Identifier of switch matrix input
<b>Feedback [Sig]</b>	Signal index of feedback signal
<b>Feedback Src</b>	Source directed to switch matrix { Pin   IR   MC }
<b>Fanout</b>	PAL block inputs assigned to signal

**PTERM steering of clusters**

The report shows PTERM steering for three PTERM clusters per macrocell. The three clusters are:

- E** Equation which consists of two PTERMs which are always part of the data equation
- A** Asynchronous which consists of the two PTERMs which are either used as part of the data equation or used as asynchronous clock and reset.
- S** Single consists of the single PTERM which is either part of the data equation or half of the XOR equation.

The following character flags indicate the steering of these clusters:

- =** Local macrocell
- u** Up one macrocell
- d** Down one macrocell
- U** Up two macrocells

**Note** *“Up” and “down” do not mean physically higher or lower in the printout. Up refers to a lower-numbered macrocell, while down refers to a higher-numbered macrocell. In odd numbered PAL blocks (blocks B, D, F, H) the macrocells are numbered in reverse order compared to the pins. Since this printout is ordered by physical pins, the macrocells in those blocks show up in reverse order. However, down from any macrocell 3 is always macrocell 4.*

---

# MACH 5 Device-Specific Fitting

---

## 9

### Chapter Overview

This chapter describes how to control the fitting process for specific MACH 5 devices. Topics include:

- A comparison of MACH 5 devices to other MACH architectures, *page [9-2](#)*
- Tips and device details, *pages [9-5](#) through [9-20](#)*
- The document file, *page [9-21](#)*

# Comparing the MACH 5 to Other MACH Architectures

AMD's MACH5xx architecture represents a departure from previous MACH (MACH1xx/2xx/3xx/4xx) families. The earlier MACH families have extremely predictable timing because all signals follow the same paths through internal matrices. The MACH 1 and 2 families have a single internal matrix through which all input signals are routed to PAL blocks. The MACH 3 and 4 families extend internal routability by using Input, Central and Output matrices (see Figure 9-1).

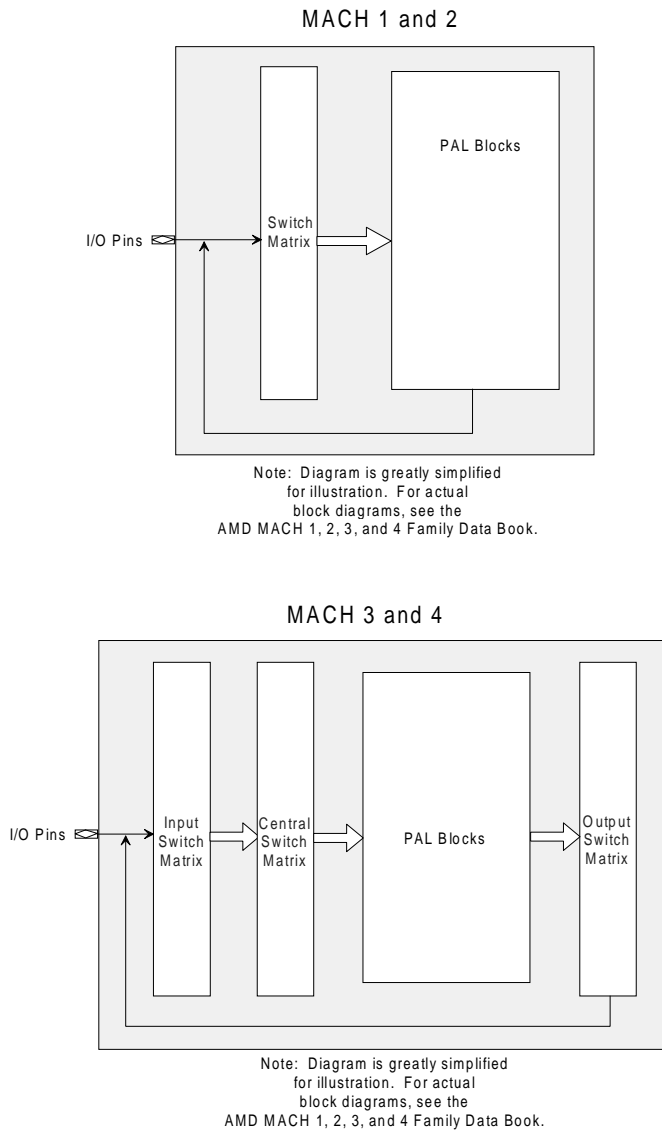
The MACH 5 architecture has a hierarchical interconnect system with internal routability of 100%. It also allows you to send signals directly to the PAL blocks without going through the interconnect matrices (as long as equations have fewer than 16 pterms). Even if functions have more than 16 pterms, the Block Interconnect (see Figure 9-2) can connect the signals to another PAL block within the segment.

When equations become too large to fit into a segment (4 PAL blocks), they can be routed to other segments via the Segment Interconnect. This means that tpd becomes longer with increased equation size, with boundaries at 16 and 48 pterms. However, this increased tpd is offset by the fact that the MACH 5 can connect any two signals internally. This lessens the necessity of having to use up I/O pins to connect signals (and eliminates some of the timing problems created by going "off chip"). Thus the MACH 5 can make refitting much easier.

The major differences between MACH1xx/2xx/3xx/4xx and MACH5 are shown graphically in the following two figures.

## MACH1xx/2xx/3xx/4xx

- Timing paths are the same for all signals, making for very predictable propagation delays.



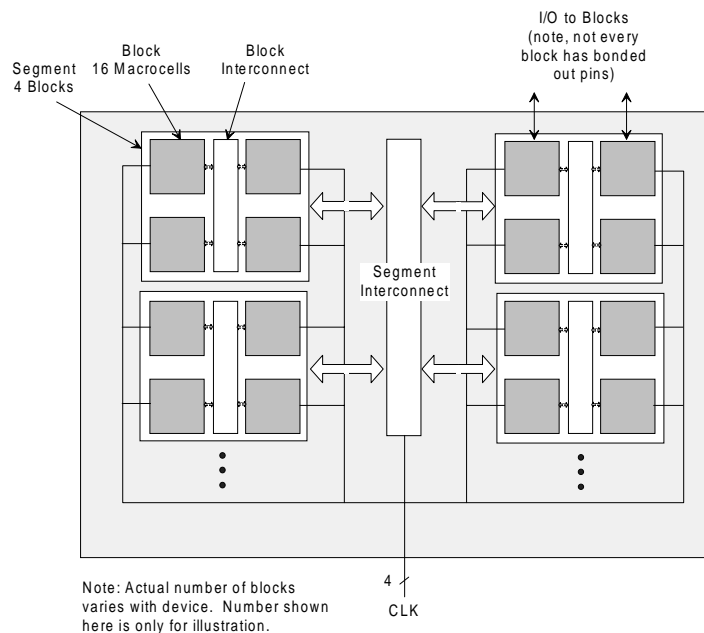
**Figure 9-1** *Simplified MACH 1xx/2xx/3xx/4xx Block Diagrams*

- To maintain timing paths, all I/Os go through a Switch Matrix (MACH 1 and 2) or an Input Switch Matrix (MACH 3 and 4). There are no direct paths from the outside world (except clocks) to the macrocells.

- Highly configurable, due to use of internal routing matrices.

## MACH5xx

- While timing paths are not all the same for the MACH 5 (unlike previous MACH families), this less predictable nature has significant advantages. For example, a block with bonded-out I/O pins could be used as a fast PLD for timing-critical signals. These signals need not go through the internal interconnect matrices.
- The use of hierarchical interconnect matrices in the MACH 5 yields increased internal routability (up to 100%). To say it another way, any two signals in an MACH 5 can be connected via the interconnect matrices.



**Figure 9-2** *Simplified 5xx Block Diagram*

# Using the .pi File to Control MACH 5 Fitting

MACH 5 devices are handled like any other PLD with full support for automatic device selection and partitioning. As with PLDs, you can also control implementation using the .pi file.

The following is a list of .pi file properties unique to the MACH 5.

FANOUT	POWER
FORCE_LOCAL_FB	SLEW_RATE
LOCAL_TOGGLE_FEEDBACK	

For additional device-specific information, refer to the *MACH Family Data Book* from AMD.

See Chapter 6, *Controlling the Fitting Process Using the .pi File* and the *PIL Reference* in PLSyn online help for more information on the .pi file.

## Routing in a Segment and Block

The SECTION construct and the TARGET statement are used to specify how signals are routed in a Segment and Block of an MACH 5 device.

### Syntax

```
TARGET  S<seg_id>[B<block_id>]
```

where

*seg\_id* is an optional segment identifier from 0..7

*block\_id* is an optional block identifier from a..d

You can specify just the targeted segment with:

```
TARGET 'S0';"section targeted at segment 0
```

or specify both targeted segment and block with:

```
TARGET 'S0Ba' "section targeted at segment 0, block A
```

### Example

```
DEVICE
```

```

TARGET 'TEMPLATE MV256_68 QFP-100-M256':
"place group into MV256

SECTION
  TARGET 'S1';
  "force q1 into MACH5 segment 1

  q1:8;
END SECTION;

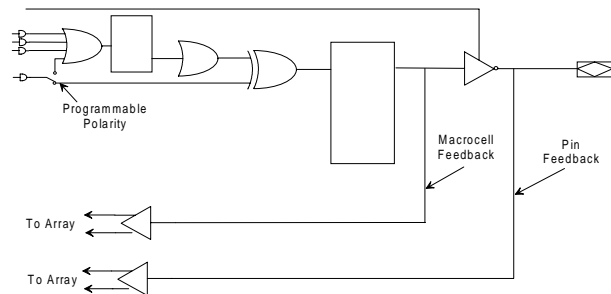
SECTION
  TARGET 'S0Ba';
  "force out7..out10 into
  "MACH5 segment 0
  "block A
  ...
  out7:5, out8:6;
  "assignment with physical pin
  "or node numbers
END SECTION;

END DEVICE;

```

## Assigning Pins and Nodes

An MACH 5 device has both physical (or absolute) pins (the ones on the device package) and relative node numbers (that is, node locations within the device). For each node number there is a corresponding node name (which is generally easier to remember than the node number).



**Figure 9-3** Mach 5 Architecture

There are two feedbacks associated with each macrocell in an MACH 5:



- Macrocell feedback which feeds back immediately after the macrocell, to the Block Interconnect.
- Pin feedback which feeds back after the tristate buffer. The pin feedback may or may not bond-out to a physical pin.

## Syntax

[S<seg\_id>][B<block\_id>] <feedback\_id>

where

seg\_id is an optional segment identifier from 0..7 (for MACH 5)

block\_id is an optional block identifier from a..d

feedback\_id = M<mcell\_no> | P<mcell\_no>

mcell\_no is the macrocell number from 0..16

M<mcell\_no> is the macrocell feedback

P<mcell\_no> is the pin feedback

Note that there is a node number for every pin feedback, regardless of whether or not the pin feedback is bonded-out to a physical pin. If the pin feedback is bonded-out, there is also a corresponding absolute (physical) pin number. If you specify absolute (physical) pin numbers, they will be reproduced in all output files of the fitter.

The following table defines relative node names and their virtual pin numbers.

**Note** *Virtual pin numbers are defined for internal device use only and are unique for the entire MACH 5 family.*

**Table 9-1** MACH 5 Node Names and Pin Numbers

Relative Node Names	Virtual Pin Names
S0BaM00..S0BaM15 . . . . .	0-15
S0BaP00..S0BaP15 . . . . .	16-31
S0BbM00..S0BbM15 . . . . .	32-47
S0BbP00..S0BbP15 . . . . .	48-63
S0BcM00..S0BcM15 . . . . .	64-79

Table 9-1 MACH 5 Node Names and Pin Numbers

Relative Node Names	Virtual Pin Names
S0BcP00..S0BcP15 .....	80-95
S0BdM00..S0BcM15 .....	96-111
S0BdP00..S0BcP15 .....	112-127
S1BaM00..S1BaM15 .....	128-143
S1BaP00..S0BaP15 .....	144-159
S1BbM00..S1BbM15 .....	160-175
S1BbP00..S1BbP15 .....	176-191
S1BcM00..S1BcM15 .....	192-207
S1BcP00..S1BcP15 .....	208-223
S1BdM00..S1BdM15 .....	224-239
S1BdP00..S1BdP15 .....	240-255
S7BaM00..S7BaM15 .....	896-911
S7BaP00..S7BaP15 .....	912-927
S7BbM00..S7BbM15 .....	928-943
S7BbP00..S7BbP15 .....	944-959
S7BcM00..S7BcM15 .....	960-975
S7BcP00..S7BcP15 .....	976-991
S7BdM00..S7BdM15 .....	992-1007
S7BdP00..S7BdP15 .....	1008-1023

Example

INPUT j1:S2BaM00 "route j1 to mux S2BaM00

# Placing a Signal on an Input Register or Latch

The .pi file property UNARY is used to place a signal on an input register or input latch. The UNARY property must be specified on the output signal of the unary function.

For more information on unaries, see *Accessing Internal Points in a PLD Device* on page 7-2.

## Example

### Source File

```
INPUT ui, iclk;
OUTPUT uo CLOCKED_BY iclk;
uo = ui;
```

### Physical Information File

```
DEVICE
SECTION
  TARGET 'S0Ba';
  INPUT ui;
  OUTPUT uo { UNARY };
END DEVICE;
```

# Using Dual Feedback

Dual feedback is the simultaneous use of both feedback paths, internal and pin. There are no DSL or .pi constructs for specifying dual feedback.

## To specify dual feedback

- 1 Write an intermediate node equation.
- 2 Set the pin feedback equal to node feedback.

The PLSyn fitter looks for such dual feedback equations and places them on the internal and pin feedback of the same macrocell.

**Note** *The node collapsing in the optimizer will collapse the intermediate node away unless you preserve the intermediate node in the .pi file.*

**Note** *There might be exceptions (unknown at this time) that will not allow placement of such dual feedback equations on the same macrocell internal/pin feedback. One possible example is lack of routing resources.*

## Example

```
INPUT i1, i2, i3, i4;
OUTPUTS out1, out2, out3;
OUTPUT pin_fb;      "pin_fb has to be placed
                    "on bonded out pin. If you
                    "consider this wasting an
                    "I/O pin, declare this a
                    "node instead.

NODE node_fb;

node_fb = i1 * i2 + i3 * i4;
                    "node feedback equation
pin_fb = node_fb;
                    "intermediate node equation

out1 = i2 * i3 * node_fb;
                    "node feedback used
out2 = i2 + i4 * pin_fb;
                    "pin feedback used
out3 = i2 * node_fb + i3 * pin_fb;
                    "both node and pin feedback
                    "on same eqn
```

# Forcing the Feedback Path to be Local

There are cases, for timing reasons, you may want all feedbacks to be contained within the same PAL block. You can do this in the MACH 5 with the `FORCE_LOCAL_FB` property. This property can be used at the `DEVICE`, `SECTION`, or signal level in the `.pi` file.

## Examples

### Source File

```
INPUT clk, rst, load, up_down, data[7..0] ;iclk;
OUTPUT count[7..0] CLOCKED_BY clk RESET_BY rst;
IF load = 0 THEN
  IF up_down = 1 THEN
    count = count .+. 1;
  ELSE
    count = count .-. 1;
  END IF;
ELSE
```

```
count = data;  
END IF; = ui;
```

## Physical Information File (Case 1)

"This example shows the use of the FORCE\_LOCAL\_FB at a  
"device level. This forces local feedback on all fanout  
"signals in the device.

```
DEVICE  
  TARGET 'PART_NUMBER AMD MACH 5-256/160-7HC';  
  {FORCE_LOCAL_FB}; "force local feedback on  
                    "all signals in the device  
END DEVICE;
```

### Physical Information File (Case 2)

"This example shows the use of the FORCE\_LOCAL\_FB in a "GROUP and SECTION. Note that you have to specify "FORCE\_LOCAL\_FB at the signal level in a GROUP.

```
DEVICE
  TARGET 'PART_NUMBER AMD MACH 5-256/160-7HC';
  GROUP
    COUNT[7];
    COUNT[6];
    COUNT[5];
    COUNT[4];
    DATA[7];{FORCE_LOCAL_FB};
    DATA[6];{FORCE_LOCAL_FB};
  END GROUP;

  SECTION
    TARGET 'SlBb';
    {FORCE_LOCAL_FB};
    COUNT[3];
    COUNT[2];
    COUNT[1];
    COUNT[0];
    DATA[5];
    DATA[4];
  END SECTION
END DEVICE;
```

## Specifying Fanout

To route a signal between two points, the fitter needs to know the signal's:

- Fanout destinations: The PAL block inputs are the signal's destinations.
- Path: If the destination is within a segment, no path information is required. If the fanout crosses segment boundaries via the segment interconnect bus, the intersegment line has to be specified.

## Syntax

```
{FANOUTS
'S<seg_id>B<block_id>M<mux_id><mux_line>
S<intersegment_line>';
```

where

```
seg_id = 0 .. 7
block_id = a / b / c / d (must be lower case)
mux_id = 0 .. 31
mux_line = 0 .. 7
intersegment_line = 0 .. 191
```

You can use the syntax shown above to specify a local feedback by using I7 for *mux\_line*. This assumes an 8:1 Level 1 mux.

## Example

A FANOUT specification of S0BaM01S100 means route to:

- Level 1 mux S0BaM0.
- Select line 1 via intersegment line 100.

```
DEVICE
" fully specified signal, NOT within a section
" The third fanout for j1 specifies a local feedback
INPUT j1:S2BaM00 { FANOUTS 'S2BaM0,S1BaM1I1,S2BaM00I7' };
NODE j2:S3BaP15 { FANOUTS 'S2BaM1I7' }; "This line will
"produce an error, local feedback incorrectly specified
NODE j2:S3BaP15 { FANOUTS 'I7' };
"correct j2 fanout spec, local feedback

SECTION
TARGET 'S0Ba'; "force out7..out8 into
" MACH5 segment 0 block A
q1:M00; "placements are local to block
INPUT i1:P01 { FANOUTS 'M0I0,S1BbM2I3' };
" 2 fanouts: 1st fanout is
" S0BaM0I0 if fully specified
NODE f1:M01 { FANOUTS 'M15I10' };
"route to L1 mux 15, line 15 of this
"block signal origin is M01 of this
"block. Note that i2 has been removed
"from this section and the fanout moved
"to its block of origin.
END SECTION;

SECTION
TARGET 'S1Bb'; "force out7..out8 into
```

```
"MACH5 segment 1 block b
q2:M00;      "placements are local to block
INPUT i2:P02; { FANOUTS 'M0I1,S0BaM0I1' };
" 2 fanouts
f2:M01 { FANOUTS 'M1I2' }; "route to L1
"mux 1, line 2 of this block
"Note that i1 has been removed from
"this section and the fanout moved
"to its block of origin.
END SECTION;
END DEVICE;
```

## Implementing Toggle Register Feedback

A toggle (T) register is implemented by taking the feedback of the register output Q and XORing it with the D register input. The toggle feedback can be

- A local feedback.
- Routed via a level 2 demux and the segment bus (non-local).

The property `LOCAL_TOGGLE_FEEDBACK` is used to force local toggle feedback.

The `LOCAL_TOGGLE_FEEDBACK` property can be specified at the device, SECTION or signal (outputs and nodes only) level.

If a local feedback path cannot be found for the toggle feedback, the fitter generates a warning.

## Implementing Dual-Edge Clocking

The MACH 5 has three clocking options:

- Selectable positive/negative edge clocking.



- Clocking on both edges.
- Complementary clocking, creating an inverse of clock line 3 (CLK2) on clock line 4 (CLK3).

The DSL control modifier `CLOCKED_BY BOTH_EDGES` lets you make use of either or both edges of the specified clock. You can use enables to specify negative or positive edge clocking by means of two keywords:

- `CLOCK_ENABLED_BY NEG_EDGE`
- `CLOCK_ENABLED_BY POS_EDGE`.

If a `CLOCK_ENABLED_BY` is not specified with the `CLOCKED_BY BOTH_EDGES` construct, the equation defaults to clocking on `BOTH` edges.

Complementary clocking is available if the macrocell is not controlled by a `CLOCKED_BY BOTH_EDGES` construct. Complementary clocking uses clock line 3 (CLK2) as the primary clock and clock line 4 (CLK3) as the inverted clock.

## Syntax

```
OUTPUT signal_name CLOCKED_BY BOTH_EDGES_OF clk_name  
    CLOCK_ENABLED_BY POS_EDGE enable_name;  
    CLOCK_ENABLED_BY NEG_EDGE enable_name;
```

## Example

```
INPUT clk1, clk2, ce1, ce2;
OUTPUT out1 CLOCKED_BY BOTH_EDGES_OF clk1;
    "clocks out1 on both edges of clk1

OUTPUT out2 CLOCKED_BY BOTH_EDGES_OF clk2
    CLOCK_ENABLED_BY POS_EDGE_OF ce1
    CLOCK_ENABLED_BY NEG_EDGE_OF ce2;
    "clocks out2 on either edge of clk2,
    "determined by enables ce1 and ce2
```

# Specifying Reserve Capacity

The MACH\_UTILIZATION property specifies the amount of reserve capacity to leave available in a device. This affects the use of pterms and macrocells.

## Syntax

```
{MACH_UTILIZATION percent} ;
```

where *percent* is the percentage of device resources to be used. The range of values is 0 to 100.

The unused resources are distributed throughout the device. There are two reasons to reserve some resources in a device.

- To allow for expansion of logic.
- To ease and speed the fitting process. Simply put, it is easier for the fitter to place and route a solution at 80% utilization than at 100% utilization. If design iteration speed is more important than density (for example, earlier in the design cycle or for refitting), set the utilization factor to a lower value.

# Constraining the Size of Combinatorial Nodes

You can constrain the size of combinatorial nodes PLSyn collapses during the optimization process, thereby affecting how the logic fits into MACH devices.

## To constrain the size of combinatorial nodes

- 1 Use the MAX\_PTERMS property in your .pi file using the syntax:

```
{MAX_PTERMS p};
```

where *p* is the maximum number of PTERMs to which the optimizer can collapse.

The PLSyn optimizer collapses combinatorial nodes up to a size specified by MAX\_PTERMS.

## Making Adjustments

### Using lower MAX\_PTERMS generally results in

- Less node collapsing
- Smaller functions
- Slower implementation
- May increase routing requirements

If the value is low, the design will typically be implemented as a larger number of smaller equations. This makes placement somewhat easier because smaller functions do not place demand on the pterm allocation mechanism, but more smaller functions may require more routing resources and may require more overall macrocell logic.

### Using higher MAX\_PTERMS generally results in

- More node collapsing
- Larger functions
- Faster implementation
- May increase routing requirements

Fewer larger functions may ease the routing requirements, but be harder to place, because the demand for pterms may cause conflicts in placing functions together in a PAL block.

**Note** *For optimal fitting, you should try a number of values to determine the best value for your design.*

### To see the exact effect of changing the optimizing parameters

- 1 After optimizing, open the .doc file.
- 2 Check the number of nodes. The number of nodes generally goes down as the MAX\_PTERMS parameter goes up.



## A Few Considerations

- Either High or Low MAX\_PTERMS can cause greater routing demand.
- Lower MAX\_PTERMS can produce more internal nodes which must be routed to the equations where they are used.
- Higher MAX\_PTERMS can allow a node to be collapsed into multiple equations so that the signals required to generate the node may be needed in multiple places. Furthermore, large equations may require large numbers of signals to be routed into the block where the equation is placed, producing a locally high routing demand.

## Other Optimizing Parameters

For general purposes, the following parameters may be used in the .pi file for designs targeting MACH5 devices.

MAX_PTERMS	32
MAX_XOR_PTERMS	31
MACH_UTILIZATION	100
MAX_SYMBOLS	32
POLARITY_CONTROL	TRUE
XOR_POLARITY_CONTROL	TRUE

## Controlling Power Levels

The syntax for specifying power level is:

```
POWER LOW | MED_LOW | MED_HIGH | HIGH
```

Power levels can be specified at a signal, SECTION or device level. The fitter will check the power levels for consistency across the various levels. Error messages will be printed out when the power levels specified do not match. If none is specified, the default power level will be HIGH.

### Example

```
SECTION
  TARGET 'S1Bb';      "force out7..out8 into MACH5
                      "segment 1 block b
  q2:M00;             "placements are local to block
  INPUT o2:P02; { FANOUTS 'M0I0', POWER LOW };
                      "power level for o2 is low
  f2:M01 { FANOUTS 'M1I1' }; "Use default slew rate
                      "which is FAST
END SECTION;
```

# Controlling Slew Rates

The syntax for specifying slew rate is:

```
SLEW_RATE    SLOW | FAST
```

Slew rate can be specified for signal, SECTION or device level. The fitter will check the slew rates for consistency across the various levels. If slew rates specified do not match, the fitter will generate an error.

## Example

```
SECTION
  TARGET 'S1Bb'; "force out7..out8 into MACH5
                                "segment 1 block b
q2:M00;                        "placements are local to block
                                "{ SLEW_RATE FAST }
INPUT o2:P02; { FANOUTS 'M0I0', SLEW_RATE SLOW };
                                "slew_rate is SLOW

f2:M01 { FANOUTS 'M1I1' }; "Use default slew
                                "rate which is FAST

END SECTION;
```

There is also a factory-programmed device-level downgrade to SLOW. When set to SLOW, it overrides the FAST slew rate attribute for individual signals. If individual signals are explicitly specified with a FAST slew rate and the device-level slew rate has been downgraded to SLOW, the fitter will generate a warning.

# The Document File

The document file, *design\_name.doc*, contains information about the various stages of compilation and partitioning. The following information is contained in the *.doc* file:

- Information about the design (title, designer, date, company, etc.) and switch values specified for compiler and optimizer functions.
- Explicit (or reduced) design equations that are realized in the final layout.
- A list of the solutions generated for the design.
- Partitioning criteria used in generating the device solutions.
- Pinout diagrams of the device solution selected.
- A list of possible devices for the templates in the solution.
- A wire list.

# The Report File

In addition to the `.doc` file, a report file, `design_name.rpt`, will be generated for an MACH 5 device. The report file generally contains the sections described below.

## Heading

This section generally contains the following information:

- Date when the design was run through the fitter
- Part type and device number
- Package type
- User supplied design information

## Example

```
DATE:      Fri Jan 26 14:44:48 1996
DESIGN:    probl.fb
DEVICE:    MV256_160:1
```



# Summary Statistics

This section summarizes the design in terms of number-of-clocks, inputs, nodes and outputs at the device level and its various sub-partitions namely, segments and PAL blocks. Power levels for each block are specified here.

## Example

```
SUMMARY STATISTICS:

  10 Inputs
  32 Outputs
   0 Tri-states
 124 Nodes

Functions by block:
S0:  8  7 12 12
S1:  8  7 12 12
S2:  8  7 12 12
S3:  8  7 12 12

D Register Macrocells    36
T Register Macrocells    24
D Latch Macrocells       0
Combinatorial Macrocells 92
D Input Registers        0
D Input Latches          0

Xor Equations            24
Single-Pterm Equations   23
Total Pterms Required    867
```

## Power Resource Utilization

The POWER SUMMARY section shows the following:

- Number of blocks with power set to LOW
- Number of blocks with power set to MED\_LOW
- Number of blocks with power set to MED\_HIGH
- Number of blocks with power set to HIGH

### Example

```
POWER SUMMARY:
Number of blocks with power set to LOW is 0
Number of blocks with power set to MED_LOW is 0
Number of blocks with power set to MED_HIGH is 0
Number of blocks with power set to HIGH is 16
```

## Device Resource Utilization

The DEVICE RESOURCE UTILIZATION section provides utilization statistics for the different device resources at the device, segment and PAL block partitions. A table is provided for each partition with the following columns:

Resource	Name of resource; the resources available for each block may be different
Available	Available resource count for the partition
Used	Used resource count for the partition
Remaining	Unused resource count for the partition
Percent	Percentage resource utilization for the partition

The resource types referenced in these tables are defined as follows:

<b>Clocks</b>	Clock pins used for clock signals
<b>Pins</b>	Input and I/O pins used in any capacity
<b>I/O Pins</b>	Number of bonded-out pin feedbacks
<b>Input Regs</b>	Macrocells used as input registers
<b>Macrocells</b>	Macrocells without output/buried distinction
<b>Pterms</b>	AND array rows used in equation generation
<b>Feedbacks</b>	Inputs to the Switch Matrix
<b>Fanouts</b>	Inputs to the AND Arrays
<b>Blk Clocks</b>	Number of selectable clock lines for each block

The resource types for the device and segment partitions are:

Clocks	Pins
Input Regs	Macrocells
Pterms	Feedbacks
Fanouts	

The resource types for the PAL block partitions are divided into two groups:

<b>Clock generator block:</b>	Clocks
	Pterms
	Blk Clocks
<b>Macrocell block:</b>	I/O Pins
	Input Regs
	Macrocells
	Pterms
	Feedbacks
	Fanouts

## Example

### DEVICE RESOURCE UTILIZATION:

Resource	Available	Used	Remaining	%
DEVICE				
Clock Pins:	4	1	3	25
I/O Pins:	160	41	119	25
Input Regs:	32	0	32	0
Macrocells	256	156	100	60
Control Pterms	144	16	128	11
Cluster Pterms	1024	876	148	85
1-pt Clusters:	256	180	76	70
3-pt Clusters:	256	252	4	98
Signal Resources	512	133	379	25
Array Inputs	512	264	248	51
Intersegment Lines:	128	9	119	7
SEGMENT 0				
Clock Pins:	4	1	3	25
Pins:	40	17	23	42
Input Regs:	8	0	8	0
Macrocells:	64	39	25	60
Control Pterms:	36	4	32	11
Cluster Pterms:	256	219	37	85
1-pt Clusters:	64	45	19	70
3-pt Clusters:	64	63	1	98
Signal Resources:	128	40	88	31
Array Inputs:	128	66	62	51
Segment Lines:	128	40	88	31
CONTROL BLOCK 'S0Ba'				
Clock Pins:	4	1	3	25
Blk Pins:	4	1	3	25
Enable Pterms:	2	0	2	0
Init Pterms:	3	1	2	33
Clock Pterms:	4	0	4	0
MACROCELL BLOCK 'S0Ba'				
I/O Pins:	16	8	8	50
Input Regs:	2	0	2	0
Macrocells	16	8	8	50
Cluster Pterms	64	61	3	95
1-pt Cluster	16	14	2	87
3-pt Clusters:	16	16	0	10
Signal Resources	32	10	22	31
Array Inputs	32	14	18	43

## Partition Groups

This section shows how functions (outputs and nodes) are assigned to the PAL blocks. It shows which signals must be routed to the PAL block to generate the functions assigned to the block. It also shows how many unique clocks, enables and register preset/reset equations are required for the assigned functions.

### Example

#### PARTITION GROUPS:

```
Block 'S0Ba'Partition 0;      Group-type FIXED_GROUP;
1 Clocks;  0 Enables;  1 Register Sets
8 Functions
O3[4] O3[2]  O3[1]
prep_4.12.large-0prep_4.12.large-1prep_4.12.large-2
prep_4.12.large-3prep_4.12.large-4

15 Signals
clk  rst      q8[7]
q8[6] q8[5]  q8[4]
q8[3] q8[2]  q8[1]
q8[0] prep_4.12.large-0prep_4.12.large-1
prep_4.12.large-2prep_4.12.large-3prep_4.12.large-4

Block 'S0Bb'Partition 1;      Group-type FIXED_GROUP;
1 Clocks;  0 Enables;  1 Register Sets
7 Functions
q8[5] q8[4]  prep_4.11.large-0
prep_4.11.large-1prep_4.11.large-2prep_4.11.large-3
prep_4.11.large-4

15 Signals
clk  rst      q7[7]
q7[6] q7[5]  q7[4]
q7[3] q7[2]  q7[1]
q7[0] prep_4.11.large-0prep_4.11.large-1
prep_4.11.large-2prep_4.11.large-3prep_4.11.large-4

Block 'S0Bc'Partition 2;      Group-type FIXED_GROUP;
1 Clocks;  0 Enables;  1 Register Sets
12 Functions
q7[7] q7[6]          q7[1]
q7[0] q8[7]          q8[6]
q8[3] q8[2]          q8[1]
q8[0] prep_4.10.large-0 prep_4.10.large-3
```

```
20 Signals
I[7]  I[6]  I[5]
I[4]  I[3]  I[2]
I[1]  I[0]  clk
rst    prep_4.10.large-0prep_4.10.large-1
prep_4.10.large-2prep_4.10.large-3prep_4.10.large-4
prep_4.11.large-0prep_4.11.large-1prep_4.11.large-2
prep_4.11.large-3prep_4.11.large-4
```

## Signal Directory

Clocks, inputs, outputs and nodes on the part are listed with specific assignment information for each signal. Slew rate which is on a per-signal basis on the MACH 5 will also be listed here.

The signal directory table will have the following columns:

<b>Signal #</b>	The index number used to reference the signal
<b>Signal Name</b>	The user identifier for the signal
<b>Source Type</b>	{Input   Hidden   Output   Biput   Internal} with register type qualifiers
<b>PalBlk</b>	Pal Block where output or node is assigned
<b>Clusters: Used</b>	Number of Pterm Clusters used to generate function
<b>Clusters: Unused PTs</b>	Unused Pterms left in used clusters
<b>Pal Block Inputs</b>	Array input lines for Signal Fanouts

## Example

SIGNAL DIRECTORY:

Notes:

Register type suffix '\_X' indicates XOR used;  
 Register type suffix '\_LT' indicates function is LOW\_TRUE.  
 'RS\_SWAP' flags functions which are preset at power-on.  
 'OE' flags tri-state functions.

```
[ 0] Output: O[7]
Pin 168 (I/O)Block S3Bd Macrocell_02 4 Pterm COMB

[ 1] Output: O[6]
Pin 169 (I/O)Block S3Bd Macrocell_03 2 Pterm COMB

[ 2] Output: O[5]
Pin 170 (I/O)Block S3Bd Macrocell_04 2 Pterm COMB

[ 3] Output: O[4]
Pin 165 (I/O)Block S3Ba Macrocell_00 1 Pterm COMB

[ 4] Output: O[3]
Pin 171 (I/O)Block S3Bd Macrocell_05 2 Pterm COMB

[ 5] Output: O[2]
Pin 163 (I/O) Block S3Ba Macrocell_02 1 Pterm COMB

[ 6] Output: O[1]
Pin 164 (I/O)Block S3Ba Macrocell_01 3 Pterm COMB

[ 7] Output: O[0]
Pin 172 (I/O)Block S3Bd Macrocell_06 2 Pterm COMB

[ 8] Node:   q1[7]
      S3BcM2 Block S3Bc Macrocell_02 4 Pterm COMB

[ 9] Node:   q1[6]
      S3BcM1 Block S3Bc Macrocell_01 2 Pterm COMB

[10] Node:   q1[5]
      S3BdM1 Block S3Bd Macrocell_01 2 Pterm COMB

[11] Node:   q1[4]
      S3BdM0 Block S3Bd Macrocell_00 1 Pterm COMB

[12] Node:   q1[3]
      S3BdM12 Block S3Bd Macrocell_12 2 Pterm COMB
```

# Fanout Table

The headings in the table have the following meanings:

<b>Signal_Src</b>	Signal name from the SIGNAL DIRECTORY LIST
<b>ISL#</b>	Intersegment line number to which Signal_Src connects
<b>SL#</b>	Segment line number
<b>Src SL#</b>	Segment line number for the same segment as the source signal



Example

FANOUT TABLE:

PASS/					Src	Fanouts-----						
FAIL	Signal_Src	ISL#	SL#	Blk	SL#	Mux	Blk	SL#	Mux	Blk	SL#	Mux
Block S0Ba:												
PASS[151]	S0BaM3	---	---	S0Ba	66	M17I2	S0Bd	66	M16I3			
PASS[152]	S0BaM4	---	---	S0Ba	97	M24I0	S0Bd	97	M22I5			
PASS[153]	S0BaM5	---	---	S0Ba	74	M16I1	S0Bd	74	M18I4			
PASS[154]	S0BaM8	---	---	S0Ba	75	M27I2	S0Bd	75	M17I4			
PASS [155]	S0BaM11	---	---	S0Ba	104	M07I0	S0Bd	104	M06I5			
PASS [162]	S0BaP4	35	116	S0Bc	116	M11I5	S0Bd	116	M29I4	S1Bc	119	M30I6
				S1Bd	119	M05I6	S2Bc	98	M09I5	S2Bd	98	M31I5
				S3Bc	102	M22I5	S3Bd	102	M19I5			
PASS [165]	S0BaP5	11	117	S0Ba	117	M01I0	S0Bb	117	M13I1	S0Bc	117	M04I5
				S0Bd	117	M03I6	S1Ba	101	M21I3	S1Bb	101	M00I1
				S1Bc	101	M15I5	S1Bd	101	M18I5	S2Ba	114	M12I0
				S2Bb	114	M04I1	S2Bc	114	M24I6	S2Bd	114	M07I6
				S3Ba	100	M29I0	S3Bb	100	M12I1	S3Bc	100	M08I5
				S3Bd	100	M13I5						
PASS [156]	S0BaP6	90	86	S0Bc	86	M10I3	S0Bd	86	M04I3	S1Bc	87	M01I4
				S1Bd	87	M28I4	S2Bc	65	M22I3	S2Bd	65	M01I3
				S3Bc	71	M20I3	S3Bd	71	M18I3			
PASS [163]	S0BaP7	23	119	S0Bc	119	M03I5	S0Bd	119	M05I6	S1Bc	99	M19I5
				S1Bd	99	M16I5	S2Bc	115	M01I5	S2Bd	115	M11I6
				S3Bc	111	M31I5	S3Bd	111	M30I6			
PASS [157]	S0BaP9	126	89	S0Bc	89	M09I4	S0Bd	89	M07I4	S1Bc	73	M06I3
				S1Bd	73	M23I5	S2Bc	92	M14I4	S2Bd	92	M03I4
				S3Bc	88	M10I4	S3Bd	88	M02I4			

# Power Table

## Example

POWER TABLE:				
	BLOCK A	BLOCK B	BLOCK C	BLOCK D
SEGMENT 0:	HIGH	HIGH	HIGH	HIGH
SEGMENT 1:	HIGH	HIGH	HIGH	HIGH
SEGMENT 2:	HIGH	HIGH	HIGH	HIGH
SEGMENT 3:	HIGH	HIGH	HIGH	HIGH

# Block Configuration Tables

## Example

```
BLOCK CONFIGURATION TABLES:
Notes: '*' indicates that the pin is bonded-out
BLOCK 'S0Ba': POWER=HIGH
CONTROL PTERMS:
    RST0 = rst ;
BLOCK CLOCKS:
    BLK_CLK 2 (PIN_CLOCK,POL=HIGH) : clk ;
    BLK_CLK 3 (PIN_CLOCK,POL=LOW) : clk ;
ARRAY INPUTS:
[---]    [165]    [---]    [138]    [139]    [---]    [135]    [155]    Inputs I0 to I7
[140]    [---]    [---]    [---]    [---]    [---]    [---]    [---]    Inputs I8 to I15
[153]    [151]    [---]    [---]    [136]    [134]    [133]    [---]    Inputs I16 to I23
[152]    [---]    [137]    [154]    [---]    [---]    [---]    [---]    Inputs I24 to I31
```

**Note**    *Array inputs I0 through I31 are assigned signal names from the SIGNAL DIRECTORY list.*

In the following segment, labels have the following meanings:

Pterms Used	Number of pterms used on this macrocell; if the column has 1+7, it means 7 pterms were used and one pterm was steered from elsewhere
Pterms Avl	Number of pterms available for this macrocell
PT Map	Indicates whether pterm was applied to the XOR or product term cluster (OR input)
POL	Indicates polarity of the signal
CLK	Indicates which clock from the clock generator was used
Reg Ctrl	Indicates whether signal was combinatorial or registered
Slew	Slew rate set
OE	Indicates whether the output enable was high or low
Node	Relative node number
Pin	Actual pin number

Example

C C C C C C C C C C C C C C C C															P C											
0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1															Pterms		PT	O	L	Reg						
MC	0 1 2 3 4 5															Used	Avl	Map	L	K	Ctrl	Slew	Node	OE	Pin	
00	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	SUM	L	3	COMB	FAST	[---]	VCC	[120]	*
01	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	0	SUM	1	3	COMB	FAST	[---]	VCC	[123]	*
02	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	1	0	SUM	L	3	COMB	FAST	[---]	VCC	[122]	*
03	-	4	3	1	-	-	-	-	-	-	-	-	-	-	-	1+7	0	XOR	H	2	RST0	SLOW	[151]	GND	[---]	*
04	-	-	-	3	1	4	4	-	-	-	-	-	-	-	-	1+11	0	XOR	H	2	RST0	SLOW	[152]	GND	[162]	*
05	-	-	-	-	3	-	-	4	4	-	-	-	-	-	-	11	0	SUM	H	2	RST0	SLOW	[153]	GND	[165]	*
06	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	---	-	-	----	----	[---]	GND	[156]	*
07	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	---	-	-	---	---	[---]	GND	[163]	*
08	-	-	-	-	-	-	-	-	3	3	4	4	-	-	-	13	1	SUM	H	2	RST0	SLOW	[154]	GND	[---]	*
09	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	---	-	-	---	---	[---]	GND	[157]	*
10	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	---	-	-	---	---	[---]	GND	[---]	*

---

# ATV5000 Device-Specific Fitting

---

# 10

## Chapter Overview

This chapter describes how to control the fitting process for Atmel's ATV5000 architecture. Topics include:

- General information about designing with the ATV5000, *page [10-2](#)*
- Tips and device details, *pages [10-2](#) through [10-17](#)*
- The report file, *page [10-18](#)*

## Designing with the ATV5000

The Atmel ATV5000 CPLD is supported by PLSyn through automatic device selection and automatic partitioning/fitting. The ATV5000 is a sophisticated device, with many unique features. ATV5000-Specific Optimization

See Chapter 6, *Controlling the Fitting Process Using the .pi File* and the *PIL Reference* in PLSyn online help for more information on the .pi file.

There are several .pi properties that control optimization of the design. While these properties are not specific to the ATV5000, they provide a means of tuning the optimization to best fit a design into ATV5000 parts.

## Constraining the Size of Combinatorial Nodes

The MAX\_PTERMS and MAX\_SYMBOLS properties are the key optimizer properties for fitting into the ATV5000. For most designs, the following settings for MAX\_PTERMS and MAX\_SYMBOLS are suggested:

```
{  
  MAX_PTERMS    13 ,  
  MAX_SYMBOLS   40  
}
```

## The Effect of MAX\_PTERMS

The MAX\_PTERMS property is the most critical property for optimizing a design for the ATV5000. The effect of changing MAX\_PTERMS is summarized here.

### Using higher MAX\_PTERMS generally results in this

- Fewer leftover combinatorial nodes
- Larger functions
- Faster implementation
- Increased number of sum terms required

Setting MAX\_PTERMS higher may increase the number of sum terms needed. The PLSyn fitter may place small registered nodes on logic cell register Q2 or on the buried logic cells. However, as MAX\_PTERMS is increased, the registered nodes increase in size beyond the capacity of the sum terms feeding register Q2 and the buried logic cells. The only option remaining may be to use more logic cell sum-terms to feed register Q1, possibly leaving register Q2 unusable.



### Using lower MAX\_PTERMS generally results in this

- More leftover combinatorial nodes
- Smaller functions
- Slower implementation
- Increased regionalization requirements

Setting MAX\_PTERMS lower may increase regionalization requirements. The regionalization requirements depend on the number of universal PTERMs in each function. Increasing MAX\_PTERMS may increase the number of PTERMs in each function, but the number of universal PTERMs in each function does not necessarily also increase. This is so because in the ATV5000, combinatorial shadow nodes feed back into the universal bus. Lowering MAX\_PTERMS will cause more combinatorial nodes to remain after node collapsing, and these additional combinatorial nodes may cause certain PTERMs to



be universal rather than regional, possibly increasing regionalization requirements.

### To see the exact effect of changing the optimizing parameters

- 1 Open the .doc file after optimizing and check the number of nodes. The number of nodes generally goes down as the MAX\_PTERMS parameter goes up.



It is advantageous to keep the number of combinatorial nodes low. This is because the combinatorial shadow nodes in the ATV5000 (the nodes in the logic cell where combinatorial node signals are placed) do double duty as RU converters. However, this depends on the particular design. If there are not many signals that must be routed from a quadrant's regional bus to the universal bus, it may be more advantageous to keep the size of the functions smaller.

In critical fitting cases, it may be necessary to try several settings for MAX\_PTERMS to get satisfactory results.

## The Effect of MAX\_SYMBOLS

Increasing MAX\_SYMBOLS will increase the number of inputs per PTERM in output and node signals. We suggest setting MAX\_SYMBOLS to 40 because the smallest product terms are the regional product terms, which have 40 input signals available. Increasing MAX\_SYMBOLS will potentially create PTERMs that are too big for the regional rows in the ATV5000.

# Specifying Device Utilization

## To specify the amount of reserve capacity to leave available in a device

- 1 Use the `ATV5_UTILIZATION` property in your `.pi` file using the syntax:

```
{ATV5_UTILIZATION percent};
```

where `percent` is the percentage of device resources to be used. The range of values is 0 to 100.

This affects the use of PTERMs, macrocells, and pins. The unused resources are distributed throughout the device. There are two reasons to reserve some resources in a device:

- Resources may be reserved to allow for expansion of logic.
- Resources may be reserved to ease and speed the fitting process. It is easier for the PLSyn fitter to place and route a solution at 80% utilization than at 100% utilization. If design iteration speed is more important than density (e.g., earlier in the design cycle), set the utilization factor to a lower value.

## Using the Flip-Flop Clock Option

The flip-flop clock option in the ATV5000 architecture can provide the clock for the registers from two locations:

- One product term.
- One product term ANDed with a clock pin signal.

The PLSyn fitter uses this flip-flop clock option to:

- Provide enabled clocking functionality.



- Allow you to control the source of the clock signal.

## Enabling Clocking

A registered output or node signal may be declared in the `.src` file to have a clock enable through the DSL

`CLOCK_ENABLED_BY` declaration. The PLSyn fitter will implement clock enable functionality by using the flip-flop clock option as follows:

- The clock signal is placed on the regional clock pin.
- The PTERM given in the `CLOCK_ENABLED_BY` declaration is placed on the clock product term.

Therefore, the clock signal will not be seen by the register until the `CLOCK_ENABLED_BY` PTERM is asserted.

The clock for the registered output or node signal must be a single signal. The clock enable may be a single signal or a single PTERM.

There is no on-chip synchronization circuitry between the clock signal and the clock product term. It is your responsibility to assure that the signals that feed the flip-flop clock option are glitch-free.

### Example

```
SOURCE FILE
INPUT i, clk, ce1..ce2;
OUTPUT o CLOCKED_BY clk CLOCK_ENABLED_BY
ce1*ce2;
o = i;
```

## Controlling the Clock Source

The flip-flop clock option in the ATV5000 allows the clock for registered output and node signals (with no clock enable), to be provided by one of two sources:

For more information on `CLOCK_ENABLED_BY`, refer to the *PIL Reference* in PLSyn online help.

- a dedicated clock pin, one per quadrant
- the clock product term

By default, the PLSyn fitter will place the clock on the clock product term, saving the quadrant clock pin for inputs to the regional bus. However, if you need the speed provided by the quadrant clock pin, you can specify that the clock be placed on the quadrant clock pin. This is done through the `CLOCK_BY_PIN .pi` property.

This property cannot be used if the signal is clocked by an equation (for example, `CLOCKED_BY a*b`).

## Example

### SOURCE FILE

```
INPUT i, clk;
OUTPUT o CLOCKED_BY clk;
o = i;
```

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'TEMPLATE ATV5000 JLCC-68-STD';
  o {CLOCK_BY_PIN}; "Force the clock to
    "come from the quadrant clock pin
END DEVICE;
```

You can also explicitly specify that the clock is to be supplied by the clock product term. This is done through the `CLOCK_BY_ROW .pi` property.

`CLOCK_BY_ROW` is the default for registered outputs and nodes.

## Example

### SOURCE FILE

```
INPUT i, clk1..clk2;
OUTPUT o CLOCKED_BY clk1*clk2;
o = i;
```

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'TEMPLATE ATV5000 JLCC-68-STD';
```

## 10-8 ATV5000 Device-Specific Fitting

---

```
    ○ {CLOCK_BY_ROW};  
END DEVICE;
```

# Using the I/O Pin Latches

The I/O pins in the ATV5000 architecture can direct an input signal through a latch. You can use the I/O pin latches through the latched unary concept.

The PLSyn fitter considers the I/O pin latches to be unary nodes, and hence possible locations for placing unary functions.

To declare a unary function in your design, declare a node signal in the `.src` file with the following characteristics:

- Declared as a latched node signal.
- Latched by either a low-true or an inverted signal.
- Fed by a single signal.

The PLSyn fitter may place the unary functions automatically, or you can place them manually through `.pi` file assignments. See the following section for the unary pin names.

For more information on unaries, see *Accessing Internal Points in a PLD Device* on page 7-2.

## Identifying Pins and Nodes

This section describes the pin and node names for the ATV5000. This information is useful to manually assign signals to pins and nodes in the `.pi` file. It is also useful for interpreting where signals were fit in the `.rpt` and the `.npi` files.

The ATV5000 has both physical pins and virtual pins. Physical pins are the pins that physically appear on the device package. Virtual pins are device node locations where node signals may be placed.

Physical pins are referenced by the pin number in the package diagram.

Virtual pins are named according to their characteristics and their location in the device. The names imply the characteristics

of the device nodes, their location within the logic cell or buried logic cell, and the physical pin number of the logic cell that they are associated with.

**REG\_SHADOW\_OF\_** Registered shadow pins are located on register Q1, with the logic cell disconnected from the I/O pin. The I/O pin then functions as an input. Registered shadow pins have access to sum terms A, B, and C. Registered node signals may be placed on registered shadow pins.

**COMB\_SHADOW\_OF\_** Combinatorial shadow pins are located on sum term B, with the feedback going into the universal bus. Combinatorial shadow pins have access to sum term B. Combinatorial node signals may be placed on combinatorial shadow pins.

**BURIED\_OF\_** Buried pins are located on register Q2. Buried pins have access to sum term C. Registered node signals may be placed on buried pins.

**UNARY\_OF\_** Unary pins are located on the I/O pin latch. Unary functions may be placed of binary pins.

**BLMC** Designator for the buried logic cells. Combinatorial node signals or registered node signals may be placed on the buried logic cells.

# Targeting Quadrants in the ATV5000

You can specify which output and node signals are to be placed together in the same quadrant of a device. This specification is done in the `.pi` file. There are several reasons for explicitly grouping signals in the ATV5000, including:

- Critical timing may require you to keep a group of signals in the same quadrant, minimizing speed lost in RU conversion.
- PCB layout may be easier when related signals are kept together.
- Critical fitting cases may require you to manually tune the partitions created by the PLSyn fitter in order to achieve a successful fit.

## Using the GROUP Construct

The `.pi` file GROUP construct allows you to specify a set of output and node signals that are to be fit together into the same quadrant, without specifying which quadrant and without keeping multiple GROUPs from being fit together in the same quadrant.

For more information on GROUP, refer to the *PIL Reference* in PLSyn online help.

### Example

#### SOURCE FILE

```
INPUT i[8];
OUTPUT ogroup1[8];
OUTPUT ogroup2[8];
ogroup1 = i;
ogroup2 = i;

PHYSICAL INFORMATION FILE

DEVICE
  TARGET 'TEMPLATE ATV5000 JLCC-68-STD';
  GROUP
    ogroup1;      "all ogroup1 signals will go
                  "into the same quadrant
  END GROUP;
  GROUP
    ogroup2;      "all ogroup2 signals may or
                  "may not also go into
                  "ogroup1's quadrant
  END GROUP;
END DEVICE;
```

## Using the SECTION Construct

For more information on SECTION, refer to the *PIL Reference* in PLSyn online help.

The .pi file SECTION construct allows you to specify a set of signals that are to be fit together in the same quadrant. Two different SECTIONS will not be fit into the same quadrant.

In addition, you can specify which quadrant to fit the SECTION into with the TARGET construct.

Syntax is:

```
TARGET 'quadrant_name' ;
```

The list below details the names of the quadrants in the ATV5000.

## Quadrant Names

Quadrant 1

Quadrant 2

Quadrant 3

Quadrant 4

If a SECTION isn't targeted to a specific quadrant, PLSyn will place the SECTION into a quadrant automatically.

## Example

### SOURCE FILE

```
INPUT i[8];
OUTPUT ogroup1[8];
OUTPUT ogroup2[8];
ogroup1 = i;
ogroup2 = i;
```

### PHYSICAL INFORMATION FILE

```
DEVICE
  TARGET 'TEMPLATE ATV5000 JLCC-68-STD';
  SECTION
    TARGET 'Quadrant 1';
    ogroup1; "all ogroup1 signals
             "will go into quadrant 1
  END GROUP;
  SECTION
    TARGET 'Quadrant 2';
    ogroup2; "all ogroup2 signals
             "will go into quadrant 2
  END GROUP;
END DEVICE;
```



## Placing Node Signals on Buried Logic Cells

The PLSyn fitter will not automatically place node signals on the buried logic cells. However, you can manually place combinatorial or registered node signals on the buried nodes. This is accomplished through pin assignments in the `.pi` file.

You can sometimes reduce the number of resources used for regionalization by manually placing combinatorial node signals on the buried logic cells rather than on the combinatorial shadow nodes. Since the buried logic cells feed back into the regional bus rather than the universal bus, as the combinatorial shadow nodes do, regionalization resources may be saved. However, you must weigh this savings against potentially incurring RU conversion if the signal placed on a buried logic cell is needed in another quadrant.

The PLSyn fitter uses the buried logic cells for regionalization when fitting output and node signals. We recommended that you do not assign node signals, especially registered node signals, to the buried logic cells unless you are sure that you have enough buried logic cells to satisfy regionalization requirements.

### Example

#### SOURCE FILE

```
input i, clk;  
node n clocked_by clk;  
n = i;
```

#### PHYSICAL INFORMATION FILE

```
DEVICE  
  TARGET 'TEMPLATE ATV5000 JLCC-68-STD';  
  n: BLMC23;    "place node n on buried  
                "logic cell 23  
END DEVICE;
```

# Understanding RU Conversion

When a design is partitioned across the quadrants of an ATV5000, there are often signals fed back into one quadrant's regional bus via the Q1 and Q2 register feedbacks that are needed by output or node signals in a different quadrant. By routing the necessary regional bus signals to the universal bus, the signals can become available to the output and node signals in other quadrants. This routing process is called RU conversion (Regional - Universal conversion).

The PLSyn fitter performs RU conversion automatically by using the logic cell configuration that gives sum term B a feedback path into the universal bus. The regional bus signal to be RU converted is placed on one of the regional rows feeding sum term B, and the signal then takes the feedback path into the universal bus. In this configuration, sum term B functions as an *RU converter*.

When a logic cell's sum term B is used as an RU converter, it becomes unavailable for any other use (such as a combinatorial shadow node).

## Understanding Regionalization

Regionalization is the process of manipulating a universal PTERM so that it may be fit on a regional row in the ATV5000. Regionalization is used during the process of fitting the PTERMs of an output or node signal.

### Universal and regional PTERMs

Universal PTERMs have at least one signal that is available only in the universal bus or in the regional bus of a different quadrant

than the output or node signal is assigned to. Universal PTERMs can go only on the universal rows of the ATV5000.

All the signals of regional PTERMs are available in the regional bus of the same quadrant that the output or node signal is assigned to. Regional PTERMs may go on universal or regional rows of the ATV5000.

### **Regionalization, sum-term combining, and fitting PTERMs**

In a difficult-to-fit design, the key to directing PLSyn's PLSyn fitter to a successful fit is simply understanding how the fitter is attempting to fit the universal PTERMs. It also helps to know how sum-term combining and regionalization are interrelated for a particular design. You can use the `.rpt` file to obtain much information about the results of sum term combining and regionalization for a fit attempt.

When the PLSyn fitter fits the PTERMs of an output or node signal, it attempts to get enough regional and universal rows by combining sum terms. If this fails to supply enough universal rows, then regionalization is used to convert some of the output or node signal's universal PTERMs to regional PTERMs, allowing placement of PTERMs on the otherwise unused regional rows.

In addition, if a node signal has a universal PTERM that must go on the regional row feeding the asynchronous preset of register Q2, regionalization will be used to convert that universal PTERM to a regional PTERM.

Regionalization is handled automatically by the PLSyn fitter. There are no provisions to manually force regionalization of PTERMs.

There are two basic techniques used in regionalization:

- signal regionalization
- PTERM regionalization

## How PLSyn Does Regionalization

The PLSyn fitter always performs signal regionalization via input pins when attempting to fit a design. This is done before any other regionalization technique is used.

UR conversion and PTERM regionalization are complementary regionalization techniques. Some designs can only be fit via UR conversion, but others can be fit only via PTERM regionalization.

During a fit attempt, the PLSyn fitter varies the number of buried logic cells available per quadrant for PTERM regionalization, from 0 to 6, as it attempts to place the PTERMs of output and node signals. The remainder of the buried logic cells are used for UR conversion. This allows the best mix of these complementary regionalization techniques to be used.

## Signal Regionalization

Signal regionalization is the process of routing universal signals to the regional bus of a quadrant. By regionalizing the universal signals in a universal PTERM, the universal PTERM may become regional. Often, many universal PTERMs that have few universal signals can be regionalized by regionalizing just a few universal signals. The PLSyn fitter uses the input/clock pins and UR conversion to regionalize signals.

### Using input pins

The PLSyn fitter will place universal input signals on any input/clock pins that are not used to supply clock signals to registers or latches. The fitter uses the path from the pins into all four regional buses to regionalize the universal input signals.

Input signals are selected for regionalization via input pins based on the number of universal PTERMs that need each universal input signal across the entire device.

### Using feedback paths (UR conversion)

UR conversion (Universal - Regional conversion) is the process of regionalizing universal signals, using the feedback path from the buried logic cells into the regional bus.

The PLSyn fitter performs UR conversion by placing a universal signal on the universal row of a buried logic cell and configuring the buried logic cell for combinatorial operation. When used in this manner, the buried logic cell functions as a *UR converter*.

## PTERM Regionalization

Pterm regionalization is the process of regionalizing an entire universal PTERM, using a buried logic cell. Since the entire universal PTERM is regionalized at once, universal PTERMs that have a lot of universal signals can be regionalized via PTERM regionalization.

The PLSyn fitter performs PTERM regionalization by placing the entire universal PTERM on the universal row of a buried logic cell and configuring the buried logic cell for combinatorial operation. Therefore, a signal representing the entire universal PTERM is available in the regional bus, and substitutes for the original universal PTERM in any output or node signals that have the universal PTERM in their equations.

# The Report File

The `.rpt` file is written by the PLSyn fitter during the process of fitting part or all of a design into ATV5000 devices. The `.rpt` file is useful as an aid in:

- Understanding why designs do not fit.
- Directing the PLSyn fitter to success in fitting a difficult design.
- Determining how a design was fit and the device resources that were used.

The `.rpt` file is complementary to the `.doc` file. It contains information about the design and about the attempt made by the PLSyn fitter to implement the design. This information is specific to the ATV5000. In-depth information about the input, output, and node signals is given, along with assignments to device resources made by the PLSyn fitter.

## Obtaining Report File

### To obtain a report file

- 1 Create a `.pi` file with a DEVICE that is targeted towards an ATV5000.

No other specifications in the `.pi` file are necessary. PLSyn will generate automatically a report file named `design_name<nn>.rpt` where `<nn>` is a sequence number representing the edition of the report.

## Example

### PHYSICAL INFORMATION FILE

```
DEVICE
    TARGET 'TEMPLATE ATV5000 JLCC-68-STD';
END DEVICE;
```

If you think the design will take more than one device, put as many DEVICES in the .pi file as you think the design will need. A different .rpt file will be created for each device in the solution, named *design\_name01.rpt*, *design\_name02.rpt*, and so on.

Pin assignments, properties, and other .pi file constructs may be placed in the DEVICES if needed. They will not affect creation of the .rpt file.

## Heading

The header contains the date and time the design was run through PLSyn. It also contains the user-supplied design information from the .src file. This gives a way of identifying the .rpt file.

```
DATE:      Fri Sep  2 15:18:45 1994Date design was run
DESIGN:    drink           Design name
DEVICE:    ATV5000:1       Part name and position in PI file

DEVICE statement list

TITLE:     drink           User-supplied information from
ENGINEER:  ATV5000 Designer.src file
COMPANY:   Atmel Corporation
PROJECT:   ATV5000 .rpt example
REVISION:  1.0
COMMENT:   Example of ATV5000 .rpt file using example drink.src
```

## Failure-to-Partition Disclaimer

If the PLSyn fitter fails to partition the design successfully across the quadrants of the device, a disclaimer is printed immediately following the heading. This lets you know that partitioning failed.

If the design partitions successfully, no disclaimer will be printed.

## Partitioner Report

This section shows:

- The functions (output and node signals) assigned to each quadrant.
- The signals that must be available in each quadrant.
- How many unique clocks, latch enables, enables, and register reset/preset equations are in each quadrant.

## Signal Directory

This section contains information about the design that is specific to the ATV5000. All input, output and node signals assigned to the device are listed.

For each signal, the buses that the signals are available in are listed.

For output and node signals, the universal and regional PTERMs are listed. Also shown for output and node signals is the equation form used (DFF, TFF, or DeMorganized).

The information in the signal directory is taken before any device resources are assigned to. Therefore, some signals may become available in different buses during function placement. Also, some universal PTERMs may be regionalized during function placement.



## Example

SIGNAL DIRECTORY:

Notes: Universal PTERMs may become regional  
during function placement.  
'BAR' indicates DeMorganized form equation used.  
'DFF' indicates D flip-flop form equation used.  
'TFF' indicates T flip-flop form equation used.

Input: nickel

Buses: Univ

Input: dime

Buses: Univ

Output: return\_dime

Buses: Univ

Universal PTERMs:

/nickel\*/dime\*quarter\*/drink\_machine-1\*drink\_machine-2;

/nickel\*/dime\*quarter\*drink\_machine-0\*

drink\_machine-1\*/drink\_machine-2 ;

Regional PTERMs:

/drink\_machine-0\*drink\_machine-1\*drink\_machine-2 ;

Node:DFF drink\_machine-0

Buses: Univ Q1

Universal PTERMs:

nickel\*/drink\_machine-0\*/drink\_machine-2 ;

nickel\*/drink\_machine-0\*/drink\_machine-1 ;

/nickel\*/quarter\*drink\_machine-0\*/drink\_machine-2 ;

/dime\*quarter\*/drink\_machine-0\*/drink\_machine-1\*

/drink\_machine-2 ;

/nickel\*/dime\*/quarter\*drink\_machine-0\*

/drink\_machine-1 ;

/nickel\*dime\*drink\_machine-0\*/drink\_machine-2 ;

Regional PTERMs:

In this example, the input signals nickel and dime are available in the universal bus. The output signal return\_dime is available in the universal bus, has two universal PTERMs, and one regional PTERM. The node signal drink\_machine-0 is available in the universal bus and quadrant 1's regional bus, and has 6 universal PTERMs.

## Signals Universalized on Sum Term B

The signals that underwent RU conversion are listed here, quadrant by quadrant, as follows:

SIGNALS UNIVERSALIZED ON SUM TERM B:

```
Quadrant 1
-----
drink_machine-2
drink_machine-1
drink_machine-0
```

In this example, RU conversion was performed only in quadrant 1.

## Signals Regionalized on Input Pins

Signals that were regionalized on input/clock pins are listed here. Signals that only supply register clocks or latches from the input/clock pins are listed also, since they are available in all regional buses.

## Function Placement Report

The function placement report provides information about the actions of the PLSyn fitter during output and node signal placement. Information about UR conversion, PTERM regionalization, and output/node signal placement is provided. If the PLSyn fitter failed to fit the design, this information is especially valuable as an aid in guiding the PLSyn fitter to a successful fit.

### Quadrant sections

The function placement report is organized on an primary level around quadrant sections. In each quadrant section, function

placement progress for each quadrant is reported. Information about quadrant 1 is reported first, then quadrants 2, 3, and 4, if any functions were assigned to those quadrants. Within each quadrant section, each line in the `.rpt` file is preceded by a quadrant indicator to remind you of the current quadrant.

## **Fit attempt sections**

Within each quadrant section, the function placement report is organized on a secondary level around fit attempt sections for each quadrant. Each fit attempt section contains information about function placement for a fit attempt within each quadrant. Each fit attempt represents an attempt the PLSyn fitter made at placing the functions in the quadrant, with a different number of buried logic cells available for PTERM regionalization in each fit attempt. There may be up to 7 fit attempts. See the preceding section on Understanding Regionalization for more information on regionalization and the fitting process.

Within each fit attempt section is an UR conversion report, a PTERM regionalization report, and an output/node signal placement report. These three reports give information about regionalization and function placement progress for a fit attempt.

## **UR conversion report**

The signals that underwent UR conversion during the fit attempt are listed in this table, along with the buried logic cells serving as UR converters.

## **Pterm regionalization report**

The PTERMs that underwent PTERM regionalization during the fit attempt are listed in this table, along with the buried logic cells each PTERM was regionalized on.

## **Output/node signal placement report**

Each output and node signal that was successfully placed during the fit attempt is listed in this table, with the pin the signal was assigned to and the sum terms in the logic cell that were used by

the signal. This lets you examine how sum term combining was performed.

Example

The output/node signal placement report for the drink example for quadrant 1, fit attempt 1 looks like:

```
Q1: OUTPUT/NODE SIGNAL PLACEMENT REPORT:
Q1:
Q1:      Device Pin      Sum terms used      Signal
Q1: -----
Q1: REG_SHADOW_OF_13      a b      drink_machine-0
Q1: REG_SHADOW_OF_12      a b      drink_machine-1
Q1: REG_SHADOW_OF_11      a b      drink_machine-2
```

Input Signal Placement Report

This table lists each input signal that was placed on an input/clock or I/O pin. Signals that were regionalized via input are also listed. If all the input signals could not be placed, the failure is notes.

Failure-to-Fit Disclaimer

If the PLSyn fitter fails to place all output, node, and input signals in the partitioned design, a disclaimer is printed immediately following the input signal placement report. This lets you know that fitting failed.

If the design fit successfully, a message is printed with the number of functions successfully fit in the device.

---

# The Documentation File

---

## A

### Appendix Overview

This appendix describes the sections of the documentation file that PLSyn creates whenever you try to physically implement a programmable logic design.

# Summary of Documentation File Contents

PLSyn generates a documentation file for the design throughout the physical implementation process. This file is called *design\_name.doc*, by default, and contains the following information:

- Compiler and optimizer run-time options (switch values).
- Reduced design equations.
- Solutions generated for the design.
- Partitioning criteria.
- Pinout diagrams for the chosen implementation.
- A list of possible devices for each architecture in the solution list.
- A wire list.

## To view the documentation file

- 1 In PLSyn, from the File menu, select Examine Doc File.

# Reduced Design Equations

When compiling and optimizing your programmable logic, PLSyn synthesizes the equations which represent the logic thereby creating additional alternative equations. The additional equations give the PLSyn fitter more options when attempting to fit your design. This also means that the .doc file might include equations in addition to those supplied by you in the design source file.

Example: If you specify a JK flip-flop as part of the design, the PLSyn compiler generates equations for all other flip-flop types as well. The synthesized equations are simply logically-equivalent versions of the flip-flop you specified.

## Equation Extensions Used in the .doc File

Table 10-1 lists equation types and the equation extension you might see in the .doc file.

**Table 10-1** *Equation Extensions Used in the .doc File*

<b>.doc File Extension</b>	<b>Description</b>	<b>Example</b>
.XORL*	if $y = a (+) b$ , then $y.xorl = a$ (left side of XOR operation)	Y.XORL
.XORR*	if $y = a (+) b$ , then $y.xorr = b$ (right side of XOR operation)	Y.XORR
.EQN	Combinatorial equation (no CLOCKED_BY on output, biput, or node)	A.EQN
.D	D flip-flop equation	FLOP.D
.J	J flip-flop equation	FLOP.J
.K	K flip-flop equation	FLOP.K
.S	S flip-flop equation	FLOP.S
.R	R flip-flop equation	FLOP.R
.T	T flip-flop equation	FLOP.T

**Table 10-1**    *Equation Extensions Used in the .doc File*

<b>.doc File Extensio n</b>	<b>Description</b>	<b>Example</b>
.CLK	clock equation	X.CLK = /A  OUTPUT x CLOCKED_BY /a
.RESET	reset equation	X.RESET = RST  OUTPUT x CLOCKED_BY /a RESET_BY rst
.PRESET	preset equation	X.PRESET = PRST  OUTPUT x CLOCKED_BY /a PRESET_BY prst
.OE	OE enabled equation	X.OE = OE  OUTPUT x ENABLED_BY oe
.LATCH	latched equation	X.LATCH = LAT1  OUTPUT X LATCHED_BY lat1
.CE	clock-enabled equation	X.CE = CE

\*. The compiler/optimizer may generate an XOR equation, even if none was specified in the original .dsl file. Examples include synthesis from T flops, arithmetic operators .+, and .-., etc.

**Note** *PLSyn can always generate the complemented (DeMorgan) version of the equations. But when the version of an equation is non-complemented, PLSyn might not be able to generate it because of its size.*

## DeMorgan Equations

In addition to the equations listed in the previous table, PLSyn might generate DeMorgan versions of the same equations. These, too, are candidates for device fitting.

In the .doc file, PLSyn marks the DeMorgan version of an equation with a tilde (~) after the equation name.

### Example

Suppose you have declared equations as follows:

```
INPUT a, b, oe;  
OUTPUT or1 ENABLED_BY oe;  
or1 = a + b;
```



After synthesis, PLSyn writes the .doc file equations as follows:

```
OR1.EQN =  A + B;  
.OE =  OE;  
OR1.EQN(~) =  /A * /B;  
.OE(~) =  /OE;
```

## Equation Display

Equations can fall into four categories:

Primary	Equations used to describe the signal.
Synthesized	Equations generated by the compiler/optimizer.
DeMorgan	Complemented equations generated by the compiler/optimizer.
Fit	Form of the equations (primary, synthesized, or the DeMorgan of the two) that PLSyn actually fits into the device.

By default, the .doc file includes either:

- the version of the equation that was used during fitting, or
- the primary equation version if fitting has not yet been done.

## Partitioning Criteria

The Partitioning Criteria section lists the constraints in effect during the partitioning/fitting process.

**Note** *A warning appears in the .doc file if you updated the constraints used during partitioning after PLSyn generated the solutions. This tells you that the partitioning criteria displayed in the .doc file might be incorrect.*

PLSyn writes the partitioning criteria to the .doc file after having created the list of possible devices from the available (.avl) file and the enabled constraints.

## Solutions List

The Solutions List section lists the architectures that the PLSyn fitter found to fit your programmable logic. This is the same information that PLSyn displays in the solutions list in the PLSyn window.

## Fuse Map Files

The Fuse Map Files section associates which fuse maps go to which device for a particular solution. You will only see this section if you ran the Fuse Map Generator command from the Tools menu.

# Pinout Diagrams

The Pinout Diagrams section contains for each device in your chosen implementation either:

- a diagram, for a DIP or CDIP package type, or,
- a pinout table, for all other package types.

that shows the device, the pin types (INPUT, OUTPUT, BIPUT), and an indicator of the signal/pin placement. PLSyn writes this information to the `.doc` file after having completed fitting and partitioning.

## Possible Devices List

When PLSyn generates device solutions, the solution list in the PLSyn window contains architecture names, not manufacturers' names, for devices. The Possible Devices List section in the `.doc` file provides a list of the actual devices that are available for a given architecture.

## Wire List

The Wire List section lists for your chosen implementation, which signals to connect to which pins.

---

# Summary of Files

---

**B**

## Appendix Overview

This appendix describes each of the file types that PLSyn uses.

# Files Used by PLSyn

File Extension	Description	Source
.afb	Database containing compiled logic equations. Used for simulations and as input to the PLSyn optimizer.	PLSyn compiler
.avl	Available parts file. You can copy <code>plsynlib.avl</code> to create a custom available file for your site.	System-installed <code>plsynlib.avl</code> file
.cst	Constraint file. A temporary file used to specify the partitioning constraints and priorities.	PLSyn
.doc	Design documentation file, updated during physical implementation.	PLSyn
.dsl	DSL source code files. If the design is schematic-based, the file named <i>design_name.dsl</i> is reserved for use by PLSyn.	user, schematic-to-DSL translator
.edf	EDIF netlist containing the design's programmable logic.	Schematics
.fb	Database containing optimized logic equations and implementation data.	PLSyn
.j1,.j2, ...	Fuse map files, in JEDEC format.	PLSyn fuse maps generator
.log	PLSyn log/error file, updated during physical implementation.	PLSyn
.npi	PIL file containing a description of the design's fitting/partitioning. Can be used to repeat the implementation on subsequent iterations by copying to the <code>.pi</code> file.	PLSyn, after fuse map is generated
.pi	Physical information file containing optimization, fitting, and partitioning statements. You can customize the <code>.pi</code> file to control the implementation.	For new designs, a copy of <code>default.pi</code> , found in the MicroSim root directory
.sch	Schematic file.	Schematics
.slb	Symbol library file.	Schematics
.plb	Package library file.	Schematics
.tv	Test vectors file.	PSpice/PLogic

---

# AMD MACH Device Tables

---

## C

### Appendix Overview

This appendix contains lookup tables for pin names and fuse commands for AMD MACH device architectures. These are the notations you can use in your `.pi` file.

[Pin Name Tables on page C-2](#) lists the pin reference name for each macrocell in a PAL block.

[MACH 1xx and 2xx: Fuse Commands for Driving Outputs on page C-12](#) lists the fuse commands you can use to force the named pin to be driven.

# Pin Name Tables

The following tables list the reference name for each macrocell in a PAL block.

## To determine the exact name for a pin

Replace the ## characters in the listed Reference Name with the corresponding two digit Macrocell Number.

### MACH 110

Macrocell Number (##)					
Bloc k	Pins	Reference Name	Output	Buried	Input
A	2-9	MACROCELL_A##	00 - 07		
	14-21		08 - 15		
B	24-31	MACROCELL_B##	00 - 07		
	36-43		08 - 15		

### MACH 111, 111SP

Macrocell Number (##)					
Bloc k	Pins	Reference Name	Output	Buried	Input
A	2-9	MACROCELL_A##	00 - 07		
	14-21		08 - 15		
B	24-31	MACROCELL_B##	15 - 08		
	36-43		07 - 00		

**MACH 120, 121**

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	2-7	MACROCELL_A##	00 - 05		
	9-14		06-11		
B	21-26	MACROCELL_B##	11-06		
	28-33		05 - 00		
C	36-41	MACROCELL_C##	00 - 05		
	43-48		06-11		
D	55-60	MACROCELL_D##	11-06		
	62-67		05 - 00		

**MACH 130, 131, 131SP**

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	3-10	MACROCELL_A##	00 - 07		
	12-19		08 - 15		
B	24-31	MACROCELL_B##	15 - 08		
	33-40		07- 00		
C	45-52	MACROCELL_C##	00 - 07		
	54-61		08 - 15		
D	66-73	MACROCELL_D##	15 - 08		
	75-82		07- 00		



MACH 210, 211, 211SP

Bloc k	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	2-9	MACROCELL_A##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
B	14-21	MACROCELL_B##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	
C	21-31	MACROCELL_C##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
D	36-43	MACROCELL_D##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	

**MACH 215**

			Macrocell Number (##)		
Bloc k	Pins	Reference Name	Output	Buried	Input
A	2-9	MACROCELL_A##	00,02,04, 06,08,10, 12,14		
		IN_REG_A##			01,03,05, 07,09,11, 13,15
B	14-21	MACROCELL_B##	14,12,10, 08,06,04, 02,00		
		IN_REG_B##			15,13,11, 09,07,05, 03,00
C	24-31	MACROCELL_C##	00,02,04, 06, 08,10,12, 14		
		IN_REG_C##			01,03,05, 07,09,11, 13,15
D	36-43	MACROCELL_D##	14,12,10, 08,06,04, 02,00		
		IN_REG_D##			15,13,11, 09,07,05, 03,00

MACH 220, 221, 221SP

Bloc k	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	2-7	MACROCELL_A##	00,02,04, 06,08,10	01,03,05, 07,09,11	
B	9-14	MACROCELL_B##	10,08,06, 04,02,00	11,09,07, 05,03,01	
C	21-26	MACROCELL_C##	00,02,04, 06,08,10	01,03,05, 07,09,11	
D	28-33	MACROCELL_D##	10,08,06, 04,02,00	11,09,07, 05,03,01	
E	36-41	MACROCELL_E##	00,02,04, 06,08,10	01,03,05, 07,09,11	
F	43-48	MACROCELL_F##	10,08,06, 04,02,00	11,09,07, 05,03,01	
G	55-60	MACROCELL_G##	00,02,04, 06,08,10	01,03,05, 07,09,11	
H	62-67	MACROCELL_H##	10,08,06, 04,02,00	11,09,07, 05,03,01	

**MACH 230, 231**

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	3-10	MACROCELL_A##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
B	12-19	MACROCELL_B##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	
C	24-31	MACROCELL_C##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
D	33-40	MACROCELL_D##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	
E	45-52	MACROCELL_E##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
F	54-61	MACROCELL_F##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	
G	66-73	MACROCELL_G##	00,02,04, 06,08,10, 12,14	01,03,05, 07,09,11, 13,15	
H	75-82	MACROCELL_H##	14,12,10, 08,06,04, 02,00	15,13,11, 09,07,05, 03,01	

MACH 435, 436

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	3-10	MACROCELL_A##	00 - 15		
		IN_REG_A##			00 - 07
B	12-19	MACROCELL_B##	00 - 15		
		IN_REG_B##			07 - 00
C	24-31	MACROCELL_C##	00 - 15		
		IN_REG_C##			00 - 07
D	33-40	MACROCELL_D##	00 - 15		
		IN_REG_D##			07 - 00
E	45-52	MACROCELL_E##	00 - 15		
		IN_REG_E##			00 - 07
F	45-52	MACROCELL_F##	00 - 15		
		IN_REG_F##			07 - 00
G	66-73	MACROCELL_G##	00 - 15		
		IN_REG_G##			00 - 07
H	75-82	MACROCELL_H##	00 - 15		
		IN_REG_H##			07 - 00

**MACH 445, 446**

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
B	5-12	MACROCELL_B##	00 - 15		
		IN_REG_B##			07 - 00
C	19-26	MACROCELL_C##	00 - 15		
		IN_REG_C##			00 - 07
D	31-38	MACROCELL_D##	00 - 15		
		IN_REG_D##			07 - 00
E	43-50	MACROCELL_E##	00 - 15		
		IN_REG_E##			00 - 07
F	55-62	MACROCELL_F##	00 - 15		
		IN_REG_F##			07 - 00
G	69-76	MACROCELL_G##	00 - 15		
		IN_REG_G##			00 - 07
H	81-88	MACROCELL_H##	00 - 15		
		IN_REG_H##			07 - 00
A	93-100	MACROCELL_G##	00 - 15		
		IN_REG_G##			00 - 07

MACH 465, 466

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
C	3-10	MACROCELL_C##	00 - 15		
		IN_REG_C##			07 - 00
D	13-20	MACROCELL_D##	00 - 15		
		IN_REG_D##			07 - 00
E	32-39	MACROCELL_E##	00 - 15		
		IN_REG_E##			00 - 07
F	42-49	MACROCELL_F##	00 - 15		
		IN_REG_F##			00 - 07
G	54-61	MACROCELL_G##	00 - 15		
		IN_REG_G##			07 - 00
H	64-71	MACROCELL_H##	00 - 15		
		IN_REG_H##			07 - 00
I	86-93	MACROCELL_I##	00 - 15		
		IN_REG_I##			00 - 07
J	96-103	MACROCELL_J##	00 - 15		
		IN_REG_J##			00 - 07
K	107-114	MACROCELL_K##	00 - 15		
		IN_REG_K##			07 - 00
L	117-124	MACROCELL_L##	00 - 15		
		IN_REG_L##			07 - 00
M	136-143	MACROCELL_M##	00 - 15		
		IN_REG_M##			00 - 07
N	146-153	MACROCELL_N##	00 - 15		
		IN_REG_N##			00 - 07
O	158-165	MACROCELL_O##	00 - 15		
		IN_REG_O##			07 - 00
P	168-175	MACROCELL_P##	00 - 15		
		IN_REG_P##			07 - 00

Block	Pins	Reference Name	Macrocell Number (##)		
			Output	Buried	Input
A	190-197	MACROCELL_A##	00 - 15		
		IN_REG_A##			00 - 07
B	200-207	MACROCELL_B##	00 - 15		
		IN_REG_B##			00 - 07



# MACH 1xx and 2xx: Fuse Commands for Driving Outputs

The following tables give the fuse commands for the .pi file to force the named pin to be driven.

## MACH 110

**Table 10-2**    *MACH 110 OE Fuse Commands*

Pin 02:	INTACT 6166 ;	BLOWN 6167 ;
Pin 03:	INTACT 6174 ;	BLOWN 6175 ;
Pin 04:	INTACT 6182 ;	BLOWN 6183 ;
Pin 05:	INTACT 6190 ;	BLOWN 6191 ;
Pin 06:	INTACT 6198 ;	BLOWN 6199 ;
Pin 07:	INTACT 6206 ;	BLOWN 6207 ;
Pin 08:	INTACT 6214 ;	BLOWN 6215 ;
Pin 09:	INTACT 6222 ;	BLOWN 6223 ;
Pin 14:	INTACT 6230 ;	BLOWN 6231 ;
Pin 15:	INTACT 6238 ;	BLOWN 6239 ;
Pin 16:	INTACT 6246 ;	BLOWN 6247 ;
Pin 17:	INTACT 6254 ;	BLOWN 6255 ;
Pin 18:	INTACT 6262 ;	BLOWN 6263 ;
Pin 19:	INTACT 6270 ;	BLOWN 6271 ;
Pin 20:	INTACT 6278 ;	BLOWN 6279 ;
Pin 21:	INTACT 6286 ;	BLOWN 6287 ;
Pin 24:	INTACT 6294 ;	BLOWN 6295 ;
Pin 25:	INTACT 6302 ;	BLOWN 6303 ;
Pin 26:	INTACT 6310 ;	BLOWN 6311 ;
Pin 27:	INTACT 6318 ;	BLOWN 6319 ;
Pin 28:	INTACT 6326 ;	BLOWN 6327 ;
Pin 29:	INTACT 6334 ;	BLOWN 6335 ;
Pin 30:	INTACT 6342 ;	BLOWN 6343 ;
Pin 31:	INTACT 6350 ;	BLOWN 6351 ;
Pin 36:	INTACT 6358 ;	BLOWN 6359 ;
Pin 37:	INTACT 6366 ;	BLOWN 6367 ;

**Table 10-2** *MACH 110 OE Fuse Commands*

Pin 38:	INTACT 6374 ;	BLOWN 6375 ;
Pin 39:	INTACT 6382 ;	BLOWN 6383 ;
Pin 40:	INTACT 6390 ;	BLOWN 6391 ;
Pin 41:	INTACT 6398 ;	BLOWN 6399 ;
Pin 42:	INTACT 6406 ;	BLOWN 6407 ;
Pin 43:	INTACT 6414 ;	BLOWN 6415 ;

**MACH 120****Table 10-3** *MACH 120 OE Fuse Commands*

Pin 02:	INTACT 2918 ;	BLOWN 2919 ;
Pin 03:	INTACT 2927 ;	BLOWN 2928 ;
Pin 04:	INTACT 2936 ;	BLOWN 2937 ;
Pin 05:	INTACT 2945 ;	BLOWN 2946 ;
Pin 06:	INTACT 2954 ;	BLOWN 2955 ;
Pin 07:	INTACT 2963 ;	BLOWN 2964 ;
Pin 09:	INTACT 2972 ;	BLOWN 2973 ;
Pin 10:	INTACT 2981 ;	BLOWN 2982 ;
Pin 11:	INTACT 2990 ;	BLOWN 2991 ;
Pin 12:	INTACT 2999 ;	BLOWN 3000 ;
Pin 13:	INTACT 3008 ;	BLOWN 3009 ;
Pin 14:	INTACT 3017 ;	BLOWN 3018 ;
Pin 21:	INTACT 6037 ;	BLOWN 6038 ;
Pin 22:	INTACT 6028 ;	BLOWN 6029 ;
Pin 23:	INTACT 6019 ;	BLOWN 6020 ;
Pin 24:	INTACT 6010 ;	BLOWN 6011 ;
Pin 25:	INTACT 6001 ;	BLOWN 6002 ;
Pin 26:	INTACT 5992 ;	BLOWN 5993 ;
Pin 28:	INTACT 5983 ;	BLOWN 5984 ;
Pin 29:	INTACT 5974 ;	BLOWN 5975 ;
Pin 30:	INTACT 5965 ;	BLOWN 5966 ;
Pin 31:	INTACT 5956 ;	BLOWN 5957 ;
Pin 32:	INTACT 5947 ;	BLOWN 5948 ;
Pin 33:	INTACT 5938 ;	BLOWN 5939 ;
Pin 36:	INTACT 8958 ;	BLOWN 8959 ;
Pin 37:	INTACT 8967 ;	BLOWN 8968 ;

**Table 10-3** *MACH 120 OE Fuse Commands (continued)*

Pin 38:	INTACT 8976 ;	BLOWN 8977 ;
Pin 39:	INTACT 8985 ;	BLOWN 8986 ;
Pin 40:	INTACT 8994 ;	BLOWN 8995 ;
Pin 41:	INTACT 9003 ;	BLOWN 9004 ;
Pin 43:	INTACT 9012 ;	BLOWN 9013 ;
Pin 44:	INTACT 9021 ;	BLOWN 9022 ;
Pin 45:	INTACT 9030 ;	BLOWN 9031 ;
Pin 46:	INTACT 9039 ;	BLOWN 9040 ;
Pin 47:	INTACT 9048 ;	BLOWN 9049 ;
Pin 48:	INTACT 9057 ;	BLOWN 9058 ;
Pin 55:	INTACT 12077 ;	BLOWN 12078 ;
Pin 56:	INTACT 12068 ;	BLOWN 12069 ;
Pin 57:	INTACT 12059 ;	BLOWN 12060 ;
Pin 58:	INTACT 12050 ;	BLOWN 12051 ;
Pin 59:	INTACT 12041 ;	BLOWN 12042 ;
Pin 60:	INTACT 12032 ;	BLOWN 12033 ;
Pin 62:	INTACT 12023 ;	BLOWN 12024 ;
Pin 63:	INTACT 12014 ;	BLOWN 12015 ;
Pin 64:	INTACT 12005 ;	BLOWN 12006 ;
Pin 65:	INTACT 11996 ;	BLOWN 11997 ;
Pin 66:	INTACT 11987 ;	BLOWN 11988 ;
Pin 67:	INTACT 11978 ;	BLOWN 11979 ;

**MACH 130****Table 10-4** *MACH 130 OE Fuse Commands*

Pin 03:	INTACT 3750 ;	BLOWN 3751 ;
Pin 04:	INTACT 3759 ;	BLOWN 3760 ;
Pin 05:	INTACT 3768 ;	BLOWN 3769 ;
Pin 06:	INTACT 3777 ;	BLOWN 3778 ;
Pin 07:	INTACT 3786 ;	BLOWN 3787 ;
Pin 08:	INTACT 3795 ;	BLOWN 3796 ;
Pin 09:	INTACT 3804 ;	BLOWN 3805 ;
Pin 10:	INTACT 3813 ;	BLOWN 3814 ;
Pin 12:	INTACT 3822 ;	BLOWN 3823 ;
Pin 13:	INTACT 3831 ;	BLOWN 3832 ;

**Table 10-4** *MACH 130 OE Fuse Commands (continued)*

Pin 14:	INTACT 3840 ;	BLOWN 3841 ;
Pin 15:	INTACT 3849 ;	BLOWN 3850 ;
Pin 16:	INTACT 3858 ;	BLOWN 3859 ;
Pin 17:	INTACT 3867 ;	BLOWN 3868 ;
Pin 18:	INTACT 3876 ;	BLOWN 3877 ;
Pin 19:	INTACT 3885 ;	BLOWN 3886 ;
Pin 24:	INTACT 7773 ;	BLOWN 7774 ;
Pin 25:	INTACT 7764 ;	BLOWN 7765 ;
Pin 26:	INTACT 7755 ;	BLOWN 7756 ;
Pin 27:	INTACT 7746 ;	BLOWN 7747 ;
Pin 28:	INTACT 7737 ;	BLOWN 7738 ;
Pin 29:	INTACT 7728 ;	BLOWN 7729 ;
Pin 30:	INTACT 7719 ;	BLOWN 7720 ;
Pin 31:	INTACT 7710 ;	BLOWN 7711 ;
Pin 33:	INTACT 7701 ;	BLOWN 7702 ;
Pin 34:	INTACT 7692 ;	BLOWN 7693 ;
Pin 35:	INTACT 7683 ;	BLOWN 7684 ;
Pin 36:	INTACT 7674 ;	BLOWN 7675 ;
Pin 37:	INTACT 7665 ;	BLOWN 7666 ;
Pin 38:	INTACT 7656 ;	BLOWN 7657 ;
Pin 39:	INTACT 7647 ;	BLOWN 7648 ;
Pin 40:	INTACT 7638 ;	BLOWN 7639 ;
Pin 45:	INTACT 11526 ;	BLOWN 11527 ;
Pin 46:	INTACT 11535 ;	BLOWN 11536 ;
Pin 47:	INTACT 11544 ;	BLOWN 11545 ;
Pin 48:	INTACT 11553 ;	BLOWN 11554 ;
Pin 49:	INTACT 11562 ;	BLOWN 11563 ;
Pin 50:	INTACT 11571 ;	BLOWN 11572 ;
Pin 51:	INTACT 11580 ;	BLOWN 11581 ;
Pin 52:	INTACT 11589 ;	BLOWN 11590 ;
Pin 54:	INTACT 11598 ;	BLOWN 11599 ;
Pin 55:	INTACT 11607 ;	BLOWN 11608 ;
Pin 56:	INTACT 11616 ;	BLOWN 11617 ;
Pin 57:	INTACT 11625 ;	BLOWN 11626 ;
Pin 58:	INTACT 11634 ;	BLOWN 11635 ;
Pin 59:	INTACT 11643 ;	BLOWN 11644 ;

**Table 10-4** *MACH 130 OE Fuse Commands (continued)*

Pin 60:	INTACT 11652 ;	BLOWN 11653 ;
Pin 61:	INTACT 11661 ;	BLOWN 11662 ;
Pin 66:	INTACT 15549 ;	BLOWN 15550 ;
Pin 67:	INTACT 15540 ;	BLOWN 15541 ;
Pin 68:	INTACT 15531 ;	BLOWN 15532 ;
Pin 69:	INTACT 15522 ;	BLOWN 15523 ;
Pin 70:	INTACT 15513 ;	BLOWN 15514 ;
Pin 71:	INTACT 15504 ;	BLOWN 15505 ;
Pin 72:	INTACT 15495 ;	BLOWN 15496 ;
Pin 73:	INTACT 15486 ;	BLOWN 15487 ;
Pin 75:	INTACT 15477 ;	BLOWN 15478 ;
Pin 76:	INTACT 15468 ;	BLOWN 15469 ;
Pin 77:	INTACT 15459 ;	BLOWN 15460 ;
Pin 78:	INTACT 15450 ;	BLOWN 15451 ;
Pin 79:	INTACT 15441 ;	BLOWN 15442 ;
Pin 80:	INTACT 15432 ;	BLOWN 15433 ;
Pin 81:	INTACT 15423 ;	BLOWN 15424 ;
Pin 82:	INTACT 15414 ;	BLOWN 15415 ;

**MACH 210****Table 10-5** *MACH 210 OE Fuse Commands*

Pin 02:	INTACT 3086 ;	BLOWN 3087 ;
Pin 03:	INTACT 3102 ;	BLOWN 3103 ;
Pin 04:	INTACT 3118 ;	BLOWN 3119 ;
Pin 05:	INTACT 3134 ;	BLOWN 3135 ;
Pin 06:	INTACT 3150 ;	BLOWN 3151 ;
Pin 07:	INTACT 3166 ;	BLOWN 3167 ;
Pin 08:	INTACT 3182 ;	BLOWN 3183 ;
Pin 09:	INTACT 3198 ;	BLOWN 3199 ;
Pin 14:	INTACT 6406 ;	BLOWN 6407 ;
Pin 15:	INTACT 6390 ;	BLOWN 6391 ;
Pin 16:	INTACT 6374 ;	BLOWN 6375 ;
Pin 17:	INTACT 6358 ;	BLOWN 6359 ;
Pin 18:	INTACT 6342 ;	BLOWN 6343 ;
Pin 19:	INTACT 6326 ;	BLOWN 6327 ;

**Table 10-5** *MACH 210 OE Fuse Commands (continued)*

Pin 20:	INTACT 6310 ;	BLOWN 6311 ;
Pin 21:	INTACT 6294 ;	BLOWN 6295 ;
Pin 24:	INTACT 9502 ;	BLOWN 9503 ;
Pin 25:	INTACT 9518 ;	BLOWN 9519 ;
Pin 26:	INTACT 9534 ;	BLOWN 9535 ;
Pin 27:	INTACT 9550 ;	BLOWN 9551 ;
Pin 28:	INTACT 9566 ;	BLOWN 9567 ;
Pin 29:	INTACT 9582 ;	BLOWN 9583 ;
Pin 30:	INTACT 9598 ;	BLOWN 9599 ;
Pin 31:	INTACT 9614 ;	BLOWN 9615 ;
Pin 36:	INTACT 12822 ;	BLOWN 12823 ;
Pin 37:	INTACT 12806 ;	BLOWN 12807 ;
Pin 38:	INTACT 12790 ;	BLOWN 12791 ;
Pin 39:	INTACT 12774 ;	BLOWN 12775 ;
Pin 40:	INTACT 12758 ;	BLOWN 12759 ;
Pin 41:	INTACT 12742 ;	BLOWN 12743 ;
Pin 42:	INTACT 12726 ;	BLOWN 12727 ;
Pin 43:	INTACT 12710 ;	BLOWN 12711 ;

**MACH 215****Table 10-6** *MACH 215 OE Fuse Commands*

Pin 02:	BLOWN 88 .. 131 ;
Pin 03:	BLOWN 440 .. 483 ;
Pin 04:	BLOWN 792 .. 835 ;
Pin 05:	BLOWN 1144 .. 1187 ;
Pin 06:	BLOWN 1496 .. 1539 ;
Pin 07:	BLOWN 1848 .. 1891 ;
Pin 08:	BLOWN 2200 .. 2243 ;
Pin 09:	BLOWN 2552 .. 2595 ;
Pin 14:	BLOWN 5536 .. 5579 ;
Pin 15:	BLOWN 5184 .. 5227 ;
Pin 16:	BLOWN 4832 .. 4875 ;
Pin 17:	BLOWN 4480 .. 4523 ;
Pin 18:	BLOWN 4128 .. 4171 ;
Pin 19:	BLOWN 3776 .. 3819 ;

**Table 10-6**    *MACH 215 OE Fuse Commands (continued)*

Pin 20:	BLOWN 3424 .. 3467 ;
Pin 21:	BLOWN 3072 .. 3115 ;
Pin 24:	BLOWN 6056 .. 6099 ;
Pin 25:	BLOWN 6408 .. 6451 ;
Pin 26:	BLOWN 6760 .. 6803 ;
Pin 27:	BLOWN 7112 .. 7155 ;
Pin 28:	BLOWN 7464 .. 7507 ;
Pin 29:	BLOWN 7816 .. 7859 ;
Pin 30:	BLOWN 8168 .. 8211 ;
Pin 31:	BLOWN 8520 .. 8563 ;
Pin 36:	BLOWN 11504 .. 11547 ;
Pin 37:	BLOWN 11152 .. 11195 ;
Pin 38:	BLOWN 10800 .. 10843 ;
Pin 39:	BLOWN 10448 .. 10491 ;
Pin 40:	BLOWN 10096 .. 10139 ;
Pin 41:	BLOWN 9744 .. 9787 ;
Pin 42:	BLOWN 9392 .. 9435 ;
Pin 43:	BLOWN 9040 .. 9083 ;

**MACH 220**

**Table 10-7**    *MACH 220 OE Fuse Commands*

Pin 02:	INTACT 2814 ;	BLOWN 2815 ;
Pin 03:	INTACT 2830 ;	BLOWN 2831 ;
Pin 04:	INTACT 2846 ;	BLOWN 2847 ;
Pin 05:	INTACT 2862 ;	BLOWN 2863 ;
Pin 06:	INTACT 2878 ;	BLOWN 2879 ;
Pin 07:	INTACT 2894 ;	BLOWN 2895 ;
Pin 09:	INTACT 5798 ;	BLOWN 5799 ;
Pin 10:	INTACT 5782 ;	BLOWN 5783 ;
Pin 11:	INTACT 5766 ;	BLOWN 5767 ;
Pin 12:	INTACT 5750 ;	BLOWN 5751 ;
Pin 13:	INTACT 5734 ;	BLOWN 5735 ;
Pin 14:	INTACT 5718 ;	BLOWN 5719 ;
Pin 21:	INTACT 8622 ;	BLOWN 8623 ;
Pin 22:	INTACT 8638 ;	BLOWN 8639 ;

**Table 10-7** *MACH 220 OE Fuse Commands (continued)*

Pin 23:	INTACT 8654 ;	BLOWN 8655 ;
Pin 24:	INTACT 8670 ;	BLOWN 8671 ;
Pin 25:	INTACT 8686 ;	BLOWN 8687 ;
Pin 26:	INTACT 8702 ;	BLOWN 8703 ;
Pin 28:	INTACT 11606 ;	BLOWN 11607 ;
Pin 29:	INTACT 11590 ;	BLOWN 11591 ;
Pin 30:	INTACT 11574 ;	BLOWN 11575 ;
Pin 31:	INTACT 11558 ;	BLOWN 11559 ;
Pin 32:	INTACT 11542 ;	BLOWN 11543 ;
Pin 33:	INTACT 11526 ;	BLOWN 11527 ;
Pin 36:	INTACT 14430 ;	BLOWN 14431 ;
Pin 37:	INTACT 14446 ;	BLOWN 14447 ;
Pin 38:	INTACT 14462 ;	BLOWN 14463 ;
Pin 39:	INTACT 14478 ;	BLOWN 14479 ;
Pin 40:	INTACT 14494 ;	BLOWN 14495 ;
Pin 41:	INTACT 14510 ;	BLOWN 14511 ;
Pin 43:	INTACT 17414 ;	BLOWN 17415 ;
Pin 44:	INTACT 17398 ;	BLOWN 17399 ;
Pin 45:	INTACT 17382 ;	BLOWN 17383 ;
Pin 46:	INTACT 17366 ;	BLOWN 17367 ;
Pin 47:	INTACT 17350 ;	BLOWN 17351 ;
Pin 48:	INTACT 17334 ;	BLOWN 17335 ;
Pin 55:	INTACT 20238 ;	BLOWN 20239 ;
Pin 56:	INTACT 20254 ;	BLOWN 20255 ;
Pin 57:	INTACT 20270 ;	BLOWN 20271 ;
Pin 58:	INTACT 20286 ;	BLOWN 20287 ;
Pin 59:	INTACT 20302 ;	BLOWN 20303 ;
Pin 60:	INTACT 20318 ;	BLOWN 20319 ;
Pin 62:	INTACT 23222 ;	BLOWN 23223 ;



**MACH 230****Table 10-8** *MACH 230 OE Fuse Commands*

Pin 03:	INTACT 3646 ;	BLOWN 3647 ;
Pin 04:	INTACT 3662 ;	BLOWN 3663 ;
Pin 05:	INTACT 3678 ;	BLOWN 3679 ;
Pin 06:	INTACT 3694 ;	BLOWN 3695 ;
Pin 07:	INTACT 3710 ;	BLOWN 3711 ;
Pin 08:	INTACT 3726 ;	BLOWN 3727 ;
Pin 09:	INTACT 3742 ;	BLOWN 3743 ;
Pin 10:	INTACT 3758 ;	BLOWN 3759 ;
Pin 12:	INTACT 7526 ;	BLOWN 7527 ;
Pin 13:	INTACT 7510 ;	BLOWN 7511 ;
Pin 14:	INTACT 7494 ;	BLOWN 7495 ;
Pin 15:	INTACT 7478 ;	BLOWN 7479 ;
Pin 16:	INTACT 7462 ;	BLOWN 7463 ;
Pin 17:	INTACT 7446 ;	BLOWN 7447 ;
Pin 18:	INTACT 7430 ;	BLOWN 7431 ;
Pin 19:	INTACT 7414 ;	BLOWN 7415 ;
Pin 24:	INTACT 11182 ;	BLOWN 11183 ;
Pin 25:	INTACT 11198 ;	BLOWN 11199 ;
Pin 26:	INTACT 11214 ;	BLOWN 11215 ;
Pin 27:	INTACT 11230 ;	BLOWN 11231 ;
Pin 28:	INTACT 11246 ;	BLOWN 11247 ;
Pin 29:	INTACT 11262 ;	BLOWN 11263 ;
Pin 30:	INTACT 11278 ;	BLOWN 11279 ;
Pin 31:	INTACT 11294 ;	BLOWN 11295 ;
Pin 33:	INTACT 15062 ;	BLOWN 15063 ;
Pin 34:	INTACT 15046 ;	BLOWN 15047 ;
Pin 35:	INTACT 15030 ;	BLOWN 15031 ;
Pin 36:	INTACT 15014 ;	BLOWN 15015 ;
Pin 37:	INTACT 14998 ;	BLOWN 14999 ;
Pin 38:	INTACT 14982 ;	BLOWN 14983 ;
Pin 39:	INTACT 14966 ;	BLOWN 14967 ;
Pin 40:	INTACT 14950 ;	BLOWN 14951 ;
Pin 45:	INTACT 18718 ;	BLOWN 18719 ;
Pin 46:	INTACT 18734 ;	BLOWN 18735 ;
Pin 47:	INTACT 18750 ;	BLOWN 18751 ;

**Table 10-8** *MACH 230 OE Fuse Commands (continued)*

Pin 48:	INTACT 18766 ;	BLOWN 18767 ;
Pin 49:	INTACT 18782 ;	BLOWN 18783 ;
Pin 50:	INTACT 18798 ;	BLOWN 18799 ;
Pin 51:	INTACT 18814 ;	BLOWN 18815 ;
Pin 52:	INTACT 18830 ;	BLOWN 18831 ;
Pin 54:	INTACT 22598 ;	BLOWN 22599 ;
Pin 55:	INTACT 22582 ;	BLOWN 22583 ;
Pin 56:	INTACT 22566 ;	BLOWN 22567 ;
Pin 57:	INTACT 22550 ;	BLOWN 22551 ;
Pin 58:	INTACT 22534 ;	BLOWN 22535 ;
Pin 59:	INTACT 22518 ;	BLOWN 22519 ;
Pin 60:	INTACT 22502 ;	BLOWN 22503 ;
Pin 61:	INTACT 22486 ;	BLOWN 22487 ;
Pin 66:	INTACT 26254 ;	BLOWN 26255 ;
Pin 67:	INTACT 26270 ;	BLOWN 26271 ;
Pin 68:	INTACT 26286 ;	BLOWN 26287 ;
Pin 69:	INTACT 26302 ;	BLOWN 26303 ;
Pin 70:	INTACT 26318 ;	BLOWN 26319 ;
Pin 71:	INTACT 26334 ;	BLOWN 26335 ;
Pin 72:	INTACT 26350 ;	BLOWN 26351 ;
Pin 73:	INTACT 26366 ;	BLOWN 26367 ;
Pin 75:	INTACT 30134 ;	BLOWN 30135 ;
Pin 76:	INTACT 30118 ;	BLOWN 30119 ;
Pin 77:	INTACT 30102 ;	BLOWN 30103 ;
Pin 78:	INTACT 30086 ;	BLOWN 30087 ;
Pin 79:	INTACT 30070 ;	BLOWN 30071 ;
Pin 80:	INTACT 30054 ;	BLOWN 30055 ;
Pin 81:	INTACT 30038 ;	BLOWN 30039 ;
Pin 82:	INTACT 30022 ;	BLOWN 30023 ;

---

# Index

---

## Symbols

.afb, B-2  
.avl, 5-16, B-2  
.cst, B-2  
.doc, A-3, B-2  
    MACH5, 9-21  
.dsl, 3-4, B-2  
.edf, B-2  
.fb, B-2  
.lg, B-2  
.npi, 5-28, B-2  
.pi, B-2  
.plb, B-2  
.sch, B-2  
.slb, B-2  
.tv, B-2

## A

A/D interface, 4-4  
active-low nodes, 3-15  
architecture constraint, 5-18  
available file, 5-16  
Available File text box, 5-20

## B

back annotation, 5-28  
back annotation (schematic), 5-28  
blocks  
    creating DSL, 3-6

## C

changing designs with PLDs, 5-30  
Compile Library command, 5-8  
Compiler command, 5-7  
compiling, 1-3, 5-7, 5-8  
    command, 5-7  
    Create Nodes option, 5-9  
    Output Warnings option, 5-8  
    Product Term option, 5-9  
constants, 3-16  
constraints, 1-4  
    architecture, 5-18  
    available file, 5-20  
    current usage, 5-19  
    device template, 5-18  
    frequency, 5-19  
    logic family, 5-18  
    manufacturer, 5-18

- number of devices, 5-20
- package type, 5-18
- propagation delay, 5-19
- setting up, 5-18
- temperature, 5-18
- user-defined, 5-20

converting nodes, 3-15

Create Nodes check box, 5-9

creating

- DSL blocks, 3-6
- fuse maps, 5-27
- PCB netlists, 5-29

current constraint, 5-19

## D

defining pin names, 3-6

DeMorgan

- equations, A-4
- optimization method, 5-11

design flow, 1-2

device

- accessing internal points, 7-2
- constraints, 5-18
- maximum number, 5-20
- programming, 1-5
- selection, 1-5, 5-26

Device Templates button, 5-19

device templates constraint, 5-18

dig\_prim.lib, 3-2

dig\_prim.slb, 3-2

directed partitioning

- 16V8HD, 22VP10, and 16VP10 devices, 7-17
- AMD MACH devices, 8-2, 9-5
- controlling equation size, 6-10
- FIT\_AS\_OUTPUT property, 6-6
- fitting signals together, 6-13
- fitting to a single device, 6-16
- fitting to multiple devices, 6-17
- maintaining pin assignments, 6-15
- mixing automatic and directed modes, 6-17
- P1800 devices, 7-16
- PLD utilization, 6-5
- specifying devices, 6-14
- specifying footprints, 6-18
- synthesis control properties, 6-9

document file

- MACH5, 9-21
- reduced design equations, A-3

don't care generation, 5-11

DSL blocks, 3-4

- changing the interface, 3-8
- creating, 3-6
- placing, 3-5
- procedures, 3-5

DSL Model Editor, 3-7

## E

editing a DSL Model, 3-7

equation

- display, A-5
- extensions, A-3

exclusive-OR synthesis, 5-12

## F

FANOUTS property (MACH5), 9-13

file extensions, B-1

fitting, 1-4

- introduction, 5-14
- starting, 5-25

FLOAT\_NODES property, 8-34

FORCE\_INTERNAL\_FB property, 8-38

FORCE\_LOCAL\_FB property (MACH5), 9-11

frequency

- constraint, 5-19
- priority, 5-24

Fuse Map Generator command, 5-27

fuse maps

- creating, 5-27
- Fuse Map Generator command, 5-27

## G

generic logic symbols, 3-2

## H

HI symbol, 3-16

hidden node, 7-2

HIGH property (MACH5), 9-19

## I

I/O models, 4-5

I/O parameters, 4-5  
 INCLUDE statement, 3-12  
 interface nodes  
   naming & labeling, 3-14  
 internal nodes, 3-13

## J

JEDEC file, 4-6, 6-12, B-2

## L

labeling interface nodes, 3-14  
 library, 5-7, 5-8  
 LO symbol, 3-16  
 LOCAL\_TOGGLE\_FEEDBACK property (MACH5), 9-14  
 logic  
   constants, 3-16  
   minimization, 5-12  
   symbols, 3-2  
 Logic Family button, 5-19  
 logic family constraints, 5-18  
 LOW property (MACH5), 9-19  
 LOW\_TRUE port, 3-15

## M

MACH\_UTILIZATION property (MACH5), 9-16  
 MACH\_ZERO\_HOLD\_INPUT property, 8-47  
 Manufacturer button, 5-19  
 manufacturer constraints, 5-18  
 markers, 4-11  
 Max Current Usage text box, 5-19  
 Max Devices text box, 5-20  
 Max Frequency text box, 5-19  
 Max Prop Delay text box, 5-19  
 MED\_HIGH property (MACH5), 9-19  
 MED\_LOW property (MACH5), 9-19

## N

naming restrictions, 3-16  
 Navigate/Push command, 3-6  
 netlist  
   PCB, 5-29  
 node collapsing, 5-12  
 node naming restrictions, 3-14

nodes  
   active-low, 3-15  
 non-programmable logic, 1-3  
 Number of Pins priority, 5-23

## O

optimization, 5-10  
   command, 5-10  
   DeMorganization, 5-11  
   don't care generation, 5-11  
   exclusive-OR, 5-12  
   logic minimization, 5-12  
   node collapsing, 5-12  
   register synthesis, 5-11  
   selecting method, 5-13  
   XOR synthesis, 5-12  
 Optimizer command, 5-10  
 Options command, 5-7, 5-8, 5-9  
   compiler options  
     Create Nodes text box, 5-9  
     Output Warnings checkbox, 5-8  
     Product Term text box, 5-9  
   optimizer options  
     Optimization Method list, 5-13

## P

Package Type button, 5-19  
 package type constraints, 5-18  
 parameter  
   Edit Parameter dialog box controls, 5-19, 5-23  
 partitioning, 1-4  
   criteria, A-6  
   introduction, 5-14  
   starting, 5-25  
 PCB netlists  
   creating, 5-29  
 physical nodes, 3-15  
 PIL (Physical Implementation Language), 6-2  
 pin naming, 3-6  
 pinout  
   diagrams, A-7  
   preserving, 8-27  
 PLDs  
   change designs with, 5-30  
   designing with, 1-3, 3-1  
   simulation, 1-3  
   symbols, 3-2

PLogic, 4-5

PLSyn

- design flow, 1-2
- product overview, xviii
- standard features, xxiii
- starting, 5-5

plsynlib.avl, 5-16

Price priority, 5-23

priorities, 1-4, 5-23

- frequency, 5-24
- number of pins, 5-23
- price, 5-23
- propagation delay, 5-24
- size, 5-24
- supply current, 5-24
- user-defined, 5-24

Probe markers

- using, 4-11

procedures

- DSL block, 3-5

product overview, xviii

Product Term text box, 5-9

programmable logic

- design methods, 1-2
- designing with, 3-1
- Interface nodes, 3-14
- internal nodes, 3-13
- node name restrictions, 3-14
- simulation, 4-1

Prop Delay priority, 5-24

propagation delay constraint, 5-19

## R

reduced design equations, A-3

register synthesis, 5-11

## S

schematic

- back-annotation page, 5-28

selecting devices, 5-26

shadow node, 7-3

SIGNATURE property, 8-48

simulation, 1-3

- A/D interface, 4-4
- setting up and starting, 4-3
- test vectors, 4-6
- timing, 1-5, 4-6

with programmable logic, 4-1

Size priority, 5-24

SLEW\_RATE property (MACH5), 9-20

solutions list, A-6

source code

- using existing DSL source, 3-9

starting PLSyn, 5-5

supply current priority, 5-24

symbols

- 74xx series, 3-3
- generic logic, 3-2
- PLDs, 1-3, 3-2

## T

Temperature button, 5-19

temperature constraints, 5-18

test vectors, 4-6

timing simulations, 1-5, 4-6

Tools menu

- Compile Library command, 5-8
- Compiler command, 5-7
- Fuse Map Generator command, 5-27
- Optimizer command, 5-10
- Options command, 5-7, 5-8, 5-9
- Update Schematic command, 5-28

## U

unary node, 7-4

Update Schematic command, 5-28

updating the schematic, 5-28

USE statement, 3-12

User 1 priority, 5-24

User 1 text box, 5-20

User 2 priority, 5-24

User 2 text box, 5-20

using Probe markers, 4-11

## W

wire list, A-7

## X

XOR synthesis, 5-12