AMD

MACHXL® Software User's Guide

1995

MACHXL® Software User's Guide

1995

Advanced
Micro
Devices

MACHXL Software User's Guide

P.O. Box 3453
Sunnyvale, CA 94088

TEL:408-732-2400
TWX: 910339-9280
TELEX: 34-6306
TOLL FREE: 800-538-8450

APPLICATIONS HOTLINE:
800-222-9323 (US)
44-(0)-256-811101 (UK)
0590-8621 (France)
0130-813875 (Germany)
1678-77224 (Italy)

Document . evision 3.0
Published July, 1995. Printed in U.S.A.

Contents

Chapter 3 Windows Interface

1 Brushing Up on PLDs/CPLDs

Contents

PLD Architectures

Given the number of families and their different architectures, a single chapter would be insufficient to cover all the necessary data. So, instead of trying to present a brief subset of the information, we suggest you refer to AMD's PLD/CPLD data book for complete documentation about AMD's families of programmable devices.

2 About MACHXL

Contents

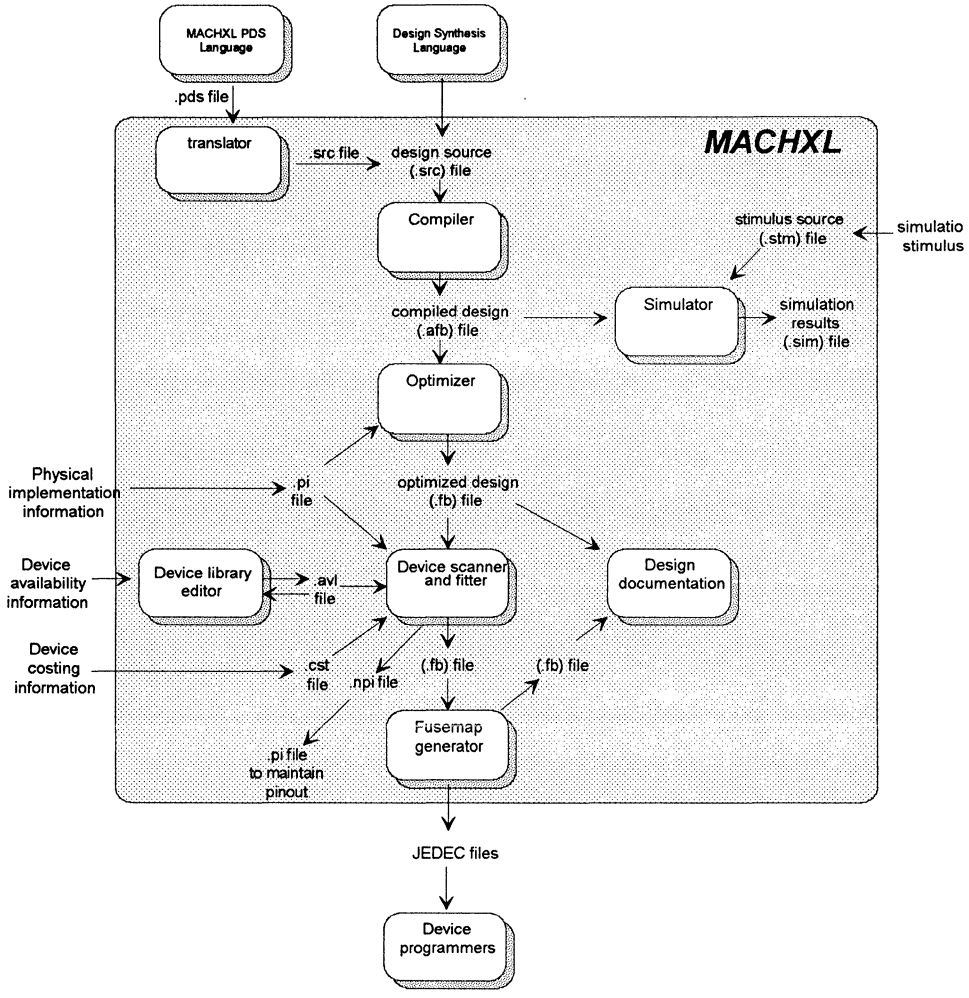
MACHXL Overview and Architecture

MACHXL is a complete, universal programmable logic device (PLD, CPLD) development tool that enables you to program PLDs for specialized applications simply and efficiently. You determine what to design and MACHXL supports you by:

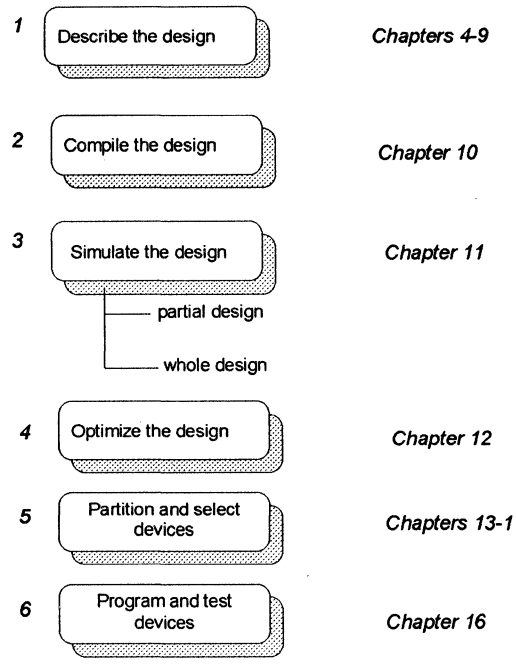
- Allowing you to describe a design using the most suitable method for your application–state machine, truth table, or equations.
- Optimizing and reducing a design to the smallest set of gates, using industry-standard methods best utilizing the resources of selected devices.
- Simulating functionality of your design while it is still in the design phase, before committing to hardware.
- Automatically selecting devices based on your design criteria. MACHXL maps your design into various device architectures and presents the best solutions from which to choose.
- Automatic or manual placement of input and output signals in selected programmable devices including fitting the design across as many as 20 devices (optional).
- Programming the devices using automatic fusemap generation and easy device programmer communications.
- Testing the programmable device(s) by generating test vectors from the functional simulator's results and downloading them to the programmer with the fusemap file.
- Prototyping ASICs using programmable devices.

MACHXL provides full automated support, supplying a design environment allowing you to concentrate on your design, not on the device. As a matter of fact, you do not need to understand the inner workings of PLDs in order to do PLD designs. And when you finish the design with MACHXL, the software automatically selects the best devices, based on criteria you set (like price, package, number of devices, etc.)

The following is a block diagram of the MACHXL system.



The following figure shows the basic design flow when using MACHXL and where the information for each step can be found in the User's Guide.



Design Flow

Programmable Logic Design Synthesis is the process of describing a design by schematic or language entry and synthesizing that information into an optimized form used to program one or more programmable devices. MACHXL is a full-featured programmable Design Synthesis Tool letting you concentrate on your design, not the operational details of the programmable devices.

MACHXL synthesizes the path from design description to actual programmable devices.

You run MACHXL through the Windows menu system (see **Chapter 3**).

Key features of MACHXL include:

- Multiple design entry modes
- Full range of device support from PLDs to CPLDs

Each of these key features is discussed in the following sections.

Design Entry

Flexible Design Methodology

MACHXL provides a device-independent approach letting you enter your logic design without specifying devices for implementation. If desired, you may choose specific devices during design entry.

Design entry may be accomplished by one or a combination of methods. This feature allows you to describe each function using the entry method best suited for that particular function.

Design Synthesis Language (DSL)

DSL is a high-level behavioral language developed for use with programmable logic. DSL provides constructs for state machine descriptions, truth tables and Boolean equations. DSL also allows hierarchical design with procedures and functions. Program control-flow statements such as IF and CASE, combined with multiple nesting and hierarchical design capabilities, let you describe complex designs quickly and easily. You may also create macros to perform text-substitution.

PDS Language

MACHXL allows you to use PDS source files as language input for those designs developed with PALASM.

Compiling

Once the design is entered in DSL, it must be compiled. Compiling your design creates an internal representation of the design with all high-level constructs converted to Boolean equations. The compiler handles multiple design files via the USE construct. The USE construct resolves all invocations of procedures, functions, and components to create a set of low-level synthesized gates.

Functional Simulation

MACHXL's functional simulator lets you verify the functionality of your design before you commit it to programmable devices. By detecting problem areas early, you can modify the design while still in the design process. This simulator can be used to verify individual procedures and functions or entire systems. In addition to simulating each procedure and function to verify that they describe the logic properly, the entire design can be simulated at the system level. This assures your design's complete functionality at both function and system levels.

Simulation in MACHXL is accomplished by a high-level test language similar in construction to its Design Synthesis Language (DSL). The test language lets you create high-level constructs like iterative loops and truth tables to make it easier to simulate your design.

This test language also lets you generate test vectors that can be used to verify the devices after they have been programmed. Verification is done by sending the programmed device stimulus vectors and checking the responses against those from the simulator.

Optimizing

MACHXL uses various optimization techniques to find the necessary product terms and select the smallest set to describe the original equations. All optimization forms are stored and are available for device selection and implementation.

By taking advantage of digital logic design rules, MACHXL utilizes fully the device architecture capability.

Automatic DeMorganization

This feature allows the partitioning system to invert signals internally 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 of a particular device. It also gives maximum flexibility to the partitioning system, which may allow larger, more complex designs to be placed into fewer devices.

Automatic Flip-Flop Synthesis

Another logic design rule is synthesizing the proper flip-flop type to optimize equation placement within a device. For example, a set of equations may be described in terms of J-K flip flops in the design and MACHXL can place these equations in a device that has only D flip flops by synthesizing the D-equation equivalents. A more common application is the use of T flip-flop equations, instead of D flip flops, to produce smaller equations.

Automatic Don't Care Generation

Don't Care conditions can be expressed in IF/THEN/ELSE, CASE, TRUTH TABLE and STATE MACHINE statements as well as assigned to signals. Unspecified output values are assumed to be Don't Care, allowing the optimizer to assign either a zero or one value, depending upon which value generates the most optimal equation. Signals can also be set explicitly to Don't Care values. This feature gives you the potential to create highly-optimized designs resulting in smaller hardware solutions.

XOR Synthesis

The compiler and optimizer maintain an exclusive-OR representation of all equations for which such a representation can be built. This gives the partitioning system the ability to use the exclusive-OR representation in devices with that capability or to use the sum-of-products representation in devices without exclusive-ORs.

NODE Collapsing

The optimizer minimizes the use of intermediate nodes in the design. It removes nodes, collapsing their equations into any equations referencing the removed node. This collapsing process can be controlled by the designer to produce the best results for the target hardware.

Logic Minimization

Reduction levels used by MACHXL include various combinations of industry-standard heuristic and exact methods to meet your design goals. These reduction levels include:

- Espresso
- Espresso (Exact)
- Quine-McCluskey

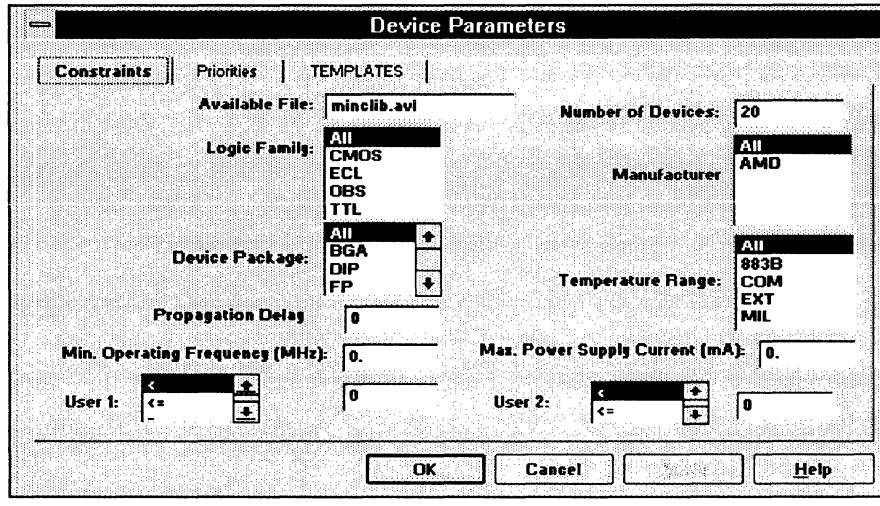
You may also specify NO_REDUCE, which performs logic conversion only with no logic minimalization.

Device Selection

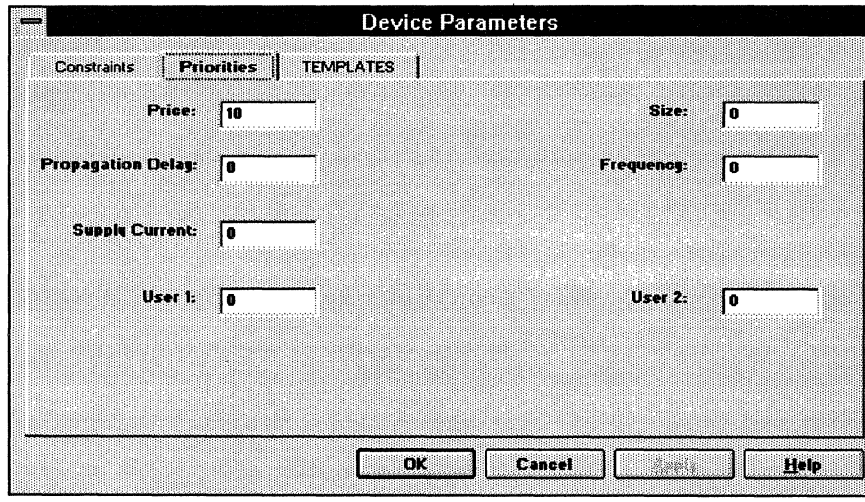
PLDs/CPLDs

MACHXL automates the selection of the best PLD/CPLD architectures and specific devices for your design. Based on the device characteristics and the design constraints, the device selection system searches the master library for devices that match your constraints. Your design is then mapped into combinations of the selected device architectures.

If the design requires more than one device, the design is automatically partitioned across multiple devices (this capability is optional.) MACHXL also lets you choose among many speed, power and package type variations offered by the IC vendors. The following screen shows the menu used to set Constraints.



The following screen shows the menu used to prioritize the constraints by adding a relative weighting. For example, price may be given a weight that is twice as important as power as is shown in the following menu.



Two user constraint fields let you enter data for a device that is specific to your design or manufacturing environment.

If manual device selection is preferred, you may specify the devices for implementation in a Physical Information (*.pi*) file. Specifying a device in a Physical Information file will override the automatic device selection process.

Design Partitioning

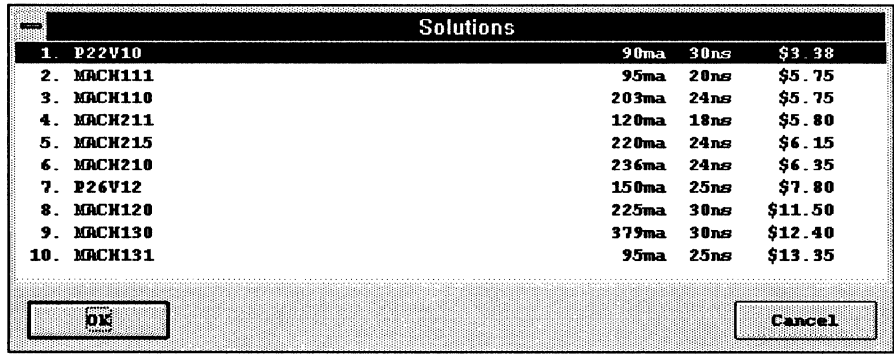
Design equations are automatically divided among multiple PLDs/CPLDs to create design solutions. The partitioning system searches PLD/CPLD device architectures in the master library for the combination of devices creating the best solutions for the design.

The partitioning system generates and displays the top ten design solutions, using your design constraints in conjunction with the device requirements from the design description (see the following screen.) The solutions are prioritized

using the design constraints and displayed in order. You may stop the process at any time and select a solution.

Solutions Menu

This menu displays device solutions into which the design can fit. Design pin outs can be assigned automatically or manually through the use of MACHXL's physical information (*.pi*) file.



Devices Library

- Contains design data on all AMD devices
- Supported devices include AMD's PLDs and CPLDs

The device library contains the most up-to-date specification information available from AMD.

For a complete list of the devices in MACHXL's device library, refer to the separate Device Library listings in **Appendix A**. **Appendix A** lists the PLD and CPLD devices supported by MACHXL.

Building a MACHXL Design Synthesis Language Source File

MACHXL lets you create a source file to describe your design. **Chapters 4 - 9** cover the elements of this source file. The following shows the general organization of a typical design source file. It also lists the chapter(s) where information about each part of the design source file is located.

Parts of a Source File (Using MACHXL's Design Synthesis Language)

Headers (information about the design)	see Chapter 4
MACRO Definitions (text substitution structures)	see Chapter 9
USE constructs (compiled Procedures and Functions to be used by this source file)	see Chapter 8
Procedure/Function Definitions (Procedures/Functions used in this design)	see Chapter 8
System-Level Declarations (declaring the signals to be used in this design)	see Chapter 5
System-Level Statements (statements and constructs that describe your design)	see Chapters 6,7

The outline above shows the main sections used in a source file. Each of the sections listed is optional. In addition to these chapters **Appendix B** contains a number of language design examples, complete with comments and explanations.

3 Windows Interface

Contents

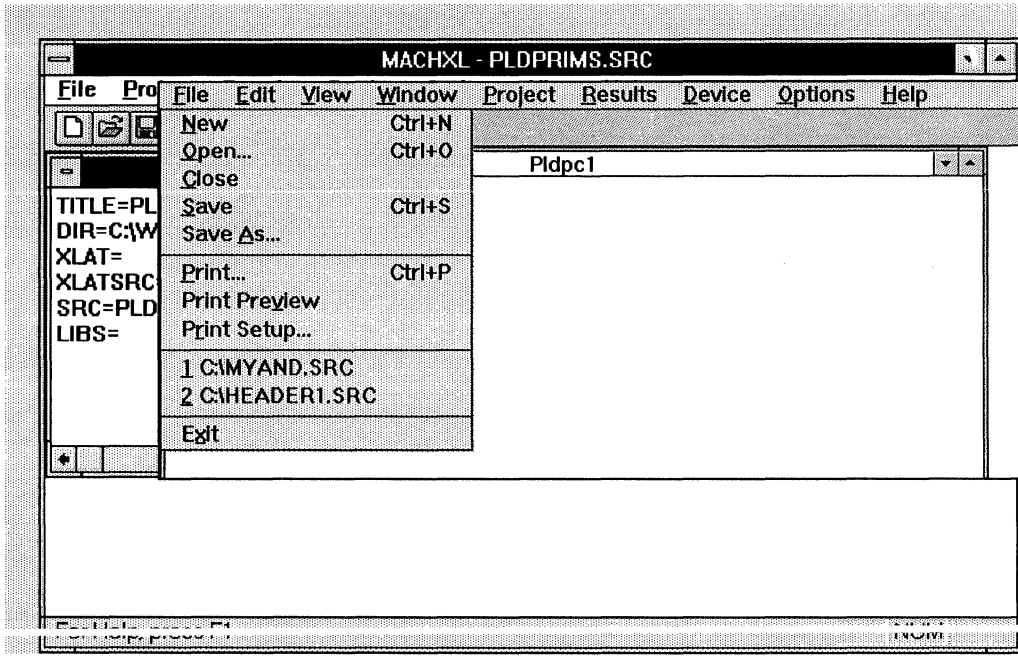
Introduction

This chapter is intended to give the necessary detail to operate MACHXL in a Microsoft Windows 3.x or later environment.

We assume a basic knowledge of Windows and will not explain commonly used Windows menus. Only those functions unique to MACHXL are explained.

The following screen shows the MACHXL main screen with all the menu functions.

Note: When you first enter MACHXL, the menu bar will show only three Windows functions. Once you've opened a file (new or existing), the full menu bar, shown below, will appear.



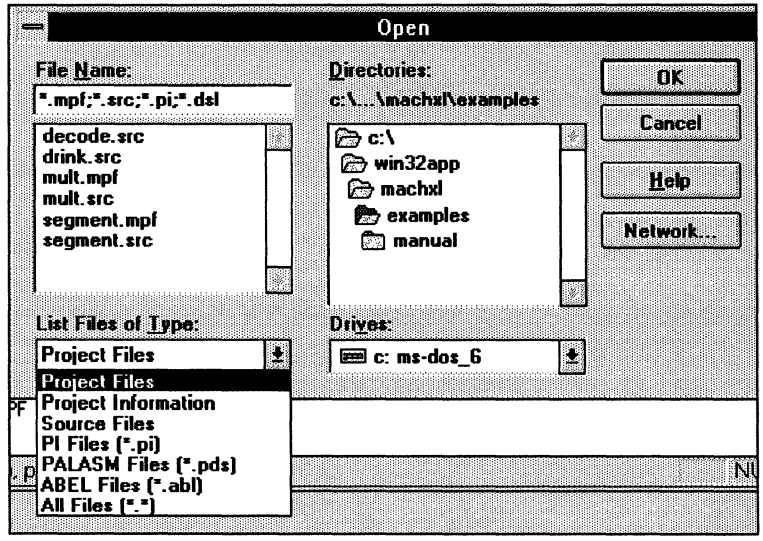
File Menu

New

Starts a new project file. Selecting New opens a new window. If there are already one or more windows open, MACHXL opens a new window without closing any of the others.

Opening a File (New or Existing)

The New and Open menu items open a dialog box containing the names of all files in the current directory with an extension matching the type of file you select in the List Files of Type box. You can change the type of file displayed by selecting List Files of Type and choosing the file type from the pulldown list.



Project Files

Those used to build a design.

- **.mpf* (Project Files)
- **.src* (source files)
- **.pi* (Physical Information files)
- **.dsl* (Design Synthesis Language files)

Project Information Files

Those containing information about how your design was "built".

Source Files

.pi Files (*.pi)

Physical Information files to control partitioning of your design.. For more information on *.pi* files, see chapters 13 through 15.

PALASM Files (*.pds)

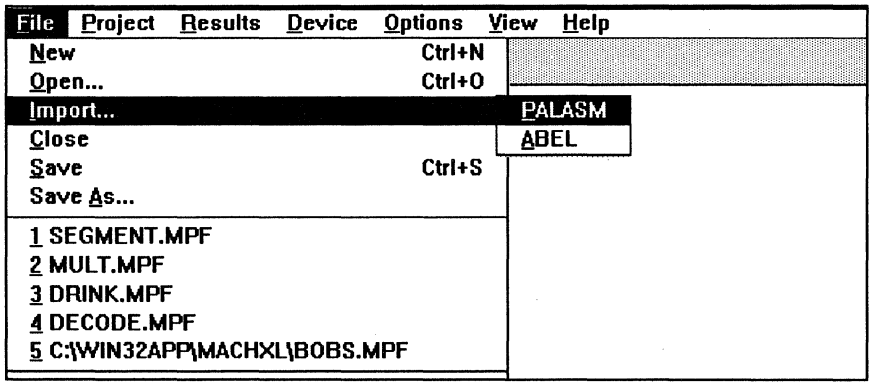
ABEL Files (*.abl)

All Files (*.*)

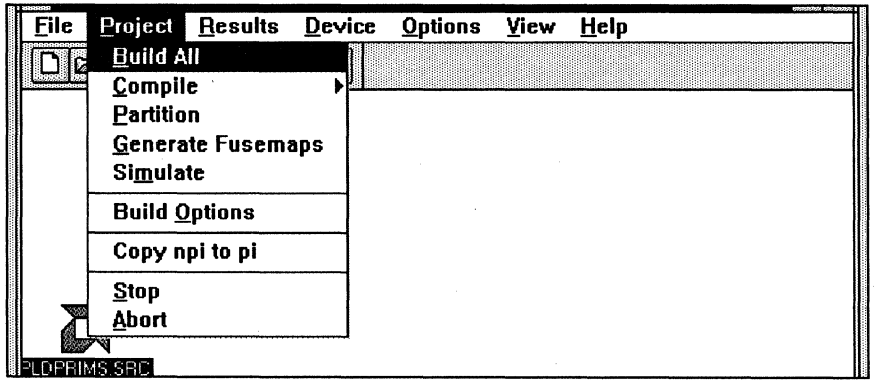
Import

Imports and translates the selected file to a *.src* (source file). Import types are:

- PALASM
- ABEL



Project Menu



Build All

Runs the MACHXL tools on the open source file. The tools are run in the following sequence:

Compiler - compiles the source file.

Optimizer - optimizes to the most efficient number of gates in the smallest possible device(s).

Simulator - runs the simulator on the design. Please note the simulator in MACHXL is a functional simulator only. The simulator will only be run if there is a *design_name.stm* file in the same directory as the design file, and if the option Automatically Simulate is set in the Simulate Options menu. See **Chapter 11** for more information on the Simulator and creating an *.stm* file.

Document File - documents the compile, optimize, and simulation processes and places the information in the file *design_name.doc*.

Device Scanner- scans the file of available devices to find those into which your design will fit.

Fitter - creates solutions for your design, based on the devices from the Scanner, and the constraints and priorities which you set (see the Device menu and the Parameters menu items later in this section for more information on setting priorities and constraints.) These solutions are listed in the solutions menu, allowing you to choose one. No fusemaps are actually created here. This step correctly partitions your design into single or multiple devices, and takes care of routing signals to each device.

Simulator - functionally simulates the design again, now that partitioning is complete. The simulator will only be run if there is a *design_name.stm* file in the same directory as the design file, and if the option Automatically Simulate is set in the Simulate Options menu. See **Chapter 10** for more information on the Simulator and creating an *.stm* file.

Fuse Mapper - creates fusemaps for the design. These fusemaps can then be downloaded to a device programmer to program the PLDs/CPLDs.

Document File - after the Scanner, Fitter, Simulator, and Fuse Mapper are run, the file *design_name.doc* is updated with information about the scan, fit, and fusemap processes. Note this information is

appended to the earlier compile, optimize, and simulate information in the file.

During the Build process a window displays its progress. An information message appears saying *Build Completed* at the end of the process.

If your design is hierarchical, each portion of the design will be compiled, optimized, and simulated.

If a failure occurs in any of the Build processes, MACHXL will abort the process.

Compile

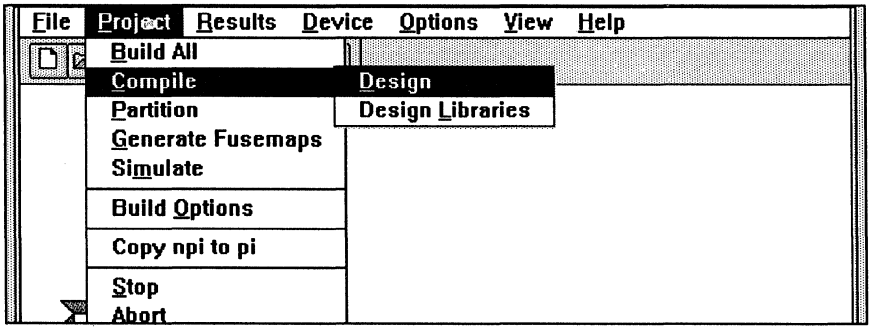
MACHXL can compile a complete design or just certain modules. This allows defining modules as symbols or library parts.

A normal design (i.e., one with system-level signals) will be run through all of the Build processes (i.e., compile, optimize, simulate, etc.)

A library part will be run through only the compile process. Since it has no system-level signals, a library part cannot be run through the optimizer.

Selecting Compile causes another pulldown menu to appear. This pulldown allows telling MACHXL whether you are compiling a design or a design library (i.e., a library part.)

When the compile if finished, an information message will display *Compile Completed.*



Design

Compiles, optimizes, and simulates a design. A design must have system-level signals.

Design Libraries

Allows compiling (but not optimizing or simulating) a module for use as a library part. A module has no system-level signals, and therefore cannot be run through the optimizer.

Partition

The process of partitioning a design involves several processes, each of which is explained below.

Device Scanner- scans the file of available devices to find those into which your design will fit.

Fitter - creates solutions for your design, based on the devices from the Scanner, and the constraints and priorities which you set (see the Device menu and the Parameters menu items later in this section for more information on setting priorities and constraints.) These solutions are listed in the solutions menu, allowing you to choose one. No fusemaps are actually created here. This step correctly partitions your design into single or multiple devices, and takes care of routing signals to each device.

Simulator - functionally simulates the design again, now that partitioning is complete. The simulator will only be run if there is a *design_name.stm* file in the same directory as the design file, and if the option Automatically Simulate is set in the Simulate Options menu. See **Chapter 10** for more information on the Simulator and creating an *.stm* file.

Fuse Mapper - creates fusemaps for the design. These fusemaps can then be downloaded to a device programmer to program the PLDs/CPLDs.

Generate Fusemaps

Generates fuse maps for PLD or CPLD devices. You do not need to run this procedure for each PLD/CPLD in the design. MACHXL will create the fusemap files for all the PLD/CPLD devices in your design (assuming there is more than one device in your design solution.)

You need to generate the fuse maps before you can program the devices.

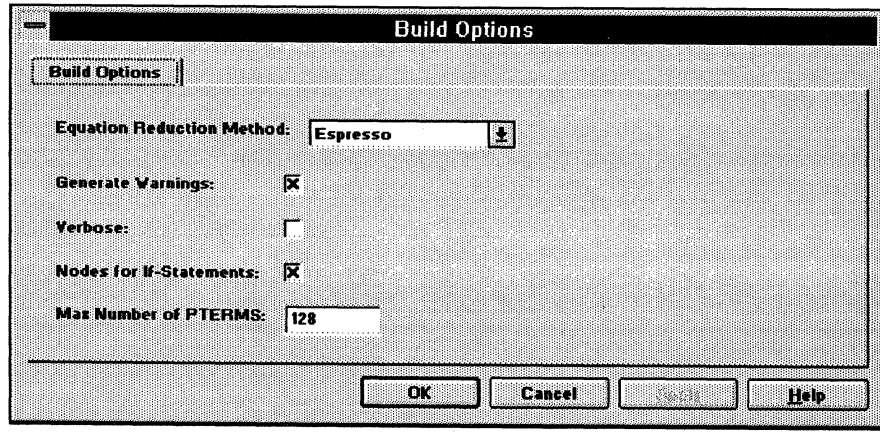
Build Options

This menu lets you view and set the equation reduction method used by the optimizer to reduce equations. It also lets you specify whether you want MACHXL to generate warning messages for conditions it deems unusual but not catastrophic.

Equation Reduction Method

Controls how the optimizer reduces the design equations.

The Espresso technique is fast and generally produces very good equations. Espresso Exact and Quine-McClusky methods are slower and use more of your PCs dynamic memory (RAM) but may result in smaller equations. Due to speed and memory use concerns, Espresso Exact and Quine-McClusky reduction techniques should be restricted to designs with relatively small equations where optimal equation reduction is critical.



The default reduction method is Espresso.

Generate Warnings

This option controls whether or not MACHXL will produce messages for conditions it deems unusual but not catastrophic.

The default is warnings to be displayed.

Verbose

MACHXL has a number of processes (compiler, optimizer, simulator, etc.), each of which can generate messages to let you know what is going on with the process. You can choose whether or not to have these messages displayed with the Verbose option. It is useful to have these messages displayed if you have a large, complex design requiring a lot processing time. However, if you have a smaller design, you may not want these messages to appear. In either case, these messages are contained in the *.log* file, so you still have access to them.

The default for Verbose is off.

Nodes for If Statements

Specifies whether the compiler should generate nodes for IF/THEN/ELSE statements.

MAX Number of Pterms

Specifies the maximum number of pterms allowed in a design equation.

Copy npi to pi

Copies the Partitioner-created *.npi* (New Physical Information) file to a *.pi* file. For more information, see the section entitled ***Using the .npi File to Recreate a Pinout*** in ***Chapter 13***.

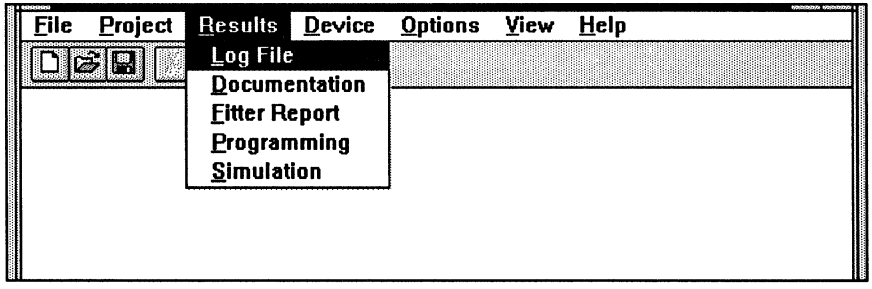
Stop

Tells MACHXL to stop a Build that's in progress and exit all processes gracefully. MACHXL will also report any errors to the file *design_name.err*.

Abort

Tells MACHXL to abort a Build that's in progress and to stop all processes. Processes will be aborted regardless of the stage they are in.

Results Menu



Log File

The *.log* file (*design_name.log*) contains any warnings or errors that occurred during the compile and partitioning phases. If you have problems during these phases, this is the file to view.

Documentation

The *.doc* file (*design_name.doc*) contains all information about your chosen design solution, including signal names, pinouts of devices, equations that were used in the solution, etc.

You can set how equations are printed in the documentation file. For more information, see the section later in this chapter entitled Documentation Options in the Options Menu.

Fitter Report

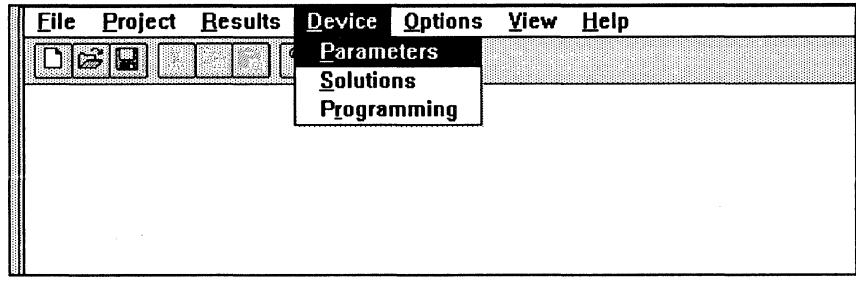
If you are using MACH devices in your design, this field will display the MACH *.rpt* (report) file. The report contains all pertinent information about fitting of MACH device(s), including the percentage of resource utilization.

For more information on reading the *.rpt* file, see **Chapter 14** and **Appendix D**.

Programming

Allows viewing of the programming (JEDEC) files for a design.

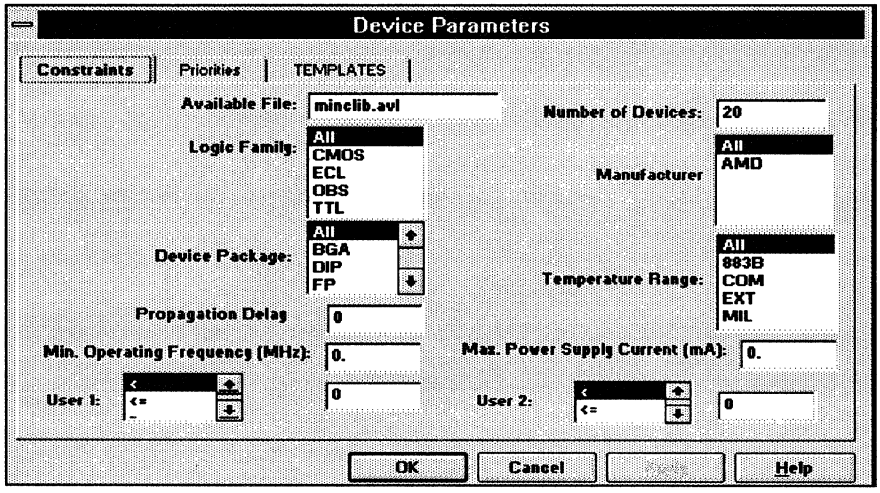
Device Menu



The Device Menu lets you run certain processes concerning devices in your design. The pulldowns and submenus are explained below.

Parameters

Constraints



Constraints allows limiting the number of devices MACHXL considers as valid solutions during the partitioning process. For example, you can set a Logic Family constraint permitting only TTL device to be considered.

While constraints are a powerful feature of MACHXL, setting arbitrarily stringent requirements may severely limit the number of devices MACHXL can fit. It may also make it impossible to fit the design into any device or devices.

Logic Family:

This field shows which logic families are considered as valid partitioning devices. The default is all logic families.

Device Package:

This field shows which device package types are considered valid during partitioning of your design. The default is all package types.

Propagation Delay (nS):
This field sets the maximum propagation delay for any device in the solution. Note this is the propagation delay for any device, not for the design as a whole. The default is 0.0nS.

Min. Operating Frequency (MHz):
Specifies the minimum operating frequency, in MHz, of any device considered for a solution. The default is 0.0 MHz.

User 1:
User-supplied field to enter your own constraint. For example, if your company keeps statistics on failure rate and MTBF, you could use User 1 and User 2 to represent these statistics for devices or families.

Number of Devices:
Lets you tell MACHXL the maximum number of devices in the solution. You may use any number from 1 to 20. Note depending on the complexity of the design, setting an arbitrarily low number of devices may force MACHXL to consider only very large devices. MACHXL may also be unable to fit the design at all. The default for this field is 20.

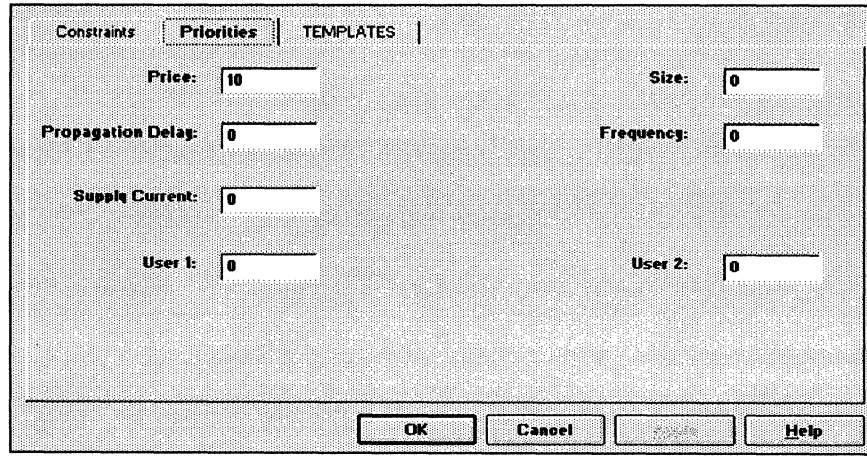
Temperature Range:
Allows selecting valid device operating temperature ranges.

The default is all temperature ranges.

Max Power Supply Current (mA):
Allows entering the maximum amount of current (in mA) a device may draw. Note this is the maximum draw for any one device, not for the whole design (if there is more than one device in the solution.) The default is 0.0 mA.

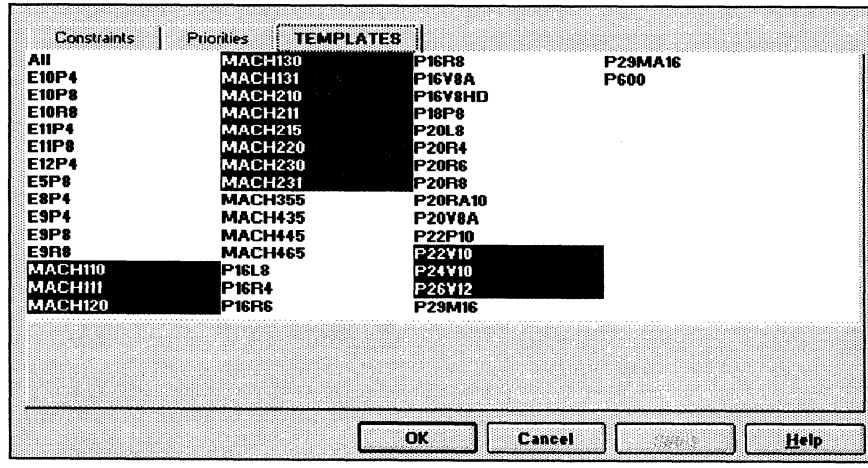
User 2:
See User 1.

Priorities



This dialog box tells MACHXL how important are certain criteria when selecting devices. Priorities are used as weighting factors to determine the order of solutions displayed in the Top 10 List. Weighting is on a scale of 1 to 10 with 10 being the most important. By means of the Priorities menu you tell MACHXL which are most important.

TEMPLATES



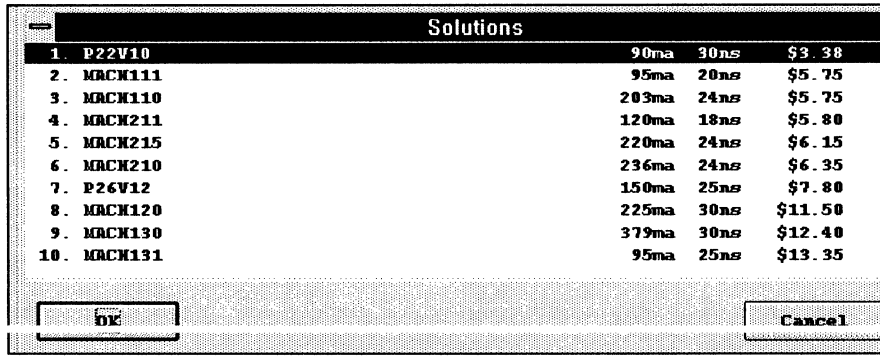
Every programmable device belongs to an architecture, which shares features with other similar devices. For example, two devices with similar part numbers may be identical inside, and vary only in their temperature range or package type. These architectures are know as templates, and are set by the manufacturer.

The templates menu allows specifying which architecturesare considered valid during partitioning. Specifying only those templates needed will considerably speed the partitioning process.

> ***Note**: You must select the Device template for each architecture you use, even if a device is specified in the Physical Information (.*pi*) file. For more information on using a* .pi *file to modify MACHXLs partitioning process, see Chapter 13.*

Solutions

After MACHXL partitions and fits a design, it displays a list of the Top 10 Solutions (if there are at least 10), from which you can choose a solution. These solutions are developed from your design and include constraints and priorities you set.



You may go back to the Solutions menu at any time and choose a new solution. Simple choose Solutions from the Device menu. This eliminates the need to re-compile your design each time you want to investigate a new solution.

Programming

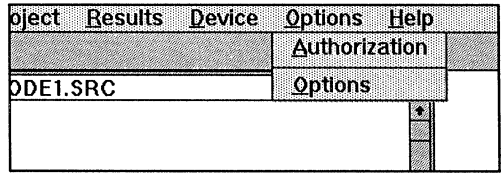
Downloads the JEDEC files for your design to the device programmer.

Options Menu

The Options Menu let you set parameters that affect the overall look and operation of MACHXL There are five option categories:

- Build Options
- Documentation Options
- Schematic Options
- Simulator Options
- System Interface Options

You can set the parameters by selecting each category, as explained in the following sections.



Authorization

Allows you to modify authorization codes for MACHXL and the AMD device modules.

Options

There are four buttons at the bottom of the Options pulldown affecting options menus.

OK

Saves the choices made in the current menu.

Cancel

Returns all values to their original state and closes the menu.

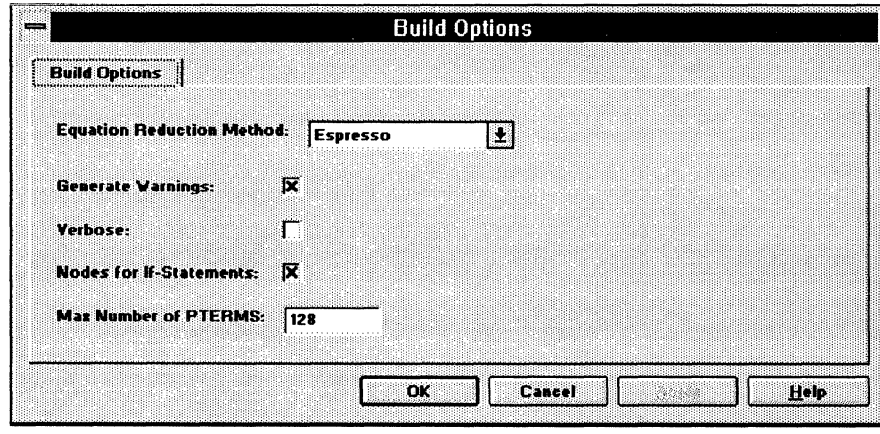
Apply

Applies the menu values to the current design.

Build Options

Equation Reduction Method

Controls how the optimizer reduces the design equations.



The Espresso technique is fast and generally produces very good equations. Espresso Exact and Quine-McClusky methods are slower

and use more of your PCs dynamic memory (RAM) but may result in smaller equations. Due to speed and memory use concerns, Espresso Exact and Quine-McClusky reduction techniques should be restricted to designs with relatively small equations where optimal equation reduction is critical.

The default reduction method is Espresso.

Generate Warnings

This option controls whether or not MACHXL will produce messages for conditions it deems unusual but not catastrophic.

The default is for warnings to be displayed.

Verbose

MACHXL has a number of processes (compiler, optimizer, simulator, etc.), each of which can generate messages to let you know what is going on with the process. You can choose whether or not to have these messages displayed with the Verbose option. It is useful to have these messages displayed if you have a large, complex design requiring a lot processing time. However, if you have a smaller design, you may not want these messages to appear. In either case, these messages are contained in the *.log* file, so you still have access to them.

The default for Verbose is off.

Nodes for If Statements

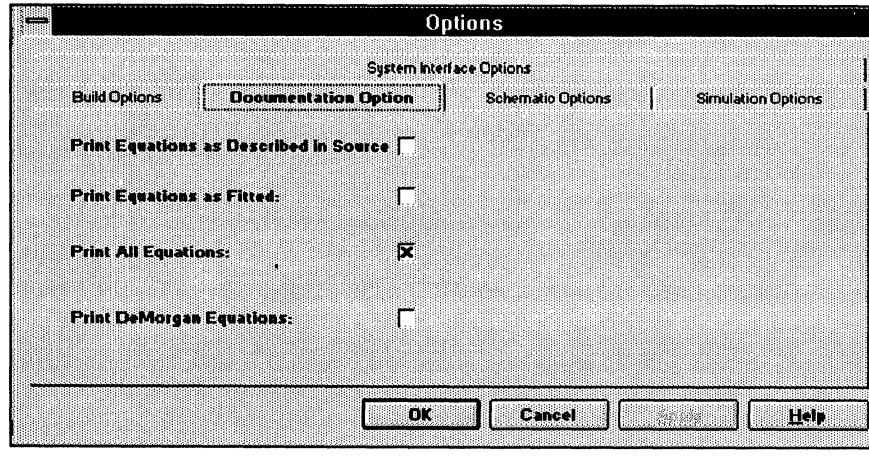
Specifies whether the compiler should generate nodes for IF/THEN/ELSE statements.

MAX Number of Pterms

Sets the maximum number of pterms allowed in any design equation.

Documentation Options

The documentation options allow you to set how equations will be displayed in the documentation (*design_name.doc*) file.



Print Equations as Described in the Source

This option tells MACHXL to print the equations as specified in the original source file. For example, if you specify *x* as a JK flop, you will see both J and K equations in the *.doc* file.

Print Equations as Fitted

This option tells MACHXL to print the equations as fit onto the devices. Because of the operations of the optimizer, these equations may appear considerably different than those originally specified.

Print All Equations

The compiler takes the original equations you specify and attempts to synthesize as many functionally-equivalent equations as possible. This is to maximize the number of devices that MACHXL can fit your design onto. This is done also to minimize the size of the design, allowing it to be fit onto smaller, less-expensive devices.

This option writes the equations to the *.doc* file as you originally described them, as well as All equations synthesized by the compiler.

Print DeMorgan Equations

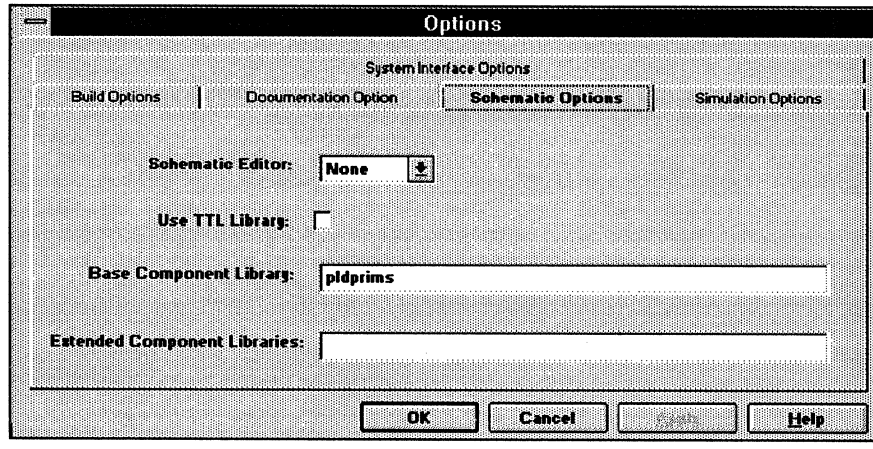
This option prints all the DeMorgan equivalents of the equations used during fitting.

Schematic Options

MACHXL uses schematic and language (source) files as input. The following set options applying to schematic input.

Schematic Editor

Allows choosing which MACHXL-supported schematic editor will be used to edit a schematic design file.



Use TTL Library

This options allows you to specify whether or not to use the EDIF 200 TTL library.

Base Component Library

The default component library used by MACHXL's schematic netlist compiler is called PLDPRIMS. If you choose, you can replace this

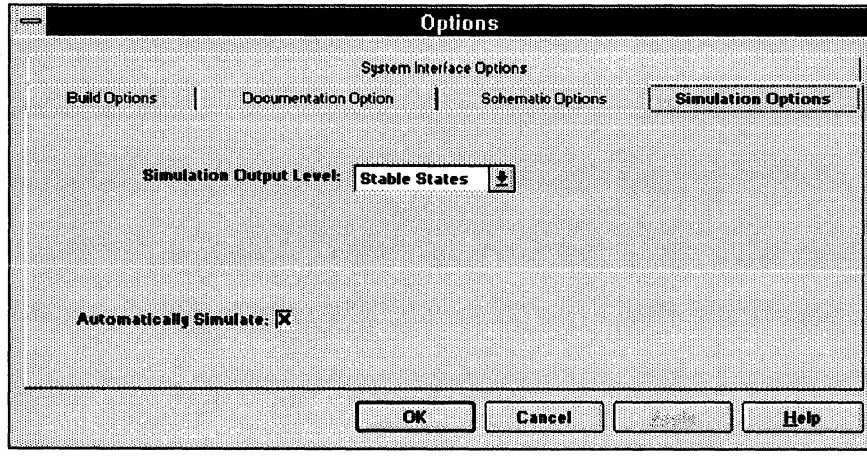
library with another. Enter the path and name of the library in the field provided.

Extended Component Libraries

You can extend the component library provided with MACHXL (PLDPRIMS) by adding schematic components of your own, or component libraries from another source. MACHXL will recognize up to five of these component libraries. These are used in addition to the base component library, PLDPRIMS.

Simulation Options

Allows you to set options relating to MACHXL's functional simulator.



Simulation Output Level

These options change the way the simulator outputs information to its listing (*.sim*) file. For more information on the simulator's operation, see **Chapter 11**.

All States

Causes both unstable and stable states to be written to the simulator listing file.

Unstable States
Writes only unstable simulator states to the simulator listing file.

Stable States
Writes only stable simulator states to the simulator listing file.

Automatically Simulate
Runs the simulator during a normal Build process. The following shows the order processes are run during the Build.

Compiler - compiles the source file.

Optimizer - optimizes the design, reducing it to the most efficient number of gates into the smallest possible device(s).

Simulator - runs the simulator on the design. Please note the simulator in MACHXL is a functional simulator only. The simulator runs only if there is a *design_name.stm* file in the same directory as the design file, and if the option Automatically Simulate is set in the Simulate Options menu. See **Chapter 11** for more information on the Simulator and creating an *.stm* file.

Document File - documents the compile, optimize, and simulation processes and places the information in the file *design_name.doc*.

Device Scanner- scans the file of available devices to find those into which your design will fit.

Fitter - creates solutions for your design, based on the devices from the Scanner, and the constraints and priorities which you set (see the Device menu and the Parameters menu items later in this section for more information on setting priorities and constraints.) These solutions are listed in the solutions menu, allowing you to choose one. No fusemaps are actually created here. This step correctly partitions your design into single or multiple devices, and takes care of routing signals to each device.

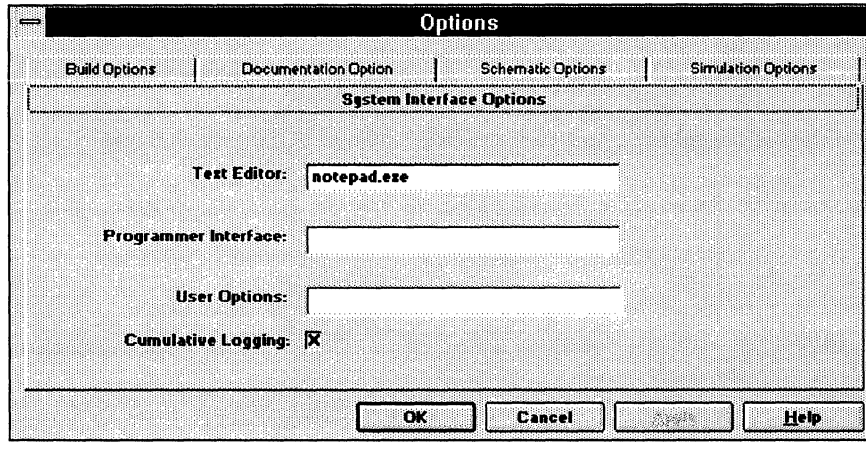
Simulator - functionally simulates the design again, to create test vector files.

Fuse Mapper - creates fusemaps for the design. These fusemaps can then be downloaded to a device programmer to program the PLDs/CPLDs.

Document File - after the Scanner, Fitter, Simulator, and Fuse Mapper are run, the file *design_name.doc* is updated with information about the scan, fit, and fusemap processes. Note this information is appended to the earlier compile, optimize, and simulate information in the file.

Notice the simulator is run twice through a normal build and partition cycle. By changing the Automatically Simulate field, you can tell MACHXL not to run the simulator. If you do not need to simulate your design, disabling the Simulator can speed the build and partition processes.

System Interface Options



Text Editor

Tells MACHXL which text editor to use to create source files. You need to supply the name of the editor's executable file as well as the path. Windows Notepad is default.

Programmer Interface

Tells MACHXL the name and path of the device programmer's communication software. You need to supply this before you can download fusemaps to your device programmer.

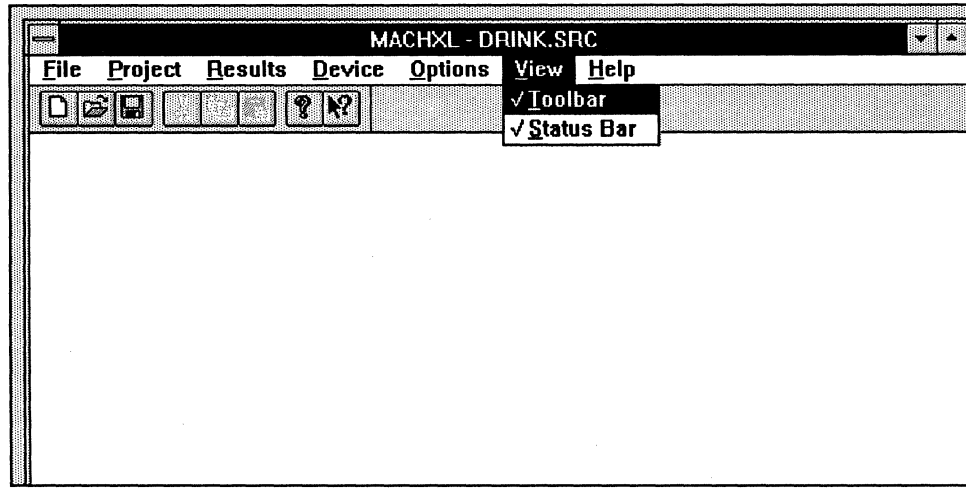
User Options

Allows you to enter command strings or set parameters for the text editor.

Cumulative Logging

When MACHXL logs the prcess results to the *design_name.log* file, it overwrites previous information. This field tells MACHXL to concatenate the new information to the file instead of overwriting.

View Menu



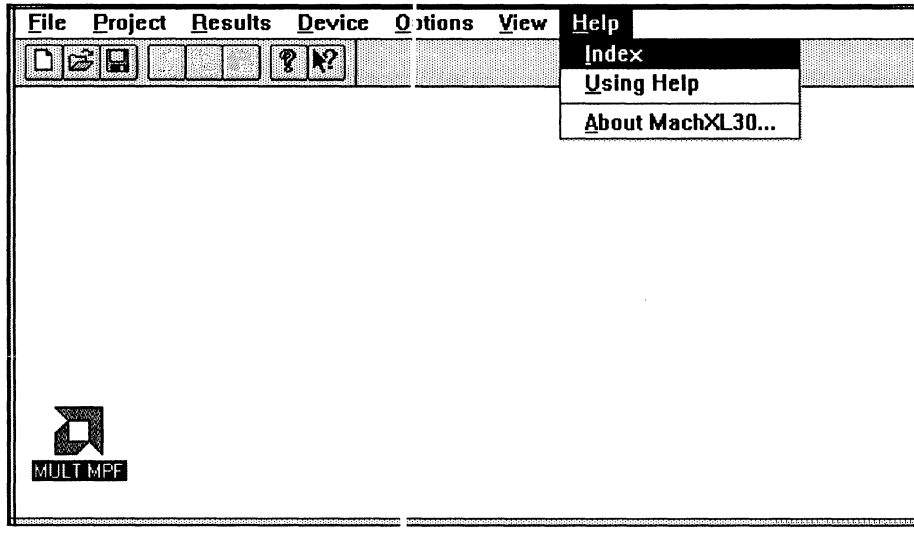
Toolbar

Lets you specify whether or not to display the MACHXL tool bar. If the tool bar is turned on, a check mark will appear to the left of the label. Turning the toolbar off increases the size of the Main Window.

Status Bar

Lets you specify whether or not to display the MACHXL status bar at the bottom of the Main Window. If the status bar is turned on, a check mark will appear to the left of the label. Turning the status bar off increases the size of the Main Window, but will not allow you to see the status of MACHXL functions.

Help Menu



Index

Displays index of items available for help.

Using Help

A short tutorial on using help features.

About MACHXL

Diplays the version number of MACHXL.

4 Conventions and Syntax

Contents

Introduction to Design Synthesis Language (DSL)

The Design Synthesis Language (DSL) is a high-level behavioral language developed for use with programmable logic. You can use DSL to build a source file to describe your design. DSL provides constructs for state-machine descriptions, truth tables, and Boolean. DSL also allows hierarchical design with procedures and functions. Program control statements such as IF and CASE, combined with multiple nesting and hierarchical design capabilities let you describe complex designs quickly and easily. You can also create macros to perform text substitution.

There are two kinds of DSL files, each providing different functions:

- Source file- this is a functional description of your design using DSL. The source file describes the behavior of your design.
- Physical Information (*.pi*) file-this controls how a design is implemented. This optional file can be used to specify:
  - physical devices used when implementing the design
  - pin out of each device in the design
  - optimization techniques used on the design
  - device-specific features required by the design

The structure and syntax of DSL are described in the remainder of this chapter and in chapters 5 through 9. The structure of the physical information (.pi) file and device specific information are found in chapters 14 and 15.

Description of a DSL Source File

As mentioned earlier, the DSL source file contains the functional description of your design. DSL has a structure similar to many programming languages. If you have experience with a programming language, you'll probably recognize many of the constructs of DSL.

A DSL source file will contain the following information:

1. Procedure and function definitions for frequently used descriptions. Much like their programming language counterparts, procedures and functions are declared before they are invoked in DSL.
2. Signal declarations that define the characteristics of the signals in the design. Signal descriptions are equivalent to variable declarations in a programming language.
3. Statements (including procedure and function instantiations) make up the logic that is implemented in your programmable devices.

The following shows the suggested organization of a DSL source file and the chapters where information on each part may be found:

Headers	See this chapter
MACRO definitions	See chapter 9
USE constructs	See chapter 8
Procedure/Function definitions	See chapter 8
System-level signal declarations	See chapter 7
System-level statements	See chapter 7

Each section of the DSL design description is optional. For example, you may create a simple DSL design description that consists only of a System-level declaration and the system-level statements. Or, you may create a DSL source file that includes only Procedure and Function definitions. This source file could then in turn be used as a library of handy routines that can be accessed by other DSL source files.

Examples of DSL source files can be found in ***Appendix B, Language-Based Design Examples***.

Conventions Used by Design Synthesis Language

The following table shows the conventions used by DSL.

Notation	Description
[ ]	Brackets indicate optional features of the language construct
[ ]	Bold Brackets indicate that brackets are a necessary part of a construct
{ }	Curly braces identify where 1 or more items may be repeated in a construct
CAPITALS	Capitals indicate the parts of a construct needed for a valid statement or definition (e.g., keywords).
italics	Italics indicate user-supplied parts of a construct.
name_list	List of named items (identifiers, expressions, etc.) separated by commas. The shorthand range operator [..] may be used to express a list of identifiers that differ only in a trailing number (see Groups and Ranges in Chapter 7.)

Identifiers

Identifiers are names given to specific items in a source file. Named items include signals, macros, procedures, functions, state machines, states in a state machine, and test language variables.

The rules for forming identifiers are:

1. The first character of an identifier must be a letter ("A" through "Z", or "a" through "z") or an underscore (_).
2. Succeeding characters may be any sequence of letters (A..Z, a..z), digits (0..9), the dollar sign ($), or the underscore (_).
3. You may use any combination of upper-case and lower-case letters in an identifier. The Design Synthesis Language is case-insensitive; thus, upper-case and lower-case letters are treated alike.

4. Identifiers cannot contain spaces. Use the underscore character to separate words in long identifiers to make them easier to read.
5. Identifiers may be of any length.

Keywords

The identifiers listed below are reserved by the language as keywords and may not be used for other identifier purposes.

AND
BIN
BIPUT
BLOWN
CASE
CLOCK_ENABLED_BY
CLOCKED_BY
CLOCKF
COMP_OFF
COMP_ON
D_FLOP
D_LATCH
DEC
DEFAULT
DEFAULT_TO
DEMORGAN_SYNTH
DEVICE
DISABLED_ONLY_FOR_
TEST
DO
ELSE
ELSIF
ENABLED_BY
END
FF_SYNTH
FIT_WITH
FIXED
FOOTPRINT
FOR
FUNCTION
GOTO
GRAY_CODE
GROUP
HEX
HIGH_VALUE
IF
INCLUDE
INITIAL
INITIAL_TO
INPUT
INTACT
JK_FLOP
LAST_VALUE
LATCHED_BY
LOW_TRUE
LOW_VALUE
MACRO
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
MESSAGE
MOD
NAME
NO_COLLAPSE
NO_CONNECT

NO_REDUCE
NODE
NOT
OCT
ONE_HOT
OR
OUTPUT
PART_NUMBER
PHYSICAL
POLARITY_CONTROL
PRESET_BY
PROCEDURE
RESET_BY
RETURN
SECTION
SET
SIMULATION
SR_FLOP
STATE
STATE_BITS
STATE_MACHINE
STATE_VALUES
STEP
SYSTEM_TEST
T_FLOP
TARGET
TEMPLATE
TEST_VECTORS
THEN
TO
TRACE
TRUTH_TABLE
USE
VAR
VIRTUAL
WHEN
WHILE
WIRED_BUS
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

Integer Constants

Integer constants are used in DSL to assign a fixed value to a signal, for arithmetic operations, or as part of a conditional test. Constants must follow these rules:

- Constants must be integers.
- Constants may be of any length. Operations in DSL are performed with unlimited precision.
- The first character of a constant must be a digit; otherwise the compiler will interpret the character string as an identifier.

- Constants assigned to single bit or non-array signals can only be 0 or 1.
- If no base is specified, the constant is assumed to be decimal.
- Constants used in conditions or arithmetic operations can represent values in four bases (binary, octal, decimal, or hexadecimal). To set the base of the constant, add the first letter of the base name to the end of the constant. For example, to represent the character C as the hexadecimal value for 12, add a leading zero to the letter C and follow it with h (for hexadecimal): 0Ch. This distinguishes it from the letter C. Either upper case or lower case may be used for the letter of the base name.

The following are examples of legal and illegal constants:

Legal Constants:

```
0101b       "binary constant
0732o       "octal constant
973         "decimal constant
973d        "decimal constant
0A0Ah       "hexadecimal constant
```

Illegal Constants:

```
2.54        "constant must be an integer
A0Ah        "constants must start with a digit
0AC0d       "constant must match the base specified
```

Comments

Providing comments in your source file makes it easier to understand the intent of certain sections of code for later reference. Commented code can be particularly useful for design teams working on a project so each member can better understand the other team members' pieces of a project.

Comments begin with a quotation mark (") followed by text. A new line indicates the end of a comment.

Comments used as notes throughout a source file should not be confused with the COMMENT keyword. The COMMENT keyword is used to include comments in a JEDEC file.

For instance, with the comment next to the STATE allred; statement, it is clear that allred is the first state of the state machine:

```
STATE allred:      "First state
```

Headers

Headers are used to place design information in the source file.

Header statements, if used, must appear at the beginning of a source file. Six optional header types are recognized by the Design Synthesis Language: TITLE, ENGINEER, COMPANY, REVISION, and COMMENT.

A design may use any combination of header types, in any order, or none at all. Each header has an associated string. The format for a header is:

```
header_type 'header_information';
```

Where:

header_type is one of the six header keywords: TITLE, ENGINEER, COMPANY, REVISION, PROJECT, or COMMENT.

header_information is text describing the header type information for the design. This text is enclosed in single quote marks.

Examples

```
#TITLE       'X1000 MEMORY GLUE LOGIC';
#ENGINEER    'JOE SILICON';
#REVISION    '2.02';
```

To place multiple lines in the JEDEC file, use separate lines of text enclosed by single-quote marks:

```
#COMMENT     'This design implements the glue'
             'logic between the X1000 and its memory. ';
```

5 Signal Declarations and Modifiers

Contents

Introduction

This chapter discusses the types of signals that the Design Synthesis Language recognizes. Discussions include how to declare signals, as well as the modifiers you may use on them. The types of signal declarations available include:

- INPUTS
- OUTPUTS/BIPUTS
- NODES
- WIRED_BUS

Arrays of these signal types may also be declared.

Modifiers to the signal declarations allow you to declare signals as:

- low true
- flip-flops
- latches

as well as setting the clocking/latching and their default states.

Declarations

Different types of signal declarations made at the beginning of a source file define and name signals (identifiers) to be used in a design and indicate to the compiler, optimizer, and fitting tools how these various signal identifiers will function in the design. Signals may be declared at both the system and local levels (see the next section, System and Local Signal Declarations).

The types of signals available in the Design Synthesis Language include: INPUT, NODE, OUTPUT/ BIPUT, and WIRED_BUS. Any of these signal types may also be declared as an array. A description of each follows.

System and Local Signal Declarations

Signal declarations may appear inside or outside of a procedure or function. A signal declaration made outside of a procedure or function is known as a system signal, and is available at the system level. The signal will not be recognized within any procedure or function.

All procedure and function descriptions must appear before any system level design information, including system-signal declarations and system-level statements.

A signal declaration made inside a procedure or function is local and is not visible to any other procedure or function even at the system level.

Thus, a local signal can have the same name as a system signal and will exist only until the end of the function or procedure. A system signal with the same name as a local signal is immune to any changes made to the local signal unless the value is passed explicitly through a procedure output.

Arrays

An array is a set of logically related signals that can be treated separately or as a unit. All types of signals may be declared as arrays (Signals types INPUTs, NODEs, OUTPUTs, BIPUTs, and WIRED_BUSes.)

The array identifier is listed along with a number or a range of numbers that determines the size of the array. For instance, you may declare a range for an array using beginning and ending indexes:

```
identifier[index_1..index_n];
```

Indexes can be given in either ascending or descending order. When an array is used in an expression, the first index is always the Most Significant Bit and the last index is the Least Significant Bit.

The declaration:

```
OUTPUT addr[15..0];        "array addr declared using
                           a range of indexes
```

Specifies 16 elements to the array: addr[15], addr[14], addr[13], addr[12], addr[11], addr[10] through addr[0].

As an alternative, you may simply specify the size of an array, which becomes a shorthand way of giving a range of indexes from (array_size - 1) descending to 0:

```
identifier[array_size];
```

The same array declaration given in the previous example, specifying 16 elements to the OUTPUT addr, can be declared as follows:

```
OUTPUT addr[16];  "array addr declared using an
                  "array size
```

Again, the elements in the array include: addr[15], addr[14], addr[13], addr[12], addr[11], addr[10] through addr[0].

The following array declaration for q:

```
OUTPUT q[4..7];
```

Has four elements q[4], q[5], q[6], and q[7]. Note that this array has ascending indexes: q[4] being the Most Significant Bit and q[7] being the Least Significant Bit.

Each index can be a constant expression made up of constants and operators. For example, the following:

```
INPUT in[2.*.5]
```

is exactly the same as:

```
INPUT in[10]      " .*. is the DSL operator for
                  " multiplication
```

Input Signals

Signals that serve only as inputs to a design may be declared using the keyword INPUT.

The syntax for declaring input signals is as follows:

```
[LOW_TRUE] INPUT identifier_or_array_list;
```

The optional LOW_TRUE modifier may be used to indicate that a low voltage will represent the true state of the declared input signal(s). (See Low_True in the Declaration Modifiers section, later in this chapter for more information.)

Each signal name in the identifier list must be separated by a comma, and the declaration must be followed by a semi-colon.

Examples

```
INPUT x,y[4],z;              "declares inputs x, y[3],
                             "y[2],y[1], y[0], and z
LOW_TRUE INPUT x,y,z;        "declares inputs x, y, and
                             "z as low-true
INPUT /x,y,z[7..5];          "declares inputs x as low-
                             "true, y, z[7], z[6], z[5]
```

Output/Biput Signals

Signals that will be visible outside a design can be declared using the OUTPUT keyword. BIPUT may be used as a synonym for OUTPUT when symbols are used for bi-directional operation. The syntax for declaring outputs is as follows:

```
[LOW_TRUE] [flip_flop_type] OUTPUT
      identifier_or_array_list [control_info]
      [default_info];
```

OUTPUTs may be used without modifiers as a way to get signals out of a design. On the other hand, NODEs with modifiers are a way of creating internal design elements. An OUTPUT declared with the same modifiers as a NODE is a shorthand or alternate way of representing a NODE that feeds a regular OUTPUT.

Example

```
INPUT  a, b;
D_FLOP OUTPUT x CLOCKED_BY clk;
x = a * b;
```

is equivalent to:

```
INPUT  a, b;
D_FLOP  NODE  x_node  CLOCKED_BY  clk;
OUTPUT  x;
x = x_node;
x_node = a * b;
```

Biput Signal Usage

From a language structure view point, an output statement that contains an ENABLED_BY can be used as a bidirectional signal. However, the following information gives some insight into proper usage of BIPUTs:

The statement:

```
OUTPUT xx ENABLED_BY oe;
```

usually represents an output pin that will be driven with an input value if the ENABLED_BY (i.e., oe) signal is asserted.

The statement

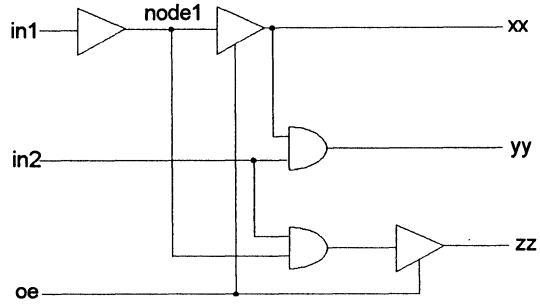
```
BIPUT  xx ENABLED_BY oe;
```

usually represents a biput pin that is driven by internal logic when the ENABLED_BY (i.e., oe) signal is asserted. This same pin is driven by an input value when the ENABLED_BY (i.e., oe) signal is not asserted.

When the ENABLED_BY pin (i.e., oe) of an OUTPUT/BIPUT signal is not asserted, the OUTPUT/ BIPUT signal will have a high impedance (.Z.) state.

The following example and screen show how OUTPUT and BIPUT statements with an ENABLED_BY modifier should be used. The example also shows how signal feedback can be accessed before/or after the ENABLED_BY modifier.

In the following example, *xx* is used as a BIPUT pin, *yy* is used as an OUTPUT pin, and *zz* is used as an OUTPUT pin that is enabled and uses internal feedback from *node1*.




```
Input in1,in2, oe;
PHYSICAL NODE node1;
BIPUT xx ENABLED_BY oe;
OUTPUT yy;
OUTPUT zz ENABLED_BY oe;
node1 = in1;
xx = node 1;
yy = xx * in2;
zz = node1 * in2;
```

Nodes

Nodes are signals in a design that are not visible outside the design (unlike INPUTs and OUTPUTs.) A node simply identifies a point in a logic design. This point (node) may be an actual physical point, or a virtual point that is collapsed during optimization. Physical and Virtual nodes are discussed in more detail later in this chapter.

A node without a clock (i.e., no CLOCKED_BY) will be a combinatorial node. Combinatorial nodes are useful building blocks for connecting separate pieces of combinatorial logic (much like a schematic net.) An equation for a node may be created in one part of a design and referenced in other parts.

The logic optimizer may choose to leave nodes in the design to be fit as physical nodes in hardware. The optimizer may also choose to remove a node by passing its equation logic to all equations that reference the node (this is called node collapsing.) One of two modifiers, PHYSICAL or VIRTUAL, may be used with the keyword NODE to control node collapsing. The PHYSICAL modifier is used to force the optimizer to create a physical node in hardware. VIRTUAL is used to force the optimizer to collapse a node during optimizing.

There are other control mechanisms for controlling node collapsing. These mechanisms are properties that are placed in a Physical Information (.pi) file. For more information on controlling node collapsing, see Chapter 13

Even though you may use the modifiers VIRTUAL and PHYSICAL, we recommend that NODE be used without either unless there is a specific reason to control node collapsing (e.g., when you need to duplicate a design.) By not using the modifiers except when absolutely necessary, you give the optimizer maximum freedom to reach the optimal equation sizes for the target hardware.

Nodes are declared with the NODE keyword using the following syntax:

```
[LOW_TRUE][flip_flop_type] [VIRTUAL|PHYSICAL] NODE
      identifier_or_array_list [control_info]
      [default_info];
```

Example

```
NODE x, y[4], z;  "declares combinatorial NODEs
                  "x, y[3], y[2], y[1], y[0], and z

JK_FLOP NODE x, y, z[6..4]  CLOCKED_BY clk;
                  "declares JK flip-flops x, y,
                  " z[6],z[5], z[4]
```

For example, with the declaration of i as a virtual node and its assignment as a*b:

```
INPUT  a, b, c;
VIRTUAL NODE  i;
OUTPUT  o;
```

```
i = a * b;
o = i * c;
```

The resulting assignment statement for o is:

```
o = a * b * c;
```

In the example given above, the VIRTUAL modifier forces the optimizer to remove the node. However, if the VIRTUAL modifer were not given, the optimizer would still have collapsed the node since the resulting equation is smaller.

In the following example, changing the PHYSICAL NODE declaration to VIRTUAL NODE also changes the generated equation:

```
INPUT  a, b, c;
OUTPUT  q;
PHYSICAL NODE  x;

x = a * b;
q = x * c;

x declared as a
PHYSICAL NODE is
implemented as:
q = a * b * c;
x = a * b;
```

```
INPUT  a, b, c;
OUTPUT  q;
VIRTUAL NODE  x;

x = a * b;
q = x * c;

x declared as a
VIRTUAL NODE is
implemented as:
q = x * c;
```

Note: Node collapsing is also dependent on other properties you may have specified (see Chapter 12).

Wired-Bus Signals

The WIRED_BUS declaration defines a group of nodes or outputs that are to be tied together electrically. Each node or output must have an ENABLED_BY expression that is independent of all others in the group, since

no two nodes or outputs may be enabled at the same time. A group of nodes can be referenced in expressions by declaring them as WIRED_BUS signals. The identifier named for the group has the value of whichever node is enabled.

The syntax for declaring wired bus signals is:

```
WIRED_BUS  identifier: node_1, node_2, .. node_n;
```

Where: *node_n* is an enabled node or output.

Alternatively, you can declare an array of wired bus signals:

```
WIRED_BUS  identifier[size]:  group_1, group_2, ..
group_n;
```

Where:group_n is a group or array of enabled nodes or outputs of the same width as the declared array.

With the following signal declarations:

```
INPUT  a, b;
NODE  x1  ENABLED_BY  a * b;
NODE  x2  ENABLED_BY  a * /b;
NODE  x3  ENABLED_BY  /a * b;
```

The wired bus declaration for w:

```
WIRED_BUS  w:  x1, x2, x3;
```

defines the new name w to be equal to the value of x1 when a*b = 1, the value of x2 when a*/b = 1, and the value of x3 when /a*b = 1. w can be used in expressions just like any other signal.

Several arrays and individual signals are declared as follows:

```
INPUT  a, b;
NODE  x1[4]  ENABLED_BY  a * b;
NODE  x2[4]  ENABLED_BY  a * /b;
NODE  q, r, s, t  ENABLED_BY  /a * b;
```

can be tied together by declaring a WIRED_BUS:

```
WIRED_BUS  w[4]:  x1, x2, [q, r, s, t];
```

This creates the following connections:

w[3] represents the connection of x1[3], x2[3], and q.

w[2] represents the connection of x1[2], x2[2], and r.

w[1] represents the connection of x1[1], x2[1], and s.

w[0] represents the connection of x1[0], x2[0], and t.

Declaration Modifiers

Declaration modifiers are optional parameters that may be used when declaring certain kinds of signals. These include:

- the low-true designator for inputs, nodes, outputs, and biputs
- the flip-flop type designators for outputs, biputs, and nodes
- control information for outputs, biputs, and nodes
- default information for outputs, biputs, nodes, and return values of functions
- LOW_TRUE

The optional LOW_TRUE modifier is used to define an input, output, biput, or node as low-true, indicating that a low voltage will represent the true state. The LOW_TRUE modifier appears first in a signal declaration:

```
LOW_TRUE  INPUT  x, y, z;
```

A signal may also be declared as low true by preceding the signal name with the logical negation symbol (/) in the declaration:

```
INPUT  x, /y, z;  "Declares inputs x, y as low-true,
                  "and z
```

If low-true is not indicated by either the LOW_TRUE modifier or the logical negation symbol when the signal is declared, the signal will default to high true.

Flip-Flop Types

Node and output declarations may be preceded by a flip-flop type that allows the signal to be described as the designated flip-flop or latch. The optimizer will synthesize equations for other flip-flop types, allowing the fitting tools to implement the signal using the most efficient actual hardware flip-flop type. The declared type allows the design to be described in the most convenient way for the user.

When a flip-flop type is declared, an accompanying CLOCKED_BY expression or LATCHED_BY expression (in the case of a D_LATCH) must also be declared. If a flip-flop type is declared without a CLOCKED_BY or LATCHED_BY expression, the compiler will generate an error.

The syntax for declaring a node or output is:

```
[flip_flop_type] NODE identifier_or_array_list
      [control_info] [default_info];
[flip_flop_type] OUTPUT identifier_or_array_list
      [control_info] [default_info];
```

Where *flip_flop_type* is D_FLOP, D_LATCH, JK_FLOP, SR_FLOP, or T_FLOP

When referencing a node or output signal that has a JK, SR, or T flip-flop type, the corresponding suffix (.J, .K, .R, .S, or .T) must be appended to the node or output signal name when an expression is assigned to it. .D is optional for D flip-flops.

A declaration without a flip-flop type and without a CLOCKED_BY or LATCHED_BY modifier will be combinatorial.

D_FLOP

D_FLOP defines a node or output to be a D flip flop. If no flip-flop type is specified and a CLOCKED_BY expression is used when declaring a node or output, a D-type flip-flop will be assumed.

Since the D-type flip-flop is the default register type, D-type node or output signals do not require a .D suffix when an expression is assigned to it.

The optimizer will synthesize all other flip-flop types for this equation, allowing MACHXL to fit any type.

Valid declarations and uses of D_FLOP include:

```
D_FLOP NODE a CLOCKED_BY clk; "D_FLOP is optional
NODE a CLOCKED_BY clk;        "since it is the
a = b;                        " default suffix.
                              ".D not required for
                              " node a with D_FLOP
```

An invalid use of a D_FLOP in an assignment statement would be:

```
NODE x CLOCKED_BY clk;  "x is declared by default
x.j = 1;                "as a D_FLOP, not a
                        "JK_FLOP
```

D_LATCH

D_LATCH defines a node or output to be a latched signal for a D-latch type device. The declaration modifier LATCHED_BY (rather than CLOCKED_BY) must be used in the declaration statement when D_LATCH is specified for flip_flop_type. If a flip-flop type other than D_LATCH is declared with a LATCHED_BY expression, the compiler will generate an error.

Note: D_FLOP gives partitioning a greater number of device architectures to choose from in selecting devices for a design than does D_LATCH. For this reason, we recommend using D_FLOP whenever possible, rather than D_LATCH.

The following is a valid LATCHED_BY declaration:

```
D_LATCH NODE b LATCHED_BY latch;
```

An invalid D_LATCH declaration would be:

```
D_LATCH  NODE b CLOCKED_BY clk;      "D_LATCH requires a
                                     "LATCHED_BY
                                     "expression
```

JK_FLOP

JK_FLOP defines a node or output to be a JK flip-flop type. Expressions are assigned to JK flops by appending the .J or .K suffix to the signal name (e.g., FLOP1.J, FLOP1.K). If an expression is assigned to a signal using the .J or .K suffix but has not been declared as the JK_FLOP type, the compiler will generate an error.

MACHXL's optimizer will synthesize versions of all other flop types, allowing the tools to fit any of these versions.

The following two examples indicate valid declarations and uses of JK_FLOP:

```
JK_FLOP  OUTPUT  jk1  CLOCKED_BY  clk;
JK_FLOP  NODE  jk1  CLOCKED_BY  clk;
jk1.j = 1;
jk1.k = 0;
```

Invalid uses of JK_FLOP include:

```
JK_FLOP  NODE  jk1;      "declaration missing
                         "CLOCKED_BY expression
jk1 = a;                 "jk1 missing .J or .K
                         "suffix in assignment
                         "statement
```

SR_FLOP

SR_FLOP defines a node or output to be an SR flip-flop. Expressions are assigned to SR flops by appending the .S or .R suffix to the signal name (e.g., FLOP1.S, FLOP1.R).

MACHXL's optimizer will synthesize versions of all other flop types, allowing the tools to fit any of these versions.

Valid declarations and uses of SR_FLOP include:

```
SR_FLOP NODE sr1 CLOCKED_BY clk;
SR_FLOP NODE sr1 CLOCKED_BY clk;
sr1.s = 0;
sr1.r = 1;
```

Invalid uses of SR_FLOP include:

```
SR_FLOP  OUTPUT  sr1;    "missing CLOCKED_BY
                         "expression in declaration
sr1 = a;                 "sr1 missing .S or .R
                         "suffix in assignment
                         "statement
```

T_FLOP

T_FLOP defines a node or output to be a T flip-flop. Expressions are assigned to T nodes or outputs by appending the .T suffix to the signal name (e.g., FLOP1.T).

MACHXL's optimizer will synthesize versions of all other flop types, allowing the tools to fit any of these versions.

Valid declarations and uses of T_FLOP include:

```
T_FLOP OUTPUT t1 CLOCKED_BY clk;
t1.t = 0;
T_FLOP NODE t1 CLOCKED_BY clk;
t1.t = 1;
```

Invalid T_FLOP uses include:

```
T_FLOP NODE t1;    "missing CLOCKED_BY expression in
                   "declaration
t1 = a;            "t1 missing .T suffix in
                   "assignment statement
```

Control Information Constructs

Nodes or output signals can be declared with any combination of one or more optional control information constructs. Each construct may be used only once per declaration. Control information constructs include:

```
CLOCKED_BY expression
      [CLOCK_ENABLED_BY expression]
LATCHED_BY expression
ENABLED_BY expression
RESET_BY expression
PRESET_BY expression
```

Note that in a declaration, control information constructs come after the identifier list:

```
[LOW_TRUE] [flip_flop_type] NODE identifier_list
      [control_info] [default_info];
[LOW_TRUE] [flip_flop_type] OUTPUT identifier_list
      [control_info] [default_info];
```

Each control information construct is described in its own section.

CLOCKED_BY

CLOCKED_BY defines an expression for clocking the register of a flip-flop. When the CLOCKED_BY expression becomes true, signals on the input of the register are clocked to the output of the register on the positive edge. For D-latches, use LATCHED_BY rather than CLOCKED_BY.

LATCHED_BY

LATCHED_BY defines an expression for timing the latch of a D_LATCH. As long as a LATCHED_BY expression is true, signals on the input of the latch are transferred to the output of the latch (i.e., the latch becomes

transparent from input to output). When the LATCHED_BY expression is false, the latch holds the last value.

CLOCK_ENABLED_BY

Valid only when preceded by a corresponding CLOCKED_BY expression. Defines an enabling expression that must be true in order for the CLOCKED_BY expression to be seen by the register.

RESET_BY

RESET_BY defines an expression for the asynchronous resetting of the register. When the RESET_BY expression is true, the corresponding register is false.

Note: Since only registers can be reset, a RESET_BY statement must be accompanied by CLOCKED_BY or LATCHED_BY.

PRESET_BY

PRESET_BY defines an expression for the asynchronous presetting of the register. When the PRESET_BY expression is true, the register is true.

Note: Since only registers can be reset, a PRESET_BY statement must be accompanied by CLOCKED_BY or LATCHED_BY.

ENABLED_BY

ENABLED_BY defines an expression to be used as an enable control on a registered or combinatorial node. The node's value is enabled when the ENABLED_BY expression is true. When it becomes false, the node's value is .Z. (tri-state).

Note: Assignment of .Z. to a signal is an alternate method of creating an ENABLED_BY expression. For example:

NODE n;
*IF a*b THEN*
*n=c*d*
ELSE
n=.Z.;
END IF;
is exactly the same as:
*NODE n ENABLED_BY a*b;*
*n=c*d;*

Default Information Constructs

Optional default information statements may be used with nodes, outputs, output parameters of procedures, and return values of functions. The default information *[default_info] constructs include:*

```
DEFAULT_TO expression
DEFAULT_TO expression, expression
DEFAULT_TO LAST_VALUE
DEFAULT_TO .X.
NO_REDUCE
```

DEFAULT_TO

DEFAULT_TO defines a value to which a node will default if not explicitly assigned a value. These situations include the following:

- unspecified ELSE clause of an IF construct,
- unspecified conditions of a CASE statement,
- unspecified conditions in a truth table,
- unspecified STATE values in a state machine,
- any other conditions in which there is no assignment to the signal of interest.

The following default values can be specified for a node or output: an expression (such as 0, 1, a*b), .X. (DON'T CARE), and LAST_VALUE. Commonly, the default value used will be 0 (denoting false) since this allows the designer to specify only those cases when a signal is true.

If no DEFAULT_TO statement is given, the compiler assumes the default value of DON'T CARE (.X.). The DON'T CARE value allows the optimizer to produce the smallest equations possible. However, this also means that the value of the signal in the default condition is unpredictable.

LAST_VALUE causes a register to default to itself so that its value will not change unless otherwise specified. LAST_VALUE can be used only with registers (i.e., signals with CLOCKED_BY.)

If a node has been declared as a JK or SR flip-flop type, DEFAULT_TO is followed by two expressions separated by commas:

```
DEFAULT_TO  expression1, expression2;
```

The first expression is the default value for the .J or .S inputs and the second expression is the default for the .K or .R inputs.

For example, the DEFAULT_TO statement in the following declaration of the JK flip-flop output `out1` causes `out1.j` to default to 0 and `out1.k` to default to 1:

```
JK_FLOP OUTPUT out1 CLOCKED_BY clk DEFAULT_TO 0, 1;
```

Note: When using the DEFAULT_TO in a statement with other modifiers, DEFAULT_TO must be the last item in the statement. The following is a legal use of DEFAULT_TO:

JK_FLOP OUTPUT out1 CLOCKED_BY clk DEFAULT_TO 0, 1;

The following is an illegal use of DEFAULT_TO:

JK_FLOP OUTPUT out1 DEFAULT_TO 0, 1 CLOCKED_BY clk;

NO_REDUCE

NO_REDUCE can be substituted for DEFAULT_TO in a node or output signal declaration in order to inhibit reduction of an equation or group of symbols. NO_REDUCE also prevents the optimizer from performing flip-flop synthesis. The declared flip-flop type will be used. Reduction within a single product term is still performed, as demonstrated in the following example:

When the nodes hmem and hblock are declared with the NO_REDUCE default information statement:

```
D_FLOP NODE hmem, hblock CLOCKED_BY clk NO_REDUCE;
```

Normal reduction will not be performed on the equations:

```
hmem=a*b*c+a*b+a*c+b*c;
hblock=a*b*c*a*b*a+a*b
```

However, the equation for hblock will be reduced to a*b*c+a*b because there are duplicate signals in the first product term (a*b*c*a*b*a).

When used in a function declaration, the NO_REDUCE modifier tells the compiler not to reduce the function's return value. When NO_REDUCE is used in a procedure output declaration, the compiler will not reduce the procedure output equation.

One purpose of NO_REDUCE is to allow the creation of hazard-free equations. Redundant product terms can be added where these product terms would otherwise be reduced out.

The following declarations indicate that output c of procedure p and the return value b of the function compare will not be reduced:

```
PROCEDURE  p(INPUT  a,b;  OUTPUT  c  NO_REDUCE);
FUNCTION  compare(a, b)  NO_REDUCE;
```

6 Expressions

Contents

Introduction

This chapter discusses the operators used to construct expressions in the Design Synthesis Language, operator precedence, and several types of expressions.

Combinations of one or more identifiers, signals, and/or constants that are related by operators are called expressions. Operators specify the operation to be performed among identifiers, signals, constants, and expressions.

Identifiers

There are three types of operators used in the Design Synthesis Language: arithmetic, logical, and relational.

Each operator has an operator precedence relative to other operators. This precedence affects the order of evaluation in an expression.

The following table is a listing of all of the expression types and usable operators in the Design Synthesis Language. The table indicates the relative precedence of the operators. All binary operators of equal precedence associate left to right.

Expression/ Operation	**Description**
constant	constant expression
identifier	simple signal or array
identifier[index [.. index]]	array reference
identifier(expression_list)	function invocation
[expression_list]	group
(expression)	parentheses for overriding default precedence

Expression/ Operation	Description	Precedence	Operator Type
/a	NOT	1	logical
*(a,b,c,d)	AND	1	expression shorthand (logical)
/*(a,b,c,d)	NAND	1	expression shorthand (logical)
+(a,b,c)	OR	1	expression shorthand (logical
/+(a,b,c,d)	NOR	1	expression shorthand (logical)
(+)(a,b,c)	XOR	1	expression shorthand (logical)
/(+)(a,b,c)	XNOR	1	expression shorthand (logical)
constant .*. constant	multiplication	2	arithmetic
constant ./. constant	division	2	arithmetic
constant .MOD. constant	modulo	2	arithmetic
a * b	AND	2	logical
a /* b	NAND	2	logical
a .+. b	addition	3	arithmetic
a .-. b	subtraction	3	arithmetic
a + b	OR	3	logical
a /+ b	NOR	3	logical
a (+) b	XOR	3	logical
a /(+) b	XNOR	3	logical
a=b	equal	4	relational
a<>b	not equal	4	relational
a<b	less than	4	relational
a>b	greater than	4	relational
a<=b	less than or equal	4	relational
a>=b	greater than or equal	4	relational
NOT a	logical negation	5	logical
a AND b	logical AND	6	logical
a OR b	logical OR	7	logical

Logical Operators

Logical operators are used to describe logical relationships among signals in expressions. The language supports the standard logical operators used to perform Boolean functions in programmable logic design.

Symbol	Description	Precedence
/a	NOT	1
a * b	AND	1
a /* b	NAND	1
a + b	OR	1
a /+ b	NOR	1
a (+) b	XOR	1
a /(+) b	XNOR	1

Equations built with the (+) exclusive-OR operator can be fit into devices with exclusive ORs and devices without exclusive ORs. Both representations of the equation are maintained throughout the system, allowing automatic partitioning to use either form.

Expression Shorthand (ES)

Expression shorthand provides a convenient way to express an operation on many expressions. Expression shorthand may be used for the commonly used logical operators: *, +, (+), /*, /+, and /(+).

The syntax of expression shorthand is:

```
ES_Operator(expression_list)
```

Where `ES_Operator` is one of the following logical operators: *, +, (+), /*, /+, /(+).

The Expression Shorthand Operators all have highest precedence.

Shorthand Operator	Example	Evaluates to	Precedence
*	*(A,B,E(+)F)	A*B*(E(+)F)	1
+	+(A,B,D*E)	A+B+(D*E)	1
(+)	(+)(A,B,E)	A(+)B(+)E	1
/*	/*(B,C,D)	/(B*C*D)	1
/+	/+(B,C,D)	/(B+C+D)	1
/(+)	/(+)(B,D,F)	/(B(+)D(+)F)	1

The binary operation:

```
out1 = a7 * a6 * a5 * a4 * a3 * a2 * a1 * a0;
```

Using expression shorthand for the logical AND operator (*), may be shortened to:

```
out1 = *(a7 .. a0);
```

Relational Operators

The relational operators are used for comparing expressions (including identifiers, constants, and other expressions). Relational operations always give a *one (true)* or *zero (false)* value as their result.

Symbol	Description	Precedence
a = b	equal	4
a <> b	not equal	4
a < b	less than	4
a > b	greater than	4
a <= b	less than or equal	4
a >= b	greater than or equal	4

For the relational operators <, >, <=, and =>, the compiler will by default insert a node at each bit position of the operation. MACHXL's optimizer will then remove most of these nodes, resulting in optimal equation sizes, according to the constraints placed on the optimizer. For more information on the compiler and optimizer operations, see **Chapters 11** and **13.**

The logical operators OR, AND, and NOT have the same behavior as +, *, and / but have lower precedence than the relational operators. These operators are useful for combining relational expressions.

Relational operations may be performed on arrays and groups, as in the following example:

```
IF [a[1..4] >= 5 AND a[1..4] <= 2 THEN
      x = (a=17);          "If array a has value 17, x
END IF;                    "= 1.Otherwise, x = 0.
```

Some comparison expressions involving relational operators and their results include the following:

Operation	Result
a = 1	True, if a has a value of 1
a = 1	False, if a has a value of 0
b <> c	True, if b has a value of 1 and c has a value of 0 or vice versa
a = b OR a = c	True, if a has a value as b or c

Arithmetic Operators

The arithmetic operators are used for performing arithmetic operations on arrays, groups or constants.

Operator	Description	Example	Precedence
constant .*. constant	multiplication	5.*.7	2
constant ./. constant	division	10./.2	2
constant .MOD. constant	modulo	17.MOD.3	2
a.+.b	addition	a.+.b	3
a.-.b	subtraction	a.-.b	3

The arithmetic operators .*. (multiplication), ./. (division), and .MOD. (modulo) can only be used with constants, as shown in the table above. The .+. (addition) and .-. (subtraction) operations may be performed on any array or group built from signals or constants. The compiler will, by default,

generate a node at each bit of an addition or subtraction operation. MACHXL's optimizer will collapse most of these nodes to produce an optimal set of equations, regardless of the form of the operands. For example, constants in operands require less logic and will result in more nodes being collapsed. For more information on the operation of the optimizer, see **Chapter 12.**

The result of the .+. (addition) and .-. (subtraction) operations will be the same array size as the operands. This means that if a carry bit is generated, it is thrown away. In the following example, the array count can represent values from 0 to 1023. If the value of count is 1023 and 1 is added to the array, the count rolls over to 0 and the carry bit is lost.

```
NODE count[10] CLOCKED_BY clk;
count=count .+. 1;        "counts by 1, rolls over
                          "at 1023,no carry bit
```

If you need to keep the carry bit, pad the operands with leading zeros, as shown in the following example.

```
INPUT a[10], b[10];
OUTPUT x[11];             "define an array 1-bit
                          "wider to accept the carry
                    "bit

x=[0,a] .+. [0,b]         "add arrays a and b into x
                          "including the carry bit
```

Constant Expressions

Constants can be used alone or with operators to form expressions. An operator that acts only on constant expressions results in a constant expression. If an operator acts on a constant and a non-constant, then the constant is assumed to have a bit width equal to that of the non-constant expression. If the value of the constant can not be represented in the available bits then an error is generated.

Constant expressions are required in contexts such as array size declarations.

Examples

```
5 * 7 + 128          "This results in the constant
                     "163.
5 * [a,b,c]          "This results in [a, 0, c]
13 * [a,b,c]         "This is an error since 13 cannot
                     "be represented in 3 bits.

MACRO size 10;
INPUT in [size];
OUTPUT out [size .*. 2];
```

Using Parentheses to Change Precedence

Precedence in an expression may be overridden by use of parentheses. For example, since logical AND (*) has higher precedence than logical OR (+), the following expression:

```
a * b + c
```

will be evaluated as follows:

```
(a * b) + c
```

However, by using parentheses, you may override the default. Thus, in the expression:

```
a * (b + c)
```

(b + c) will be evaluated first, then the result will be AND'd with a.

Groups and Ranges

A group of signals that will perform similar functions and which you want to treat similarly can be referred to using brackets [ ]. Signals grouped together within brackets can be assigned a single value or can be specified to take on the values of another set of signals.

In the following assignment statement, the four signals a, b, c, and d are all set to zero.

```
[a,b,c,d] = 0;
```

Without group notation the previous operation would require four assignment statements as shown below:

```
a = 0;
b = 0;
c = 0;
d = 0;
```

The order in which signals are listed in a group is important. The first (left-most) signal in the group (e.g., a in the previous example) is the Most Significant Bit and the last signal (right-most) specified (d) is the Least Significant Bit. This is important when you set a group of signals equal to a value.

You may combine group notation with the range notation. The range statement,

```
[q3..q0] = 5;
```

is interpreted by the Design Synthesis Language as:

```
[q3,q2,q1,q0] = [0,1,0,1];
```

q3 is listed first, so the range is in descending order: q3, q2, q1, q0. The binary representation of the numeral 5 is 0101, so the signals will be set to the following values:

```
q3 = 0;
q2 = 1;
q1 = 0;
q0 = 1;
```

If the order in the range is reversed ([q0..q3] = 5), the Most Significant Bit would be q0, and the values for the assignments would become:

```
q0 = 0;
q1 = 1;
```

```
q2 = 0;
q3 = 1;
```

To assign a group of signals to another group of signals, you can use the group notation and the assignment operator:

```
[q3..q0] = [d3..d0];
```

In assigning groups, you may use the numerals 0 and 1, the don't care symbol .X., and the tri-state symbol .Z., in the group on the right side of the assignment operator:

```
[a,b,c,d] = [a,0,.X.,1];
```

The don't care symbol .X. may also be used within ranges that are acted upon by relational operators. Wherever .X. appears in the range, the compiler will ignore that term when doing a comparison of ranges.

Thus, the statement:

```
IF [a15..a0] >= [b15..b6,1,.X.,.X.,.X.,0,b0] THEN
      x = y;
END IF;
```

is exactly the same as:

```
IF [a15..a5,a1,a0] >= [b15..b6,1,0,b0] THEN
      x=y;
END IF;
```

You may also perform operations on groups, such as the following:

```
[a,b,c,d] = /[a,0,a+b,1]*addr[3..6];
```

A member of any group that is itself a group will be unfolded so that its members become members of the containing group.

Thus, with the following node declaration:

```
NODE a[2],b[2],c[4];
```

The statement:

```
[a[1..0],b[1..0]] = c[3..0];
```

has the same meaning as:

```
[a[1],a[0],b[1],b[0]] = [c[3],c[2],c[1],c[0]];
```

And

```
[a,c] = [1,c,0];
```

is equivalent to:

```
[a[1],a[0],c[3],c[2],c[1],c[0]] =
[1,c[3],c[2],c[1],c[0],0].
```

The following example shows a number of range and group notations as they might appear in the context of other source code.

Example

```
INPUT rd, wt, dir;
OUTPUT q7..q0, up, down;

IF ([rd,wt]=01b) THEN
      [q7..q0] = 00000000b;
      [up,down] = 00b;
ELSE
      [q7..q0] = [q0,q7..q1];  "performs a rotate
      [up,down] = 01b;
END IF;
```

Array Expressions

An array is a set of logically related signals that can be treated separately or as a unit. (See **Chapter 5**, and the section on Arrays.)

An array, a subrange of an array, or an individual array element may be assigned a single value. Or they may be specified to take on the values of another set of signals.

Each element of an array can be indexed and used as an ordinary signal. Each array element, a range of array elements, or the array as a whole may also be given individual assignment statements.

For instance, for the array addr declared as follows:

```
OUTPUT addr[16] ;
```

The value of an individual element can be referenced:

```
a = addr[5];
```

In this case, addr[5] is being assigned to a.

You may also assign a subrange of addr to individual signals as shown below:

```
addr [10..0] = [x10, x9, x8, x7, x6, x5, x4, x3, x2, x1,
x0];
```

A subrange of addr may also be referenced:

```
addr[2..6] = 21;
```

In this case, array elements addr[2] through addr[6] are assigned the corresponding values on the right. addr[2] is the Most Significant Bit and addr[6] is the Least Significant Bit.

This assignment is equivalent to:

```
[addr[2],addr[3],addr[4],addr[5],addr[6]] = [1,0,1,0,1];
```

In addition, you may assign the array addr to a combination of another smaller array and individual signals, as shown below:

```
addr = [ a[ 9..0 ] , x1, x2, x3, y1, y2, y3 ] ;
```

In this case addr[15] through addr[6] would be assigned to array elements a[9] through a[0], and addr[5] through addr[0] would be assigned to x1, x2, x3, y1, y2, and y3 respectively.

The following array assignment ANDs the array addr with the hexadecimal constant value FF:

```
addr = addr*00ffH
```

which is also equivalent to:

```
addr[15..0] = [0,0,0,0,0,0,0,0,addr[7..0]]
```

For the array *q* declared as follows:

```
OUTPUT q[4..7] CLOCKED_BY clk;
```

the assignment

```
q[7..5] = q[4..6];
```

is equivalent to

```
q[7]=q[4];
q[6]=q[5];
q[5]=q[6];
```

The above assignments would cause q[5] and q[6] to take on the same values.

Don't Care Condition

The Don't Care condition is denoted by .X in the Design Synthesis Language. You can use the Don't Care condition explicitly when describing the value of a signal. The optimizer will then assign either a 0 or 1 to the signal, depending on which produces the smallest equation.

Examples

The following is an example of a valid usage of Don't Care:

```
f = a * /b * .X.;          ".X. should be at the end
                           "of the equation
```

The following is an example of an invalid usage of Don't Care:

```
f = .X. * a * /b;          "This will produce
                           "incorrect results
```

You can also use .X to describe the behavior of undeclared states in a state machine. The following example completely specifies all possible conditions of a state machine, and ensures the most optimal equation generation:

```
STATE_MACHINE dont_care CLOCKED_BY clk;

STATE one:
      IF count THEN
            clear = 1;
            IF (pulse = 0) THEN
                  count8 = 1;
                  GOTO one;
            ELSE
                  GOTO one;
            END IF;
      END IF;

STATE two:
. . .

ELSE
      GOTO .X.;
      clear = .X.;
      count8 = .X.;
END dont_care;
```

7 Statements and Constructs

Contents

Introduction

The Design Synthesis Language provides various kinds of statements and constructs that may be used to build design equations. The types available include:

- Assignment statements
- IF statements
- CASE statements
- TRUTH TABLE constructs
- STATE MACHINE constructs.

Each of these is discussed in detail in this chapter.

Assignment Statements

The assignment statement is used to describe the values of the assignable signals (OUTPUT, NODE, BIPUT) in a design. An expression is assigned to a signal or group of signals by means of the assignment operator (=).

The syntax of the assignment statement is:

```
assignment_expression = expression;
```

Where:

```
assignment_expression:  identifier [suffix]
                        identifier [index] [suffix]
                        identifier [index..index] [suffix]
                        [assignment_expression_list]

suffix: is one of the following:    .D
                                    .J
                                    .K
                                    .R
                                    .S
                                    .T
```

In the assignment of flip-flop signals, an optional suffix may be used to indicate which of a flip-flop's equations is being assigned: D, J, K, R, S, or T. The .D suffix is optional on D_FLOPs. As with expressions, arrays can be assigned in whole or in part.

Examples

```
INPUT  a, b;
OUTPUT  x;
D_FLOP  d,  darr[4]  CLOCKED_BY  a;
JK_FLOP  jk,  jkarr[4]  CLOCKED_BY  a;
SR_FLOP  sr  CLOCKED_BY  a;
T_FLOP  t1..t4  CLOCKED_BY  a;

x = a * b;
d.D = a + b;
darr = jkarr;
jk.J = a;
jk.K = b;
jkarr[2..1].J = [a, b];
jkarr.K = [a, b, 1, 0];
[sr.R, sr.S, t1.T..t4.T] = [a, b, a * b, 1, 0, a + b];
```

IF Statements

The IF statement allows an expression's value to determine whether a body of statements will take effect. The syntax of an IF statement is:

```
IF expression THEN
      statements
{ELSIF expression THEN
      statements}
[ELSE
      statements]
END IF;
```

The expressions in an IF statement must be single-bit values; they cannot be multi-bit width arrays or groups. If an expression has a value of 1, then it is a true condition; otherwise it is a false condition.

The statements contained inside the THEN take effect only when the corresponding expression is true.

An IF statement may contain any number of optional ELSIF clauses. The statements contained in an ELSIF clause take effect if its expression is true and all preceding expressions are false.

An IF statement may contain an optional ELSE clause. The statements contained in an ELSE take effect if all other expressions are false.

The IF statement ends with END IF;.

Example

```
IF a * b THEN
      x = c;
ELSE
      x = d;
END IF;
```

The resulting equation for x will be x = a*b*c + /a*d + /b*d.

CASE Construct

The CASE construct allows you to compare an expression against a list of values, each of which has associated statements. If the expression matches a given value, the associated statements take effect. Multiple values and ranges may be given with each value.

The CASE construct may include an ELSE statement that processes any value not specified by the listed values. The CASE construct ends with an END CASE statement.

The syntax of the CASE construct is:

```
CASE expression
      WHEN value_range =>
            statements
      [ELSE
            statements]
END CASE;
```

Where:

value_range is a list of numbers or a range of numbers.

The value of the CASE expression is compared to each of the values in the *value_ranges* of the WHEN clauses. If the value of the expression matches any of the values in a value_range, the associated WHEN statements take effect.

Example

```
INPUT  a[8];
OUTPUT  x, y, z;

CASE a
WHEN 5=>            "The following statements take
      x = y;        "effect if a = 5
y = x;

WHEN 7..15=>        "The following statement takes
      z = x;        "effect if 7 ≤ a ≤ 15.

WHEN  30..41, 53, 57, 100..113=>    "The following
      z = y;              "statement takes effect if
                          "a = any of these values

ELSE                "The ELSE statement takes effect
      y = z;        "if a does not match any of the
                    "WHEN values
END CASE;
```

TRUTH_TABLE

A truth table provides a convenient way to list output values for selected input expression combinations. Any or all of the possible input combinations may be used.

The syntax of a truth table is:

```
TRUTH_TABLE
      expression_list :: assignment_expression_list;
            value_range_list :: expression_list;
            ELSE:: expression_list;
END TRUTH_TABLE;
```

Where:

expression is defined in **Chapter 7**;
assignment_expression is defined earlier in this chapter in the **Assignments Statements** section;
value_range is defined in the preceding section under the heading **CASE Construct**.

To set up a truth table, list all input expressions to the left of a double colon (::) and all signals that are to be assigned to on the right of the double colon. List corresponding values for the signals in column format under the signal names.

A .X. input value tells the compiler to ignore the corresponding input expression when creating the condition. A .X. output value tells the compiler to assign DON'T CARE to the corresponding output symbol.

For a .Z. output value, the compiler will build the necessary equation for the output enable to cause a high impedance value for the corresponding output signal. In the following example, if *a* and *enable* are low, the output *x* will be placed in a high impedance (.Z.) state.

```
TRUTH_TABLE
      a, enable   ::   x;
      "_____________________
      0,  0       ::  .Z.;
END TRUTH_TABLE;
```

The compiler automatically checks for duplicate input terms that yield different output values. For example, the following will generate a compiler error because the input values overlap:

```
TRUTH_TABLE
      a, b, c :: x;
"-----------------
      0, 0, 0 :: 0;
      0, 0, 0 :: 1;
END TRUTH_TABLE;
```

The ELSE statement may be used in a truth table to process unspecified input conditions.

As with other statements, the truth table construct may be nested within other constructs (IF, CASE, etc.). When a truth table is nested within another construct, the resulting equations will be affected by the conditions of the parent construct.

The following example sets up a truth table using an array of nodes and individual node identifiers:

```
NODE a[4], b, c, x, j;

TRUTH_TABLE
a,  b,  c  ::  x,  j,  a[0..3];     "An array can be used
                                    "in the expression
                                    "list
"-----------------------------------
0,  1,  0  ::  d,  b*c,  [b, c, x, j];   "In this case,
                                         "j = b * c
1,  1,  x  ::  0,  x,  5;     "In this case, a[0..3] =
                              "[b, c, x, j]
15, 1,  x  ::  1,  0,  5;     "In this case, c is tested
                              "against x
ELSE       ::  x,  1,  15;    "Outputs for all other
                              "cases
END TRUTH_TABLE;
```

The next example demonstrates how a truth table may be used inside an IF statement:

```
IF a = b THEN
TRUTH_TABLE
      c, d :: e, f;
      "------------
      0, 0 :: 1, 1;
      0, 1 :: 1, 0;
      1, 0 :: 0, 1;
      1, 1 :: .X., .X.;              "Don't Care
END TRUTH_TABLE;

ELSE
      f = 0;
      e = 0;
END IF;
```

STATE_MACHINE Construct

The STATE_MACHINE construct is an efficient way to describe sequential logic. A state machine features a set of unique states; each state performs a set of operations, including branching to the next state in the state machine sequence.

The syntax of a state machine is as follows:

```
STATE_MACHINE  identifier  [state_machine_control_info];
      {STATE  state_name  [[value]]:      "Description of
                              "first state including
      statements}             "GOTO statements
      {STATE  state_name  [[value]]:      "Description of
                              "second state including
      statements}             "GOTO statements
      .
      .
      .
```

```
[ELSE                "Description of behavior of
                     "undeclared states including
      statements]    "GOTO statements

END  identifier;
```

Where:

```
state_machine_control_info = [CLOCKED_BY expression]
                             [RESET_BY expression]
                             [default info * ]
                             [STATE_BITS array]
                             [STATE_BITS group]
                             [STATE_VALUES identifier]
```

* *default_info* is discussed in **Chapter 6** in the **Declaration Modifiers** section.

State machines use hardware signals to keep track of which state the state machine is in. These hardware signals are called state bits. If state bits are not explicitly declared with the STATE_BITS construct, the DSL compiler will automatically generate nodes to act as state bits for the design (the STATE_BITS construct is discussed later in this chapter).

If the state machine is declared with a CLOCKED_BY construct, the state machine will be a *synchronous* state machine.

If the state machine does not have a CLOCKED_BY construct and the state bits are combinatorial, the state machine will be an *asynchronous* state machine.

STATE_MACHINE statements can be nested within other constructs, (i.e., CASE, IF, Functions, Procedures, TRUTH_TABLES) or may be nested within themselves.

The elements of the state machine description are discussed in the following sections.

CLOCKED_BY (in a STATE_MACHINE)

The CLOCKED_BY construct controls when the state machine will advance to the next state. IF the state machine declaration includes a CLOCKED_BY construct, the state machine will be a synchronous state machine. A synchronous state machine advances to the next state in the sequence when the CLOCKED_BY expression goes true.

The syntax of CLOCKED_BY is as follows:

```
CLOCKED_BY expression
```

The signal declaration for the state bits can also determine if the state machine is a synchronous state machine. If explicitly declared state bits are registered signals (i.e., declared with a CLOCKED_BY construct in the NODE, OUTPUT, or BIPUT statements), the state machine will also be considered a synchronous state machine.

If the state machine does not have a CLOCKED_BY construct, and if the explicitly declared state bits are combinatorial, the state machine will be an asynchronous state machine. An asynchronous state machine will advance to the next state when a GOTO statement is encountered in a STATE declaration. For additional information on asynchronous state machines, see the section later in this chapter entitled ***Asynchronous State Machines***.

Rules for Using CLOCKED_BY in a State Machine

If the state machine includes explicitly declared state bits (using the STATE_BITS construct), the following rules apply to the state machine CLOCKED_BY expression:

- The CLOCKED_BY expression for the state machine must match the CLOCKED_BY expression for all the state bit signals. The CLOCKED_BY expression for the state bit signals is included in the NODE, OUTPUT, or BIPUT statement that is used to declare the state bit signals.

- If the state machine is an asynchronous state machine, the state bit signals must be declared combinatorial (i.e., no CLOCKED_BY construct in the NODE, OUTPUT, or BIPUT statements).

- If the explicitly declared state bits are registered signals (i.e., declared with a CLOCKED_BY expression), the state machine will be considered a synchronous state machine.

For additional information on declaring state bits, see the ***STATE_BITS (in a STATE_MACHINE)*** later in this chapter.

Examples

The following example shows a synchronous state machine with explicitly declared state bits. Note the CLOCKED_BY expression for the state bits matches the CLOCKED_BY expression for the state machine:

```
NODE sb[4]  CLOCKED_BY (/clk);

STATE_MACHINE  sync_machine
      STATE_BITS  sb  CLOCKED_BY  (/clk);
      .
      .
      .
```

The following example shows another way to declare a synchronous state machine. In this case, the STATE_MACHINE statement does not include a CLOCKED_BY statement. The state machine is forced to be a synchronous machine by the explicitly declared state bits with a CLOCKED_BY statement.

```
NODE  sb[4]  CLOCKED_BY (/clk);

STATE_MACHINE  sync_machine
      STATE_BITS  sb;
      .
      .
      .
```

The following example shows an asynchronous state machine with explicitly declared state bits. Note that the state machine declaration does not include a CLOCKED_BY statement and that the state bits are also declared without a CLOCKED_BY statement.

```
NODE   sb [4];

STATE_MACHINE async_machine
       STATE_BITS sb;
       .
       .
       .
```

RESET_BY (in a STATE_MACHINE)

The RESET_BY statement lets you force (asynchronously) the state machine to the first declared state in the state machine. To force this transition, the individual state bits are asynchronously reset or preset to match the values of the first state.

The format for using RESET_BY in a state machine is as follows:

```
RESET BY expression
```

RULES for Using RESET_BY in a State Machine

- The RESET_BY construct may be used only with synchronous state machines (i.e., state machines that are declared with a CLOCKED_BY statement).

- If state bits are declared explicitly (using the STATE_BITS construct), the state bit signal declarations cannot include a RESET_BY or PRESET_BY statement. The DSL compiler will automatically determine the appropriate reset or preset expression for each individual state bit signal from the state machine RESET_BY statement.

For more information on declaring state bits, see the next section entitled ***STATE_BITS (In a STATE_MACHINE).***

Example

The following example shows a synchronous state machine with explicitly declared state bits and a RESET_BY statement. Note that the state bit signal declaration does not include RESET_BY or PRESET_BY statements.

In this example the *sb* signals will be set immediately to the value 0101b when *reset* is true. Setting the state bits to this value forces the state machine to the idle state.

```
NODE sb [4]  CLOCKED_BY clk;

STATE_MACHINE reset_machine
      STATE_BITS sb  CLOCKED_BY  RESET_BY reset;

      STATE idle [ 0101b ]:
      .
      .
      .
```

STATE_BITS (in a STATE_MACHINE)

State machines use hardware signals to keep track of the state a state machine is in. These hardware signals are called *state bits*.

A design can explicitly declare the state bits for a state machine by using the STATE_BITS construct. If state bits are not explicitly declared, the DSL compiler will automatically generate nodes to act as state bits for the design.

The format for the STATE_BITS construct is:

```
STATE_BITS array
      or
STATE_BITS group
```

Where:

> *array* is an array of signals previously declared with a NODE, OUTPUT, or BIPUT statement.

group is a group of signals previously declared with a NODE, OUTPUT, or BIPUT statement.

Rules for Using the STATE_BITS Construct in a State Machine

- State bit signals must be declared using the NODE, OUTPUT, or BIPUT statements before they can be used in a state machine.
- All of the state bits must be clocked by the same expression in a **synchronous state machine.** The CLOCKED_BY expression in the state machine must match the CLOCKED_BY expression in the NODE, OUTPUT, or BIPUT statements that declare the state bit signals.
- All of the state bits must be combinatorial (i.e., declared without a CLOCKED_BY expression in the NODE, OUTPUT, or BIPUT statements) in an **asynchronous state machine** (i.e., a state machine declared without a CLOCKED_BY expression).
- If a **synchronous state machine** includes a RESET_BY statement, the NODE, OUTPUT, or BIPUT statements that declare the state bits cannot have a RESET_BY or PRESET_BY statement. The DSL compiler will automatically determine the appropriate reset or preset expression for each indiviual state bit signal from the state machine RESET_BY statement. This eliminates any possibility of reset or preset conflicts.
- If a state machine includes default information, the NODE, OUTPUT, or BIPUT stetements that declare the state bits cannot have default information. The DSL compiler will automatically determine the appropriate default information for each individual state bit signal from the state machine default information. This eliminates any possibility of default conflicts.

You can assign unique values to the state bits for each state using one of the three following methods:

- Specify explicitly the state bit value in each state as part of the STATE declaration. With this method you must first specify the state bits using the STATE_BITS construct. See the heading ***STATE Declarations*** later in this chapter for more information.

- Specify an algorithm for assigning state bit values with the STATE_VALUES construct. This construct lets you use a gray code or one-hot assignment algorithm without having to specify explicitly each state bit value. See the heading ***STATE_VALUES*** later in this section for more information.

- Let the DSL compiler assign values automatically. With this method, the compler will assign the value of 0 (zero) to the first state in the state machine, 1 (one) to the second state, 2 to the third state, and so on. The state bit assignment process is a simple binary counter that starts at 0 (zero). The values are assigned by the compiler in the order in which the states are declared.

Example

This example uses a group of individual signals for the state bits. This state machine explicitly assigns a value to each state.

```
INPUT a, b, clk;
NODE c3 . . c0 CLOCKED_BY clk;

STATE_MACHINE counter  STATE_BITS [c3 . . c0]  CLOCKED_BY
      clk;
STATE one [0001b]:
      GOTO two;
STATE two [0010b]:
      IF a THEN
            GOTO three;
      ELSIF b THEN
            GOTO two;
      ELSE
            GOTO one;
      END IF;
```

```
STATE three[1100b]:
      GOTO one;
END counter;
```

STATE_VALUES

The STATE_VALUES construct lets the user control how state-bit values are assigned to states without explicitly assigning each value. The user declares the assignment algorithm with the STATE_VALUES construct.

The syntax of the STATE_VALUES construct is:

```
STATE_VALUES ONE_HOT
      or
STATE_VALUES GRAY_CODE
```

Rules for Using the STATE_VALUES Construct

- When you use the STATE_VALUES construct, you cannot explicitly assign state bit values to states. This would result in assigning two different values to the same state.

- If the STATE_VALUES construct is not used and the user does not explicitly assign state bit values for each state, the DSL compiler will automatically assign state bit values. In this case the compiler will assign the value of 0 (zero) to the first state in the state machine, 1 (one) to the second state, 2 to the third state, and so on. The default state bit assignment is a simple binary counter that starts at zero. The values are assigned in the order in which the states are declared.

ONE_HOT

The "one hot" algorithm assigns a unique state bit to each state (shown in the following example). This method is useful when targeting register-rich architectures. The format for the one hot bit selection method is:

```
STATE_MACHINE  sm_name  CLOCKED_BY clk  STATE_VALUES
      ONE_HOT;
```

Example

In the following state machine:

```
STATE_MACHINE  sm1  CLOCKED_BY clk  STATE_VALUES ONE_HOT;
      STATE  one: ...
      STATE  two: ...
      STATE  three: ...
      STATE  four: ...
```

The state values will be:

```
one   [0001b]
two   [0010b]
three [0100b]
four  [1000b]
```

With the one-hot bit selection method, the number of states is equal to the number of state bits. This makes the one-hot bit selection method less register-efficient than the default or GRAY_CODE methods. However, the equations for each state bit will be very efficient.

GRAY_CODE

An alternate algorithm, GRAY_CODE, causes the compiler to assign state bits like a gray-code counter.

With the gray-code counting method, consecutive state values are defined by changing only one bit, as shown in the following example. This reduces the possibility of race conditions when going from one state to a consecutive state in an asynchronous state machine. It may also result in smaller equations for JK, RS, and T flip-flop state machines.

```
STATE_MACHINE  gray  STATE_VALUES  GRAY_CODE;
      STATE  first: ...
      STATE  second: ...
      STATE  third: ...
      STATE  fourth: ...
```

```
      STATE  fifth: ...
      STATE  sixth: ...
END  gray;
```

Using the GRAY_CODE algorithm, the compiler will assign state values as follows:

```
first       [000b]
second      [001b]
third       [011b]
fourth      [010b]
fifth       [110b]
sixth       [111b]
```

STATE Declarations

The STATE construct allows you to declare the individual states in a state machine. The syntax of the STATE construct is as follows:

```
STATE identifier [[value]];
      statements
```

Where:

value is the optional state bit value that should be assigned in this state.

statements are DSL statements that define the behavior of this state. The statements can be used to assign values to signals. They can also be used to define transitions to other states. IF, CASE, TRUTH_TABLE, and other STATE_MACHINE statements can be used within a STATE declaration.

Rules for Using the STATE Construct

- The identifier must be a unique identifier in the design description. The design description cannot have a state with the same name as a signal or other identifier.

- The GOTO statement is used to transition to other states. If the state declaration does not include a GOTO statement, the transition will depend on the DEFAULT_TO construct for the state machine. The following table shows how the DEFAULT_TO construct controls this transition:

DEFAULT_TO value	Transition to
0	state whose state bit value is 0 (zero)
1	state whose state bit value is all ones
LAST VALUE	same state
.X.	unknown state

- The state machine can include as many states as necessary to implement the design

Example

The following example shows a state machine with multiple states, including conditional branching out of each state.

```
INPUT clk,  pwr_up,  start,  stop,  reset;
OUTPUT time[16]  CLOCKED_BY clk  RESET_BY pwr_up;
NODE sbits[2]  CLOCKED_BY clk  RESET_BY pwr_up;

STATE_MACHINE  stop_watch
      CLOCKED_BY clk
      DEFAULT_TO LAST_VALUE
      STATE_BITS  sbits;
```

```
STATE idle [00b]:          "Wait until the start
                           "button is pressed
      IF (start) THEN
            time = 1;
            GOTO count;
      ELSE
            time = 0;
            GOTO idle;
      END IF;

STATE count [01b] :        "Count up until the stop button
                           "is pressed
      IF (stop) THEN
            time = time;
            GOTO display_time;
      ELSE
            time = time .+. 1;
            GOTO count;
      END IF;

STATE display_time [10b]:      "Display the time until the
                               "reset button is pressed
      IF (reset) THEN
            time = 0;
            GOTO idle;
      END IF;
ELSE
      GOTO .X.;
      time = .X.;
END stop_watch;
```

GOTO Statement

The GOTO statement directs the transition from one state to another in a state machine. The syntax of a GOTO statement is:

```
GOTO state_name;
```

GOTO is allowed anywhere statements can occur in a STATE declaration.

Asynchronous State Machines

Sometimes you may need to create asynchronous state machines in order to avoid clocking delays. If a CLOCKED_BY expression is not declared for the STATE_MACHINE or state bits, the resulting state machine will be asynchronous.

Since registers are not used for the state bits in an asynchronous state machine, a circuit may depend on the device propagation delays to be stable. Also, logical hazards in the design may lead to unexpected transitions of the state machine. For these reasons, circuits should be designed to avoid race conditions and logical hazards.

One approach that may help reduce race conditions and logical hazards involves selecting state-bit values that cause only a single state bit to change when moving from one sequential state to another. The STATE_VALUES GRAY_CODE construct will perform this automatically for you.

In addition, the NO_REDUCE default information may help reduce logical hazards. If the state-bit equations contain redundant logic to avoid hazards, the NO_REDUCE construct will ensure that this extra logic is not reduced out of the design equations.

Example

```
STATE_MACHINE  states  STATE_BITS[s4..s0];
      STATE  one[00010b]:
             y = x;
             GOTO  two;
      STATE  two[00110b]:
             y = a;
             GOTO  three;
      STATE  three[01110b]:
             y = b
             GOTO one;
END states;
```

You can also use .X. to describe the behavior of undeclared states in a state machine. The following example specifies completely all possible conditions of a state machine, and ensures the most optimal equation generation.

```
STATE_MACHINE dont_care CLOCKED_BY clk;

STATE one;
      IF count THEN
            clear = 1;
            IF (pulse = 0) THEN
                  count8 = 1;
                  GOTO one
      ELSE
            GOTO one
            END IF;
      END IF;

STATE two;
. . .
```

8 Procedures and Functions

Contents

Introduction

Procedures and functions let you create logically distinct design blocks that are independent of the rest of the design. This lets you create hierarchical designs (i.e., designs that build complex functions from lower level blocks.)

Procedure and function descriptions do not create physical hardware. Their purpose is to describe functionality that can be used any number of times in a design. Only a function or procedure invoked at the system level (outside of the function or procedure description) results in actual hardware.

Procedures and functions may invoke other procedures or functions but may not invoke themselves. All statement constructs discussed in Chapter 8 (assignment, IF, CASE, TRUTH_TABLE, STATE_MACHINE, GOTO) may be used in a procedure or function.

All procedure and function descriptions must appear before any system-level design information. This includes system-signal declarations and system-level statements. A procedure or function must also appear before it is called by another function or procedure. See ***Chapter 2*** and the section ***entitled Building a MACHXL Design Synthesis Language Source File*** for an overview of how Procedure and Function definitions fit into the overall source file.

Procedures

Procedures are the main building blocks of hierarchical design. A hierarchical block diagram of a design can be easily mapped to a Design Synthesis Language description by mapping each section of the block diagram to a procedure in the language. The inputs and outputs of each block correspond directly to the inputs and outputs of a procedure.

Procedures are invoked at the system level and have both input and output parameters, allowing them to explicitly pass values in and out.

Declaring a Procedure

The syntax for declaring a procedure is:

```
PROCEDURE procedure_name
      (INPUT identifier_or_array_list;
            [flip-flop type] OUTPUT
            identifier_or_array_list
            [control_info][default_info]);
      local declarations
      statements
END procedure_name;
```

The following procedure description declares *and1* as having three parameters. The first two (*a, b*) are input parameters and the third (*x*) is an output parameter:

```
PROCEDURE and1(INPUT a, b; OUTPUT x);
      x = a * b;
END and1;
```

Once a procedure is declared, it can be invoked from within other procedures, functions, and at the system level. Procedures can be invoked anywhere an ordinary statement can be appear.

Invoking a Procedure

The format for invoking a procedure is:

```
procedure_name(expression_or_signal_list);
```

Where:

expression_or_signal_list consists of two parts:

1. expressions in corresponding positions to those in the input parameters, and
2. assignable signals (OUTPUT, BIPUT, NODE) in corresponding positions to those in the output parameters.

As an example, we use the following steps to create a system level design using the procedure *and1* shown above:

1. declare the actual inputs and outputs,
2. invoke the procedure with the appropriate expressions in corresponding positions to the input and output parameters of the procedure description, as shown below.

```
INPUT  in1, in2;
OUTPUT  result;

and1(in1, in2, result); "invoke and1, passing in1,
                        "in2 as input parameters
                        "and result as an output
                        "parameter
```

For more information about input and output parameters, see the sections following entitled ***Input Parameters*** and ***Output Parameters***.

As another example, the following procedure implements a 4-bit parity generator. *parity4* has two parameters: a 4-bit input array *x*, and a one-bit output *y*.

```
PROCEDURE parity4(INPUT x[4]; OUTPUT y);
      y = x[0] (+) x[1] (+) x[2] (+) x[3];
END parity4;
```

The *parity4* procedure description does not in and of itself cause hardware creation.

The following two invocations of the procedure *parity4* cause hardware to be created because they are invoked at the system level. These invocations implement two separate 4-bit parity generators.

```
INPUT a[4], b[4];
OUTPUT out1, out2;

parity4(a, out1);
parity4(b, out2);
```

Notice that the actual INPUTs and OUTPUTs for the two parity generators are also declared at the system level.

Functions

Functions are a useful way to describe distinct pieces of logic that result in an expression value.

Functions are invoked from within an expression. They have input parameters that allow them to accept values into the function, but generate as output a return value that is passed back to the original expression.

Declaring a Function

The syntax for declaring a function is:

```
FUNCTION function_name( [INPUT] identifier_or_array_list)
            [[size]][default_info];
      local declarations;
      statements;        "including RETURN statements
END function_name;
```

Functions take declared input parameters and generate a return value. Because functions only have input parameters, the INPUT keyword is optional in the parameter declaration (unlike a procedure which must have input and output paramenters). The input parameters for a function are the same as for a procedure. (See the section following entitled ***Input Parameters***.)

For information about default_info, see the heading Default Information in Chapter 5.

The return value of a function can be any width. A size for the return value can optionally be specified following the right parenthesis of the input parameter declaration. As an example, the following function returns a 4-bit array that is the bit-wise AND of arrays *a* and *b*.

```
FUNCTION and4(a[4], b[4])[4];
      RETURN a*b;
END and4;
```

If [*size*] is omitted, a width of 1 is assumed.

The return value is used to pass signals out of a function. The return value of a function is assigned by the RETURN statement:

```
RETURN expression;
```

A RETURN statement can appear anywhere in the function that statements can occur. The width of expression must match the [size] declared for the return value.

Function return values and procedure output parameters can be given default values just like ordinary signal declarations. If no DEFAULT_TO statement is given, .X. (DON'T CARE) is assumed.

Invoking a Function

A function is invoked from within an expression. Its return value becomes the value of the expression where the function is invoked.

To invoke a function:

```
function_name(expression_list)
```

The following example illustrates how to invoke a function:

```
FUNCTION or1(x, y);
      RETURN x+y;
END or1;
```

The function or1 is invoked from within an expression to create hardware to implement *q = a * (b * c + d):*

```
INPUT a,b,c,d;
OUTPUT q;

q = a * or1(b*c, d);
```

The value of *b*c* is passed to the function as input parameter *x*, and *d* is passed as the input parameter *y*.

Input Parameters

Input parameters are used to pass signals into a procedure or function. When a procedure or function is invoked, any equal-width expression or group of expressions can be passed to an input parameter. The passed expressions will drive the inputs in the invocation of the procedure or function.

The procedure *and1*:

```
PROCEDURE and1(INPUT a, b; OUTPUT x);
      x = a * b;
END and1;
```

can be invoked at the system level to create hardware to *implement q = (x+y) * (y+z*x)*. To do this, declare the actual inputs (*x, y, z*) and outputs () of the system-level design and invoke the procedure with the appropriate expressions or signals in corrresponding positions to the input and output parameters of the procedure description, as shown in the following example:

```
INPUT x, y, z;
OUTPUT q;

and1(x+y,  y+z*x,  q);
```

Output Parameters

Output parameters are the means of passing equations out of a procedure. Ultimately, all of the statements (IF, CASE, etc.) in a procedure will result in a single equation for each output parameter.

When a procedure is invoked, each output parameter must be passed an argument that is an assignable signal (NODE, OUTPUT, BIPUT) or group of equal-width assignable signals in its expression_or_signal_list. The output of the procedure will be assigned to the passed argument.

For instance, using the procedure *and1* shown previously, the assignable signal result corresponds to the output parameter *x*:

```
INPUT  in1, in2;
```

```
OUTPUT  result;

and1(in1, in2, result);
```

The equation for an output parameter defined by the procedure will have its inputs driven by the corresponding arguments, and the resulting equation will be assigned to the output signal.

Output parameters can be given control information and default information just like ordinary signal declarations. For information about control_info and default_info, see ***Control Information*** and ***Default Information*** in ***Chapter 6***.

Local Declarations

Local signals can be declared within procedures and functions. Procedure or function signal declarations remain local to the procedure or function in which they are defined.

These signals are only visible within the procedure or function in which they are declared and will not conflict with other signals of the same name in other procedures, functions, and at the system level.

These local signals may not be referenced at the system level or in any other function or procedure.

What Happens When a Procedure or Function is Invoked?

When a procedure or function is invoked, a new instance of its local signals is created at the invocation level. An instance of a NODE is also created for each input and output parameter of a procedure or function.

Each time a procedure or function is invoked, each local signal is given a global name. The DSL compiler keeps each of these signals unique so that the same name can be used in different procedures/functions. This also means

that the same procedure/function may be invoked multiple times without name conflicts. The form of these unique names is as follows:

```
Procedure_name.instance_number.local_name
Function_name.instance_number.local_name
```

Where:

> *instance_number* starts at 1 and increments each time the procedure or function is invoked within a particular procedure or function.
>
> *local_name* is the variable name within the procedure or function.

While this naming scheme works well to give each signal in a design a unique name, it can cause problems if the design is still subject to change. For example, suppose that a procedure named *add2* has a local signal named *a*. The signal name for *a* in its first invocation would be:

```
add2.1.a
```

This says that the procedure_name is *add2*, the signal was created the first time the procedure was invoked (its instance_number), and the local_name is *a*.

Let's further assume that this signal (*a*) is assigned to an output pin after fitting. If the language source file is changed such that an invocation of *add2* is added before the current one, this new invocation becomes instance_number 1, and the previous invocation becomes instance_number 2. The signals are renamed during the compile and the wrong signal *add2.1.a* is assigned to the hardware pin.

The DSL compiler gives you the capability to label an instance of an invocation (rather than having the compiler do it) so that signal names are immune to changes in the design file. To label an invocation, use the following:

```
[label:] procedure_name (argument_list);
[label:] function_name (argument_list);
```

Where *label* is any legal identifier.

This changes the signal's name to:

procedure_name.label.local_name.

To use the previous example, if we gave the procedure *add2* a label of *func_1* the procedure invocation would look like the following:

```
func_1:add2 (argument list);
```

and the global signal name for signal *a* would become:

```
add2.func_1.a.
```

In the following procedure declaration of a divide-by-two frequency divider:

```
PROCEDURE frequency_divider  (INPUT in; OUTPUT out);
      NODE x  CLOCKED_BY in;
      x = /x;
      out=x;
END frequency_divider;
```

the local declaration of NODE x is used to perform the frequency division.

A divide-by-four frequency divider could be implemented using the divide-by-two procedure:

```
INPUT in;
OUTPUT out;
NODE tmp;

frequency_divider(in,tmp);
frequency_divider(tmp,out);
```

Each invocation of frequency_divider creates a NODE at the system level to perform each divide-by-two. The names of the NODEs as they will appear in the documentation file are:

```
frequency_divider.1.x
```

and

```
frequency_divider.2.x
```

This same example can be implemented as a function:

```
FUNCTION frequency_divider(in);
      NODE x  CLOCKED_BY in;
      x = /x;
      RETURN x;
END frequency_divider;
```

To create the divide-by-four counter, invoke the function as follows:

```
INPUT in;
OUTPUT out;
out = frequency_divider(frequency_divider(in));
```

The following example describes the procedure *and4*, which has three parameters: 4-bit wide inputs *a* and *b*, and a 4-bit wide output *x*. The equation for *x* is *a*b*.

```
PROCEDURE and4(INPUT a[4], b[4]; OUTPUT x[4]);
      x = a*b;
END and4;
```

This procedure can be invoked at the system level to create hardware as follows:

```
INPUT  a, b, c, d, e[4];
OUTPUT  w, x, y, z;

and4([a, b, c, d], e, [w, x, y, z]);
```

This results in *[w, x, y, z] = [a, b, c, d] * [e[3], e[2], e[1], e[0]]*.

In the following example, *c* will have the value 1 if *a=b*. Otherwise, *c* will take on the value of *a*:

```
PROCEDURE p (INPUT a,b; OUTPUT c DEFAULT_TO a);
      IF a=b THEN
      c=1;
      END IF;
END p;
```

Invoking Procedures and Functions From Other Files

The keyword USE allows procedures and functions from other compiled source files to be used in a design. Two formats for using procedures and functions from other files are available. The first format:

```
USE 'filename';
```

makes all procedures and functions from the referenced file available to the current design file.

The second format

```
USE  ' filename ' . name;
```

makes available only a named procedure or function from another file.

Note that the filename is enclosed by single quote marks (').

In the following example, the function and1 from the *design1.src* file is used in *design2.src*:

```
File design1.src:

FUNCTION and1(a,b);
      RETURN a*b;
END  and1;


File design2.src:

USE  'design1'.and1;
INPUT  x, y;
OUTPUT  z;
z = and1(x, y);
```

This capability allows a design to be broken up into multiple files for better organization, and allows parallel development by several designers. It also gives you the capability of developing a library of useful procedures and functions that can be shared by many designs.

9 Text Processing

Contents

Introduction

To assist in the tedious aspects of entering a source code design, the Design Synthesis Language provides several means to help you avoid retyping frequently used sections of text. Macros (using the MACRO keyword) gives you the ability to perform text substitution. You may also include text from other source files in your current design, using the INCLUDE keyword.

In addition, there is a quick method to comment out blocks of code for debugging purposes using the COMP_OFF and COMP_ON constructs.

This chapter discusses these time-saving constructs in detail.

Macros

Macros allow the user to create an identifier that will be replaced by an associated block of text. Unlike procedures or functions, macros simply perform text substitution. They are used as a type of shorthand to free the designer from redundant typing.

The syntax of a macro definition is:

```
MACRO macro_name [(parameters)] text;
MACRO macro_name [(parameters)] {multi-line text}
```

A macro, when defined, is given a name, optional parameters (separated by commas), and text. Unlike function or procedure parameters, nothing is passed into or out of a macro since a macro simply substitutes a line or lines of text. When a macro name is encountered by the compiler, the compiler substitutes the pre-defined text for the MACRO name. Thus, if a MACRO is defined as:

```
MACRO  adder(a, b)  a(+)b;
```

The compiler will replace every occurrence of adder(a, b) with the text a(+)b.

Each macro in a source file is global. This means that you cannot have two macros with the same name or the DSL compiler will indicate that you have tried to redeclare a macro. This also means that a macro may be used anywhere in the source file (after its declaration) regardless of where the declaration occurs.

Macro names may also be used as part of an expression or equation. For example, MACRO adder(a, b), defined previously, used in the following assignment:

```
x = adder(y, z);
```

Will expand to:

```
x = y(+)z;
```

Some additional examples of valid macro definitions include:

```
MACRO  bit_mask  00FFH;

MACRO  fill(a, b, c)  {
       x = a;
       y = b;
       z = c;
       }
```

Note: Macros containing semicolons must be surrounded by curly brackets ({ }).

Including Other Files in a Design

The INCLUDE statement allows you to include other text files in your current design file. This can eliminate retyping of often used structures or macros. Included files may in turn include other files. The text of the included file will be inserted in your current design at the point where the keyword INCLUDE occurs.

INCLUDE statements can occur anywhere in the design. The format for including a file in your current design is:

```
INCLUDE  ' filename ';
```

The filename includes any filename extension (no default extension is appended.) The filename must be enclosed by single quote marks (').

Commenting Out Blocks of Code

The Design Synthesis Language allows you to tell the compiler to ignore sections of the design file at compilation time. COMP_OFF indicates the start of a section of code to be ignored, and COMP_ON indicates the end of that section.

The syntax for using COMP_OFF and COMP_ON is:

```
COMP_OFF
section_of_source_code
COMP_ON
```

No semi-colons are used at the end of the COMP_OFF and COMP_ON keywords.

You can either use the keywords COMP_OFF and COMP_ON to comment out a section of code, or you can use the comment symbol (") at the beginning of every line you want the compiler to ignore.

In the following example, the compiler will ignore the ELSIF clause in the following IF statement:

```
IF (reset) THEN
      [q3..q0] = 0;
COMP_OFF
ELSIF ([q3..q0] = 9) THEN
      [q3,q2,q1,q0] = 0;
COMP_ON
ELSE
      [q3..q0] = 5;
END IF;
```

10 Compiling a Design

Contents

Introduction

Once you have created the logic description (the source file) for a design, you are ready to compile the file. The compiler converts the source file (*filename.src*) into an internal representation of the design (*filename.afb*). The *.afb* can be used by:

- the simulator to simulate the design,
- the optimizer to prepare a system-level design for device fitting,
- another run of the compiler on another *.src* file that USEs this file.

Compilation

The compiler's responsibility is to interpret the source language (described in Chapters 4 - 9) and create the internal representation file (*.afb*). During this process the compiler converts the high-level constructs (declaration, expression, and statement) of the source language into a simple list of signals with associated equations, each of which are stored in the internal representation. The compiler performs error checking on the design to make sure it follows the rules and restrictions described in Chapters 4 - 9. The equations that drive each signal are created so that the action described in the high-level source language is implemented by the equations.

The output of the compiler (*filename.afb*) contains the signals and associated equations. This file can then be used by the simulator to verify the behavior of the design. For more information on the simulator, see Chapter 11.

If the compiled source file (*.afb*) contains a system-level design, this design can be passed on to the optimizer and the remainder of the tool chain for fitting into devices. A system-level design is one described outside of any Procedure or Function (see Chapter 8 for more information on Procedures and Functions.)

Multiple File Designs

MACHXL lets you implement a design in multiple source files. An advantage of having multiple source files is that only the portions of a design that are

affected by file changes need be recompiled. If a bottom-up hierarchical approach is used in your design, Procedures and Functions can be placed in their own file that is compiled only once. Any additional design work that requires these Procedures and Functions can still be done in another file without the need to recompile these completed Procedures and Functions each time.

Another advantage of multiple design files is that it lets you develop libraries of generally useful Procedures and Functions and place them in their own files. These Procedures and Functions can be used (via the USE command shown in Chapter 8) by many different designs, giving greater leverage of design effort.

There is always one parent source file that contains the system-level portion of the design. All other files will contain only Procedures and Functions that may be 'USEd' in the parent file to create the final design. A 'USEd' source file must be compiled before any other source file that USEs its Procedures and Functions.

Errors in Compilation

If errors are encountered in a language design, the errors and line numbers can be found in the file *filename.err*. Appendix C, MACHXL Error and Warning Messages, includes Design Synthesis Language error messages.

To correct language design errors, you will need to go to the line that contains the error in the language source file (*filename.src*) and make the necessary corrections, save the file, and recompile.

For information on the compiler menus and the actual process involved in compilation, see Chapter 3.

11 Simulating and Testing a Design

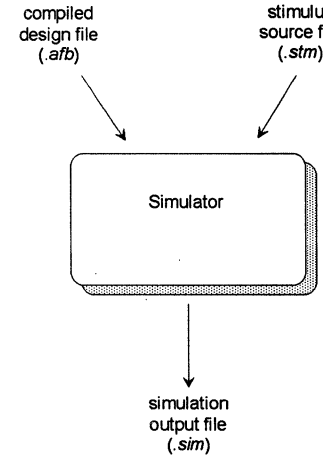
Contents

Introduction

An important feature of the MACHXL software is its simulation and test-language capabilities. The simulator gives you the ability to:

- simulate modules (procedures and functions) to verify correct operation,
- simulate the complete design to verify correct operation,
- generate test vectors to verify correct operation of the programmed devices.

The simulator in MACHXL does not do timing simulation. It is a functional stimulator only.



You must have a compiled design (*design_name.afb*) file and a stimulus source file (*design_name.stm*) to run MACHXL's simulator. The remainder of this chapter discusses how to use MACHXL's test language to create a simulation source file (the figure at right shows the files used by and produced by the simulator). This chapter also describes how to interpret the results. Example simulation files may be found at the end of ***Appendix B, Language-Based Examples***.

The simulator takes input values provided by the designer in the *.stm* file (via the Test Language) and applies them to the section to be simulated. The simulated output is then checked against the expected output and any discrepancies or unstable states are written to the simulator listing file (*filename.sim*).

Device testing is done in the same fashion by sending the simulator-generated input vectors to the device programmer via the JEDEC file. These actual output vectors are then compared against the simulator-generated output vectors to verify the device.

Input vectors and expected output vectors are specified to the simulator by means of the Test Language. The Test Language lets you specify which variables to use in the simulation and which signals to trace. The Test Language also provides operations needed to construct the test vectors.

Test Language Reference

The *.stm* file is a source file you create (using MACHXL's Test Language) to give instructions to the simulator. The *.stm* file can be created using any editor or word processor, the same as your design source file. This section describes the commands and syntax of the Test Language and how to use them in the *.stm* file. An example *.stm* file with explanations is provided at the end of this section. Additional simulation examples are in Appendix B.

General Structure of a Simulation or Test File

A *.stm* file has sections like other source language files. In the declaration section signals and variables are declared and a step duration is set. In the body of the source file you give the simulator specific instructions that initialize signal values and compute the values for input and output signals. Flow control constructs (like IF/THEN, WHILE-DO and FOR-DO) give control over the simulation process. For more information on the internal operation of the simulator, an explanatory section is provided at the end of this chapter.

The simulator lets you simulate a module (i.e., a Procedure or Function) by using the keyword SIMULATION with the Procedure's or Function's name The general form of a module simulation section is shown below:

Procedure/function simulation:

```
SIMULATION procedure_name|function_name ;
      {declarations}
      {statements}
END SIMULATION ;
```

The simulator also lets you simulate the whole design by using a SIMULATION section at the global level (i.e., outside of any Proceedure or Function). Any SIMULATION section without a Procedure or Function name is considered global. The general form of the design-file SIMULATION section is shown below:

Design file simulation:

```
SIMULATION ;
      {declarations}
      {statements}
END SIMULATION ;
```

There is also capability in the simulator to generate test vectors that can be stored in the JEDEC file (that is sent to the device programmer). These test vectors can check actual device outputs against simulated outputs to ensure the device is working as expected. The SYSTEM_TEST keyword is used when you want to generate vectors that test the programmed devices. The following rules apply to using SYSTEM_TEST:

- The SYSTEM_TEST keyword is a system-level command placing test vectors in the JEDEC file of the design. This differs from a SIMULATION section that does not place simulation vectors in the JEDEC file.

- The SYSTEM_TEST keyword is not allowed in Functions or Procedures because it is a system-level command, not a module-level command.

The general form of SYSTEM_TEST section is shown below:

System test vector generation:

```
SYSTEM_TEST ;
      {declarations}
      {statements}
END SYSTEM_TEST;
```

The following is a more explicit example of the form for system SIMULATION and TEST_SYSTEM sections showing the general usage of some of the keywords. Each step and the keywords are explained in following sections.

```
SIMULATION|SYSTEM_TEST;
      Declarations
            clock resolution (STEP)
            variable declarations (VAR)
            signals to display in simulation output
            (TRACE)

      Body
            assign initial values to signals (INITIAL)
            assign values to signals by table
            assign values to signals by assignment (SET)
            insert messages for simulation output
            (MESSAGE)
            compute values for input, output signals
            (arithmetic operators)
            flow control (IF/THEN/ELSE, WHILE-DO, FOR-DO)
END SIMULATION|SYSTEM_TEST ;
```

Keywords

The identifiers listed below are reserved by the simulator as keywords and may not be used as signal names, procedure names, function names, or variable names.

AND	IF	THEN
BIN	INITIAL	TO
CASE	MESSAGE	TRACE
CLOCKF	NOT	VAR
DEC	OCT	WHEN
DO	OR	WHILE
ELSE	RETURN	
ELSIF	SET	
END	SIMULATION	
FOR	STEP	
HEX	SYSTEM_TEST	

Declarations

The following sections describe the three types of declaration statements for the simulator. The types of declarations are as follows:

```
STEP  time step labeling information
VAR   variable declarations
TRACE output listing order and format.
```

These declarations must appear after the header (SIMULATION or SYSTEM_TEST) and before any statements. They can be mixed in any order.

Specifying the Clock Resolution

The STEP statement allows the user to specify how the time steps in the simulation listing file are to be labeled. This does not affect the behavior of the simulation, as the simulator is strictly functional. STEP lets the user specify the time label associated with each simulation step. The general form of the command is as follows:

```
STEP time_units;
```

Where:

> *time_units* is an integer value and a time unit specification (ns, us, ms, or s). If the STEP statement is omitted, the default step value is 10ns. If you use multiple STEP statements, a warning is generated and only the last declaration is used.

For example, the following specifies that each step in the simulation should be labeled in 50 ns intervals.

```
STEP 50ns;  "Labelling in time units
```

Valid time units and their abbreviations are:

Symbol	Time Unit
ns	nanoseconds
us	microseconds
ms	milliseconds
s	seconds

The integer value and the time unit symbol cannot have spaces between them. They must be adjacent. For example:

```
STEP 50ns;        "valid
STEP 50  ns;      "invalid
```

Note: The simulator in MACHXL is a functional simulator only. The simulator does not do any timing simulation. Device delays are not represented in the simulation results.

Variable and Signal Expressions

There are two types of expressions used in the test language: variable and signal expressions. Variable expressions are made up of variables and operators while signal expressions are made up of signals, variables, signal values, and operators. Variables are defined in the simulation file and used to control the flow of the simulation or to assign values to signals. Signals are defined in the design file and are part of the design.

Declaring Variables

The VAR declaration lets a user allocate local integer variables that can be used in:

- generating values assigned to signals
- signal expressions
- control or conditional constructs (e.g., IF/THEN, CASE, FOR-DO, WHILE-DO).

Variables are declared using the VAR keyword:

```
VAR var_name {,var_name};
```

Where:

var_name is one or more identifiers naming variables for the simulation or test section. Variable names are separated by commas.

The following statement declares *j* a variable:

```
VAR j;
```

Note: A variable should not be confused with a signal. A signal is declared in the design language and assigned expected input or output values in a simulation or test section for simulation. A variable is declared and used only in a simulation or test section to keep track of the flow of test operations or to assign values to signals in the simulation section. Variables are assigned values using variable assignment statements (Example: j = 0; or, j = j .+. 1;). Signals are assigned values using the INITIAL and SET keywords.

The variable *i* is declared and initialized to 0 in the following example and used as a counter to keep track of the number of WHILE-DO iterations performed. It is also used in the SET statement to assign signals *A7* through *A0* the value of *i*:

```
VAR i;
i = 0;

WHILE (i < 255) DO
      SET [A7..A0] = i;
      CLOCKF;
      i = i .+. 1;
END WHILE;
```

Tracing Signals

The TRACE declaration allows the user to specify which signals are written to the simulation listing file. It also specifies how those signals are to be formatted.

If no TRACE statement is given, all the signals in this section of the design will appear in the simulation output in binary form.

If a TRACE statement is used in a simulation or test section, all the signals in a design will be used in the simulation, but only the signals specified by the TRACE statement will be displayed in the simulation listing file. Signals in the listing file are written in the same order as given in the TRACE statement.

The format for using TRACE is:

```
TRACE signal [BIN|DEC|HEX|OCT] {,signal
      [BIN|DEC|HEX|OCT]};
```

Where:

> *signal* is one or more signals or groups of signals separated by commas. Range notation and groups of signals may also be used. No comma is used between a signal and its numeric base specification. Signals must be previously declared in the language design under test. The keyword RETURN may be used to trace a FUNCTION return value. Groups of signals displayed in DEC format must be less than 31 bits wide.

BIN|DEC|HEX|OCT are optional specifications to display the associated signal or signals in binary, decimal, hexadecimal, or octal form respectively. If no numerical base is specified, the default is BIN.

Different bases can be used for different signals. The base representation for a signal will only be visible in the table format of the simulation output. If no base representation is supplied, binary is assumed.

Note: Only one TRACE statement can be used per SIMULATION or SYSTEM_TEST section.

The following TRACE statement shows how to specify individual signals (`clk, count1, count2`) or groups of signals (`[d7..d0], x[5..0]`), each with different numeric base representations.

```
TRACE clk, count1, count2, [d7..d0] DEC, x[5..0] OCT;
```

Since signals `clk, count1`, and `count2` are not given a base specification, they will all be displayed in binary. The signals `d7..d0` are followed by the base declaration DEC, and will be represented in decimal form in the simulation listing file. The signals `x5` through `x0` have a base declaration of OCT and will be displayed in octal format.

When displaying groups of signals, any signal in the group with a value of DON'T CARE or is in a HIGH IMPEDANCE (tri-state) condition will cause the group to be displayed with asterisks (*).

Statements

Statements are used in a simulation or test section to construct vectors. You can construct vectors manually using the table format to specify values for inputs and outputs. You may also use the SET and CLOCKF language constructs to create vectors. The high-level IF/THEN/ELSE, FOR-DO, and WHILE-DO control flow constructs used with the SET and CLOCKF keywords automate vector generation.

The following sections discuss the table format and language constructs to create simulation and test vectors.

Using the Table Format to Create Vectors

The TEST_VECTORS statement lets the user enter both the input signal values and the expected output values for each simulation step.

The form of the TEST_VECTORS statement is as follows:

```
TEST_VECTORS
       signal_name [,signal_name];
       var_expres [,var_expres];
       .
       .
       .
       var_expres [,var_expres];
END TEST_VECTORS;
```

Where:

signal_name is the list of signals affected by this statement.

var_expres contains values for input, output, or biput signals on incremental clocks. Values for inputs, outputs, and biputs are shown below:

Signal type	Allowable values
input	0 (set to binary 0) 1 (set to binary 1) .X. (don't care) .C. (clock the pin)
output	0 (set to binary 0) 1 (set to binary 1) .X. (don't care) .Z. (high Impedance) .S. (calculated during simulation)

Signal type	Allowable values
biput	0 (set to binary 0) 1 (set to binary 1) .X. (don't care) .Z. (high Impedance) .S. (calculated during simulation)

When the TEST_VECTORS statement is executed, each line following the signal list line is used to generate a simulation step. For each signal in the signal list, the corresponding *var_expres* from this state is used to set the input value for this signal. Once all the signal values are set, the simulator executes the step just as if a CLOCKF statement were executed. In fact, the two following examples, one using the TEST_VECTORS statement and the other using the SET and CLOCKF statements, are equivalent:

```
TEST_VECTORS
      signal_name1,..., signal_namen;
      var_expres1,..., var_expresn;
      .
      .
      .
      var_expres1,..., var_expresn;
END TEST_VECTORS;

SET [signal_name1,...,signal_namen] =
      [var_expres1,...var_expresn];
CLOCKF;
SET [signal_name1,...,signal_namen] =
      [var_expres1,...var_expresn];
CLOCKF;
```

The TEST_VECTORS statement provides a shorthand method of setting variables to signals. It eliminates the need for SET and CLOCKF statements for each simulation step.

To enter test information in table format, list the individual input and output signals or groups with their corresponding values.

The following Gray-Code Counter example uses the table format to create simulation test vectors for the procedure *gr4_truth*. The Design Synthesis Language source section is shown first, followed by the Test Language source section.

```
#TITLE       '4-Bit Gray-code Counter with Reset';
#ENGINEER    'J. Engineer';
#COMPANY     'Hytek Co.';
" This file contains a procedure for a 4-Bit Gray-code
" counter using the TRUTH_TABLE construct.  The previous-
" state output values are used as inputs in the truth
" table to generate the next-state output values.  The
" reset line is forced by using it as an input.

PROCEDURE gr4_truth( ) ;
INPUT   clk, reset ;
OUTPUT  p3, p2, p1, p0   CLOCKED_BY  clk ;

      TRUTH_TABLE

     reset, p3, p2,  p1,  p0   ::   p3,  p2,  p1,  p0 ;
"    ------------------------------------------------
       0,   X,  X,   X,   X    ::   0,   0,   0,   0 ;

       1,   0,  0,   0,   0    ::   0,   0,   0,   1 ;
       1,   0,  0,   0,   1    ::   0,   0,   1,   1 ;
       1,   0,  0,   1,   1    ::   0,   0,   1,   0 ;
       1,   0,  0,   1,   0    ::   0,   1,   1,   0 ;

       1,   0,  1,   1,   0    ::   0,   1,   0,   0 ;
       1,   0,  1,   0,   0    ::   0,   1,   0,   1 ;
       1,   0,  1,   0,   1    ::   0,   1,   1,   1 ;
       1,   0,  1,   1,   1    ::   1,   1,   1,   1 ;

       1,   1,  1,   1,   1    ::   1,   1,   1,   0 ;
       1,   1,  1,   1,   0    ::   1,   1,   0,   0 ;
       1,   1,  1,   0,   0    ::   1,   1,   0,   1 ;
       1,   1,  1,   0,   1    ::   1,   0,   0,   1 ;
```

```
        1,   1,  0,   0,   1    ::   1,   0,   1,   1 ;
        1,   1,  0,   1,   1    ::   1,   0,   1,   0 ;
        1,   1,  0,   1,   0    ::   1,   0,   0,   0 ;
        1,   1,  0,   0,   0    ::   0,   0,   0,   0 ;

    END TRUTH_TABLE;
END gr4_truth;

    SIMULATION gr4_truth ;
        TRACE clk, reset, [p3..p0] HEX ;

        TEST_VECTORS
        clk,  reset,p3,   p2,   p1,   p0;
        .c.,  0,    0,    0,    0,    0;

        .c.,  1,    0,    0,    0,    1;
        .c.,  1,    0,    0,    1,    0;
        .c.,  1,    0,    1,    1,    0;

        .c.,  1,    0,    1,    0,    0;
        .c.,  1,    0,    1,    0,    1;
        .c.,  1,    0,    1,    1,    1;
        .c.,  1,    1,    1,    1,    1;

        .c.,  1,    1,    1,    1,    0;
        .c.,  1,    1,    1,    0,    0;
        .c.,  1,    1,    1,    0,    1;
        .c.,  1,    1,    0,    0,    1;

        .c.,  1,    1,    0,    1,    1;
        .c.,  1,    1,    0,    1,    0;
        .c.,  1,    1,    0,    0,    0;
        .c.,  1,    0,    0,    0,    0;

        END TEST_VECTORS;
    END SIMULATION;
```

Using Test Language Constructs to Create Vectors

The test language extends the table concept by introducing the SET and CLOCKF constructs. With SET and CLOCKF you only need to set values that change at a specified time unit, without listing values that stay the same.

SET assigns values to input signals and expected values to output signals. CLOCKF advances the simulation or test vector to the next time unit, which in table format is the next row.

SET and CLOCKF allow mixing the table format and the language:

```
SET [ a, b, c ] = 0;
SET [ e, f, g  ] = 110b;
CLOCKF;

TEST_VECTORS
      a, b, c, e, f, g;
      1, 1, 1, 0, Z, Z;
      1, X, 1, 1, 0, 0;
END TEST_VECTORS;
```

The rest of this section discusses the test language operators and constructs used in building vectors. The constructs include: SET, CLOCKF, INITIAL, MESSAGE, FOR-DO, IF/THEN/ELSE, and WHILE-DO.

SET

Test vectors can be created in the test language, without using table format at all, by assigning values to input signals, assigning expected values to output signals, and advancing the simulation using CLOCKF.

Values are assigned to signals using the SET keyword:

```
SET signal =  variable expression|.C.|.S.|.X.|.Z.
      {, signal =  variable expression|.C.|.S.|.X.|.Z.};
```

Where:

signal is one or more signals previously declared in the design under test. Signals are separated by commas. Range and group notation may also be used. The special signal name RETURN can be used to refer to a function return value.

variable expression is any test language mathematical expression. A mathematical expression can be a number, a variable, or an expression used by itself or with any of the arithmetic operators in the test language (e.g., .*., .+., ./., .-., .MOD., etc.).

The test language has 6 different values that can be assigned to signals. These values are shown below .

Value	Meaning
0	Set to Binary 0
1	Set to Binary 1
.C.	Clock the pin
.S.	Calculated during simulation
.X.	Don't Care
.Z.	High Impedance

Note: All signal names must be defined in the design file. Any signal name used in the simulation file and not defined in a design file generates an error message.

The binary values 0 and 1 represent false and true conditions depending on the type of signal assigned the values. For example, if a signal is defined as HIGH_TRUE then a binary 1 represents the asserted condition. If a signal is defined as LOW_TRUE then a binary 0 represents the asserted condition.

A signal set to a value of .C. will be clocked during the simulation. This value is typically assigned to signals connected to the clock input of a register device (i.e., D-type flip flop, SR-type flip flop, etc.) but can be assigned to any signal.

The .S. value tells the simulator to calculate the signal value. This value is often used to automate test vector generation using output values generated by the simulator.

Any signals set to a value of .X. will not be checked during the simulation. The difference between using .S. and .X. is important during test vector generation as opposed to simulation vector generation. An output set to a value of .S. will take on the calculated simulation value in the test vector. An output set to .X. is set to X in the test vector.

A signal set to a value of .Z. forces the signal to a high impedance or tri-state condition. This is useful when several output signals are connected as in an address bus.

Setting an output signal to any value other than .S. tests the simulator-generated output value against the SET value, generating an error on a mismatch.

A signal holds a value until another SET statement is specified.

If values are not initially specified for signals, the input signals are automatically set to .X. (Don't Care), and the output signals are set to .S. (computed by simulator).

Assign different signals to different values using one SET statement by separating the assignments with commas:

```
SET a = 1, b = 0, c = 1;
```

To set values to a group of signals, use the group notation. For example,

```
SET [X, Y, Z] = 4;
```

Assigns the value of 4 (in binary) to X, Y, and Z. Thus, the signals X, Y, and Z contains the following values:

```
X = 1 ;
Y = 0 ;
Z = 0 ;
```

The values, .S., .X., .Z., and .C. can be assigned to a group of signals. As an example, the statement,

```
SET [X, Y, Z] = .S. ;
```

sets signals X, Y, and Z to .S. (calculate during the simulation).

Another statement,

```
SET [A, B, C] = .Z. ;
```

sets outputs A, B, and C to the High Impedance value.

It is also possible to set one group of signals to another group of signals as long as both groups have the same number of signals. As an example, the statement,

```
SET [X, Y, Z] = [A, B, C] ;
```

sets signal X equal to A, signal Y equal to B, and signal Z equal to C.

CLOCKF

After assigning values to signals using SET statements, use the CLOCKF keyword to advance the simulation one time step. The syntax for CLOCKF is:

```
CLOCKF [clock_signal {,clock_signal}] ;
```

Where:

> *clock_signal* is one or more input signals used to clock a registered output. Separate clock signals with commas.

To advance the simulation one time step, use one CLOCKF statement:

```
CLOCKF ;  "advance 1 time unit
```

To advance the simulation several time steps, use CLOCKF commands in succession:

```
CLOCKF ;  "advance 4 time steps
CLOCKF ;
CLOCKF ;
CLOCKF ;
```

You can force a registered output to be clocked using CLOCKF with a clock signal. The statement:

```
CLOCKF clk;        "This is identical to:  SET clk = .C.;
                   "CLOCKF;
```

generates a pulse on the clk input and moves the simulator to the next time step. This is equivalent to entering a C for the clk signal in the table format as follows:

```
input1, input2, ...,    clk, inputN :: output1,...
                             C,    ::
                             C,    ::
```

One CLOCKF statement can be used to clock a group of clock inputs:

```
CLOCKF clk1, clk2, clk3;
```

The following example shows how the SET and CLOCKF constructs can be used to test a design. In this design, the outputs latch the inputs when clocked by the clk signal. The simulator software checks out this functionality.

```
INPUT  In7..In0, clk;
OUTPUT  Out7..Out0  CLOCKED_BY clk;

[Out7..Out0] = [In7..In0];
```

```
SYSTEM_TEST;
      SET [In7..In0] = 55h;
      CLOCKF clk;
      SET [In7..In0] = 0AAh;
      CLOCKF clk;
END SYSTEM_TEST;
```

The resulting simulation is as follows:

```
                                        O O O O O O O O
                  I I I I I I I I C U U U U U U U U
                  N N N N N N N N L T T T T T T T T
Time us           7 6 5 4 3 2 1 0 K 7 6 5 4 3 2 1 0
-------------------------------------------------------
0                 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
10                0 1 0 1 0 1 0 1 C 0 1 0 1 0 1 0 1
20                1 0 1 0 1 0 1 0 C 1 0 1 0 1 0 1 0
```

A CLOCKF statement with no signal list is used to generate a combinatorial step; i.e., unless the user is explicitly generating a clock, the simulator will advance one step but no clock edges will be generated. This is useful when testing latches. The following example shows how to explicitly generate a clock:

```
SET clk = 0 ;
CLOCKF ;
SET clk = 1 ;
CLOCKF ;
```

INITIAL

INITIAL sets the internal value of signals and creates a special simulation step. In the case of inputs, the result of INITIAL and SET are similar. In the case of outputs, the internal value is changed, but the pin and driven values remain the same. During the propagation step, the values will be propagated through the circuit

(see the section in this chapter entitled ***Internal Simulator Operation***.) The format for using INITIAL is the same as for SET.

```
INITIAL signal {, signal} = expression|.Z.|.S.|.X.|.C.;
```

Examples

The following example sets the simple signal *x* to 1:

```
INITIAL x = 1;
```

In this example, the bus *Y[16]* is set to 1001011000111100:

```
INITIAL Y = 0963CH;
      or
INITIAL Y = 1001011000111100B;
```

The following sets simple signals *A, B, C*, and *D* to 1, 0, 1, 0 respectively:

```
INITIAL [A, B, C, D] = 1010B;
      or
INITIAL [A, B, C, D] = 10; "default base 10 (decimal)
```

A group of INITIAL statements create an initial step in the simulator list. Any non-INITIAL statement will separate INITIAL statements into separate INITIAL steps. For example,

```
SET X = 0; Q = 0;
INITIAL Y = 1;
INITIAL Z = 0;
FOR I = 0 TO 31 DO
. . .
```

will form one initial step containing *Y = 1* and *Z = 0*.

```
SET X = 0;
INITIAL Y = 1;
SET Q = 0;
INITIAL Z = 0;
FOR I = 0 TO 31 DO
. . .
```

will form two initial steps,

1) *Y = 1* and *Z* having its previous value, and
2) *Y* having it's new value and *Z = 1*.

Note the initial step only propagates combinatorial values, but cannot clock registers. This means that the only way to change a register value during an INITIAL is to set the register signal name to the value.

Note also that INITIAL changes only the value of the signal in the INITIAL statement. To set a register inside of a count procedure, set the internal register value, not the output signal.

```
PROCEDURE cnt16 (INPUT clk, rst; OUTPUT Q[16]);
NODE int_Q[16] CLOCKED_BY clk, RESET_BY rst;
int_Q = int_Q .+. 1;
END cnt16;


INPUT c,r;
OUTPUT cnt_out[16];

cnt16 (c, r, cnt_out);
```

In this example, *cnt_out* is a combinatorial output connected to the register. Setting *cnt_out* to a value will not change the register. To initialize the counter to 3FFH, you would have to enter the following into your stimulus (*.stm*) file:

```
INITIAL cnt16.1. int_Q = 3FFH;
```

To find the fully qualified name, run the optimizer and the documentor, and look in the resulting *.doc* file for the name.

When an INITIAL statement is part of the SYSTEM_TEST section , the resulting initial steps will be included in the test vectors. When a signal is set to .C., the corresponding position in the test vector will contain a P, which on many devices will cause the device programmer to pre-load the data on the registered output pins into the internal registers.

INITIAL_TO

INITIAL_TO assigns the same initial value to all output signals with a single command. The INITIAL_TO command must appear before any SIMULATION or SYSTEM_TEST sections but does not have to appear before any macro definitions. The format for using the INITIAL_TO command is:

```
INITIAL_TO value;
```

Where:

value is one of the following values: 0, 1, and .X.

This value is overridden for any signals that appear in INITIAL statements that may appear in a simulation or test section. All signals are set to the specified value before the first simulation step.

MESSAGE

Messages you want to appear in the simulation output can be inserted in the test code. Messages act as signposts when you examine the simulation output, helping you determine where you are in the simulation process. To have a message appear in the output, use the format:

```
MESSAGE ( ' message text ' );
```

Messages tag the next simulation step marked by a CLOCKF statement. Any message statement placed after the last CLOCKF statement in a test section will be ignored.

In the following test section for a rolling dice design, a message statement appears where the new roll of the dice begins:

```
SET oe = .X. ;
SET d1, d2 = 0 ;
CLOCKF clk ;
CLOCKF clk ;
CLOCKF clk ;
CLOCKF clk ;
CLOCKF clk ;
CLOCKF clk ;
CLOCKF clk ;
MESSAGE( 'New Roll' );
CLOCKF clk ;
```

The simulation output displays "New Roll" after generating the vectors previous to the new roll:

```
Time ns      OE     D1     D2     Messages
-----------------------------------------------------
0            X      0      0
10           0      1      1
20           0      1      1
30           0      1      1
40           0      1      1
50           0      1      1
60           0      0      1      New Roll
```

Note: Only one MESSAGE statement should appear between any two of the statements stepping the simulator (i.e., CLOCKF or INITIAL.) If more than one MESSAGE statement is used between two statements, a warning is issued and the first message encountered is used.

RETURN

When simulating a function, the keyword RETURN may be used wherever an output signal is allowed. This indicates the return value of the function. In the following example, RETURN is used to inspect the simulation results.

Examples

.*src* (source) file

```
FUNCTION xor (INPUT a, b);
      RETURN a (+) b;
END xor;
```

.*stm* (stimulus) file

```
SIMULATION xor;
      TRACE a, b, RETURN;

      VAR i, j;

      FOR i = 0 to 1 DO
      SET a = 1;
            FOR j = 0 to 1 DO
                  SET b = j;
                  SET RETURN = i (+) j;
                  CLOCKF;
            END FOR;
      END FOR
END SIMULATION;
```

Output of Simulator

```
Simulation of xor
      Design:     xor.fb
      Stimulus:   xor.stm
```

```
                          R
                          E
                          T
                          U
                          R
Time nSec    a      b     N      Messages

-----------BBB---BBB---BBB-----------------------------

init         X      X     X
  10         0      0     0
  20         0      1     1
  30         1      0     1
  40         1      1     0
```

When the function returns a signal vector, the return value is the full range of the index, therefore RETURN should not be specified with an index.

.*src* (source) file

```
FUNCTION sprod ( INPUT a, b[8] ) [8];

      NODE t[8];

      t[0] = a * b[0];
      t[1] = a * b[1];
      t[2] = a * b[2];
      t[3] = a * b[3];
      t[4] = a * b[4];
      t[5] = a * b[5];
      t[6] = a * b[6];
      t[7] = a * b[7];
      RETURN t;
END sprod;
```

.stm (stimulus) file

```
SIMULATION sprod;
      TRACE a, b, RETURN DEC;
      VAR i, j;

      FOR i = 0 TO 1 DO
            SET a = i;
            FOR j = 0 to 255 DO
                  SET b = j;
                  CLOCKF;
            END FOR;
      END FOR;
END SIMULATION
```

Output of Simulator

```
Simulation of sprod
      Design: sprod.fb
    Stimulus: sprod.stm
```

```
                                        R R
                                        E E
                                        T T
                                        . .
                          b       b N N
                          [       [ [ [
                          7       0 7 0
   Time nSec      a   ].....] ].]      Messages

----------------BBB-BBBBBBBB-DDD-------------------------

        init       X  XXXXXXXX $$$
          10       0  00000000 000
          20       0  00000001 000
          30       0  00000010 000
          40       0  00000011 000
          50       0  00000100 000
          60       0  00000101 000
          70       0  00000110 000
          80       0  00000111 000
          90       0  00001000 000
         100       0  00001001 000
         110       0  00001010 000
         120       0  00001011 000
         130       0  00001100 000
         140       0  00001101 000
         150       0  00001110 000
         160       0  00001111 000
         170       0  00010000 000
```

Test Language Operators

Three types of operators are available in the simulation and test sections: arithmetic, bit oriented, and relational. Arithmetic operators are used to compute values for input and output signals, and values to be assigned to local test language variables. Bit-oriented operators are used to perform bit-wise Boolean operations.

Relational expressions evaluate to a 1 if the expression is true; otherwise they evaluate to 0. They are used to determine which clause in an IF/THEN/ELSE construct to perform, or the number of times to perform the statements in a WHILE-DO loop.

The test language operators are shown below:

Operation	Description	Operator Type
a.+.b	sum of a and b	arithmetic
a.-.b	difference of a and b	arithmetic
a.*.b	product of a and b	arithmetic
a./.b	quotient of a and b	arithmetic
a.	mod.b modulus of a and b	arithmetic
.-.a	negation of a	arithmetic
a+b	bit-wise OR of a and B	bit oriented
a*b	bit-wise AND of a and b	bit oriented
/a	bit-wise complement of a	bit oriented
a < b	if a < b, then 1 else 0	relational
a > b	if a > b, then 1 else 0	relational
a <= b	if a < or = b, then 1 else 0	relational
a >= b	if a > or = b, then 1 else 0	relational
a = b	if a = b, then 1 else 0	relational
a <> b	if a <> b, then 1 else 0	relational
a and b	if both a and b = 1, then 1 else 0	relational
a or b	if either a or b = 1 then 1 else 0	relational
not a	if a = 0 then 1 else 0	relational
a.<<.b	shift a left b bits	arithmetic
a.>>.b	shift a right b bits	arithmetic

The FOR-DO Construct

The FOR-DO construct allows you to create vectors iteratively. It bases its looping on an inclusive counter value. The syntax of FOR-DO is:

```
FOR var_name = lower_limit TO upper_limit DO
      statements
END FOR;
```

Where:

var_name is a local variable declared in this simulation section.

lower_limit is any variable expression.

upper_limit is any variable expression. If the value of *lower_limit* is greater than the value of *upper_limit*, the loop will not be performed.

statements is one or more test operations.

The following FOR loop performs eleven CLOCKF statements:

```
FOR i=0 TO 10 DO
      CLOCKF;
END FOR;
```

The next example shows the test section for a free-running two-bit ring counter. It uses the FOR construct to perform 33 CLOCKF statements, advancing the simulator 33 time steps.

```
SYSTEM_TEST;
      STEP 100ns;
      VAR i;
      TRACE clk, tcount1, tcount2;

      FOR i = 0 TO 32 DO
            CLOCKF clk;
      END FOR;
END SYSTEM_TEST;
```

FOR-DO statements can be nested within other FOR-DO statements. The following example demonstrates nested FOR-DO statements.

```
SET powerup = 1 ;
FOR j = 0 TO 10 DO
      SET timeout = 1 ;
      CLOCKF ;
      CLOCKF ;
      CLOCKF ;
      CLOCKF ;
      SET timeout = 0 ;
      FOR i = 0 TO 10 DO    " nested FOR-DO statement
            CLOCKF ;
      END FOR;
      END FOR;
```

IF/THEN/ELSE

The test language IF/THEN/ELSE construct has the format:

```
IF variable expression THEN
      statements
{ELSIF variable expression THEN
      statements}
[ELSE
      statements]
END IF;
```

If the variable expression in the IF statement is true, the statements between THEN and ELSE are performed. Otherwise, the statements following ELSE are performed (if existing). If there is no ELSE section in an IF statement, and if the value of the variable expression is true, the statements between the IF and END IF statements are performed.

In the following example, an IF/THEN/ELSE construct is nested in a FOR loop with additional IF statements nested in the ELSE clause:

```
SIMULATION ;
      VAR i, j ;
      TRACE ei, [i7..i0], [aa2..aa0], ggs, eeo ;
      MESSAGE( ' Test encoding ' ) ;
      SET ei=0 ;
      j = 0 ;
      FOR i=0 TO 255 DO
            SET [i7..i0] = 255-i ;
            IF (i=0) THEN " nested IF/THEN/ELSE
                  SET ggs=1, eeo=0 ;
            ELSE
                        SET ggs=0, eeo=1 ;
                  IF (i=0) THEN
                        SET [aa2..aa0] = 111b ; "nested
                                                "IFs
                  ELSIF (i=1) THEN
                        SET [aa2..aa0] = 111b ;
                  ELSIF (i=2) THEN
                        SET [aa2..aa0] = 110b ;
                  ELSIF (i=4) THEN
                        SET [aa2..aa0] = 101b ;
                  ELSIF (i=8) THEN
                        SET [aa2..aa0] = 100b ;
                  ELSIF (i=16) THEN
                        SET [aa2..aa0] = 011b ;
                  ELSIF (i=32) THEN
                        SET [aa2..aa0] = 010b ;
                  ELSIF (i=64) THEN
                        SET [aa2..aa0] = 001b ;
                  ELSIF (i=128) THEN
                        SET [aa2..aa0] = 000b ;
                  END IF;
                  CLOCKF ;
            END IF;
      END FOR;
END SIMULATION;
```

WHILE-DO

The WHILE statement will repeat a group of statements as long as the WHILE variable expression is true. If the variable expression is initially false then the statements are not performed.

```
WHILE variable expression DO
      statements
END WHILE;
```

The following example demonstrates how to use a WHILE loop to perform eleven CLOCKF statements:

```
i = 0;  " initialize variable i
WHILE i <= 10 DO
      CLOCKF;
      i = i .+. 1;
END WHILE;
```

The next example demonstrates the WHILE-DO statement, and generates the sequence, 0, 255, 1, 254, 2, 253, ..., 128 for the variable j:

```
SIMULATION
      VAR j ;

      j = 0 ; "initialize variable j

      WHILE j <> 128 DO
            SET [in7..in0] = j ;
            CLOCKF clk ;
            j = 255 .-. j ;
            IF (j < 128) THEN
                  j = j .+. 1 ;
            END IF;
      END WHILE;

      SET [in7..in0] = j ;
      CLOCKF clk ;
END SIMULATION;
```

An Example Simulation Section and Results

The following are example design (*.src*), stimulus (*.stm*), and simulation result (*.sim*) files used to explain the basic concepts of the simulator.

Design File *(design_name.src)*

```
"
" Design file for address decoder example
"
LOW_TRUE INPUT oe ;
INPUT    a1, a0, clk ;
OUTPUT   rom CLOCKED_BY clk ENABLED_BY oe ;
OUTPUT   ram CLOCKED_BY clk ENABLED_BY oe ;
OUTPUT   i_o CLOCKED_BY clk ENABLED_BY oe ;
OUTPUT   a_to_d CLOCKED_BY clk ENABLED_BY oe ;

rom = /a1 * /a0 ;
ram = /a1 * a0 ;
i_o = a1 * /a0 ;
a_to_d = a1 * a0 ;
```

Stimulus File (*design_name.stm*)

```
"
" Stimulus file for address decoder example  Line
"     Number
SIMULATION ;                                          "1
      VAR i, j;                                       "2
      STEP 10ns;                                      "3
      TRACE clk, oe, a1, a0, rom, ram, i_o, a_to_d;   "4

      SET clk = .C. ;                                 "5
      SET oe = 0 ;                                    "6
```

```
      FOR i = 0 TO 1 DO                                    "7
            j = 0 ;                                        "8
            WHILE (j < 2) DO                               "9
                  SET a1 = i ;                             "10
                  SET a0 = j ;                             "11
                  j = j .+. 1;                             "12
                  CLOCKF clk ;                             "13
            END WHILE ;                                    "14
            IF ( i AND j ) THEN                            "15
                  SET oe = 1;                              "16
            END IF ;                                       "17
      END FOR ;                                            "18
      CLOCKF clk ;                                         "19
END SIMULATION ;                                           "20
```

Line 1 of this stimulus file begins with the keyword, SIMULATION. For this simple design there is only one simulation section. A more complex design might have several simulation sections each testing part of the overall design.

Line 2 declares two variables, *i* and *j*, which are used to control the simulation and to assign values to the signals in the design. Don't confuse a variable with a signal. A variable is used only by the simulator and is not part of the design.

Line 3 tells the simulator how you want to label each step in the simulation results file. In this case, each step will be labeled in 10 ns increments. The STEP statement is provided for your convenience and doesn't affect the behavior of the simulator. The default is 10 ns.

Line 4 is a TRACE statement telling the simulator which signals should appear in the simulation output file and in what format to display them. For this simulation, all signals will appear in the listing file and the default display format of binary is used.

Lines 5 and 6 contain signal expressions. The signal, *clk*, is assigned a value of .C. indicating to the simulator this signal is to be clocked during the simulation. The SET statement of line 6 assigns a value of 0 to the signal OE.

This signal will maintain this value until the end of the simulation section or until another SET statement changes this value. SET statements are only used for assigning values to signals and not to variables.

Line 7 is the beginning statement of a looping construct. Looping constructs allow automating parts of a simulation section. The variable `i` is set initially to a value of 0. This value is then compared to the value 1 which appears on the right side of the keyword TO. If the value of `i` is greater than this value, the loop is terminated; otherwise all statements appearing between lines 7 and 18 are performed. With each pass through the loop, the value of `i` is incremented by 1. All FOR loop constructs must have an END FOR statement.

In **line 8**, the variable `j` is assigned a value of 0.

Line 9, contains another type of looping construct, the WHILE loop. This loop executes as long as the variable expression, between the WHILE and DO, evaluates true. As long as the value of `j` is less than 2, the variable expression is true and all statements between the WHILE statement header and the END WHILE statement are executed.

Lines 10 and 11 contain SET statements assigning the values of `i` and `j` to the input signals `a1` and `a0` respectively.

Line 12 increments the variable `j` by 1. If this statement is not present, the WHILE loop will not terminate.

The CLOCKF statement in **line 13** is used to advance the simulator one time step. When the simulator executes this statement, the `rom, ram, i_o`, and `a_to_d` outputs are evaluated, the clock signal, `clk`, is clocked, and the results are printed to the simulation output file.

Line 14 terminates the WHILE-DO loop.

The IF statement of **line 15** evaluates the variable expression, `i AND j`.

If the value is true, **line 16** is executed. This condition occurs when the value of `i` is non-zero and the value of `j` is non-zero (i = 1, j = 2). Any variable

expression evaluating to a value of 0 is considered false; otherwise, the variable expression is true.

Line 17 ends the IF construct started in line 15.

Line 18 ends the FOR construct started in line 7.

Line 19 causes the simulator to advance one time step.

The final statement of the simulation section is the END SIMULATION statement. At this point it is possible to start another simulation section or system test section. However, for this example, this is the only simulation section.

The following is the simulation output file (*design_name.sim*) for our example.

Simulation Results File (*design_name.sim*)

```
Simulation of SYSTEM
           Design: DALE.FB
       Simulation: DALE.STM


                                                    A
                                                    _
                                                    T
             C                       R    R    I    O
             L    O    A    A       O    A    _    _
Time nSec    K    E    1    0       M    M    O    D   Messages

             BBB  BBB  BBB  BBB  BBB  BBB  BBB  BBB--------------

init         X    X    X    X    X    X    X    X
10           C    0    0    0    1    0    0    0
20           C    0    0    1    0    1    0    0
30           C    0    1    0    0    0    1    0
40           C    0    1    1    0    0    0    1
50           C    1    1    1    Z    Z    Z    Z
```

A SYSTEM_TEST Example

The following example is a SYSTEM_TEST section demonstrating many of the test constructs discussed in this chapter. Remember a SYSTEM_TEST section places test vectors in the JEDEC file. This example is given without explanation.

```
" Design file for counter example
Low_True Input oe ;
Input clk ;
Output q5..q0 clocked_by clk enabled_by oe ;

      q0 = /q0 ;
      q1 = q1 (+) q0 ;
      q2 = q2 (+) q1 * q0 ;
      q3 = q3 (+) q2 * q1 * q0 ;
      q4 = q4 (+) q3 * q2 * q1 * q0 ;
      q5 = q5 (+) q4 * q3 * q2 * q1 * q0 ;
"
" Simulation file for counter example
SYSTEM_TEST;
      VAR i, j;
      STEP 10ns;
      TRACE clk, oe, [ q5..q0 ];

      FOR i = 0 to 1 DO
            SET oe = i ;
            FOR j = 0 to 64 DO  " nested FOR construct
                  IF i = 1 THEN " nested IF/THEN/ELSE
                                " construct
                        MESSAGE('disable outputs') ;
                  ELSE
                        MESSAGE('enable outputs') ;
                  END IF ;
                  CLOCKF clk ;
            END FOR;
      END FOR;
END SYSTEM_TEST;
```

The simulation output generated for this design is as follows:

```
            C
            L     OQ....Q
Time  ns    K     E5....0          Messages

------------B     BBBBBBB---------------------------------

0           0     0000000
10          C     0000001          enable outputs
20          C     0000010          enable outputs
30          C     0000011          enable outputs
40          C     0000100          enable outputs
50          C     0000101          enable outputs
60          C     0000110          enable outputs
70          C     0000111          enable outputs
80          C     0001000          enable outputs
90          C     0001001          enable outputs
100         C     0001010          enable outputs
110         C     0001011          enable outputs
120         C     0001100          enable outputs
130         C     0001101          enable outputs
140         C     0001110          enable outputs
150         C     0001111          enable outputs
 . . .
600         C     0111100          enable outputs
610         C     0111101          enable outputs
620         C     0111110          enable outputs
630         C     0111111          enable outputs
640         C     0000000          enable outputs
650         C     0000001          enable outputs
660         C     1ZZZZZZ          disable outputs
670         C     1ZZZZZZ          disable outputs
680         C     1ZZZZZZ          disable outputs
690         C     1ZZZZZZ          disable outputs
700         C     1ZZZZZZ          disable outputs
 . . .
```

```
                C
                L       OQ....Q
Time    ns      K       E5....0             Messages

------------    B       BBBBBBB-------------------------------

1250            C       1ZZZZZZ             disable outputs
1260            C       1ZZZZZZ             disable outputs
1270            C       1ZZZZZZ             disable outputs
1280            C       1ZZZZZZ             disable outputs
1290            C       1ZZZZZZ             disable outputs
1300            C       1ZZZZZZ             disable outputs
```

Internal Simulator Operation

The simulator uses two input files (*design_name.afb* and *design_name.stm*) to produce simulation results. The *.afb* file contains compiled design information while the *.stm* file contains stimulus (SIMULATION) and system test (SYSTEM_TEST) sections. The results of the simulation are written to the output file, *.sim*. If the *.stm* file has a system test section then the simulator generated test vectors are written to a *.tv* file which is used during the fitting process. The SYSTEM_TEST section also generates test vectors that are a part of the JEDEC file. If simulation of the design is not required then removing the *.stm* file will keep the simulator from running.

Each simulation and system test section of the .stm file is compiled into a set of statement structures. These structures define signal values for the compiled design equations and control the simulation cycles. The simulation cycles create the simulation results and test vectors.

Simulation Cycle

Each time step in the simulation (specified by the STEP command) is divided into simulation cycles. Each of these cycles computes a new intermediate value for the design equations. A simulation cycle is a process which consists

of several steps generating new values for the output signals. These steps are shown below:

```
Initialize
Compute all outputs until stable
If there is a clock signal then
      Clock step
      Compute until outputs are stable
Write out the results
```

Any error or warning messages occuring during a simulation cycle are written to the *.sim* and *.log* files. The maximum number of errors or warning allowed during each simulation cycle is 10. If the number of errors or warnings exceeds 10, the remaining errors and warnings are ignored for the current cycle and a warning message is written to the *.sim* file indicating the limit was exceeded.

Initialize

The initialize step in the previous flow diagram assigns values to the input and output signals as defined in the statement structure. Any input or output signals not assigned values in the statement structure are set to either a value of unknown (X) or to the value in the INITIAL_TO statement, if this statement exists.

Compute All Outputs Until Stable

This step is a loop of statements which are executed until all combinatorial outputs are stable or until the total number of times through the loop reaches a count of 128. Outputs are considered stable if their value has not changed from the previous step. The statements executed in the loop are as follows:

1. Compute the values of all the output generators (i.e., clocks, resets, inputs, etc.).

2. Generate the values of the outputs using the newly computed output generator values.

3. If the new signal values are the same as the signal values in the previous iteration, the outputs are stable and the loop is terminated.

If the loop count reaches 128 then some of the output signals are unstable. An error message is written to the *.sim* file for each unstable signal and these signals are set to a value of unknown (X).

If There is a Clock Signal

Each input signal set to a value of .C. is clocked and the value of all output signals is calculated. If any of the output signals are unstable then the values are recalculated until the output signals are stable or until the number of calculations reaches 128. If the count reaches 128, some of the output signal values are unstable. An error message is written to the *.sim* file for all unstable signals and these signals are set to a value of unknown (X).

Write Out Results

Each signal value is set to the new value for this cycle and the results are written to the *.sim* file.

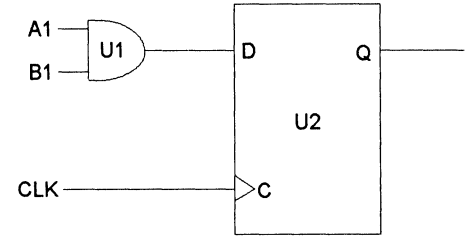
The total number of cycles performed is controlled by the statement structures compiled for each simulation and system test section. The maximum number of cycles can be 128 before the clock step and a maximum 128 cycles after the clock step.

Note: Any combinatorial outputs that are chained together in a series of more than 128 gates may not be stable.

Signal States

For each signal in the simulation, a state is stored. In the case of input signals, this value is defined by the SET command or by the value of the output which drives it. The circuit shown in the following figure contains an example of the two types of inputs. *A1*, *B1*, and *CLK* are defined by SET commands while signal *C1* is defined by the output of AND gate *U2*.

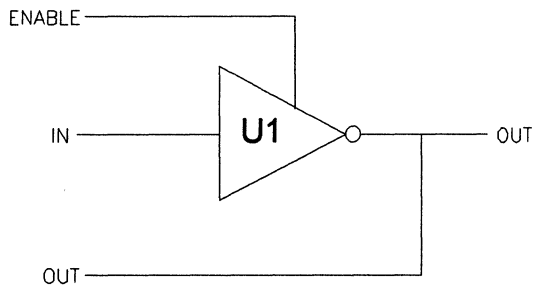
```
SIMULATION ;

TRACE A1, B1, C1, CLK, D1;
SET A1 = 0;
SET B1 = 0;
SET CLK = .C.;
   .
   .
   .

END SIMULATION ;
```



For output signals, a state is made up of three items:

1. the internal value of the signal
2. the value driven by the output onto the pin
3. the value driven by the simulator onto the pin.

The internal value corresponds to the value associated with the signal before the tri-state output and is internal to the device. The value being driven out is the value after the tri-state output. The internal value can be set by using INITIAL or INITIAL_TO statements. The value driven by the simulator onto an output is the result of using the SET command for an output signal. As an example, in the circuit and simulation section as shown in the following figure, the input signal, *IN*, is set to a value of 1. The internal value for the output signal, *OUT*, is 0. The value driven out of *U1* is Z. The reason is the signal *ENABLE* has a value of 0 thus placing *U1* in the high-impedance state. The value driven by the simulator is 1.




```
SIMULATION ;

       TRACE ENABLE,
       IN, OUT;
       SET ENABLE = 0;
       SET IN = 1;
       SET OUT = 1;
         .
         .
         .

 END SIMULATION ;
```

For output signals that do not have a tri-state output, the internal value and the value driven by the output are always equal.

The values these items can take are shown in the following table.

Symbol	Meaning
L	Low (or 0)
H	High (or 1)
X	Don't Care or Unknown
Z	High Impedance (Tri-state disabled)
C	Clocked (Pulse)
S	Simulated Value (for outputs)

Truth Tables for the Test Language Logical Operators

During each step of a simulation cycle, the equations associated with a combinatorial output are evaluated. This evaluation is made by applying the following truth tables:

Truth Table for the AND Operation

*	L	H	X	Z	C	S
L	L	L	L	L	L	S
H	L	H	X	X	C	S
X	L	X	X	X	X	S
Z	X	H	X	X	X	S
C	L	C	X	X	C	S
S	S	S	S	S	S	S

X = Don't Care

Truth Table for the OR Operation

+	L	H	X	Z	C	S
L	L	H	X	X	C	S
H	H	H	H	H	H	S
X	X	H	X	X	X	S
Z	X	H	X	X	X	S
C	C	H	X	X	C	S
S	S	S	S	S	S	S

X = Don't Care

Truth Table for the Exclusive OR Operation

(+)	L	H	X	Z	C	S
L	L	H	X	X	C	S
H	H	L	X	X	C	S
X	X	X	X	X	X	S
Z	X	H	X	X	X	S
C	C	C	X	X	C	S
S	S	S	S	S	S	S

X = Don't Care

Truth Table for the NOT (Complement) Operation

/	L	H	X	Z	C	S
	H	L	X	X	C	S

X = Don't Care

There are several possible equations associated with each output. An output may have an enable, indicating the output is a tri-state component enabled by this expression, or may have a register input with any of the optional clock, clock enable, preset and reset equations, or may be combinatorial. If the enable is not present, then it is assumed the output is always driving. The resets and presets, if present, are used to indicate an asynchronous condition taking precedence over the clock. The following truth tables describe the possible register behavior:

Truth Table for a D-type Flip Flop

D	clk	clk enable	reset	preset	Qi
X	X	X	H	L	L
X	X	X	L	H	H
X	X	X	?	X	?
X	X	X	X	?	?
X	?	X	L	L	?
X	C	?	L	L	?
X	C	L	L	L	Qi-1
X	L/H	L/H	L	L	Qi-1
L	C	H	L	L	L
H	C	H	L	L	H
?	C	H	L	L	?

X = Don't Care ? = Unknown Qi = Current State Qi-1 = Previous State

Truth Table for a JK-type Flip Flop

J/K	clk	clk enable	reset	preset	Qi
X	X	X	?	X	?
X	X	X	X	?	?
X	X	X	H	L	L
X	X	X	L	H	H
X	?	X	L	L	?
X	C	?	L	L	?
X	C	L	L	L	Qi-1
?	C	H	L	L	?
L	C	H	L	L	?
H	C	H	L	L	Qi-1
L	C	H	L	L	H
H	C	H	L	L	L
H	C	H	L	L	/Qi-1

X = Don't Care ? = Unknown Qi = Current State Qi-1 = Previous State

Truth Table for an RS-type Flip Flop

R/S	clk	clk enable	reset	preset	Qi
X	X	X	?	X	?
X	X	X	X	?	?
X	X	X	H	L	L
X	X	X	L	H	H
X	?	X	L	L	?
X	C	?	L	L	?
X	C	L	L	L	Qi-1
X	C	H	L	L	?
?	C	H	L	L	?
L	C	H	L	L	Qi-1
L	C	H	L	L	H
H	C	H	L	L	L
H	C	H	L	L	?

X = Don't Care ? = Unknown Qi = Current State Qi-1 = Previous State

Truth Table for a T-type Flip Flop

T	clk	clk enable	reset	preset	Qi
X	X	X	H	L	L
X	X	X	L	H	H
X	X	X	?	X	?
X	X	X	X	?	?
X	?	X	L	L	?
X	C	?	L	L	?
X	C	L	L	L	Qi-1
X	L/H	L/H	L	L	Qi-1
L	C	H	L	L	Qi-1
H	C	H	L	L	/Qi-1
?	C	H	L	L	?

X = Don't Care ? = Unknown Qi = Current State Qi-1 = Previous State

Truth Table for a D-type Latch

D	latch enabl	reset	preset	Qi
X	X	H	L	L
X	X	L	H	H
X	X	?	X	?
X	X	X	?	?
X	?	L	L	?
X	L	L	L	Qi-1
L	H	L	L	L
H	H	L	L	H
?	H	L	L	?

X = Don't Care ? = Unknown Qi = Current State Qi-1 = Previous State

12 Optimizing a Design

Contents

Introduction

The optimizer in MACHXL takes the abstract representations of a compiled language file (*.afb*) and converts them into physical representations. During optimization, the following functions are performed:

- nodes are collapsed out of the design when possible
- flip-flops are synthesized
- equations are reduced.

After optimization, the design file is ready for partitioning.

Optimizer Operation

The purpose of the optimizer is to reduce the size of design equations and the number of NODEs. This allows the design to fit into the fewest and smallest possible devices. PLDs implement logic using two-level logic to feed the macro cells. This means the equations feeding the inputs of the macro cells are represented in a two-level Sum-of-Products form. The circuit shown below is one example of a macro cell. The contents of a macro cell can be quite different from one PLD or CPLD device to the next. These macro cells can contain all of these logic devices, some of these logic devices, or none of these logic devices. Some PLDs have a fused inverter and fused flip-flops.

While PLDs vary in their ability to share internal hardware between macro

cells, they are all constrained by the number of input signals and by the number of product terms used in a design equation. By optimizing the design equations, it may be possible to keep the number of product terms to a number less than the maximum number of product terms allowed for a specific device. So, the goal of the optimizer is to take advantage of a particular device's architecture while not exceeding the device limits. There are three techniques used to reduce the design equations: node collapsing, register synthesis, and final reduction.

Node Collapsing

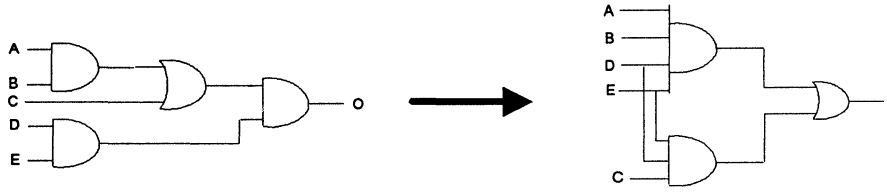
The first technique used to optimize a design is called node collapsing. Node collapsing is the process of removing an internal signal node by substituting the node's equation into any equation that references the node. As an example, the following design equations have an internal node *x*:

```
INPUT a, b, c, d ;
NODE x ;
OUTPUT y, z ;

x = a * b + c ;
y = x * d ;
z = x + b ;
```

Node collapsing results in NODE *x* being removed, yielding the following equations:

```
y = a * b * d + c * d ;
z = c + b ;
```

Note: In addition, OUTPUT z has been optimized to yield c + b.



A hardware example of node collapsing is shown in the diagram shown above. In this example, a design using 2-input logic gates is collapsed to a 2-level equivalent to increase speed.

Virtual and Physical Nodes

It is possible to explicitly control node collapsing of individual nodes by declaring them to be VIRTUAL or PHYSICAL in the design file. VIRTUAL nodes are always collapsed while PHYSICAL nodes are never collapsed (for more information on NODES and their modifiers, see **Chapter 5** and the section on ***Declarations***.)

Any node mentioned in the *.pi* (physical information) file becomes a PHYSICAL node because a node must exist physically to have properties attached to it. Individual nodes can be declared to be VIRTUAL in the *.pi* file (see Chapter 13 for more information on the *.pi* file). The following is an example portion of a *.pi* file showing both PHYSICAL and VIRTUAL nodes:

```
PHYSICAL r06 ;
VIRTUAL vn, n ;
```

This is the mechanism used in the automatically generated *.npi* file to force the optimizer to make the correct node collapsing choices. The *.npi* file can be used to guarantee a design is implemented the same way on subsequent iterations through the MACHXL design tools by copying the *.npi* file to the *.pi* file. For more information on the *.pi* file, see Chapters 13, 14, and 15.

Controlling Node Collapsing

While collapsing reduces the number of equations, it can also increase the number of terms in some equations. This can mean that there may not be enough resources in some devices to implement the design. There are five constraints you can use to limit the size of the equations produced during the node collapsing process. These constraints can be chosen to suit the requirements of a particular device. The constraints are specified as signal properties in the *.pi* file (see Chapter 14 for more information on the *.pi* file signal properties.) These global group or properties should be assigned on a

device-by-device basis for each of the different target devices. The following is a list of these properties:

MAX_SYMBOLS n

Lets you tell the optimizer the maximum number of unique symbols or signals to allow in any one equation.
Default=20.

You may set MAX_SYMBOLS equal to the maximum number of inputs and nodes feeding the array, and with some devices you may also use some of the outputs as inputs. This allows increasing the maximum number of symbols in an equation. Conversely, if you are concerned about not having enough outputs, you may want to decrease MAX_SYMBOLS.

MAX_PTERMS n

Lets you tell the optimizer the maximum number of product terms to allow in a sum-of-products version of an equation.
Default=16.

In general, this parameter can be thought of as the number of inputs to the OR gates in the device.

MAX_XOR_PTERMS n

Lets you tell the optimizer the maximum number of product terms you want to appear on one input of an exclusive OR gate, assuming the other input has one product term. If both inputs of an exclusive-OR gate have more than a single product term, the equation will exceed the MAX PTERMS constraint. If n = 0, the target device has no exclusive-OR hardware.
Default= 0.

If the XOR representation of an equation has more than one pterm on both sides then the equation will exceed the MAX_PTERMS constraint. If either the regular or the XOR pterm constraint is met, then the equation is allowable. This means that devices with XOR gates allow the most effecient form of the equation to be used.

POLARITY_CONTROL [TRUE | FALSE]

Lets you tell the optimizer whether a target device has a fusible inverter in the path to the output macrocell. If TRUE, you are telling the optimizer the device has a fusible inverter. FALSE indicates there is no fusible inverter.
Default=TRUE

XOR_POLARITY_CONTROL [TRUE | FALSE]

Lets you tell the optimizer whether a target device has a fusible inverter on its XOR. If TRUE, you are telling the optimizer the device has a fusible inverter. FALSE indicates there is no fusible inverter.
Default=FALSE

The optimizer uses these properties when collapsing a node to determine if an equation referencing a node will exceed these constraints.

If the target device has a fusible inverter (i.e., POLARITY_CONTROL or XOR_POLARITY_CONTROL is TRUE), the optimizer can take the smaller of the ordinary or DeMorgan equation set. This is because the optimizer can make use of the fusible inverter to take the smaller of the two equations.

If the target device does not have a fusible inverter (i.e., POLARITY_CONTROL or XOR_POLARITY_CONTROL is FALSE), the optimizer will take the larger of the ordinary equation set or the DeMorgan equation set. This is to make sure it can fit the larger of the two equations should it need to (since it cannot use the fusible inverter to select the smaller of the two equations). The maximum number of pterms the ordinary or DeMorgan set can have is specified by MAX_PTERMS property.

In both cases it is important to remember the optimzer uses this information only to determine whether it can collapse a node, not to determine Fit.

Note: Any equation exceeding the size constraints prior to optimization are unaffected by the size limits. These size constraints only apply to equations created during the node collapsing process.

The MAX_XOR_PTERMS constraint applies to one input of an exclusive-OR gate if the other input is fed by a single product term. This is a common situation for many device architectures. If the exclusive-OR representation of an equation has more than a single product term on both inputs then the MAX_PTERMS constraint is applied to the non-exclusive-OR form of the equation.

If the corresponding equation representations meet either MAX_PTERMS or MAX_XOR_PTERMS constraints then that equation is acceptable. Thus, fusible exclusive-OR gates allow use of the most efficient form of an equation. As an example:

```
INPUT a, b, c, d, e ;
NODE x;
OUTPUT out ;

x = c * (d + e) ;
out = (a * b) (+) x ;
```

If the NODE x were collapsed, then the exclusive-OR form of the equation is:

```
out = (a * d) (+) (c * d + c * e) ;
```

and the ordinary form of the equation is:

```
out = (c * d */a) + (c * d */b) + (c * e */a) +
      (c * e * /b) + (a *b * /c) + (a * b */d */e) ;
```

If MAX_PTERMS is greater than or equal to 6 or MAX_XOR_PTERMS is greater than or equal to 2 then node x would be collapsed. (For the sake of simplicity, this example ignores the polarity control properties and the DeMorgan form of the equations.)

Combinatorial feedback nodes are also removed whenever possible. However, if the optimizer determines the combinatorial feedback node is required by the feedback circuit, then that node is not removed. As an example, consider the following design:

```
INPUT a, b, c ;
NODE y, x ;
OUTPUT out ;

y = a (+) x ;
x = y * b ;
out = x * c ;
```

Collapsing node x in equation y produces the following equations:

```
y = a (+) (y * b) ;
out = y * b * c;
```

Because the equation for node y contains y as one of its inputs, the node x cannot be collapsed.

Any node referenced in a control equation (CLOCKED_BY, RESET_BY, etc.) is not collapsed unless it is declared as VIRTUAL. (See the previous section on Virtual and Physical Nodes for more information.) As an example:

```
INPUT clk, reset1, reset2, a, b ;
NODE reset ;
OUTPUT out CLOCKED_BY clk RESET_BY reset ;

reset = reset1 * reset2 ;
out = a + b ;
```

The node `reset` is not collapsed since it is used in the RESET_BY equation of the signal out. If it had been collapsed, this design would not have fit into many devices that have a single reset line. Declaring reset to be VIRTUAL causes the node to be collapsed anyway.

Node Collapsing and Partitioning

The equations resulting from node collapsing will have varying sizes up to the maximum size specified by the constraints. Typically, you should set the limits to match the characteristics of the largest equation that can be handled by a target device to obtain good results.

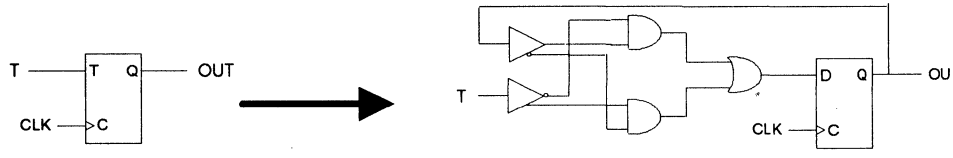
When directed partitioning is performed, the type of device each signal must fit into is specified. Since the node collapsing constraints can be specified on a device by device basis, equation sizes can be made to match each device.

When using automatic partitioning, you may prefer to use a particular set of devices for a design. In this case, setting the node collapsing constraints to that required by the largest of these devices gives good results.

Even if the types of target devices are unknown, approximate constraint values will still yield good node collapsing results. The default values give good results for a wide variety of devices. A little experimentation with the constraints can help to refine the resulting equation sizes.

Register Synthesis

The second technique used to optimize the design for the largest variety of target devices is called register synthesis. The optimizer is responsible for synthesizing the equations for alternate flip-flop types for registers (see figure below). The register type declared in the source file allows the equations for a register to be expressed in terms of the given flip-flop type. The optimizer synthesizes the equations for all the other flip-flop types to give the automatic fitting process greater flexibility in its choice of devices used to implement a register.



T-FLOP D-FLOP EQUIVALENT

If a register is declared with the NO_REDUCE modifier then it will be implemented using the declared flip-flop type and no synthesis will be done.

Equation Reduction

The final reduction technique uses one of three reduction algorithms to reduce the equations produced by node collapsing and register synthesis. The final reduction algorithm takes advantage of DON'T CARE information. During the node collapsing and register synthesis processes, the optimizer maintains ON, OFF, and DON'T CARE information sets for every equation. This allows the final reduction algorithm to best use the DON'T CARE information in reducing all collapsed and synthesized forms of the equations.

There are three final reduction algorithms available: Espresso, Espresso Exact, and Quine-McCluskey. The method used is selected by using the options menu (for more information on setting this option, see Chapter 3.) The default value for the final reduction algorithm is Espresso. The Espresso algorithm is the fastest method and usually produces results as good as the other two algorithms.

Factoring

Factoring allows for large equations to be broken up into various smaller intermediate equations, Executing from the command line, and using the MAX_PTERMS and MAX_SYMBOLS *.pi* file properties to control equation factoring, may result in a better optimal set of equations for the programmable logic device you are targeting.

13 Partitioning and Fitting (Optional)

Contents

Introduction

When you create a design with MACHXL, the design phase is separate from the device implementation phase. **Chapters 4** through **12** show the steps required to create a design using MACHXL. Assuming that the design is complete and correct, you can now concentrate on implementing the design with various programmable device architectures.

Just as the design phase was an iterative process, the hardware implementation steps of MACHXL have been set up to allow iterating on various hardware implementations of that design. MACHXL allows you to specify device characteristics and constraints for implementation of your design. Based on these settings, MACHXL searches its extensive library of PLD and CPLD architectures, looking for devices matching your criteria. MACHXL then maps your design into the specified device or devices. If the design requires more than one device, MACHXL can partition the logic across multiple devices to obtain a solution (optional).

This chapter describes the process of mapping the logical design into physical devices and discusses the ways a design can be fit and partitioned.

Partitioning Modes

When partitioning a design among the various device architectures, MACHXL allows the user to operate in one or a combination of the following three modes:

- Automatic Partitioning
- Directed Partitioning
- Manual Partitioning

The Automatic Partitioning Mode allows the partitioning software to run unconstrained. No direction is given to the software with regards to signal/pin placement or logic partitioning. This mode is the easiest to use as it requires no special files to be created or modified. For many designs, this is the only mode needed to partition a design. MACHXL software partitions automatically by using a set of constraints. Possible solutions are prioritized according to a set of user priorities. From this a list of best solutions is selected and displayed.

In the automatic mode, the software is able to generate many solutions in a short period of time. This lets you look at different scenarios (what-ifs) and decide which is best for your design implementation.

The Directed Partitioning Mode allows you to target logic into various device architectures without specific knowledge of signal-to-pin placement. The partitioning software automatically determines logic dependencies and makes certain all required logic is partitioned into the specified devices.

The Manual Partitioning Mode allows editing the Physical Information file to control every aspect of partitioning. The Manual Partitioning Mode is used when you know exactly how the logic is to be placed into one or more devices. This mode is most often used when recreating a design originally created by the automatic mode.

The Partitioning Process

There are four possible ways to control partitioning for MACHXL to give the best design implementation:

1) Directing the partitioning (i.e., setting the list of templates).
2) Selecting possible devices (available files).
3) Setting partitioning constraints.
4) Setting partitioning priorities.

This section describes each of these steps in detail and explains how they can be used to give the best implementation.

Directed Partitioning

When you determine the parts of your design to be placed into specific programmable device architectures, you must use the directed partitioning mode. Directed partitioning is accomplished through the use of a Physical Information (*.pi*) file. The *design_name.pi* file contains the specifications for device partitioning via the Physical Information Language (PIL). The Physical Information Language is similar in construct to the Design Synthesis Language, and allows you to direct partitioning aspects. For more detailed information on the *.pi* file, please see **Chapter 14**.

Placing Logic into Specified Devices

The key to directed partitioning is the ability to specify which logic is placed into a specific device. This is accomplished by placing a DEVICE construct in the *.pi* file. As an example, if the design contains two OUTPUT signals `out1` and `out2` and they are to be placed into an AMD PLD, the *.pi* file might appear as in the following example.

Example

```
DEVICE
        TARGET 'PART_NUMBER AMD PALCE22V10H-5JC/5';
        out1 ;
        out2 ;
END DEVICE;
```

If a different design has four outputs, `out1, out2, out3`, and `out4`, and it is desirable to place `out1` and `out2` into an AMD PLD and `out3` and `out4` into an AMD MACH device, the *.pi* file might read as in the following example.

Example

```
DEVICE
      TARGET 'PART_NUMBER AMD PALCE22V10H-5JC/5';
      out1 ;
      out2 ;
END DEVICE;

DEVICE
      TARGET 'PART_NUMBER AMD MACH110-15JC' ;
      out3 ;
      out4 ;
END DEVICE;
```

Placing Unspecified Logic

In the description of automatic partitioning, we mention that all logic unspecified by directed partitioning will be left to the automatic partitioning algorithms. The Physical Information Language lets you dictate where unspecified logic is placed. It also eliminates the need to specify all the logic in the system.

In order to place unspecified logic, use the default construct. In the example below, *out1* and *out2* are required in a fast (5ns) 22V10, but the rest of the logic can be placed in a slower AMD device.

Example

```
DEVICE
      TARGET 'PART_NUMBER AMD PALCE22V10H-5JC/5' ;
      out1 ;
      out2 ;
END DEVICE;

DEVICE
      TARGET 'PART_NUMBER AMD PALCE16V8Z-25PC' ;
      default ;
END DEVICE;
```

Note: The DEFAULT construct can be applied to any DEVICE. If the default construct is NOT used then automatic partitioning occurs on the unspecified logic.

Pinout and Architectural Feature Specification

Another feature of directed partitioning is the ability to specify device pinouts and architecture-specific features. Using the *.pi* file syntax, the signal-to-physical-pin assignment information and device-specific information may be passed to the partitioning and fitting software. This feature is available in all three modes of partitioning. For more information on this capability, see **Chapter 14** on *.pi* file usage, or **Chapter 15** on directed partitioning for specific devices.

Setting the Template List

You may also set up a template list for use in the MACHXL menu system. For more information, see **Chapter 3**.

Setting Partitioning Constraints

Constraints allow you to further narrow the list of devices considered by the partitioning software when producing device solutions. As mentioned earlier, only those devices appearing in the device list are considered by the partitioning software. Using this device list, the constraints are compared against the devices and only those matching the specified constraints are considered valid device solutions by the partitioning software.

By using constraints wisely, you can investigate various combinations of the available devices for your design. For example, one run of the software may generate all single device solutions. Without modifying the available file, a second run could look for three or fewer device solutions using only TTL and CMOS devices.

The following table shows the various constraints you may specify. These constraints can be entered by using the MACHXL interface or by editing the cost (*.cst*) file directly.

Constraint	Purpose
FAMILY	Use to select ECL, TTL, or CMOS devices.
MANUFACTURER	AMD.
TEMPLATE	Use as an alternative to paring down the available file to contain only those templates of interest. It is possible to maintain a large available file and use the TEMPLATE constraint to filter out those devices deemed undesirable.
NUMBER_DEVICES	The MACHXL partitioning software generates solutions containing up to 20 devices. By limiting this value, the software considers only smaller solutions and greatly improves the partitioning speed.
TEMP_RANGE	Select COM, MIL, or EXT devices.
PACKAGE	Programmable logic devices come in a variety of package types. Use this constraint to limit the package types considered valid. The section at the end of this chapter entitled ***"IC Package Descriptions"*** shows more detail on package types.
TPD	The maximum propagation delay value can be entered as a constraint. This value is used as a filter for each device checked. The propagation delay of each individual device is calculated as follows: 1) For combinatorial (non-registered) devices, the maximum propagation delay is the worst case Tpd, as published by the manufacturer. 2) For registered devices, the maximum propagation delay is the sum of Ts and Tco (setup-and-hold time and clock-to-output time). 3) For devices with both combinatorial and registered outputs, the larger of (1) or (2) is used.

Constraint	Purpose
ICC	The maximum current value can be entered as a constraint. Each individual device is checked against the entered value.
FMAX	The minimum frequency value can be entered as a constraint. Each device is checked against the entered value to assure it greater than or equal to this value.
USER1 and USER2	These user modifiable values that appear in the available file can be used as constraints as well. It is possible to select only devices with a user1 value greater than 75 or perhaps a user2 value equal to 1. One common application for these user fields is device defect rate. If your production group has failure statistics on devices (0-100), then it is possible to enter those values into your available files. A user criterion could be used to select devices with a failure rate of less than 10%.

Setting Partitioning Priorities

The device constraints described above allow limiting the list of possible devices. Assigning priorities, on the other hand, enables the MACHXL partitioning software to determine which solutions are better than others.

You can prioritize solution characteristics by assigning them values between 0 and 10. A value of 10 designates the highest priority should be assigned; a value of 1 designates the lowest priority. All criteria assigned a priority of 0 are deemed unimportant in generating device solutions.

When multiple criteria are assigned priorities, the MACHXL software uses relative weightings to determine the best solution. For price to be twice as important as size, which is twice as important as frequency, you might assign PRICE a priority of 8, SIZE a priority of 4, and FMAX a priority of 2. There is no limit to the number of priorities you may use.

The following table shows the various priorities you may specify. These priorities are entered by using the MACHXL interface or by editing the cost (*.cst*) file using a text editor.

Priority	Purpose
PRICE	Use this to find the solution with the total lowest price. If this is the only priority set, it is possible that the MACHXL software will find cheaper, multiple device solutions instead of more costly, single device solutions.
SIZE	In the MACHXL partitioning system, size equates to pin count. This priority will cause the software to attempt to minimize total pin count.
TPD	In multiple device solutions, the software determines the largest propagation delay of all the individual devices (see above) and uses this as the solution propagation delay. By prioritizing Tpd, the MACHXL software attempts to find solutions with the smallest solution propagation delay (the solution prop delay is, once again, the largest of the individual prop delays).
ICC	This is the individual Icc value for each device in the solution. By prioritizing Icc, MACHXL attempts to find solutions with the LOWEST total Icc.
FMAX	The minimum frequency value can be entered as a constraint. Each individual device is checked against the entered value to assure it is less than or equal to this value.

Constraint	Purpose
USER1	In the constraints section above, you can specify a user criteria to indicate comparison (USER1 > 4) OR equality (USER1 = 11). When comparison constraints are used, this priority can be used. If "USER1 > 0" is the constraint, then prioritizing USER1 will cause a solution with a USER1 value of 99 to be considered "better" than a solution with a USER1 value of 4. Similarly, if the constraint is "USER1 < 75", then a solution with a USER1 value of 6 is "better" than a solution with a USER1 value of 44. Note that if equality constraints are used, this priority will have no effect (since all solutions will have the SAME user value). The user value for a solution is the sum of the individual user values of the devices in that solution.
USER2	Same as USER1 above

14 Controlling Partitioning and Fitting (Optional)

Contents

Introduction

MACHXL's Partitioner/Fitter automatically partitions and fits designs without interaction. You can exercise control over how the fitter selects devices by setting constraints and priorities. The fitter will still partition and fit your design automatically, using these user-settable priorities and constraints. For most designs this is a quick and easy way to get your design into the programmable device(s).

However, there are some situations where you may need to exercise more control over the partitioning and fitting process. The following are some examples:

- Additional circuitry caused your design to outgrow its original device. You need to change to a device with another architecture but keep the same pinout as the old device.
- You have several signals in your design that are very interactive and timing among them is very critical. For timing reasons these signals should all be fit into the same device.
- The design you are working on has very tight PCB real estate constraints and you would like to force the design into a single device.
- Most of your design can be fit into a slower, less costly device. However, one block needs to be fit into a fast PLD.

One major advantage of MACHXL is that it gives you the capability to control the fitter's automatic partitioning and fitting as little or as much as your situation requires. Control (outside of constraints and priorities set in the Partitioning menu in MACHXL) is done with a Physical Information (*.pi*) file. The *.pi* file directs how the fitter does its job. If you don't need the control, the fitter will perform its functions automatically. However, if you need the control, the *.pi* file tells the fitter how to modify its fitting.

The *.pi* file contains instructions you give to the fitter on how the design should be partitioned and fit. The following are some of the functions a *.pi* file lets you control:

- Synthesis of equations, including the size of equations generated and the amount of reduction performed on each equation
- How a design is partitioned among devices
- How signals are grouped together
- How signals are assigned to pins on a device
- How individual signals are fit inside the device
- Which specific fuses are blown or left intact
- Which device specific features are used within each device

The *.pi* file instructions may be as simple as specifying a particular device or as complex as controlling node paths inside the programmable device. So, the *.pi* file gives a continuum of control from fully automatic partitioning and fitting to full user-specification of devices and signals within.

You create a *.pi* file using a text editor and the Physical Information Language (PIL). PIL is a case-insensitive addition to MACHXL's Design Synthesis Language allowing you to specify device-specific constraints. The text editor can be invoked through the menuing system (see **Chapter 3** for more information on the menuing system), or you may use any text editor you normally use to create a source file. The file must be named *design_name.pi* (where *design_name* is the name of your design) and must reside in the same directory as the design.

The information in this chapter should be used in conjunction with **Chapter 15**. This chapter gives the basic structure of the *.pi* file, while **Chapter 15** discusses the meaning of each of the device-specific *.pi* file properties.

This chapter is made up of three sections. The first section dicusses sections of the *.pi* file and the purpose of each. The second section is a reference of the

commands and constructs used in the *.pi* file, giving the syntax and usage of each command. The third section contains several examples of *.pi* files based on common design issues and how they are controlled with a *.pi* file.

How the *.pi* File Controls Partitioning

When partitioning a design among the various device architectures, MACHXL allows the user to operate in one or a combination of the following three modes:

- Automatic partitioning
- Directed partitioning
- Manual partitioning

Automatic Partitioning

The Automatic Partitioning Mode allows the partitioning software to run unconstrained. No direction is given to the software with regards to signal/pin placement or logic partitioning. This mode is the easiest to use as it requires no special files to be created or modified. For many designs, this is the only mode needed to partition a design.

Directed Partitioning

In Directed Partitioning you edit the *.pi* file to control broad-based aspects of partitioning without specifying all of the details of fitting the design.

The Directed Partitioning Mode allows you to target logic into various device architectures without specific knowledge of signal-to-pin placement. The partitioning software automatically determines logic dependencies and makes certain all required logic is partitioned into the specified devices.

Manual Partitioning

In the Manual Partitioning Mode you edit the *.pi* file to control every aspect of partitionng.

The Manual Partitioning Mode is used when you know exactly how the logic is to be placed into one or more devices. This mode is most often used when recreating a design originally created by the automatic mode.

Creating a *.pi* File

You create a *.pi* file using a text editor and the Physical Information Language (PIL). The file must be named *design_name.pi* (where *design_name* is the name of your design) and must reside in the same directory as the design. PIL is a case-insensitive, free-format language that's an addition to MACHXL's Design Synthesis Language. You can also access the *.pi* file through the menuing system. For more information on the menuing system, see **Chapter 3**.

Physical Information File Language Reference

Physical Information Language Keywords

The Physical Information Language (PIL) has keywords allowing you to describe the specifics of device partitioning and signal grouping. The following are the keywords used in PIL. Notice that some of these keywords are the same as in the Design Synthesis Language but are used in a different context. The identifiers listed below are reserved by the language as keywords and may not be used for other identifier purposes.

BLOWN	COMP_ON	DEVICE
COMP_OFF	DEFAULT	END

FIXED	LOW-VALUE	SECTION
GROUP	NAME	TARGET
HIGH-VALUE	NO_CONNECT	VIRTUAL
INPUT	OUTPUT	
INTACT	PHYSICAL	

In addition to the above reserved keywords, the following identifiers are used as property strings in the Physical Information Language (PIL).

BLOCKMODE
CLOCK_BY_PIN
CLOCK_BY_ROW
COMB_OUT_REG_FB
COMMON_SET_PTERM
DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
FIT_WITH
FLOAT_NODES
FORCE_INTERNAL_FB
FUSEMAP_FILE
JEDEC_FUSEMAP
MACH_LOW_POWER
MACH_USERCODE
MACH_UTILIZATION
MACH_ZERO_HOLD_INPUT
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
MAX_NODE_FROM_EXPANDERS
MINC_FITTER
NO_COLLAPSE
OPEN_DRAIN
PLA_FITTER
PLA_PROPERTY
PLA_PTERM_UTILIZATION
PLD_INPUT_UTILIZATION
PLD_OUTPUT_UTILIZATION
POLARITY_CONTROL
SET_PTERM
SIGNATURE
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

.pi File Syntax Rules

The following are rules for syntax in the *.pi* file.

- Signals and DEVICEs are not required to have properties attached to them in order to be listed in the *.pi* file. However, properties change the functions of the signals or DEVICEs to which they are attached. This means while the following two lines are both valid in a *.pi* file, their actions in the *.pi* file will differ.

```
sync1 . . sync5;
sync1 . . sync5 { MAX_PTERMS 8 };
```

- As with the Design Synthesis Language, each line (with the exception DEVICE, GROUP, or SECTION keywords), must end with a semicolon, as shown in the following examples:

```
DEVICE
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      out1..out5;
END DEVICE;

GROUP
      count_bits [8] ;
      sync [ 5 ]  { MAX_PTERMS  8 } ;
END GROUP ;

SECTION
      { MAX_PTERMS  8 } ;
      TARGET  'a' ;      "force out7 .. out8 into
                         "MACH block A
      out7 : 5,  out8 : 6 ;    "and onto pins 5
                               "and 6
END SECTION
```

Properties shown in this chapter with curly braces { } must be listed that way in the *.pi* file. That is, the curly braces are not optional.

- Non-numeric property arguments must be surrounded by single quotes ' '.

- The order in which properties are listed for a signal does not matter. Thus the following two lines of a *.pi* file are functionally the same.

```
INPUT in5 { MAX_PTERMS 8, MAX_SYMBOLS 4 };
INPUT in5 { MAX_SYMBOLS 4, MAX_PTERMS 8 };
```

- The *.pi* file language is case insensitive. Certain keywords and properties are shown in capital letters in this chapter for the sake of clarity.

Comments

Comments may be inserted into the .pi file in the same way as with the Design Synthesis Language. Any comment must be preceded by a double quote ("). A new line ends a comment. Each line of multiple line comments must be preceded by double quotes.

COMP_OFF and COMP_ON

As in the Design Synthesis Language, you may also use the COMP_OFF and COMP_ON commands to exclude certain sections of the file executing (for more information on COMP_OFF and COMP_ON, see **Chapter 9**.) This is useful when debugging a *.pi* file.

Input and Output Signals in the *.pi* File

A signal list consists of input, output, and biput signals. Output signals are fit on output or biput pins. There can be at most one reference to an output signal in the *.pi* file for a design. Input signals are fit on input or biput pins. There can be many references to an input signal in a *.pi* file. Signals declared as OUTPUTs or NODEs in the design source (i.e., *design_name.src*) may be used as input signals to a device.

> ***Note**: The .pi file covers inputs and outputs from the device perspective, not from the design perspective. In a design, an input or output signal may be any signal coming into or out of a design block. From the standpoint of a device, an input is a signal that can be fit on an input pin, and an output is a signal that can be fit on an output or a biput pin.*

OUTPUTs or NODEs without the modifiers INPUT or OUTPUT are assumed to be output signals. NODE signals in the design source file without the modifiers VIRTUAL or PHYSICAL are assumed to be physical nodes in the *.pi* file. Virtual nodes in the design source file may not appear as a signal in the *.pi* file.

Syntax

```
INPUT  signal_list;
OUTPUT  signal_list;
```

Example

```
out1, out2, out3;
INPUT out_as_in1, out_as_in2;
```

Note: The pin "identifier" is also the pin name, as indicated in a data book specification for the device. A pin assignment is only meaningful if a target device is given.

.pi File Structure

The following is a suggested organization of a *.pi* file:

- Global Properties
- Ungrouped signals (signals not associated with group or device specifications)
- Group specifications
- Device specifications

An explanation of each of these is given in the following sections.

Each section of the *.pi* file is optional. For example, you can create a simple *.pi* file consisting of only global properties. This allows you easy control of design optimization. Or, you could create a *.pi* file with only device specifications. This allows you to control the pinout on devices in your design.

Global Properties

Global Properties are properties applying to all of the signals or DEVICEs in the design. These properties usually affect the optimization of signals. Global properties can be overridden by properties within a device specification or by pin-specific properties. For more information on the specifics of these properties, see **Chapter 15**.

Global *.pi* File Properties

DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
MACH_UTILIZATION
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
NO_COLLAPSE
PLA_PTERM_UTILIZATION
PLD_INPUT_UTILIZATION
PLD_OUTPUT_UTILIZATION
POLARITY_CONTROL
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

Syntax

{ *global_property value* };

Example

The following example shows how to use a global property to limit the number of p-terms (OR TERMS) of all the output signals in a design.

```
{MAX_PTERMS 8};
```

Ungrouped Signals

Individual signals not associated with a Group or Device Specification are known as "ungrouped signals", and can be included in the *.pi* file. This lets you control signal optimization without specifying the device into which the signal should be fit.

Only output, biput, and node signals may be ungrouped signals. Ungrouped signals must not include pin assignments.

Any ungrouped signal will be treated as a physical node by MACHXL and will not be collapsed out during optimization (see **Chapter 12, *Optimizing a Design*** for more information on how the optimizer handles node collapsing.)

The syntax and examples of ungrouped signals are shown in the following examples:

Syntax

```
signal_name { signal_property_1  value, ..
      signal_property_n  value },
signal_name { signal_property_1  value, ..
      signal_property_n  value };
```

Example

```
sync { MAX_PTERMS 8 } ;
sig1, sig2 { MAX_PTERMS 4,  MAX_SYMBOLS 8 } ;
      "MAX_SYMBOLS and MAX_PTERMS apply
sig3, sig4 ;            "to both sig1 and sig2
```

Syntax

```
signal_name_1 .. signal_name_n  { signal_property_1
      value, .. signal_property_n  value } ;
```

Example

```
out1 .. out2 { MAX_PTERMS 5,  MAX_SYMBOLS 4} ;
            "MAX_PTERMS and MAX_SYMBOLS apply
      "to out1 and out2 since they
      "are listed as a range
```

Syntax

```
array_id {signal_property_1 value, .
      signal_property_n value} ;
```

Example

```
Y_array {MAX_PTERMS 4, MAX_SYMBOLS 4};
```

Syntax

```
array_id  [ index ] { signal_property1 value, ..
      signal_property_n value } ;   "index is an
                                    "integer
```

Example

```
sync  [ 5 ]  { MAX_PTERMS  8,  MAX_SYMBOLS 4 } ;
```

Syntax

```
array_id  [ index_1 .. index_n ]
      { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
gray_cnt [ 3 .. 8 ]  { MAX_SYMBOLS 9,  MAX_XOR_PTERMS 2
};
```

Syntax

```
DEFAULT {signal_property_1 value, .
      signal_property_n value} ;
```

Example

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

Virtual Signals

If a signal is declared in the source (*design_name.src*) file simply as a NODE, the optimizer has the option of considering this signal as a virtual node or as a physical node. If the signal is considered a virtual node, the optimizer

collapses it out during equation synthesis. If the signal is considered a physical node, the optimizer leaves it as an equation and the signal will be fit into a device. The optimizer will determine whether a simple node should be physical or virtual automatically in order to synthesize the most efficient equations. See **Chapter 5** for more information about PHYSICAL and VIRTUAL NODEs.

The VIRTUAL modifier lets you specify explicitly which nodes in your design can be collapsed out of equations during optimization. This helps ensure a design is optimized the same way every time it's optimized.

Note: By default, if a simple NODE signal is mentioned in the physical information file, it is treated as a physical node, and will not be collapsed during optimization.

The VIRTUAL modifier can only be used on ungrouped signals. Nodes specified within the GROUP or DEVICE constructs will be treated as physical nodes.

VIRTUAL nodes cannot have properties or pin assignments associated with them.

Syntax

- VIRTUAL signal_name , signal_name ;
- VIRTUAL signal_name_1 .. signal_name_n ;
- VIRTUAL array_id ;
- VIRTUAL array_id [index] ;
- VIRTUAL array_id [index_1 .. index_n] ;

Example

```
VIRTUAL out1, out2;
VIRTUAL out1..out4;
VIRTUAL yout;
VIRTUAL yin[8];
VIRTUAL yout [1..8];
```

Signal Properties for Ungrouped Signals

Ungrouped signals may be assigned signal properties affecting the optimization of the signals. These signal properties have precedence over global properties in the *.pi* file and affect only the associated ungrouped signals.

.pi File Properties for Ungrouped Signals

DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
FIT_WITH
MACH_LOW_POWER
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
NO_COLLAPSE
POLARITY_CONTROL
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

For more information on the specifics of these properties, see **Chapter 15.**

Syntax

```
signal_name { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

To assign two properties to the same ungrouped signal use:

```
out {MAX_SYMBOLS 4, MAX_PTERMS 4};
```

DEFAULT Statement for Ungrouped Signals

The DEFAULT statement lets you specify the properties and grouping for all of the signals in the design that are not otherwise listed in the *.pi* file. This allows convenient property assignment to group together many signals without explicitly specifying the signals in the *.pi* file.

When the DEFAULT statement is specified outside of a DEVICE or GROUP specification (i.e., is ungrouped), all of the signals not specified in the *.pi* file will be treated as ungrouped signals and will be affected by the DEFAULT statement.

There can be at most one DEFAULT statement in each *.pi* file.

Syntax

```
DEFAULT {signal_property_1 value, .
      signal_property_n  value};
```

Example

This example specifies all signals in the design except `a1`, `a2`, and `a3` will have no more than eight product terms in their equations. There is no similar limit on the number of product terms in the equations of `a1`, `a2`, or `a3`.

```
a1, a2, a3;
default {MAX_PTERMS 8};
```

Group Specifications

The GROUP construct lets you specify a group of signals you want fit in the same device (the device selected to fit the group is left to the MACHXL partitioner). The *.pi* file can include multiple Group Specifications, if needed. This construct is useful when you need to place a set of signals together for timing, board layout, or other reasons.

Groups of signals specified in a GROUP construct may merge together with other GROUPs and ungrouped signals to form the most efficient partitioning

solution. The ungrouped signals may consist of output, biput, or physical node signals not otherwise mentioned in the *.pi* file, or signals at the global level of the *.pi* file. Only output, biput, and node signals may be members of a GROUP. The signal list must not include pin assignments in the GROUP construct.

Syntax

```
GROUP
       [ name ]
       [ signal_list ]
       [ default ]
END GROUP;
```

The above items in the GROUP construct may appear in any order. There may be at most one NAME construct per GROUP, and one DEFAULT construct for each *.pi* file.

Naming a Group

The NAME construct is used to assign a name to a GROUP. The given name will appear with the group in the *.npi* file. For more information the the *.npi* file, see the section entitled ***Using the .npi File to Recreate a Pinout*** later in this chapter.

Naming a group can be useful for documentation purposes. Naming has no effect on the fitting process. There may be at most one NAME construct per GROUP.

Syntax

```
NAME identifier ;
```

Listing Signals in a Group

The signals list for a Group specification is a list of output, biput, and node signals to be included in the GROUP construct, as well as any signal properties for the list.

Examples and syntax of signal lists for grouped signals are shown below:

Syntax

```
signal_name { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
GROUP
sync ;
sig1,  sig2  { MAX_SYMBOLS  8 } ;   "MAX_SYMBOLS 8
                        "applies to sig1 and sig2
END GROUP ;
```

Syntax

```
signal_name_1 .. signal_name_n { signal_property_1
      value, .. signal_property_n  value }  ;
```

Example

```
GROUP
      o1 .. 08 ;
      out1 .. out2  { MAX_PTERMS  5 } ;   "MAX_PTERMS 5
applies to out1 and out2
END GROUP ;
```

Syntax

```
array_id  { signal_property_1  value, ..
      signal_property_n  value }  ;
```

Example

```
GROUP
      Xarray ;
      Yarray  { MAX_PTERMS  4,  MAX_SYMBOLS  4 } ;
END GROUP ;
```

Syntax

```
array_id [ index ] { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
GROUP
      count_bits [8] ;
      sync [ 5 ]  { MAX_PTERMS  8 } ;
END GROUP ;
```

Syntax

```
array_id [ index .. index_n ] { signal_property_1
      value, .. signal_property_n  value }  ;
```

Example

```
GROUP
      out [ 0 .. 8 ] ;
      grey_cnt [ 3 .. 8 ]  {MAX_PTERMS 8;
      MAX_SYMBOLS 16 } ;
END GROUP ;
```

Syntax

```
DEFAULT { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
GROUP
      DEFAULT  { MAX_PTERMS  8 } ;
END GROUP ;
```

Signal Properties for a Group

Signal properties for a GROUP construct are properties applying to the signals which the properties are attached. Signal properties have precedence over global properties in the *.pi* file. The properties affect the optimization of the signals.

.pi File Signal Properties Supported in the GROUP Construct:

DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
FIT_WITH
MACH_LOW_POWER
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
NO_COLLAPSE
POLARITY_CONTROL
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

For more information on the use of these properties, see **Chapter 15**.

Syntax

```
{ signal_property_1  value, .. signal_property_n
      value } ;
```

Example

To assign two properties to the same grouped signal, use:

```
GROUP
      out  { MAX_SYMBOLS  4,  MAX_PTERMS  4 } ;
END GROUP ;
```

DEFAULT Statement in a Group

The DEFAULT statement lets you specify the properties and grouping for all of the signals in the design not otherwise listed in the *.pi* file. This allows you to conveniently assign properties and group together many signals without explicitly specifying the signals in the *.pi* file.

When the DEFAULT statement is used in a GROUP construct, a group will be created containing all unspecified signals in the design.

There may be at most one DEFAULT statement in each *.pi* file. Properties on the DEFAULT statement are optional.

Syntax

```
DEFAULT  { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

This example specifies all signals in the design other than *a1*, *a2*, and *a3* will be placed in one group, and fit into the same device. The optional property MAX_PTERMS specifies signals in the group will have no more than eight product terms in their equations.

```
a1 ,  a2 ,  a3 ;

GROUP
      default  { MAX_PTERMS  8 } ;
END GROUP ;
```

Device Specifications

Device specifications let you describe device-specific information, such as the placement of signals on each device in the design. Device specifications are used as part of the Manual and Directed Partitioning modes, and give you access to device-specific features.

The DEVICE construct lets you define the device specifications. Each DEVICE construct generally corresponds to one physical device. The DEVICE construct may have embedded GROUPs or SECTIONs

(SECTIONs are discussed later in this chapter). The SECTION construct allows you to describe subsections for devices having subsections, such as the MACH devices.

Syntax

```
DEVICE
       [properties]
       [target_statement]
       [NAME]
       [signal lists]
       [DEFAULT]
       [HIGH-VALUE]
       [LOW-VALUE]
       [NO_CONNECT]
       [SECTION]
       [signal lists]
       [GROUP]
       [BLOWN fuses]
       [INTACT fuses]
END DEVICE ;
```

The above items in the DEVICE construct may appear in any order. However, there may be at most one NAME construct per DEVICE and one DEFAULT construct in a .pi file.

Device Properties

Device properties are properties applying to all signals in the device or to the device itself. These properties affect the optimization of signals, as well as how device features are utilized. Device properties have precedence over global properties. Signal properties can override device properties.

Device properties supported in the *.pi* file:

BLOCKMODE
CLOCK_BY_PIN
CLOCK_BY_ROW
COMMON_SET_PTERM
DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FLOAT_NODES
FIT_AS_OUTPUT
FORCE_INTERNAL_FB
FUSEMAP_FILE
JEDEC_FUSEMAP
MACH_LOW_POWER
MACH_UTILIZATION
MACH_ZERO_HOLD_INPUT
MAX_NODE_FROM_EXPANDERS
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
MINC_FITTER
NO_COLLAPSE
OPEN_DRAIN
PLA_PTERM_UTILIZATION
PLD_INPUT_UTILIZATION
PLD_OUTPUT_UTILIZATION
POLARITY_CONTROL
SET_PTERM
SIGNATURE
XOR_POLARITY_CONTROL

For more information on the use of these properties, see **Chapter 15**.

Syntax

```
{ device_property_1, .. device_property_n  value };
```

Example

To limit the number of p-terms (OR TERMS) of all the output signals on a device, use the following device property:

```
DEVICE
{ MAX_PTERMS  8 } ;
.
.
.
END DEVICE ;
```

Naming a Device

The NAME construct is used to assign a name to a DEVICE. The given name will appear with the group in the *.npi* file. For more information the

.npi file, see the section entitled ***Using the .npi File to Recreate a Pinout*** later in this chapter.

Naming a device can be useful for documentation purposes. Naming has no effect on the fitting process. There may be at most one NAME construct per DEVICE.

Syntax

```
NAME identifier ;
```

Targeting a Specific Device for Fitting

The TARGET construct for device specifications tells the fitters which device to use. When TARGET is used with the DEVICE construct, you can target the device, template, part number or footprint you want to use.

The TARGET construct allows you to specify devices three ways:

- You can specify the exact device using the manufacturer's part number.
- You can specify the type of device you want to use and the package type (footprint). The combination of device and footprint can help MACHXL's fitters find second-source devices for your design from its extensive Device Library.
- You can specify the footprint of the device only. The footprint specification can help MACHXL's fitters find a replacement for an existing device that may make modifications to your PCB layout unnecessary.

Syntax

```
TARGET 'PART_NUMBER manufacturer_abbreviation
      part_number' ;
TARGET 'TEMPLATE template_name  footprint_name';
TARGET 'FOOTPRINT  footprint_name';
```

Where:

manufacturer_abbreviation, *part_number*, *template_name*, and *footprint_name* can be found in **Appendix A.**

Examples

To place outputs *o1*, *o2*, and *o3* into an AMD PALCE16V8H-10JC/4, use the following entry in the *.pi* file:

```
DEVICE
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      o1,  o2,  o3 ;
END DEVICE;
```

To place outputs *o1* and *o2* on specific pins of a 22V10 DIP package, use the following entry in the *.pi* file:

```
DEVICE
      TARGET  ' TEMPLATE  P22V10  DIP-24-STD ' ;
      o1 : 14,  o2 : 15 ;
END DEVICE;
```

To place outputs *o1*, *o2*, and *o3* on specific pins of a 20-pin DIP package, use the following entry in the *.pi* file:

```
DEVICE
      TARGET  ' FOOTPRINT  DIP-20-STD ' ;
      o1 : 12,  o2 : 13,  o3 : 14 ;
END DEVICE;
```

Listing Signals in a Device

The following shows syntax and examples of how to list the signals, nodes, signal properties, and signal directions to be included in a DEVICE construct.

In the following examples, *signal_name* and *array_id* are identifiers and *index* is an integer. *Pin_assignment* is the name assigned by the device manufacturer as shown in a part data book.

Note: A pin assignment is meaningful only if a target device is specified with the TARGET construct.

Syntax

```
INPUT|OUTPUT*  signal_name  : pin_assignment
      { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
DEVICE
OUTPUT sync  { MAX_PTERMS  8 } ;
INPUT in1 : 5 ,  in2 : 6 ;     "INPUT applies to in1
                               "and in2
.
.
END DEVICE ;
```

Syntax

```
INPUT|OUTPUT*  signal_name  : pin_assignment
      { signal_property_1  value, ..
      signal_property_n  value } ,signal_name  :
      pin_assignment{ signal_property_1  value, ..
      signal_property_n  value } ;
```

* INPUT and OUTPUT are optional

Example

```
DEVICE
TARGET  ' FOOTPRINT  DIP-20-STD ' ;
sync  { MAX_PTERMS  8 } ;
INPUT o1 : 5 ,  o2 : 6 ;       "INPUT applies to o1
                               "and o2
sig1 ,  sig2  { MAX_SYMBOLS  8 } ;
            " MAX_SYMBOLS applies to sig1 and sig2
END DEVICE ;
```

Syntax

```
INPUT|OUTPUT*  signal_name_1, .. signal_name_n
      { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
DEVICE
      OUTPUT  o1 .. o8 ;
      out 1 .. out2  { MAX_PTERMS  5 } ;
END DEVICE ;
```

Note: Pin numbers cannot be assigned when using the " . . " range indicator.

Syntax

```
INPUT|OUTPUT*  array_id  { signal_property_1    value, ..
signal_property_n  value } ;
```

* INPUT and OUTPUT are optional

Example

```
DEVICE
      INPUT  Xarray ;
      Yarray  { MAX_SYMBOLS  4,  MAX_PTERMS  4 } ;
END DEVICE ;
```

Note: Pin numbers cannot be assigned when using an array to represent a set of signals.

Syntax

```
INPUT|OUTPUT*  array_id [ index ]  : pin_assignment
      { signal_property_1  value, ..
      signal_property_n  value } ;
```

Where *index* is an integer.

Example

```
DEVICE
      TARGET  ' FOOTPRINT  DIP-20-STD ' ;
      INPUT count_bits [ 8 ] ;
      sync [ 5 ]  : 12  { MAX_PTERMS  8 } ;
END DEVICE ;
```

Syntax

```
INPUT|OUTPUT*  array_id [ index_1 .. index_n ]
      { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
DEVICE
      INPUT gray_cnt [ 3..8 ]  { MAX_SYMBOLS 8 } ;
END DEVICE ;
```

Note: Pin numbers cannot be assigned when using an array to represent a set of signals.

Syntax

```
DEFAULT { signal_property_1  value, ..
      signal_property_n  value } ;
```

Example

```
DEVICE
      DEFAULT  { MAX_PTERMS  8 } ;
END DEVICE ;
```

Note: Pin numbers cannot be assigned when using DEFAULT to represent a set of signals.

Renaming the Fusemap File of a Device

Should you need to, you can rename your fusemap files (typically JEDEC files) with the FUSEMAP_FILE statement.

Syntax

```
DEVICE
.
.
.
```

```
{ FUSEMAP_FILE   'newname.xxx' } ;
END DEVICE ;
```

If a fusemap filename is not specified, the default will be used. In the example shown above, the file would be renamed to *newname.xxx*.

Specifying Signal Directions in a Device

It is common for the output signal of one device to feed input pins on other devices in designs requiring multiple devices. To avoid ambiguity in device specification, specify explicitly the signal direction of any pin in the device specification. This will ensure the output signal is generated on the appropriate device in your design.

Syntax

```
INPUT signal_list;
OUTPUT signal_list;
```

Note: For bidirectional (BIPUT) signals, use the OUTPUT statement to specify the pin that generates the bidirectional signal.

The following rules apply when specifying the signal direction:

- Output signals (specified with the OUTPUT statement) will be fit onto output or biput pins of the device.
- The OUTPUT statement can only be used for node, output, and biput signals. If you specify the OUTPUT statement more than once for a signal, MACHXL will interpret this as specifying the signal should be generated on more than one pin. This causes MACHXL to generate an error.
- The INPUT statement can be used multiple times for a signal.
- The INPUT statement can be applied to any signal in the design. If the INPUT statement is applied to a node, output, or biput

signal, MACHXL interprets this as a signal generated on a different pin than is fed to this pin as an input.

- Inputs from the design file declared without the INPUT modifier are assumed to be input signals.
- Outputs and biputs from the design file declared without the INPUT or OUTPUT modifiers are assumed to be output signals.
- Nodes from the design file declared without the INPUT or OUTPUT modifiers are assumed to be physical nodes.

Signal Properties for a Device

Signal Properties for a DEVICE construct are properties applying only to the signals to which they are attached. Signal properties have precedence over global properties in the *.pi* file and device properties attached to the device. These properties affect the optimization of the signals and provide access to device-specific features.

.pi File Signal Properties Supported in the DEVICE Construct:

CLOCK_BY_PIN
CLOCK_BY_ROW
COMB_OUT_REG_FB
DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
FIT_WITH
FORCE_INTERNAL_FB
MACH_LOW_POWER
MAX_NODE_FROM_EXPANDERS
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
NO_COLLAPSE
OPEN_DRAIN
POLARITY_CONTROL
SET_PTERM
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

For more information on using these properties, see **Chapter 15.**

Syntax

```
{property_name_1 value,..property_name_n  value};
```

Example

To assign two properties to the same device signal, use:

```
DEVICE
      out   { MAX_SYMBOLS   4,   MAX_PTERMS 4 } ;
END DEVICE ;
```

DEFAULT Statement in a Device

The DEFAULT statement lets you specify properties for all of the signals in the design not otherwise listed in the *.pi* file. This allows convenient assignment of properties and grouping of many signals without explicitly specifying the signals in the *.pi* file.

When the DEFAULT statement is used in a DEVICE construct, a group will be created containing all unspecified signals in the design..

There may be at most one DEFAULT statement in each *.pi* file.

Syntax

```
DEFAULT   { signal_property_1   value, ..
      signal_property_n   value } ;
```

Example

This example specifies signals in the design other than *a1, a2*, and *a3* be placed in one device. The property MAX_PTERMS specifies signals in the group will have no more than eight product terms in their equations.

```
a1,   a2,   a3 ;

DEVICE
      DEFAULT   { MAX_PTERMS   8 } ;
END DEVICE ;
```

Assigning Logic Levels (High-Value, Low-Value, NO_CONNECT) to Pins of a Device

It is possible to assign logic levels to pins or to specify a pin not be used as a signal pin. The HIGH-VALUE, LOW-VALUE, and NO_CONNECT statements let you duplicate exactly the pin assignment of devices by not allowing the partitioning software to place signals on the pins.

Syntax

```
HIGH-VALUE  pin_assignment ;
LOW-VALUE  pin_assignment ;
NO_CONNECT  pin_assignment ;
```

Where:

> *pin-assignment* is an identifier (an identifier is the name assigned by the device manufacturer to a pin of the device as shown in a data book). A pin assignment is meaningful only if a target device is specified with a TARGET statement.

Example

```
DEVICE
TARGET  ' FOOTPRINT  DIP-20-STD ' ;
HIGH-VALUE  :  14 ;      "Pin connected to the high
                         "voltage source
LOW-VALUE  :  7 ;        "Pin connected to the low
                         "voltage source
NO_CONNECT  1, 2, 4;     "Pin left unconnected
END DEVICE;
```

Device Section Specifications

Some complex PLDs, like the AMD MACH, are organized into blocks or quadrants. In these devices each block can be viewed as a small PLD. Outputs from the block can be easily fed back into the same block. However,

it may not be easy or possible to feed all outputs from a block into a different block on the same device.

Because of this signal routing limitation in block-oriented devices, it may be useful to control which signals are to be placed into which blocks in the device. This helps ensure the device will be used efficiently.

The SECTION construct in a device specification lets you control which signals are placed into which blocks in a block-oriented device. This construct should only be used with block-oriented devices, such as the AMD MACH devices.

Syntax

```
SECTION
      [ properties ]
      [ target_statement ]
      signal_lists
END SECTION;
```

Section Properties

Section properties are properties applying to all of the signals in the section (i.e., a block or quadrant). These properties affect the optimization of signals, as well as how device features are utilized. Section properties have precedence over global properties and device properties. Signal properties can override section properties.

Section Properties Supported in the *.pi* File.

DEMORGAN_SYNTH
DISABLED_ONLY_FOR_TEST
FF_SYNTH
FIT_AS_OUTPUT
FLOAT_NODES
FORCE_INTERNAL_FB
MAX_PTERMS
MAX_SYMBOLS
MAX_XOR_PTERMS
NO_COLLAPSE
POLARITY_CONTROL
XOR_POLARITY_CONTROL
XOR_TO_SOP_SYNTH

For more information on using these properties, see **Chapter 15**

Syntax

```
{ property_name  value, .. property_name  value } ;
```

Example

To limit the number of p-terms (OR terms) within a section of the AMD MACH110, use the following declaration:

```
DEVICE
      TARGET  'TEMPLATE  mach110  jlcc-44-std ' ;
                         " place group into MACH110
SECTION
{ MAX_PTERMS  8 } ;
TARGET  'a' ;            "force out7 .. out8 into MACH
                         "block A
out7 : 5,  out8 : 6 ;    "and onto pins 5 and 6
END SECTION
.
.
.
END DEVICE ;
```

For more information on controlling specific devices, see **Chapter 15.**

Targeting a Block or Quadrant Within a Device

The TARGET construct for section specifications tells the partitioning process which block or quadrant to use when fitting specific signals.

For more information on Targeting specific devices, see **Chapter 15.**

Syntax

```
TARGET  'section_specification' ;
```

Where:

section_specification as well as TARGET usage is device-specific.

Example

To force the signals *out7* and *out8* into block A of an AMD MACH110, use the following declaration:

```
DEVICE
TARGET  'TEMPLATE  mach110  jlcc-44-std ' ;
                          " place group into MACH110
SECTION
{ MAX_PTERMS  8 } ;
TARGET  'a' ;             "force out7 .. out8 into MACH
out7,  out8 ;             "block A
END SECTION
.
.
.
END DEVICE ;
```

Specifying Signal Directions in a Section of a Device

Specifying signal direction in a section of a device is the same as specifying them in a device as a whole. Please refer to the section earlier in this chapter entitled ***Specifying Signal Direction in a Device***.

Listing Signals in a Section of a Device

Listing signals in a section of a device is the same as for a GROUP or DEVICE construct. Please see the section earlier in this chapter entitlied ***Listing Signals for a GROUP*** or ***Listing Signals in a Device***.

Grouping Signals Within a Device

Grouping signals within a block or quadrant of a block-oriented device is similar to using the SECTION construct. The GROUP construct lets you specify which nodes, outputs and biputs are to be grouped together.

Syntax

```
GROUP
      signal_list
END GROUP;
```

The GROUP construct within a DEVICE specification differs from a SECTION specification in three ways:

- The GROUP construct does not include a TARGET specification, where a SECTION specification may.
- You cannot specify INPUTS in the GROUP construct. All signals in the GROUP construct must be node, output, or biput signals.
- The signals in the GROUP construct may be merged with other signals or GROUPs in the design.

The behavior of the GROUP construct in a device specification is device specific for block-oriented devices. For a complete discussion of GROUP usage with a specific device, please see the appropriate section in **Chapter 15.**

Fuse-Level Programming Control

The *.pi* file allows you to control how PLD and CPLD devices are programmed at the fuse level. The fuse-level control commands, BLOWN and INTACT may only be used with an explicit target device. Fuse-level programming control commands override the programming done by the partitioning process.

BLOWN and INTACT

The *.pi* file commands, BLOWN and INTACT indicate the fuses in a device to be blown or left intact.

Syntax

```
BLOWN [fuse_list];
INTACT [fuse_list];
```

The list of fuses to be blown or left intact is represented by *fuse_list* and may be either a list of fuses separated by commas or a range. The order in which the commands are given does not matter.

Example

```
DEVICE
      TARGET 'TEMPLATE P16V8A DIP-20-STD';
      DEFAULT;
      BLOWN 2056, 2058, 2060 .. 2118;
      INTACT 2057, 2059;
END DEVICE;
```

Using the *.npi* File to Recreate a Pinout

After the design is successfully partitioned and fusemaps generated, the partitioning process creates a physical information file to document (in terms of PIL) how the design was fit into the devices in the solution. This partitioning process-generated physical information file is called the *.npi* (new pi) file, and is named *design_name.npi*. This *.npi* file may be used to recreate exactly the solution and signal placement on pins. This may be useful if, for example, the design must be changed functionally after the printed circuit board has been designed and laid out. GROUP and DEVICE constructs that were assigned a NAME will retain the given NAME in the *.npi* file. To use the *.npi* file, copy the file to the *.pi* file of the same file (i.e., *same_file_name.pi*).

Examples Using the *.pi* File

The following sections will show the form and content of the Physical Information file as you could use it in different situations. The examples are intended to illustrate the concepts behind the use of the *.pi* file. See the reference section for details about PIL constructs shown in the examples.

Example 1: Controlling the Size of Equations

You can use the *.pi* file to control the size of equations generated by the optimizer. Controlling the size of equations can have a major impact on the success of fitting and number of solutions generated by the fitter. If you know you will be using devices with macrocells having 8 or fewer pterms, you would want to keep the optimizer from collapsing nodes into equations with more than 8 pterms. In this case, the *.pi* file could contain:

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

MAX_PTERMS and MAX_SYMBOLS are examples of "properties". Properties are one means of controlling certain actions of the synthesis and fitting processes.

Example 2: Forcing Signals To Be Fit Together in the Same Device

A design implementing a counter has output signals heavily interdependent. For timing reasons the designer wants them fit together in the same device. The designer also wants the automatic device selection and partitioning to determine the best device according to your priorities. In this case, the *.pi* file could contain:

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

The signals that are members of the GROUP, *q0..q5* and *carry*, will be fit together in the same device, but there are no limitations imposed by the GROUP on the device used. In addition, other groups and ungrouped signals may be fit in the same device with this group.

Example 3: Using Specific Devices

A small prototype design has several reprogrammable P16V8As that are to be used during the debugging stage. In this case, the *.pi* file could contain:

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

Example 4: Maintaining Pin Assignments

You have an existing MACHXL design in which you have changed some logic and 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. In this case, the *.pi* file could contain:

```
DEVICE
      TARGET 'TEMPLATE P20V8A 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;
```

Here, 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.

For example, DIP-24-STD indicates a 24 pin DIP package with the standard pinout mapping (i.e., 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 would be DIP-28-A28.

Signals used as inputs to the device are marked with INPUT in the *.pi* file. The signals are assigned to pins by appending a :pin_name to the signal name, such as clk:2. Device pins to be left free are marked with NO_CONNECT. The pin names in the pin assignments and no-connect pins are the actual physical pin names for the targeted device. For example, if the targeted device is a PGA, a pin assignment will look like clk:A1.

Example 5: Fitting the Design into One Device

A designer would like to fit an entire design into one MACH110. The *.pi* file would look like the following:

```
DEVICE
      TARGET 'PART_NUMBER AMD MACH110-15JC';
      DEFAULT;
END DEVICE;
```

In this example, the DEVICE specification is marked as the DEFAULT device. The default device is the device containing all the output signals NOT mentioned elsewhere in the *.pi* file. Specifying a default device is optional. Here, it provides a quick way to put all the signals in the design into the same device. DEFAULT could even be given outside of any group or DEVICE specification (the "global level" of the *.pi* file), which means all unmentioned signals will be fit through automatic partitioning and fitting.

Example 6: Fitting the Design into More Than One Device

You have a design that will take two parts. In this case, the *.pi* file could contain:

```
DEVICE
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      out1..out5;
END DEVICE;

DEVICE
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      out6..out10;
END DEVICE;
```

Example 7: Mixing Automatic and Directed Partitioning

This example shows how automatic and directed partitioning can be mixed in the same design. Assume that your design is similar to the design of the last example. However, it has several critical signals, *state_bit_0..state_bit_7,* that must be placed into fast PLDs. In this case, the *.pi* file could contain:

```
DEVICE
      TARGET 'PART_NUMBER AMD MACH210-15JC';
      out1..out5;
END DEVICE;

DEVICE
      TARGET 'PART_NUMBER AMD MACH210-15JC';
      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, MACHXL's automatic device selection, partitioning, and fitting will find the fastest device or combination of devices available that will fit the critical functions.

Example 8: Refitting a Design Into the Same Footprint

A board is already in production, but a last minute specification change dictates a change in the logic implemented in the PLD. This causes the design to outgrow the P20R8 used. To refit the design into another architecture, but keep the pinout the same, the *.pi* file could contain:

```
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;
```

Here, the device is targeted to a FOOTPRINT (DIP-24-STD). Targeting a device to a footprint will, in effect, apply automatic device selection and fitting across devices matching the footprint. Depending on the form of the actual equations, there are up to 79 architectures potentially fitting this *.pi* file example. You can use the constraints and priorities of MACHXL's automatic device selection and partitioning to optimize the fit for price, speed, or other factors. The old pin assignments will be enforced, even without knowing in advance which architecture you will be using. This means the board layout will be preserved!

Example 9: Specifying Devices Without Specifying Signals

If you want to specify which devices to use without providing specific pin information, the DEVICE construct may be used without a signal list.

For example, to fit a design into two MACH210 devices and a MACH 130 device, the following *.pi* file will perform the task.

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


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

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

15 Device-Specific Partitioning (Optional)

Contents

Introduction

Chapter 14 introduces the Physical Information (*.pi*) file and how it is used to control device partitioning. As is discussed in that chapter, the *.pi* file lets you control MACHXL's automatic partitioning, specifically:

- How a design is partitioned among devices
- How device resources are used, including pin assignments.

While **Chapter 14** deals with the structure of the *.pi* file and the general device-control features, this chapter gives information about how to use the *.pi* file with specific devices, architectures, and families of devices.

General Device Fitting With .pi File Properties

Controlling PLD Utilization

In the case of some designs it is prudent to reserve PLD resources for future logic expansion. The NO_CONNECT construct can be used to keep specific pins free (for more information, see **Chapter 14**). There are also three additional properties for controlling the utilization of PLDs. These properties have no effect on CPLDs such as the AMD MACH devices.

- 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). The syntax for all three properties is the same, as shown below:

Syntax

```
{ PLD_INPUT_UTILIZATION percent };
{ PLD_OUTPUT_UTILIZATION percent };
{ PLA_PTERM_UTILIZATION percent };
```

Example

```
{PLD_INPUT_UTILIZATION 90};
{PLD_OUTPUT_UTILIZATION 85};
{PLA_PTERM_UTILIZATION 95};
```

In these examples, input utilization is limited to 90%, output utilization is limited to 85%, and PLA pterm utilization is limited to 95%. As an example, if a P22V10 is targeted, only 19 of the 22 available array inputs will be used, and only 8 of the 10 available outputs will be used.

Using the FIT_AS_OUTPUT Property

The FIT_AS_OUTPUT property allows you to control whether a node is fit as an OUTPUT or as a NODE. The FIT_AS_OUTPUT property can be placed on NODEs or OUTPUTs in the *.pi* file. The property 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 fitter to place this node signal on an output pin.

- The user may wish to tell the fitter to fit NODEs on OUTPUTs in PLD's. Defining a NODE to be on an OUTPUT during PLD fitting may speed-up the process significantly.

Example

SOURCE FILE

```
INPUT d,e,f,clk;
NODE d_node CLOCKED_BY clk;
output out, out1;

d_node = d;
out = d_node*e;
out1 = d_node+f;
```

PHYSICAL INFORMATION FILE

```
device
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      d_node{FIT_AS_OUTPUT};
      out;
end device;

device
      TARGET 'PART_NUMBER AMD PALCE16V8H-10JC/4';
      default;
end device;
```

Controlling How Signals Fit Together

Early in the fitting process, the fitter decides which signals fit together as one inseparable block of functionality.

For PLDs this means signals will be 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.

You can control this fitting process with two *.pi* file properties, NO_COLLAPSE and FIT_WITH.

- The NO_COLLAPSE property tells the back-end tools to fit this signal individually, separate from the fitting of any other signal.

- 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, the *.pi* file would contain:

```
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'};
```

Enables Used Only For Test

In some designs, an output is disabled only during test. During normal operation the output is never disabled and the signal on the input of the tri-state buffer is functionally equivalent to the signal on the output of the tri-state buffer.

MACHXL, however, treats two signals differently if there is an enable equation between them. The signals are considered functionally different. To indicate an output will only be disabled during testing, use the *.pi* property DISABLED_ONLY_FOR_TEST. This property tells the fitter to:

- Program the enable equation.

- Treat 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

```
out_x {DISABLED_ONLY_FOR_TEST};
```

If the output signal out_x has an enable, the enable equation will be programmed. If out_x is fed only a single signal, e.g., node_y, out_x and node_y will be interchangeable for feedback purposes. This property is for outputs in the *.pi* file, but a shorthand allows the property to be applied to a group.

Example

```
{DISABLED_ONLY_FOR_TEST};
```

This example applies the DISABLED_ONLY_FOR_TEST property to all outputs in the design.

Synthesis Control Properties

Three properties are available to control synthesis in the fitter.

- The DEMORGAN_SYNTH property controls DeMorgan synthesis of the data equations, where data equations are the D, JK, SR, T, XOR left and XOR right equations.
- The FF_SYNTH property controls flip flop synthesis.
- The XOR_TO_SOP_SYNTH property controls XOR to Sum-of-Products synthesis.

When using these properties, some things are not allowed.

- Control of DeMorganization of control equations, such as ENABLE, CLOCK, RESET, or PRESET.

- Control of DeMorganization of the J equation of a JK flip flop with no corresponding DeMorganization of the K equation.

By default, the fitter will automatically optimize the design. This means that there is generally little reason to use these properties. If the need does arise, however, the use of these properties is described in the following table:

Synthesis Control Properties for Use in the *.pi* File

Property	Value	Action
DEMORGAN_SYNTH	AUTO (default)	The back-end tool will automatically select the best DeMorganization choice.
	FORCE	Force the back-end tool to DeMorganize the primary equation (use the offset).
	OFF	Prevent the back-end tool from DeMorganizing the primary equation (use the onset).
FF_SYNTH	AUTO (default)	The back-end tool 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.
	D_FLOP	Require the target device to use a D flip flop.
	T_FLOP	Require the target device to use a T flip flop.
	JK_FLOP	Require the target device to use a JK flip flop.
	SR_FLOP	Require the target device to use an SR flip flop.
XOR_TO_SOP_SYNTH	*AUTO (default)	The back-end tool will automatically select between the XOR equation and the sum-of-products equation.
	FORCE	Force the back-end tool to use the sum-of-products equation.
	OFF	Force the back-end tool to use the XOR equation.

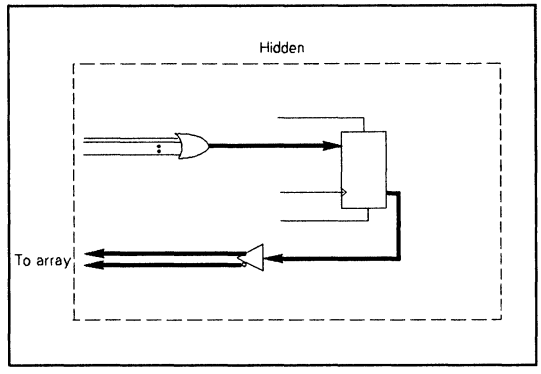
Accessing Internal Points in a Device

Hidden Nodes

A hidden node is a node not terminating in a physical pin connection.

Node signals are signals placed on hidden nodes. However, node signals are not restricted to hidden nodes; they can be placed on hidden nodes or visible pins.

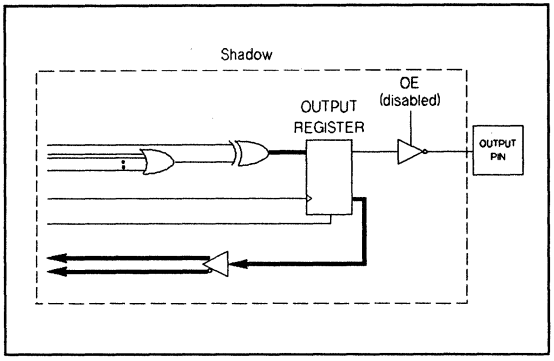
MACHXL may deliberately place a node signal on a visible pin for a variety of reasons.



Hidden Node

Shadow Nodes

A *shadow hidden node* (known simply as a *shadow node* or *shadow)* is created 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 the following figure. See Table 15-2 for the names of hidden and shadow nodes.



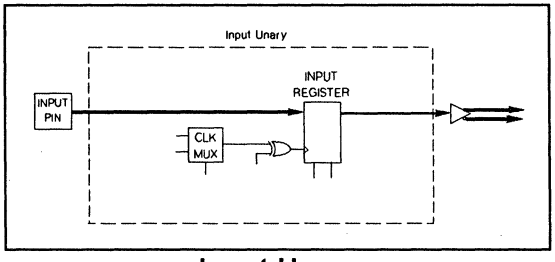
Shadow Node

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 essentially a clocked feedback path, called a *feedback unary*.

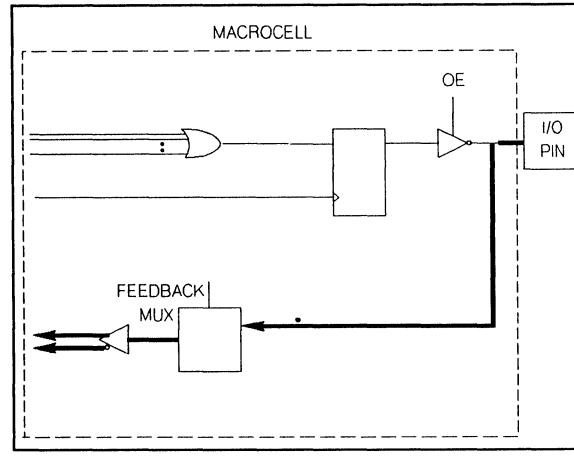
The following are diagrams and explanations of the two types of unaries.

Input Unary - A hidden unary in an input macrocell, i.e., a clocked input pin as shown below.



Input Unary

Feedback Unary - A feedback unary is a hidden unary path through the feedback register of an output macrocell, as shown below.



Feedback Unary

MACHXL allows selecting from these possible paths specifically. Nodes and unaries are specified in the *.pi* file by means of labels. A hidden node is specified with the label NODExx.

Feedback and input unaries are specified with the label UNARY_OF_xx, where xx is the manufacturer-specified pin number in the primary package, usually DIP.

A buried node is a hidden node where some external pin number is associated. A buried node or shadow node is specified with the label BURIED_OF_xx or SHADOW_OF_xx, where xx is the manufacturer-specified pin number in the primary package, usually DIP.

These labels (used in the *.pi* file) are enumerated in Tables 15-1 and 15-2, and Appendix D (for AMD MACH devices).

There are a large number of devices having general-purpose registers. The following example shows a sample in the Design Synthesis Language allowing the fitter to take advantage of these general-purpose registers.

Example

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

```
i = i_unclocked;
```

Functionally, this is equivalent to a clocked input. In this approach, however, both the clocked (*i*) and unclocked (*i_unclocked*) versions of the signal can be referenced in the design. Another advantage of this approach is it allows you to specify the hidden node in the *.pi* file. Furthermore, this description can be mapped into any device with a register. The above functionality is a *unary node*.

Devices With Unary Nodes

The templates which have unary nodes are the P16V8HD, P29M16, P29MA16, and the MACH2xx and MACH4xx parts. The MACH parts are discussed in their own sections later in this chapter.

Table 15-1. Node Descriptions and Labels by Device Template

Template	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
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

Other Device-Specific Information for PLDs

Synchronous Preset in the 22V10 Architectures

One device architecture supported by MACHXL has a synchronous preset row shared by some or all macrocells in the device.

The synchronous preset row can be used as a synchronous reset. If the fitter has DeMorganized the D equation on a device, then the asynchronous reset is now an asynchronous preset and the synchronous preset is now a synchronous reset. Given this anomaly, and the priority MACHXL places on insuring the same functionality for various device implementations, the fitter does not fit a preset equation onto any synchronous preset.

In some architectures, however, this common set can still be used (i.e., 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 the common set is accomplished by specifying the set pterm to use in the *.pi* file. This architecture is the 22V10.

In the 22V10, the synchronous preset row is common to all macrocells in the device. The pterms to use as the common set pterm for the device is specified with the COMMON_SET_PTERM property.

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;
```

In the above example, the common set pterm is *reset1*reset2*. This term sets the output low, so the fitter will automatically use the DeMorgan equation to meet this common set pterm requirement.

Accessing the Open-Drain Outputs of the P16V8HD

The P16V8HD architecture supports open-drain outputs. Unlike normal totem-pole outputs, an open-drain output will only drive 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 will depend on external loading and pull-up circuitry.

To direct outputs to be open drain, attach the OPEN_DRAIN *.pi* file 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 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.

MACHXL provides a function used to create open-drain output signals of the proper form. This function is available in the library dfeature, which resides in the *dsllib* directory under the MINC executable directory. The function is as follows:

```
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 1_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 MACHXL's simulator will simulate the functionality correctly and test vectors sent to the device programmer will also be correct. The fitter will generate two enable equations, one for open-drain capable devices and one for 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, an output can be fit on parts without the open-drain capability at the cost of increased enable equation complexity. Timing and parametric design issues should be considered independent of MACHXL's open-drain synthesis capability.

The open-drain function may also be used to aid in bus design. The following example shows bus functionality using the open-drain capablity.

Example

SOURCE FILE

```
USE 'dfeature';

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

"  Declare the two busses 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 busses
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);

"  Finally, 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 };
```

Specifying JEDEC Filenames

MACHXL places JEDEC files in the design directory, using names in the form *design_name.jn*. To specify a name for each JEDEC file you can use the FUSEMAP_FILE property in the *.pi* file. The FUSEMAP_FILE property is only allowed within a DEVICE construct.

Syntax

```
{ FUSEMAP_FILE  ' filename ' } ;
```

Example

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

AMD MACH

MACH devices are handled like any other PLD in MACHXL with full support for automatic device selection and partitioning. There are some details involved in using MACH parts that can improve utilization and help device-specific implementation issues. An overview of these issues is given in this section. For more details on targeting MACH devices, please see **Appendix D**.

MACH Pin Numbering

In the MACH family, there are six types of pins and internal nodes which may be assigned signals. They are:

- Input pins
- Input-clock pins
- Biput pins
- Shadow pins
- Buried pins
- Unary pins

For physical pins, inputs, clock-inputs and I/O pins, the MACHXL reference is identical to the device pin number. The internal nodes, called buried, shadow, and unary pins, are referenced by node numbers.

Buried and shadow pins are hidden (cannot be seen outside the device) and can be used to hold functions which are only used within the device. A buried pin is a macrocell within the device which cannot be connected to an I/O pin. A shadow pin is the internal part of an enabled output. It is simply the macrocell and its internal feedback path. Using a shadow pin rather than a biput pin allows the physical pin and its pin feedback path to be used as an input. For more information on buried, shadow, and unary pins (nodes), see the earlier sections in this chapter on ***Hidden Nodes*** , ***Shadow Nodes,*** and ***Unary Nodes***.

The macrocells 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 may go in the same order as the neighboring physical pin numbers or in the reverse order.

A buried node is specified with the label BURIED_OF_xx, where xx is the manufacturer-specified pin number in the primary package, which is JLCC for the MACH family.

A shadow node is specified with the label SHADOW_OF_xx, where xx is the manufacturer-specified pin number in the primary package, which is JLCC for the MACH family.

A unary node is specified with the label UNARY_OF_xx, where xx is the manufacturer-specified pin number in the primary package, which is JLCC for the MACH family.

These labels (used in the *.pi* file) are enumerated in the application note entitled ***Complete List of MACH Pin Names*** in **Appendix D**. This application note also contains the numbering for buried, shadow, and unary pins, as well as pin/node numbering.

The label names have the following meanings:

node label	Description
BURIED_OF_xx	Buried node associated with pin xx on the device
SHADOW_OF_xx	Shadow node associated with pin xx on the device
UNARY_OF_xx	Unary nodes associated with pin xx on the device

Using the .pi File with MACH Devices

The *.pi* (Physical Information) file allows specifying details about implementing a design in a MACH family device.

Properties and Device Utilization

The MACH_UTILIZATION property allows specifying 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.

1. Resources may be reserved to allow for logic expansion.

2. Resources may be reserved to ease and speed the fitting process. 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 (e.g., earlier in the design cycle), set the utilization factor to a lower value.

Equation Optimization

The MAX _PTERMS property provides a means of tuning the optimization to best fit a design into MACH parts. The optimization process collapses combinatorial nodes in the design up to a size specified by MAX_PTERMS. The value used for this property affects fitting into MACH parts. 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. At the other end, 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.

For more information on the use and syntax of the MAX_PTERMS property, see **Chapter 12**.

The minimum and maximum number of pterms along with a suggested value for the MAX_PTERMS value are shown in in the following table.

Family	Minimum Number of Pterms per Output	Maximum Number of Pterms per Output	Suggested Number for MAX_PTERMS
MACH 1XX	4	12	8
MACH 2XX	4	16	8 to 12*
MACH 4XX	5	20	10 to 15

* varies with the design.

For optimal fitting, you should try a number of values to determine the best value for a given design.

Note: Any optimization property (for example MAX_PTERMS or MAX_SYMBOLS) may be used in GROUPs, SECTIONS, or with any individual signals. For more information on the optimization properties, see Chapter 13.

Targeting PAL Blocks

You can specify which nodes, outputs and biputs are to be placed together in the same PAL block of a MACH device. Although in the 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.

With the MACH family, there are two ways to specify a group of signals be placed together in the same PAL block:

- GROUP specifications in the *.pi* file
- SECTION specifications in the *.pi* file.

Using GROUPs with MACH

A GROUP specification inside a DEVICE targeted to a MACH device will place all of the signals inside the GROUP into the same PAL block. Other GROUPs inside the DEVICE may or may not also be fit into that same PAL block.

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
      END GROUP;          "the same PAL block

      GROUP
            ogroup2;      "all ogroup2 signals may or may
      END GROUP;          "not also go into ogroup1's PAL
                          "block
END DEVICE;
```

Using SECTIONs with MACH

A SECTION specification inside a DEVICE targeted to a MACH device will place all of the signals inside the SECTION into the same PAL block. Signals from one SECTION will not be placed into the PAL block of another SECTION.

You can specify which PAL block a SECTION should be placed into with the TARGET construct. If a SECTION isn't targeted to a PAL block, MACHXL will determine the best PAL block for the SECTION automatically.

Syntax

```
TARGET 'pal_block_name';
```

The following table lists the names of the PAL blocks for the MACH family:

Template	PAL Block Names
MACH110	A..B
MACH120	A..D
MACH130	A..D
MACH210	A..D
MACH215	A..D
MACH220	A..H
MACH230	A..H
MACH435	A..H
MACH465	A..P

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;
```

Using FLOAT_NODES with MACH Devices

When refitting a design into a MACH device, the designer often will want to preserve the same pinout as in the original fit. However, the internal node assignments do not necessarily need to be maintained. The FLOAT_NODES property assists in the task of refitting a design in a MACH device.

The FLOAT_NODES property causes the MACH fitters to interpret a node assignment in the *.pi* file as specifying only to which PAL block the signal goes. The node assignment is released, allowing the nodes to float in the PAL block indicated by the node assignment. This often gives the MACH fitters the latitude required to successfully refit the design with a fixed pinout.

Syntax

{ FLOAT_NODES } ;

Example

SOURCE FILE

```
INPUT I1;
INPUT clk, oe;
NODE n1..n2 CLOCKED_BY clk;
OUTPUT O1 ENABLED_BY oe;
```

```
n1 - I1;
n2 = n1;
o1 = n2;
```

PHYSICAL INFORMATION FILE

```
{FLOAT_NODES};

DEVICE
TARGET 'PART_NUMBER AMD MACH110-20/BXA';

o1:2;
INPUT clk:13;
INPUT oe:32;
INPUT I1:33;
ni:SHADOW_OF_16;  "ni will be fit in PAL block A but not
                  "necessarily on node SHADOW_OF_16
END DEVICE;
```

In the example shown above, the *.pi* file was created by copying the *.npi* file created by MACHXL to a *.pi* file, then adding the FLOAT_NODES property (some constructs normally found in a *.npi* file have been eliminated for clarity). The FLOAT_NODES property is given globally, and will apply to all MACH devices in the *.pi* file.

Accessing the MACH Internal Feedback Path

In the MACH devices, outputs without an output enable can be fed back into the device through two paths:

1. Directly from the pin
2. Directly from the macrocell

These paths are functionally equivalent, but the pin feedback may be slightly slower than macrocell feedback.

By default, MACHXL will route signals using the pin-feedback path. To use the macrocell-feedback path, attach the FORCE_INTERNAL_FB property to

the appropriate signal in the *.pi* file. To use this feedback on all signals in the device, include the FORCE_INTERNAL_FB property in the DEVICE specification.

Example

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 internal
                                    "feedback
      default;
END DEVICE;
```

Configuring the MACH 445 and MACH 465 Devices 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 has the effect of increasing the set-up time for the input registers and reducing the hold time for these registers to zero.

To set the hold time on the input registers, use the MACH_ZERO_HOLD_INPUT property in the DEVICE section of the *.pi* file.

```
DEVICE
      TARGET 'PART_NUMBER AMD MACH465-15KC';
      {MACH_ZERO_HOLD_INPUT}; "Set all input registers to
                              "zero hold time
      default;
END DEVICE;
```

If the MACH_ZERO_HOLD_INPUT property is assigned to a device, all of the input registers in the device will be configured for zero-hold time.

Accessing the MACH 445 and MACH 465 Signature Bits

The MACH 445 and MACH 465 devices have a 32-bit field to hold user data. This field is called the Signature Bits (or USERCODE) field.

To place data in this field, use the SIGNATURE property in the DEVICE section of the *.pi* file.

Example
PHYSICAL INFORMATION FILE

```
DEVICE
      TARGET 'PART_NUMBER AMD MACH465-15KC';
      {SIGNATURE 'test'};
      default;
END DEVICE;
```

The argument for this property may be a string or an integer. If a string is used, up to four characters can be placed in the field. Alternately, any 32-bit signed integer can be placed in the field.

The MACH *.rpt* File

The MACH fitter has the capability to write a complete description of a fitted device showing resource utilization, all signal and routing information and full placement details including internal nodes.

The *.rpt* file is produced for MACH devices fit in the directed partitioning mode, that is, as a result of a DEVICE construct in the *.pi* file. The *.rpt* file is not produced during automatic device selection and partitioning.

If you have a solution generated by automatic partitioning and need an *.rpt* file, move the *.npi* file to *design.pi* and rerun the fitter. This should run quite quickly and will produce *.rpt* files for any MACH devices in the solution.

If you are fitting a design into a known set of MACH parts, and want a *.rpt* file on the first pass, put empty DEVICE constructs into the *.pi* file. This forces an *.rpt* file while allowing the fitter the freedom to partition the design. The following partitions a design into two MACH110's and produces a *.rpt* for each device.

```
DEVICE
      TARGET 'PART_NUMBER AMD MACH110-15JC';
END DEVICE ;
DEVICE
      TARGET 'PART_NUMBER AMD MACH110-15JC';
END DEVICE ;
```

Note: Detailed MACH-specific information can be found in Appendix D.

The MACH LOW_POWER Attribute

All MACHxx1 (i.e., MACH111, MACH231, etc.) devices have a low-power attribute that can be applied at the macrocell level. The attribute sets the macrocell for a given signal to the low power consumption mode. This can be applied globally for all signals in a device, or locally (in a DEVICE statement) for only those signals specified.

```
{LOW_POWER}        global declaration for all signals in
                   a device

DEVICE
      TARGET signal_1, signal_2 {LOW_POWER};
END DEVICE;        affects only signal_1 and signal_2
```

16 Programming and Testing Devices

Contents

Introduction

When selecting a PLD or CPLD device solution, MACHXL generates an output file containing the information necessary to program devices using a device programmer over an RS-232C communications port. This chapter discusses ways of downloading fusemaps to your device programmer, programming your devices, and testing your devices.

Programming PLDs or CPLDs

MACHXL creates device fusemap files containing all the information required by a device programmer to program the devices. The device fusemap files must be downloaded from your computer system to the device programmer before a device can be programmed.

Downloading Fusemaps

Downloading the fusemap file from a system to a programmer is done using the device programmer's communication software. This software can be executed from MACHXL's menu system. To use your device programmer's software from the MACHXL menu, select DOWNLOAD. (See Chapter 3 for more information on this menu selection.)

Using Your Device Programmer's Downloading Software

To download the device fusemaps you must run your device programmer's communication software. This software will require the name of the device fusemap file. MACHXL creates the following output files for PLD and CPLD devices:

Format	Output Filename	Device Type
JEDEC	*filename*. j1 .. jn	PALs and GALs

Where:

filename is the name of the file containing your design.

.j1 Is the extension for a JEDEC file, with the number 1 corresponding to the first device in the solution. For a counter design with three devices in its solution, the names of the output files would be: *counter.j1*, *counter.j2*, and *counter.j3*.

Connecting Your Computer System to a Device Programmer

You should use the cable recommended by the maker of your device programmer when connecting the programmer to your computer system. Refer to your device programmer's documentation for instructions on connecting the cable from your computer system to the device programmer.

Testing Devices

Test vectors are produced by the simulator and are part of the device fusemap file. The test vectors are produced only if there is a .stm file containing a SYSTEM_TEST. Refer to your device programmer's documentation for more information on how to test your device using the test vectors.

17 Documenting a Design

Contents

Introduction

MACHXL documents a design during the various stages of compilation and partitioning. All this information about a design is contained in the file *design_name.doc*. The following information is contained in the *.doc* file:

Information about the design (title, designer, date, company, etc) and switch values (options) specified for compiler and optimizer functions

- Reduced design equations
- 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

These sections of the documentation file are described in the following sections.

Title Page

The title page gives the header information optionally specified for a design, including the title, engineer, company, project, revision number, and comments about the design. The date and filename are automatically generated and included in the beginning of the documentation. The individual

MACHXL module revision numbers are also provided.

The title page also includes information about switch values specified for the compiler and optimizer.

Switch Values (options)

This section of the .doc file indicates the switch values used by the compiler and optimizer. These reduction and optimization values assure the same equation output on subsequent passes of the front end. These switch values indicate levels for:

- Compiler reduction
- Optimizer reduction
- Optimizer node generation.

For more information on the switch values available for the compiler and optimizer, see Chapters 10 and 12 respectively.

Reduced Design Equations

How Equations are Generated

When a source file is compiled and optimized, MACHXL takes the user-specified equations and synthesizes additional equations from them. For example, if specifying a JK-flip flop as part of the design, the compiler generates equations for all other flip flop types as well. These synthesized equations are simply logically-equivalent versions of the flip flop specified. These additional equations give the partitioning and fitting software more options with which to fit your design. This means that the *.doc* file may contain equations in addition to those supplied by you in the design source file.

Equation Extensions Used in the .doc File

The following tables list equation types and the equation extension used in the *.doc* file:

.*doc* File Extension	Description	Example
* .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

* The compiler/optimizer may generate an XOR equation even if none was specified in the original .src file. Examples include synthesis from T flops, arithmetic operators .+. and .-., etc. For more information see chapters 10 and 12.

.*doc* File Extension	Description	Example
.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
.CLK	clock equation OUTPUT x CLOCKED_BY /a	X.CLK = /A
.RESET	reset equation OUTPUT x CLOCKED_BY /a RESET_BY rst	X.RESET = RST

.doc File Extension	Description	Example
PRESET	preset equation OUTPUT x CLOCKED_BY /a PRESET_BY prst	X.PRESET = PRST
.OE	enabled equation OUTPUT x ENABLED_BY oe	X.OE = OE
.LATCH	latched equation OUTPUT X LATCHED_BY lat1	X.LATCH = LAT1
.CE	clock-enabled equation	X.CE = CE

DeMorgan Equations

In addition to the equations listed in the previous table, the compiler/optimizer may generate DeMorgan versions of the same equations.

> *Note: There are cases when the non-complemented version of an equation can NOT be generated by the compiler/optimizer, due to the size of the equation, but the complemented (DeMorgan) version can.*

The DeMorgan equivalent of the original or synthesized equations may be fit into a device. In the *.doc* file, a tilde (~) after an equation name, such as *OR1.EQN*(~), indicates the DeMorgan version of that equation.

For example, an equation declared with the following specifications:

```
LOW_TRUE INPUT oe;
INPUT a, b;
OUTPUT or1 ENABLED_BY oe;
or1 = a + b;
```

After the system equations are created, the *.doc* file equations are:

```
OR1.EQN       = A + B;
      .OE     = OE;
OR1.EQN(~)    = /A * /B;
      .OE(~)  = /OE;
```

Equation Display

There are four equation categories displayed in the *.doc* file. They are:

1. **Primary** - equations used to describe the signal
2. **Synthesized** - equations generated by the compiler/optimizer
3. **DeMorgan** - complemented equations generated by the compiler/optimizer
4. **Fit** - form of the equations (primary, synthesized, or the DeMorgan of the two) that was actually fit into the device

The documentor can display any (or all) of these equation categories independently. By default, the documentor will:

- Display the version of the equation that was used during Fitting
- Display the primary equation version if Fitting has not yet been done

You can also access the *.doc* file by means of the menuing system. For more information on using the menus, see Chapter 3.

Partitioning Criteria

A copy of the *.cst* (cost) file, used to specify the partitioning constraints, is placed in the *.doc* file for reference.

A warning may appear in the *.doc* file to indicate the *.cst* file used during partitioning was updated since the solutions were generated. indicates the partitioning criteria displayed in the *.doc* file may be incorrect.

This section appears in the *.doc* file only after the device scanner is run.

Solutions List

A copy of the solutions generated for a design are placed in the documentation file for quick reference. Another solution may be selected for a design by using the Solutions menu item from the Device Menu.

This section appears in the *.doc* file only after the device fitter is run.

Fusemap Files

This section indicates which fusemaps go to which device for a particular solution. This section will appear in the *.doc* file only after the fuse mapper is run.

Pinout Diagrams

Partitioning a design produces a pinout diagram (DIP or CDIP packages) or a pinout table (all other packages) shows the device , the pin types (i.e., INPUT, OUTPUT, BIPUT), and an indicator of the signal/pin placement.

Possible Devices List

When device solutions are generated, the solution list contains device template names, not manufacturers' names for devices. The design documentation displays actual devices to select for the device templates used in a solution.

Wire List

A wire list shows which signals to connect to which pins for the device solution selected.

Viewing the Documentation

For specific information on viewing the documentation file from within MACHXL, see **Chapter 3**. You may also view the file outside of MACHXL by pulling the file *filename.doc* into a text editor.

18 MACHXL Design Tips

Contents

What If Equations Are Too Large?

Equations with too many product terms or input symbols for the target devices won't fit into the devices. Very large equations may cause a fatal error from the compiler. There are several good design practices that help avoid large equations:

- Declare NODEs and use them to break up large equations or hold intermediate values shared by other equations. These NODEs give the optimizer more flexibility to do its job. It removes nodes if their removal doesn't make equations too large. See Chapter 5 for more on declaring NODEs.

- Use PROCEDUREs and FUNCTIONs to implement portions of the design logically separate or for functionality repeated in more than one place. Using PROCEDUREs and FUNCTIONs automatically introduces NODEs that help the optimizer do its job. See Chapter 5 for more on declaring NODEs. See Chapter 8 for more information on Procedures and Functions.

- If a STATE_MACHINE is used then the values assigned to each state can affect the size of equations. The STATE_VALUES ONE_HOT value assignment method produces small equations at the cost of using more registers. (However, ONE_HOT state machines do cause large intermediate equations prior to optimization.) See Chapter 7 for more on STATE_MACHINEs.

There are several *.pi* properties used by the optimizer which affect the sizes of equations produced by the optimizer. See Chapter 12 for more on the optimizer.

What If MACHXL Runs Out of Memory?

The minimum recommended memory configuration on a PC is 8 Megabytes. Large designs targeting the larger more complex devices may require more than 8 Megabytes.

In the Compiler

If this error occurs in the compiler it is probably due to an equation growing too large to be represented in the available memory. See the previous question ***What If Equations Are Too Large?*** for information on controlling equation size.

Large state machines, and especially ONE_HOT state machines, use lots of memory since they produce large intermediate equations. Again, use of NODEs to hold intermediate values of conditional expressions in each state simplifies the resulting equations.

In the Optimizer

If this error occurs in the optimizer then it is probably due to the MAX_PTERMs property in the *.pi* file being too large. The default is 16. Memory problems can crop up when MAX_PTERMs is set in the hundreds for targeting devices handling very large equations.

In the Fitter

If this error occurs in the fitter, the combination of the size of the design and the number of templates considered is too large. If some solutions have already been found then the fitter recovers gracefully allowing one of the existing solutions to be selected.

To avoid this error and allow the fitter to consider all possible solutions across the available templates, use the template constraint menu to pare away those templates not appropriate for the design. Very large designs may need the templates pared down to just a few.

What Can Be Done to Speed Things Up?

In the Compiler and Optimizer

Slow performance of the compiler and the optimizer is usually due to very large equations. See the previous question ***What If Equations Are Too Large?*** for information on controlling equation size.

In the Fitter

The most important first step is restricting the number of device architectures the fitter considers by specifying all device constraints. Prioritize according to size if appropriate. Pay attention to the physical constraints in the constraints menu, and the templates menu as well. Avoid including complex device architectures, if they are not needed, in a large solution search. See Chapter 3 for information on controlling constraints.

The *.pi* file is used to direct partitioning. If particular devices are needed as part of the solution, specify *.pi* file signal groups to assist the fitter in partitioning the design. See Chapter 15 for information on directed fitting.

What Can Be Done to Minimize the Amount of Hardware Needed to Implement a Design?

This is one of the goals of any design and depends on the design. However several capabilities of MACHXL software can help.

In the Design Files

It is important the equations being fitted are appropriate for the target hardware. See the previous question ***What If Equations Are Too Large?*** for information on controlling equation size.

The language allows the designer to explicitly control whether NODEs remain or are removed by the optimizer by declaring them to be PHYSICAL or VIRTUAL. See **Chapter 12, *Optimizing a Design*** for more on PHYSICAL and VIRTUAL NODEs.

In the Fitting Constraints

The fitter finds good solutions as long as it is allowed to search for solutions. Avoid turning synthesis options off, such as Auto-Demorganization. Avoid restricting the set of architectures considered by the fitter to those familiar when there may be better, less familiar devices for implementing a design that the fitter will find if given a chance. A good strategy is first allowing the fitter to do a wide-open, extensive partitioning search across a complete set of device architectures. For large designs, let this search run overnight or over a weekend. Then, pick a solution and, if needed, fine-tune the solution by moving the *.npi* file to the *.pi* file and modifying this new *.pi* file as needed. See Chapter 15 for information on directed fitting.

.cst File and Fitter Speed

By creating a *.cst* file containing only the template of interest (see below), you can save time during the execution of the fitter.

Example *.cst* file

TEMPLATE = MACH210;

Similar to the default *.pi* above, this information can be placed into a default constraint file (*default.cst*) which will assure its use each time a new design is fit.

A MACHXL Supported Devices

Contents

Introduction

This appendix has 8 sections. They are as follows:

- AMD PLD Design Module
- AMD MACH Design Module
- Devices listed by template number
- Device footprints
- New (added) devices (Version 3.0)
- Renamed devices (Version 3.0)
- Obsolete devices (Version 3.0)
- Deleted devices (Version 3.0)

AMD PLD Design Module

Each column is made up of the AMD abbreviation, and the AMD device part number, which are grouped under the template name (in **bold** print).

This list is used to generate the appropriate TARGET statement to specify a particular device in the *.pi* file.

Syntax

```
TARGET 'PART_NUMBER  manufacturer_abbreviation
device_part_number';
```

Example

```
TARGET 'PART_NUMBER  AMD MACH110-12JC';
```

AMD	PAL16L8-4JC
AMD	PAL16L8-5JC
AMD	PAL16L8-5PC
AMD	PAL16L8-7DC
AMD	PAL16L8-7JC
AMD	PAL16L8-7PC
AMD	PAL16L8A2CN
AMD	PAL16L8A2CNL
AMD	PAL16L8ACN
AMD	PAL16L8ACNL
AMD	PAL16L8B2CN
AMD	PAL16L8B2CNL
AMD	PAL16L8B4CJ
AMD	PAL16L8B4CN
AMD	PAL16L8B4CNL
AMD	PAL16L8BCN
AMD	PAL16L8BCNL
AMD	PAL16L8D/2JC
AMD	PAL16L8D/2PC

P16R4

AMD	5962-85155042A
AMD	5962-8515504RA
AMD	5962-85155082A
AMD	5962-8515508RA
AMD	5962-88515042A
AMD	5962-8851504RA
AMD	81036102A
AMD	8103610RA
AMD	81036142A
AMD	8103614RA
AMD	PAL16R4-4JC
AMD	PAL16R4-5JC
AMD	PAL16R4-5PC
AMD	PAL16R4-7DC
AMD	PAL16R4-7JC
AMD	PAL16R4-7PC
AMD	PAL16R4A2CN
AMD	PAL16R4A2CNL
AMD	PAL16R4ACN
AMD	PAL16R4ACNL
AMD	PAL16R4B2CN
AMD	PAL16R4B2CNL
AMD	PAL16R4B4CJ
AMD	PAL16R4B4CN
AMD	PAL16R4B4CNL
AMD	PAL16R4BCN
AMD	PAL16R4BCNL
AMD	PAL16R4CN
AMD	PAL16R4CNL
AMD	PAL16R6-4JC
AMD	PAL16R6-5JC
AMD	PAL16R6-5PC
AMD	PAL16R6-7DC
AMD	PAL16R6-7JC
AMD	PAL16R6-7PC
AMD	PAL16R6A2CN
AMD	PAL16R6A2CNL
AMD	PAL16R6ACN
AMD	PAL16R6ACNL
AMD	PAL16R6B2CN
AMD	PAL16R6B2CNL
AMD	PAL16R6B4CJ
AMD	PAL16R6B4CN
AMD	PAL16R6B4CNL
AMD	PAL16R6BCN
AMD	PAL16R6BCNL
AMD	PAL16R6CN
AMD	PAL16R6CNL
AMD	PAL16R6D/2PC

P16R8

AMD	PAL16R8-4JC
AMD	PAL16R8-5JC
AMD	PAL16R8-5PC
AMD	PAL16R8-7DC
AMD	PAL16R8-7JC
AMD	PAL16R8-7PC
AMD	PAL16R8A2CN
AMD	PAL16R8A2CNL
AMD	PAL16R8ACN
AMD	PAL16R8ACNL
AMD	PAL16R8B2CN
AMD	PAL16R8B2CNL

AMD	PAL16R8B4CJ
AMD	PAL16R8B4CN
AMD	PAL16R8B4CNL
AMD	PAL16R8BCN
AMD	PAL16R8BCNL
AMD	PAL16R8CN
AMD	PAL16R8CNL
AMD	PAL16R8D/2JC
AMD	PAL16R8D/2PC
P16V8A	
AMD	PALCE16V8H-10JC/4
AMD	PALCE16V8H-10PC/4
AMD	PALCE16V8H-10SC/4
AMD	PALCE16V8H-15JC/4
AMD	PALCE16V8H-15PC/4
AMD	PALCE16V8H-15SC/4
AMD	PALCE16V8H-25JC/4
AMD	PALCE16V8H-25PC/4
AMD	PALCE16V8H-5JC/5
AMD	PALCE16V8H-7JC/5
AMD	PALCE16V8H-7PC/5
AMD	PALCE16V8Q-10JC/5
AMD	PALCE16V8Q-15JC/4
AMD	PALCE16V8Q-15PC/4
AMD	PALCE16V8Q-25JC/4
AMD	PALCE16V8Q-25PC/4
AMD	PALCE16V8Z-15JI
AMD	PALCE16V8Z-15PI
AMD	PALCE16V8Z-25JC
AMD	PALCE16V8Z-25JI
AMD	PALCE16V8Z-25PC
AMD	PALCE16V8Z-25PI
AMD	PALLV16V8-10JC
AMD	PALLV16V8-10PC
AMD	PALLV16V8Z-20JI
AMD	PALLV16V8Z-20PI
P16V8HD	
AMD	PALCE16V8HD-15JC
AMD	PALCE16V8HD-15PC
P18P8	
AMD	AMPAL18P8ADC
AMD	AMPAL18P8AJC
AMD	AMPAL18P8ALJC
AMD	AMPAL18P8ALPC
AMD	AMPAL18P8APC
AMD	AMPAL18P8BJC
AMD	AMPAL18P8BPC
AMD	AMPAL18P8LJC
AMD	AMPAL18P8LPC
P20L8	
AMD	PAL20L8-10/2JC
AMD	PAL20L8-10/2PC
AMD	PAL20L8-5JC
AMD	PAL20L8-5PC
AMD	PAL20L8-7JC
AMD	PAL20L8-7PC
AMD	PAL20L8A2CNL
AMD	PAL20L8A2CNS
AMD	PAL20L8ACNL
AMD	PAL20L8ACNS
AMD	PAL20L8B2CFN
AMD	PAL20L8B2CNL
AMD	PAL20L8B2CNS
AMD	PAL20L8BCFN
AMD	PAL20L8BCNL
AMD	PAL20L8BCNS
P20R4	
AMD	PAL20R4-5JC
AMD	PAL20R4-5PC
AMD	PAL20R4-7DC
AMD	PAL20R4-7JC
AMD	PAL20R4-7PC
AMD	PAL20R4A2CNL
AMD	PAL20R4A2CNS
AMD	PAL20R4ACNL
AMD	PAL20R4ACNS
AMD	PAL20R4B2CFN
AMD	PAL20R4B2CNL
AMD	PAL20R4B2CNS
AMD	PAL20R4BCNL
AMD	PAL20R4BCNS
P20R6	
AMD	PAL20R6-5JC
AMD	PAL20R6-5PC

AMD	PAL20R6-7DC
AMD	PAL20R6-7JC
AMD	PAL20R6-7PC
AMD	PAL20R6A2CNL
AMD	PAL20R6A2CNS
AMD	PAL20R6ACNL
AMD	PAL20R6ACNS
AMD	PAL20R6B2CFN
AMD	PAL20R6B2CNS
AMD	PAL20R6BCNL
AMD	PAL20R6BCNS
P20R8	
AMD	PAL20R8-10/2PC
AMD	PAL20R8-5JC
AMD	PAL20R8-5PC
AMD	PAL20R8-7DC
AMD	PAL20R8-7JC
AMD	PAL20R8-7PC
AMD	PAL20R8A2CNL
AMD	PAL20R8A2CNS
AMD	PAL20R8ACNL
AMD	PAL20R8ACNS
AMD	PAL20R8B2CFN
AMD	PAL20R8B2CNL
AMD	PAL20R8B2CNS
AMD	PAL20R8BCNL
AMD	PAL20R8BCNS
P20RA10	
AMD	PAL20RA10-20CFN
AMD	PALCE20RA10H-10JC
AMD	PALCE20RA10H-10JI
AMD	PALCE20RA10H-10PC
AMD	PALCE20RA10H-10PI
AMD	PALCE20RA10H-15JC
AMD	PALCE20RA10H-15JI
AMD	PALCE20RA10H-15PC
AMD	PALCE20RA10H-15PI
AMD	PALCE20RA10H-20PC
AMD	PALCE20RA10H-7JC
AMD	PALCE20RA10H-7JI
P20V8A	
AMD	PALCE20V8H-10JC/4
AMD	PALCE20V8H-10PC/4
AMD	PALCE20V8H-15JC/4
AMD	PALCE20V8H-15JI/4
AMD	PALCE20V8H-15PC/4
AMD	PALCE20V8H-25JC/4
AMD	PALCE20V8H-25JI/4
AMD	PALCE20V8H-25PC/4
AMD	PALCE20V8H-5JC/5
AMD	PALCE20V8H-7JC/5
AMD	PALCE20V8H-7PC/5
AMD	PALCE20V8Q-15JC/4
AMD	PALCE20V8Q-15PC/4
AMD	PALCE20V8Q-20PI/4
AMD	PALCE20V8Q-25JC/4
AMD	PALCE20V8Q-25PC/4
P22P10	
AMD	AMPAL22P10AJC
AMD	AMPAL22P10ALDC
AMD	AMPAL22P10ALPC
AMD	AMPAL22P10BJC
AMD	AMPAL22P10BPC
P22V10	
AMD	AMPAL22V10AJC
AMD	AMPAL22V10APC
AMD	AMPAL22V10JC
AMD	AMPAL22V10PC
AMD	CE22V10H-15E4/BKA
AMD	PAL22V10-10JC
AMD	PAL22V10-10PC
AMD	PAL22V10-15DC
AMD	PAL22V10-15JC
AMD	PAL22V10-15PC
AMD	PALCE22V10H-10JC/5
AMD	PALCE22V10H-10PC/5
AMD	PALCE22V10H-15JC/4
AMD	PALCE22V10H-15PC/4
AMD	PALCE22V10H-15SC/4
AMD	PALCE22V10H-25JC/4
AMD	PALCE22V10H-25PC/4
AMD	PALCE22V10H-25SC/4

AMD	PALCE22V10H-5JC/5
AMD	PALCE22V10H-7JC/5
AMD	PALCE22V10H-7PC/5
AMD	PALCE22V10Q-10JC
AMD	PALCE22V10Q-10PC
AMD	PALCE22V10Q-10SC
AMD	PALCE22V10Q-15JC
AMD	PALCE22V10Q-15PC
AMD	PALCE22V10Q-25JC/4
AMD	PALCE22V10Q-25PC/4
AMD	PALCE22V10Z-15JI
AMD	PALCE22V10Z-15PI
AMD	PALCE22V10Z-15SI
AMD	PALCE22V10Z-25JC
AMD	PALCE22V10Z-25JI
AMD	PALCE22V10Z-25PC
AMD	PALCE22V10Z-25PI
AMD	PALCE22V10Z-25SC
AMD	PALCE22V10Z-25SI
AMD	PALLV22V10-10PC
AMD	PALLV22V10-7JC
AMD	PALLV22V10Z-25JI
AMD	PALLV22V10Z-25PI
AMD	PALLV22V10Z-25SI

P24V10

AMD	PALCE24V10H-15JC
AMD	PALCE24V10H-15PC
AMD	PALCE24V10H-25JC
AMD	PALCE24V10H-25PC

P26V12

AMD	PALCE26V12H-15JC/4
AMD	PALCE26V12H-15PC/4
AMD	PALCE26V12H-20JC/4
AMD	PALCE26V12H-20PC/4

P29M16

AMD	PALCE29M16H25JC/4
AMD	PALCE29M16H25PC/4

P29MA16

AMD	PALCE29MA16H25JC/4
AMD	PALCE29MA16H25PC/4

P600

AMD	PALCE610H-15JC
AMD	PALCE610H-15PC
AMD	PALCE610H-20/B3A
AMD	PALCE610H-20/BLA
AMD	PALCE610H-25JC
AMD	PALCE610H-25PC

AMD MACH Design Module

The device templates (i.e., architectures) listed below are supported by this Design Module.

MACH110	MACH220
MACH111	MACH230
MACH120	MACH231
MACH130	MACH355
MACH131	MACH435
MACH210	MACH445
MACH211	MACH465
MACH215	

Each column is made up of the AMD abbreviation, and the AMD device part number, which are grouped under the template name (in **bold** print).

This list is used to generate the appropriate TARGET statement to specify a particular device in the *.pi* file.

Syntax

```
TARGET 'PART_NUMBER  manufacturer_abbreviation
device_part_number';
```

Example

```
TARGET 'PART_NUMBER  AMD MACH110-12JC';
```

MACH110

AMD	MACH110-12JC
AMD	MACH110-14JI
AMD	MACH110-15JC
AMD	MACH110-18JI
AMD	MACH110-20JC
AMD	MACH110-24JI

MACH111

AMD	MACH111-10JC
AMD	MACH111-12JC
AMD	MACH111-15JC
AMD	MACH111-20JC
AMD	MACH111-7JC

MACH120

AMD	MACH120-12JC
AMD	MACH120-15JC
AMD	MACH120-18JI
AMD	MACH120-20JC
AMD	MACH120-24JI

MACH130

AMD	MACH130-15JC
AMD	MACH130-18JI
AMD	MACH130-20JC
AMD	MACH130-24JI

MACH131

AMD	MACH131-10JC
AMD	MACH131-12JC
AMD	MACH131-15JC
AMD	MACH131-20JC
AMD	MACH131-7JC

MACH210

AMD	MACH210-12JC
AMD	MACH210-14JI
AMD	MACH210-15JC
AMD	MACH210-18JI
AMD	MACH210-20JC
AMD	MACH210-24JI
AMD	MACH210A-10JC
AMD	MACH210A-10VC
AMD	MACH210A-12JC
AMD	MACH210A-12JI
AMD	MACH210A-12VC
AMD	MACH210A-15VC
AMD	MACH210A-20VC
AMD	MACH210A-7JC
AMD	MACH210AQ-12JC
AMD	MACH210AQ-15JC
AMD	MACH210AQ-18JI
AMD	MACH210AQ-20JC
AMD	MACH210AQ-24JI
AMD	MACHLV210-15JC
AMD	MACHLV210-18JI
AMD	MACHLV210-20JC
AMD	MACHLV210-24JI

MACH211

AMD	MACH211-12JC
AMD	MACH211-14JI
AMD	MACH211-15JC
AMD	MACH211-18JI
AMD	MACH211-20JC
AMD	MACH211-24JI
AMD	MACH211A-10JC
AMD	MACH211A-10VC
AMD	MACH211A-12JC
AMD	MACH211A-12JI
AMD	MACH211A-12VC
AMD	MACH211A-15VC
AMD	MACH211A-20VC
AMD	MACH211A-7JC
AMD	MACH211AQ-12JC
AMD	MACH211AQ-15JC
AMD	MACH211AQ-18JI
AMD	MACH211AQ-20JC
AMD	MACH211AQ-24JI
AMD	MACHLV211-15JC
AMD	MACHLV211-18JI
AMD	MACHLV211-20JC
AMD	MACHLV211-24JI

MACH215

AMD	MACH215-12JC
AMD	MACH215-14JI
AMD	MACH215-15JC
AMD	MACH215-18JI
AMD	MACH215-20JC

AMD	MACH215-24JI

MACH220

AMD	MACH220-10JC
AMD	MACH220-12JC
AMD	MACH220-15JC
AMD	MACH220-18JI
AMD	MACH220-20JC
AMD	MACH220-24JI

MACH230

AMD	MACH230-10JC
AMD	MACH230-15JC
AMD	MACH230-18JI
AMD	MACH230-20JC
AMD	MACH230-24JI

MACH231

AMD	MACH231-10JC
AMD	MACH231-12JC
AMD	MACH231-15JC
AMD	MACH231-20JC
AMD	MACH231-7JC

MACH355

AMD	MACH355-15YC
AMD	MACH355-20YC

MACH435

AMD	MACH435-12JC
AMD	MACH435-15JC
AMD	MACH435-20JC
AMD	MACH435Q-20JC
AMD	MACH435Q-25JC

MACH445

AMD	MACH445-12YC
AMD	MACH445-15YC
AMD	MACH445-20YC

MACH465

AMD	MACH465-12YC
AMD	MACH465-15YC
AMD	MACH465-20YC

Devices Listed By Template Number

This section is a listing of all AMD devices supported by MACHXL. The devices in this list are sorted alphabetically by template number. The columns in the list consist of the manufactures abbreviation, followed by the device's part number, and the footprint name.

This list can be used to generate a TARGET statement in the *.pi* file to specify a particular device.

Syntax

```
TARGET 'TEMPLATE template_name  footprint_name';
```

Example

```
TARGET 'TEMPLATE MACH110 JLCC-44-STD;
```

MACH110

AMD	MACH110-12JC	JLCC-44-STD
AMD	MACH110-14JI	JLCC-44-STD
AMD	MACH110-15JC	JLCC-44-STD
AMD	MACH110-18JI	JLCC-44-STD
AMD	MACH110-20JC	JLCC-44-STD
AMD	MACH110-24JI	JLCC-44-STD

MACH111

AMD	MACH111-10JC	JLCC-44-STD
AMD	MACH111-12JC	JLCC-44-STD
AMD	MACH111-15JC	JLCC-44-STD
AMD	MACH111-20JC	JLCC-44-STD
AMD	MACH111-7JC	JLCC-44-STD

MACH120

AMD	MACH120-12JC	JLCC-68-STD
AMD	MACH120-15JC	JLCC-68-STD
AMD	MACH120-18JI	JLCC-68-STD
AMD	MACH120-20JC	JLCC-68-STD
AMD	MACH120-24JI	JLCC-68-STD

MACH130

AMD	MACH130-15JC	JLCC-84-STD
AMD	MACH130-18JI	JLCC-84-STD
AMD	MACH130-20JC	JLCC-84-STD
AMD	MACH130-24JI	JLCC-84-STD

MACH131

AMD	MACH131-10JC	JLCC-84-STD
AMD	MACH131-12JC	JLCC-84-STD
AMD	MACH131-15JC	JLCC-84-STD
AMD	MACH131-20JC	JLCC-84-STD
AMD	MACH131-7JC	JLCC-84-STD

MACH210

AMD	MACH210-12JC	JLCC-44-STD
AMD	MACH210-14JI	JLCC-44-STD
AMD	MACH210-15JC	JLCC-44-STD
AMD	MACH210-18JI	JLCC-44-STD
AMD	MACH210-20JC	JLCC-44-STD
AMD	MACH210-24JI	JLCC-44-STD
AMD	MACH210A-10JC	JLCC-44-STD
AMD	MACH210A-10VC	TQFP-44-TQ44
AMD	MACH210A-12JC	JLCC-44-STD
AMD	MACH210A-12JI	JLCC-44-STD
AMD	MACH210A-12VC	TQFP-44-TQ44
AMD	MACH210A-15VC	TQFP-44-TQ44
AMD	MACH210A-20VC	TQFP-44-TQ44
AMD	MACH210A-7JC	JLCC-44-STD
AMD	MACH210AQ-12JC	JLCC-44-STD
AMD	MACH210AQ-15JC	JLCC-44-STD
AMD	MACH210AQ-18JI	JLCC-44-STD
AMD	MACH210AQ-20JC	JLCC-44-STD
AMD	MACH210AQ-24JI	JLCC-44-STD
AMD	MACHLV210-15JC	JLCC-44-STD
AMD	MACHLV210-18JI	JLCC-44-STD
AMD	MACHLV210-20JC	JLCC-44-STD
AMD	MACHLV210-24JI	JLCC-44-STD

MACH211

AMD	MACH211-12JC	JLCC-44-STD
AMD	MACH211-14JI	JLCC-44-STD
AMD	MACH211-15JC	JLCC-44-STD
AMD	MACH211-18JI	JLCC-44-STD
AMD	MACH211-20JC	JLCC-44-STD
AMD	MACH211-24JI	JLCC-44-STD
AMD	MACH211A-10JC	JLCC-44-STD
AMD	MACH211A-10VC	TQFP-44-TQ44
AMD	MACH211A-12JC	JLCC-44-STD
AMD	MACH211A-12JI	JLCC-44-STD
AMD	MACH211A-12VC	TQFP-44-TQ44
AMD	MACH211A-15VC	TQFP-44-TQ44
AMD	MACH211A-20VC	TQFP-44-TQ44
AMD	MACH211A-7JC	JLCC-44-STD
AMD	MACH211AQ-12JC	JLCC-44-STD
AMD	MACH211AQ-15JC	JLCC-44-STD
AMD	MACH211AQ-18JI	JLCC-44-STD
AMD	MACH211AQ-20JC	JLCC-44-STD
AMD	MACH211AQ-24JI	JLCC-44-STD
AMD	MACHLV211-15JC	JLCC-44-STD
AMD	MACHLV211-18JI	JLCC-44-STD
AMD	MACHLV211-20JC	JLCC-44-STD
AMD	MACHLV211-24JI	JLCC-44-STD

MACH215

AMD	MACH215-12JC	JLCC-44-STD
AMD	MACH215-14JI	JLCC-44-STD
AMD	MACH215-15JC	JLCC-44-STD
AMD	MACH215-18JI	JLCC-44-STD
AMD	MACH215-20JC	JLCC-44-STD
AMD	MACH215-24JI	JLCC-44-STD

MACH220

AMD	MACH220-10JC	JLCC-68-STD
AMD	MACH220-12JC	JLCC-68-STD
AMD	MACH220-15JC	JLCC-68-STD
AMD	MACH220-18JI	JLCC-68-STD
AMD	MACH220-20JC	JLCC-68-STD
AMD	MACH220-24JI	JLCC-68-STD

MACH230

AMD	MACH230-10JC	JLCC-84-STD
AMD	MACH230-15JC	JLCC-84-STD
AMD	MACH230-18JI	JLCC-84-STD
AMD	MACH230-20JC	JLCC-84-STD
AMD	MACH230-24JI	JLCC-84-STD

MACH231

AMD	MACH231-10JC	JLCC-84-STD
AMD	MACH231-12JC	JLCC-84-STD
AMD	MACH231-15JC	JLCC-84-STD
AMD	MACH231-20JC	JLCC-84-STD
AMD	MACH231-7JC	JLCC-84-STD

MACH355

AMD	MACH355-15YC	QFP-144-STD
AMD	MACH355-20YC	QFP-144-STD

MACH435

AMD	MACH435-12JC	JLCC-84-STD
AMD	MACH435-15JC	JLCC-84-STD
AMD	MACH435-20JC	JLCC-84-STD
AMD	MACH435Q-20JC	JLCC-84-STD
AMD	MACH435Q-25JC	JLCC-84-STD

MACH445

AMD	MACH445-12YC	QFP-100-STD
AMD	MACH445-15YC	QFP-100-STD
AMD	MACH445-20YC	QFP-100-STD

MACH465

AMD	MACH465-15YC	QFP-208-STD
AMD	MACH465-20YC	QFP-208-STD

P16L8

AMD	PAL16L8-4JC	JLCC-28-A28
AMD	PAL16L8-5JC	JLCC-20-STD
AMD	PAL16L8-5PC	DIP-20-STD
AMD	PAL16L8-7DC	DIP-20-STD
AMD	PAL16L8-7JC	JLCC-20-STD
AMD	PAL16L8-7PC	DIP-20-STD
AMD	PAL16L8A2CN	DIP-20-STD
AMD	PAL16L8A2CNL	JLCC-20-STD
AMD	PAL16L8ACN	DIP-20-STD
AMD	PAL16L8ACNL	JLCC-20-STD
AMD	PAL16L8B2CN	DIP-20-STD
AMD	PAL16L8B2CNL	JLCC-20-STD
AMD	PAL16L8B4CJ	DIP-20-STD
AMD	PAL16L8B4CN	DIP-20-STD
AMD	PAL16L8B4CNL	JLCC-20-STD
AMD	PAL16L8BCN	DIP-20-STD
AMD	PAL16L8BCNL	JLCC-20-STD
AMD	PAL16L8D/2JC	JLCC-20-STD
AMD	PAL16L8D/2PC	DIP-20-STD

P16R4

AMD	5962-85155042A	LCC-20-STD
AMD	PAL16R4-5JC	JLCC-20-STD
AMD	PAL16R4-5PC	DIP-20-STD
AMD	PAL16R4-7DC	DIP-20-STD
AMD	PAL16R4-7JC	JLCC-20-STD
AMD	PAL16R4-7PC	DIP-20-STD
AMD	PAL16R4A2CN	DIP-20-STD
AMD	PAL16R4A2CNL	JLCC-20-STD
AMD	PAL16R4ACN	DIP-20-STD
AMD	PAL16R4ACNL	JLCC-20-STD
AMD	PAL16R4B2CN	DIP-20-STD
AMD	PAL16R4B2CNL	JLCC-20-STD
AMD	PAL16R4B4CJ	DIP-20-STD
AMD	PAL16R4B4CN	DIP-20-STD
AMD	PAL16R4B4CNL	JLCC-20-STD
AMD	PAL16R4BCN	DIP-20-STD
AMD	PAL16R4BCNL	JLCC-20-STD
AMD	PAL16R4CN	DIP-20-STD
AMD	PAL16R4CNL	JLCC-20-STD

P16R6

AMD	PAL16R6-4JC	JLCC-28-A28
AMD	PAL16R6-5JC	JLCC-20-STD
AMD	PAL16R6-5PC	DIP-20-STD
AMD	PAL16R6-7DC	DIP-20-STD
AMD	PAL16R6-7JC	JLCC-20-STD
AMD	PAL16R6-7PC	DIP-20-STD
AMD	PAL16R6A2CN	DIP-20-STD
AMD	PAL16R6A2CNL	JLCC-20-STD
AMD	PAL16R6ACN	DIP-20-STD
AMD	PAL16R6ACNL	JLCC-20-STD
AMD	PAL16R6B2CN	DIP-20-STD
AMD	PAL16R6B2CNL	JLCC-20-STD
AMD	PAL16R6B4CJ	DIP-20-STD
AMD	PAL16R6B4CN	DIP-20-STD
AMD	PAL16R6B4CNL	JLCC-20-STD
AMD	PAL16R6BCN	DIP-20-STD

AMD	PAL16R6BCNL	JLCC-20-STD
AMD	PAL16R6CN	DIP-20-STD
AMD	PAL16R6CNL	JLCC-20-STD
AMD	PAL16R6D/2PC	DIP-20-STD

P16R8

AMD	PAL16R8-4JC	JLCC-28-A28
AMD	PAL16R8-5JC	JLCC-20-STD
AMD	PAL16R8-5PC	DIP-20-STD
AMD	PAL16R8-7DC	DIP-20-STD
AMD	PAL16R8-7JC	JLCC-20-STD
AMD	PAL16R8-7PC	DIP-20-STD
AMD	PAL16R8A2CN	DIP-20-STD
AMD	PAL16R8A2CNL	JLCC-20-STD
AMD	PAL16R8ACN	DIP-20-STD
AMD	PAL16R8ACNL	JLCC-20-STD
AMD	PAL16R8B2CN	DIP-20-STD
AMD	PAL16R8B2CNL	JLCC-20-STD
AMD	PAL16R8B4CJ	DIP-20-STD
AMD	PAL16R8B4CN	DIP-20-STD
AMD	PAL16R8B4CNL	JLCC-20-STD
AMD	PAL16R8BCN	DIP-20-STD
AMD	PAL16R8BCNL	JLCC-20-STD
AMD	PAL16R8CN	DIP-20-STD
AMD	PAL16R8CNL	JLCC-20-STD
AMD	PAL16R8D/2JC	JLCC-20-STD
AMD	PAL16R8D/2PC	DIP-20-STD

P16V8A

AMD	PALCE16V8H-10JC/4	JLCC-20-STD
AMD	PALCE16V8H-10PC/4	DIP-20-STD
AMD	PALCE16V8H-10SC/4	SOIC-20-STD
AMD	PALCE16V8H-15JC/4	JLCC-20-STD
AMD	PALCE16V8H-15PC/4	DIP-20-STD
AMD	PALCE16V8H-15SC/4	DIP-20-STD
AMD	PALCE16V8H-25JC/4	JLCC-20-STD
AMD	PALCE16V8H-25PC/4	DIP-20-STD
AMD	PALCE16V8H-5JC/5	JLCC-20-STD
AMD	PALCE16V8H-7JC/5	JLCC-20-STD
AMD	PALCE16V8H-7PC/5	DIP-20-STD
AMD	PALCE16V8Q-10JC/5	JLCC-20-STD
AMD	PALCE16V8Q-15JC/4	JLCC-20-STD
AMD	PALCE16V8Q-15PC/4	DIP-20-STD
AMD	PALCE16V8Q-25JC/4	JLCC-20-STD
AMD	PALCE16V8Q-25PC/4	DIP-20-STD
AMD	PALCE16V8Z-15JI	JLCC-20-STD
AMD	PALCE16V8Z-15PI	DIP-20-STD
AMD	PALCE16V8Z-25JC	JLCC-20-STD
AMD	PALCE16V8Z-25JI	JLCC-20-STD
AMD	PALCE16V8Z-25PC	DIP-20-STD
AMD	PALCE16V8Z-25PI	DIP-20-STD
AMD	PALLV16V8-10JC	JLCC-20-STD
AMD	PALLV16V8-10PC	DIP-20-STD
AMD	PALLV16V8Z-20JI	JLCC-20-STD
AMD	PALLV16V8Z-20PI	DIP-20-STD

P16V8HD

AMD	PALCE16V8HD-15JC	JLCC-28-P28
AMD	PALCE16V8HD-15PC	DIP-24-STD

P20L8

AMD	PAL20L8-10/2JC	JLCC-28-P28
AMD	PAL20L8-10/2PC	DIP-24-STD
AMD	PAL20L8-5JC	JLCC-28-P28
AMD	PAL20L8-5PC	DIP-24-STD
AMD	PAL20L8-7JC	JLCC-28-P28
AMD	PAL20L8-7PC	DIP-24-STD
AMD	PAL20L8A2CNL	JLCC-28-U28
AMD	PAL20L8A2CNS	DIP-24-STD
AMD	PAL20L8ACNL	JLCC-28-U28
AMD	PAL20L8ACNS	DIP-24-STD
AMD	PAL20L8B2CFN	JLCC-28-P28
AMD	PAL20L8B2CNL	JLCC-28-P28
AMD	PAL20L8B2CNS	DIP-24-STD
AMD	PAL20L8BCFN	JLCC-28-U28
AMD	PAL20L8BCNL	JLCC-28-U28
AMD	PAL20L8BCNS	DIP-24-STD

P20R4

AMD	PAL20R4-5JC	JLCC-28-P28
AMD	PAL20R4-5PC	DIP-24-STD
AMD	PAL20R4-7DC	DIP-24-STD
AMD	PAL20R4-7JC	JLCC-28-P28
AMD	PAL20R4-7PC	DIP-24-STD
AMD	PAL20R4A2CNL	JLCC-28-U28
AMD	PAL20R4A2CNS	DIP-24-STD
AMD	PAL20R4ACNL	JLCC-28-U28
AMD	PAL20R4ACNS	DIP-24-STD
AMD	PAL20R4B2CFN	JLCC-28-P28
AMD	PAL20R4B2CNL	JLCC-28-P28
AMD	PAL20R4B2CNS	DIP-24-STD
AMD	PAL20R4BCNL	JLCC-28-U28
AMD	PAL20R4BCNS	DIP-24-STD

P20R6

AMD	PAL20R6-5JC	JLCC-28-P28
AMD	PAL20R6-5PC	DIP-24-STD
AMD	PAL20R6-7DC	DIP-24-STD
AMD	PAL20R6-7JC	JLCC-28-P28
AMD	PAL20R6-7PC	DIP-24-STD
AMD	PAL20R6A2CNL	JLCC-28-U28
AMD	PAL20R6A2CNS	DIP-24-STD
AMD	PAL20R6ACNL	JLCC-28-U28
AMD	PAL20R6ACNS	DIP-24-STD
AMD	PAL20R6B2CFN	JLCC-28-P28
AMD	PAL20R6B2CNS	DIP-24-STD
AMD	PAL20R6BCNL	JLCC-28-U28
AMD	PAL20R6BCNS	DIP-24-STD

P20R8

AMD	PAL20R8-10/2PC	DIP-24-STD
AMD	PAL20R8-5JC	JLCC-28-P28
AMD	PAL20R8-5PC	DIP-24-STD
AMD	PAL20R8-7DC	DIP-24-STD
AMD	PAL20R8-7JC	JLCC-28-P28
AMD	PAL20R8-7PC	DIP-24-STD
AMD	PAL20R8A2CNL	JLCC-28-U28
AMD	PAL20R8A2CNS	DIP-24-STD
AMD	PAL20R8ACNL	JLCC-28-U28
AMD	PAL20R8ACNS	DIP-24-STD
AMD	PAL20R8B2CFN	JLCC-28-P28
AMD	PAL20R8B2CNL	JLCC-28-P28
AMD	PAL20R8B2CNS	DIP-24-STD
AMD	PAL20R8BCNL	JLCC-28-U28
AMD	PAL20R8BCNS	DIP-24-STD

P20RA10

AMD	PAL20RA10-20CFN	JLCC-28-P28
AMD	PALCE20RA10H-10JC	JLCC-28-P28
AMD	PALCE20RA10H-10JI	JLCC-28-P28
AMD	PALCE20RA10H-10PC	DIP-24-STD
AMD	PALCE20RA10H-10PI	DIP-24-STD
AMD	PALCE20RA10H-15JC	JLCC-28-P28
AMD	PALCE20RA10H-15JI	JLCC-28-P28
AMD	PALCE20RA10H-15PC	DIP-24-STD
AMD	PALCE20RA10H-15PI	DIP-24-STD
AMD	PALCE20RA10H-20PC	DIP-24-STD
AMD	PALCE20RA10H-7JC	JLCC-28-P28
AMD	PALCE20RA10H-7JI	JLCC-28-P28

P20V8A

AMD	PALCE20V8H-10JC/4	JLCC-28-P28
AMD	PALCE20V8H-10PC/4	DIP-24-STD
AMD	PALCE20V8H-15JC/4	JLCC-28-P28
AMD	PALCE20V8H-15JI/4	JLCC-28-P28
AMD	PALCE20V8H-15PC/4	DIP-24-STD
AMD	PALCE20V8H-25JC/4	JLCC-28-P28
AMD	PALCE20V8H-25JI/4	JLCC-28-P28
AMD	PALCE20V8H-25PC/4	DIP-24-STD
AMD	PALCE20V8H-5JC/5	JLCC-28-P28
AMD	PALCE20V8H-7JC/5	JLCC-28-P28
AMD	PALCE20V8H-7PC/5	DIP-24-STD
AMD	PALCE20V8Q-15JC/4	JLCC-28-P28
AMD	PALCE20V8Q-15PC/4	DIP-24-STD
AMD	PALCE20V8Q-20PI/4	DIP-24-STD
AMD	PALCE20V8Q-25JC/4	JLCC-28-P28
AMD	PALCE20V8Q-25PC/4	DIP-24-STD

P22P10

AMD	AMPAL22P10AJC	JLCC-28-P28
AMD	AMPAL22P10ALDC	DIP-24-STD
AMD	AMPAL22P10ALPC	DIP-24-STD
AMD	AMPAL22P10BJC	JLCC-28-P28
AMD	AMPAL22P10BPC	DIP-24-STD

P22V10

AMD	AMPAL22V10AJC	JLCC-28-P28
AMD	AMPAL22V10APC	DIP-24-STD
AMD	AMPAL22V10JC	JLCC-28-P28
AMD	AMPAL22V10PC	DIP-24-STD
AMD	CE22V10H-15E4/BKA	FP-24-STD
AMD	PAL22V10-10JC	JLCC-28-P28
AMD	PAL22V10-10PC	DIP-24-STD
AMD	PAL22V10-15DC	DIP-24-STD
AMD	PAL22V10-15JC	JLCC-28-P28
AMD	PAL22V10-15PC	DIP-24-STD
AMD	PALCE22V10H-10JC/5	JLCC-28-P28
AMD	PALCE22V10H-10PC/5	DIP-24-STD
AMD	PALCE22V10H-15JC/4	JLCC-28-P28
AMD	PALCE22V10H-15PC/4	DIP-24-STD
AMD	PALCE22V10H-15SC/4	FP-24-STD
AMD	PALCE22V10H-25JC/4	JLCC-28-P28
AMD	PALCE22V10H-25PC/4	DIP-24-STD
AMD	PALCE22V10H-25SC/4	FP-24-STD
AMD	PALCE22V10H-5JC/5	JLCC-28-P28
AMD	PALCE22V10H-7JC/5	JLCC-28-P28
AMD	PALCE22V10H-7PC/5	DIP-24-STD
AMD	PALCE22V10Q-10JC	JLCC-28-P28
AMD	PALCE22V10Q-10PC	DIP-24-STD
AMD	PALCE22V10Q-10SC	SOIC-24-STD
AMD	PALCE22V10Q-15JC	JLCC-28-P28
AMD	PALCE22V10Q-15PC	DIP-24-STD
AMD	PALCE22V10Q-25JC/4	JLCC-28-P28
AMD	PALCE22V10Q-25PC/4	DIP-24-STD
AMD	PALCE22V10Z-15JI	JLCC-28-P28
AMD	PALCE22V10Z-15PI	DIP-24-STD
AMD	PALCE22V10Z-15SI	SOIC-24-STD
AMD	PALCE22V10Z-25JC	JLCC-28-P28
AMD	PALCE22V10Z-25JI	JLCC-28-P28
AMD	PALCE22V10Z-25PC	DIP-24-STD
AMD	PALCE22V10Z-25PI	DIP-24-STD
AMD	PALCE22V10Z-25SC	SOIC-24-STD
AMD	PALCE22V10Z-25SI	SOIC-24-STD
AMD	PALLV22V10-10PC	DIP-24-STD
AMD	PALLV22V10-7JC	JLCC-28-P28
AMD	PALLV22V10Z-25JI	JLCC-28-P28
AMD	PALLV22V10Z-25PI	DIP-24-STD
AMD	PALLV22V10Z-25SI	SOIC-24-STD

P24V10

AMD	PALCE24V10H-15JC	JLCC-28-STD
AMD	PALCE24V10H-15PC	DIP-28-STD
AMD	PALCE24V10H-25JC	JLCC-28-STD
AMD	PALCE24V10H-25PC	DIP-28-STD

P26V12

AMD	PALCE26V12H-15JC/4	JLCC-28-STD
AMD	PALCE26V12H-15PC/4	DIP-28-STD
AMD	PALCE26V12H-20JC/4	JLCC-28-STD
AMD	PALCE26V12H-20PC/4	DIP-28-STD

P29M16

AMD	PALCE29M16H25JC/4	JLCC-28-P28
AMD	PALCE29M16H25PC/4	DIP-24-STD

P29MA16

AMD	PALCE29MA16H25JC/4	JLCC-28-P28
AMD	PALCE29MA16H25PC/4	DIP-24-STD

P600

AMD	PALCE610H-15JC	JLCC-28-S28
AMD	PALCE610H-15PC	DIP-24-STD
AMD	PALCE610H-20/B3A	JLCC-28-S28
AMD	PALCE610H-20/BLA	DIP-24-STD
AMD	PALCE610H-25JC	JLCC-28-S28
AMD	PALCE610H-25PC	DIP-24-STD

Device Footprints by Template Number

The following is an alphabetical list of MACHXL templates (i.e., architectures) and the footprints available for each.

Syntax

```
TARGET 'TEMPLATE template_name footprint_name ' ;
```

E10P4
- DIP-18-STD
- FP-18-STD
- JLCC-20-U20

E10P8
- DIP-24-STD
- JLCC-28-V28
- LCC-28-V28

E11P4
- DIP-18-STD

E11P8
- DIP-24-STD
- FP-24-STD
- JLCC-28-V28
- LCC-28-V28
- SOIC-24-STD

E12P4
- DIP-20-STD
- JLCC-20-V28

E5P8
- DIP-16-STD
- FP-16-STD
- JLCC-20-V20
- SOIC-16-STD

E8P4
- DIP-16-STD
- FP-16-STD
- JLCC-20-V20
- SOIC-16-STD

E9P4
- DIP-16-STD
- FP-16-STD
- JLCC-20-V20
- SOIC-16-STD

E9P8
- DIP-20-STD
- JLCC-20-STD

E9R8
- DIP-24-STD
- JLCC-28-V28
- LCC-28-V28

MACH110
- JLCC-44-STD

MACH111
- JLCC-44-STD

MACH120
- JLCC-68-STD

MACH130
- JLCC-84-STD

MACH131
- JLCC-84-STD

MACH210
- JLCC-44-STD
- TQFP-44-TQ44

MACH211
- JLCC-44-STD
- TQFP-44-TQ44

MACH215
- JLCC-44-STD

MACH220
- JLCC-68-STD

MACH230
- JLCC-84-STD

MACH231
- JLCC-84-STD

MACH355
- QFP-144-STD

MACH435
- JLCC-84-STD

MACH445
- QFP-100-STD

MACH465
- QFP-208-STD

P16L8
- DIP-20-STD
- FP-20-STD
- JLCC-20-STD
- JLCC-28-A28
- LCC-20-STD
- SOJ-20-STD

P16R4
- DIP-20-STD
- FP-20-STD
- JLCC-20-STD
- JLCC-28-A28
- LCC-20-STD

SOJ-20-STD

P16R6

DIP-20-STD
FP-20-STD
JLCC-20-STD
JLCC-28-A28
LCC-20-STD
SOJ-20-STD

P16R8

DIP-20-STD
FP-20-STD
JLCC-20-STD
JLCC-28-A28
LCC-20-STD
SOJ-20-STD

P16V8A

DIP-20-STD
JLCC-20-STD
LCC-20-STD
SOIC-20-STD

P16V8HD

DIP-24-STD
JLCC-28-P28

P20L8

DIP-24-STD
FP-24-STD
JLCC-28-P28
JLCC-28-U28
LCC-28-P28
LCC-28-R28

P20R4

DIP-24-STD
FP-24-STD
JLCC-28-P28
JLCC-28-U28
LCC-28-P28
LCC-28-R28

P20R6

DIP-24-STD
FP-24-STD
JLCC-28-P28
JLCC-28-U28
LCC-28-P28
LCC-28-R28

P20R8

DIP-24-STD
FP-24-STD
JLCC-28-P28
JLCC-28-U28
LCC-28-P28
LCC-28-R28

P20RA10

DIP-24-STD
JLCC-28-P28
JLCC-28-R28
JLCC-28-U28
LCC-28-P28
LCC-28-R28

P20V8A

DIP-24-STD
JLCC-24-STD
JLCC-28-P28
JLCC-28-U28
LCC-24-STD
LCC-28-P28
SOIC-24-STD

P22P10

DIP-24-STD
JLCC-28-P28

P22V10

DIP-24-STD
FP-24-STD
JLCC-24-STD
JLCC-28-P28
JLCC-28-PC28
JLCC-28-V28
JLCC-28-W28
LCC-24-STD
LCC-28-P28
LCC-28-W28
SOIC-24-STD

P24V10

DIP-28-STD
JLCC-28-STD

P26V12

DIP-28-STD
JLCC-28-STD

P29M16

DIP-24-STD
JLCC-28-P28

P29MA16

DIP-24-STD
JLCC-28-P28

P600

DIP-24-STD
JLCC-28-S28
SOIC-24-STD

New Devices

The following is a list of new devices that have been added to V3.0.

Mfg	Template Number	Manufacturer's Part Number
AMD	MACH110	MACH110-14JI
AMD	MACH110	MACH110-18JI
AMD	MACH110	MACH110-24JI
AMD	MACH111	MACH111-10JC
AMD	MACH111	MACH111-12JC
AMD	MACH111	MACH111-15JC
AMD	MACH111	MACH111-20JC
AMD	MACH111	MACH111-7JC
AMD	MACH120	MACH120-12JC
AMD	MACH120	MACH120-18JI
AMD	MACH120	MACH120-24JI
AMD	MACH130	MACCH130-18JII
AMD	MACH130	MACH130-18JI
AMD	MACH130	MACH130-24JI
AMD	MACH131	MACH131-10JC
AMD	MACH131	MACH131-12JC
AMD	MACH131	MACH131-15JC
AMD	MACH131	MACH131-20JC
AMD	MACH131	MACH131-7JC
AMD	MACH210	MACH210-14JI
AMD	MACH210	MACH210-18JI
AMD	MACH210	MACH210-24JI
AMD	MACH210	MACH210A-10VC
AMD	MACH210	MACH210A-12JI
AMD	MACH210	MACH210A-12VC
AMD	MACH210	MACH210A-15VC
AMD	MACH210	MACH210A-20VC
AMD	MACH210	MACH210A-75C
AMD	MACH210	MACH210A-7JC
AMD	MACH210	MACH210AQ-12JC
AMD	MACH210	MACH210AQ-18JI

Mfg	Template Number	Manufacturer's Part Number
AMD	MACH210	MACH210AQ-24JI
AMD	MACH210	MACHLV210-18JI
AMD	MACH210	MACHLV210-24JI
AMD	MACH215	MACH215-14JI
AMD	MACH215	MACH215-18JI
AMD	MACH215	MACH215-24JI
AMD	MACH220	MACH220-10JC
AMD	MACH220	MACH220-18JI
AMD	MACH220	MACH220-24JI
AMD	MACH230	MACH230-10JC
AMD	MACH230	MACH230-18JI
AMD	MACH230	MACH230-24JI
AMD	MACH231	MACH231-10JC
AMD	MACH231	MACH231-12JC
AMD	MACH231	MACH231-15JC
AMD	MACH231	MACH231-20JC
AMD	MACH231	MACH231-7JC
AMD	MACH355	MACH355-12JC
AMD	MACH355	MACH355-15YC
AMD	MACH355	MACH355-20YC
AMD	MACH435	MACH435-12JC
AMD	MACH435	MACH435Q-20JC
AMD	MACH445	MACH445-12YC
AMD	MACH445	MACH445-15KC
AMD	MACH445	MACH445-15YC
AMD	MACH445	MACH445-20YC
AMD	MACH465	MACH465-15YC
AMD	MACH465	MACH465-20YC
AMD	P16V8A	PALCE16V8Z-15JI
AMD	P16V8A	PALCE16V8Z-15PI
AMD	P16V8A	PALCE16V8Z-25JC

Mfg	Template Number	Manufacturer's Part Number
AMD	P16V8A	PALCE16V8Z-25PC
AMD	P16V8A	PALLV16V8-10JC
AMD	P16V8A	PALLV16V8-10PC
AMD	P16V8A	PALLV16V8Z-20JI
AMD	P16V8A	PALLV16V8Z-20PI
AMD	P20RA10	PALCE20RA10H-10JC
AMD	P20RA10	PALCE20RA10H-10JI
AMD	P20RA10	PALCE20RA10H-10PC
AMD	P20RA10	PALCE20RA10H-10PI
AMD	P20RA10	PALCE20RA10H-15JC
AMD	P20RA10	PALCE20RA10H-15JI
AMD	P20RA10	PALCE20RA10H-15PC
AMD	P20RA10	PALCE20RA10H-15PI
AMD	P20RA10	PALCE20RA10H-7JC
AMD	P20RA10	PALCE20RA10H-7JI
AMD	P20V8A	PALCE20V8H-15JI/4
AMD	P20V8A	PALCE20V8H-25JI/4
AMD	P20V8A	PALCE20V8H-7JC/5
AMD	P20V8A	PALCE20V8H-7PC/5
AMD	P20V8A	PALCE20V8Q-20PI/4
AMD	P22V10	PALLV22V10-10PC
AMD	P22V10	PALLV22V10-7JC
AMD	S128V128	PLA128V128-DUMMY
AMD	S64V32	PLA64V32-DUMMY

Renamed Devices

The following is a list of devices that have been renamed for various reasons since V1.2. MACHXL will continue to support the architectures in the Version 3.0 release.

Mfg	New Manufacturer's Part Number	Old Manufacturer's Part Number
AMD	MACH130-18JI	MACCH130-18JII
AMD	MACH210A-7JC	MACH210A-75C
AMD	MACH355-15YC	MACH355-12JC
AMD	MACH445-15YC	MACH445-15KC
AMD	MACH465-20YC	MACH465-20KC
AMD	MACH465-15YC	MACH465-15KC

Obsolete Devices

The following is a list of devices that have become obsolete since V1.2. These devices will remain obsolete for one (1) year, at which time they will no longer be supported.

Mfg	Manufacturer's Part Number
AMD	MACH211-14JI
AMD	MACH211-18JI
AMD	MACH211-24JI

Deleted Devices

The following is a list of devices that have been deleted since V1.2. They are no longer supported.

Mfg	Manufacturer's Part Number
AMD	MACH110-20/BXA
AMD	MACH111-14JI
AMD	MACH111-18JI
AMD	MACH111-24JI
AMD	MACH131-18JI
AMD	MACH131-24JI
AMD	MACH210-20/BXA
AMD	MACH220-14JI
AMD	MACH231-18JI
AMD	MACH231-24JI
AMD	P20L10-DIP-OBS
AMD	P20L10-JLCC-OBS

B Language-Based Design Examples

Contents

Introduction

The examples in this chapter are intended as a tutorial of language constructs and are ordered to introduce new concepts with each example.

Each example can be found in the *examples/manual* subdirectory of the MINC installation directory. There are also other undocumented examples in this subdirectory for reference.

Building a MACHXL Design Synthesis Language Source File

MACHXL lets you build a source file to describe your design. **Chapters 4 - 9** cover the elements of this source file. The following diagram shows the general organization of a typical design source file. It also lists the chapter(s) where information about each part of the design source file is located.

Parts of a Source File (Using MACHXL's Design Synthesis Language)

Headers (information about the design)	Chapter 4
MACRO Definitions (text substitution structures)	Chapter 9
USE constructs (compiled Procedures and Functions to be used by this source file)	Chapter 8
Procedure/Function Definitions (Procedures/Functions used in this design)	Chapter 8
System-Level Declarations (declaring the signals to be used in this design)	Chapter 5
System-Level Statements (statements and constructs that describe your design)	Chapters 6, 7

The sample above is a template of a typical source file. Each of the sections listed is optional. In addition to these chapters, this appendix contains a number of language design examples, complete with comments and explanations.

Gray_Code Counter Examples

The following eight examples implement a 4-bit gray code counter with reset. The first example uses an asynchronous reset while the others use a synchronous reset. The functioning of each is similar, however each example uses different statement constructs to illustrate the various capabilities available in the Design Synthesis Language. These examples are complete (where appropriate) with *.cst*, *.pi* (if needed), and *.stm* files and represent solutions using PLDs and CPLDs.

Example 1: Asynchronously Reset Gray Code Counter Using Simple Equations (PLDs)

This design implements a 4-bit gray code counter with an asynchronous reset. The signals *q3, q2, q1*, and *q0* represent the 4-bit counter output values. The equations were derived by writing out the 16 gray code values in order and noting all combinations where a signal transitions to a 1.

```
" GRAY1
" AMD


INPUT clock, reset;
OUTPUT q3, q2, q1, q0 CLOCKED_BY clock RESET_BY reset;


q3 = Q3*/Q2*/Q1*Q0 + Q3*/Q2*Q1*Q0 + Q3*/Q2*Q1*/Q0 +
        Q3*Q2*Q1*/Q0 + Q3*Q2*Q1*Q0 + Q3*Q2*/Q1*Q0 +
        Q3*Q2*/Q1*/Q0 + /Q3*Q2*/Q1*/Q0;

q2 = Q3*Q2*Q1*Q0 + Q3*Q2*/Q1*Q0 + Q3*Q2*/Q1*/Q0 +
        /Q3*Q2*/Q1*/Q0 + /Q3*Q2*/Q1*Q0 + /Q3*Q2*Q1*Q0 +
        /Q3*Q2*Q1*/Q0 + /Q3*/Q2*Q1*/Q0;
```

```
q1 = Q3*/Q2*Q1*/Q0 + Q3*Q2*Q1*/Q0 + Q3*Q2*Q1*Q0 +
        Q3*Q2*/Q1*Q0 + /Q3*Q2*Q1*/Q0 + /Q3*/Q2*Q1*/Q0 +
        /Q3*/Q2*Q1*Q0 + /Q3*/Q2*/Q1*Q0;

q0 = Q3*/Q2*Q1*Q0 + Q3*/Q2*Q1*/Q0 + Q3*Q2*/Q1*Q0 +
        Q3*Q2*/Q1*/Q0 + /Q3*Q2*Q1*Q0 + /Q3*Q2*Q1*/Q0 +
        /Q3*/Q2*/Q1*Q0 + /Q3*/Q2*/Q1*/Q0;
```

.stm (Stimulus) File for Example 1

```
" This is the stimulus source for the 4-bit gray code counter
" with 'asynchronous reset' in the file 'gray1.src'.

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION',

" JEDEC test vectors are produced when PLD/CPLD devices are
" targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q3,q2,q1,q0];

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;
```

```
        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

END SYSTEM_TEST;
```

Example 2: Synchronously Reset Gray Code Counter Using Simple Equations

This is the same gray code counter as in Example 1 except with a synchronous reset rather than an asynchronous reset. To implement a synchronous reset the *reset* signal has been incorporated into the equations of each counter output value.

```
" GRAY2
" AMD

INPUT clock, reset;
OUTPUT q3, q2, q1, q0 CLOCKED_BY clock;

q3 = /reset*(Q3*/Q2*/Q1*Q0 + Q3*/Q2*Q1*Q0 + Q3*/Q2*Q1*/Q0 +
        Q3*Q2*Q1*/Q0 + Q3*Q2*Q1*Q0  + Q3*Q2*/Q1*Q0 +
        Q3*Q2*/Q1*/Q0 + /Q3*Q2*/Q1*/Q0);

q2 = /reset*(Q3*Q2*Q1*Q0 + Q3*Q2*/Q1*Q0 + Q3*Q2*/Q1*/Q0 +
        /Q3*Q2*/Q1*/Q0 + /Q3*Q2*/Q1*Q0 + /Q3*Q2*Q1*Q0 +
        /Q3*Q2*Q1*/Q0 + /Q3*/Q2*Q1*/Q0);

q1 = /reset*(Q3*/Q2*Q1*/Q0 + Q3*Q2*Q1*/Q0 + Q3*Q2*Q1*Q0 +
        Q3*Q2*/Q1*Q0 + /Q3*Q2*Q1*/Q0 + /Q3*/Q2*Q1*/Q0 +
        /Q3*/Q2*Q1*Q0 + /Q3*/Q2*/Q1*Q0);
```

```
q0 = /reset*(Q3*/Q2*Q1*Q0 + Q3*/Q2*Q1*/Q0 + Q3*Q2*/Q1*Q0 +
        Q3*Q2*/Q1*/Q0 + /Q3*Q2*Q1*Q0 + /Q3*Q2*Q1*/Q0 +
        /Q3*/Q2*/Q1*Q0 + /Q3*/Q2*/Q1*/Q0);
```

.stm (Stimulus) File for Example 2

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray2.src'.

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION',

" JEDEC test vectors are produced when PLD/CPLD devices are
" targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q3,q2,q1,q0];

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;
```

```
        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

END SYSTEM_TEST;
```

Example 3: Synchronously Reset Gray Code Counter Using a Truth Table

This is another example of a 4-bit counter with a synchronous reset, this time using a truth table to specify the output values.

```
" GRAY3
" AMD

INPUT clock, reset;
OUTPUT q3, q2, q1, q0 CLOCKED_BY clock;

" This macro makes all X's be treated as don't cares.
MACRO X .X.;

TRUTH_TABLE
        reset, q3, q2, q1, q0 :: q3, q2, q1, q0;
        "--------------------------------------

         1,  X, X, X, X :: 0, 0, 0, 0;
         0,  0, 0, 0, 0 :: 0, 0, 0, 1;
         0,  0, 0, 0, 1 :: 0, 0, 1, 1;
         0,  0, 0, 1, 1 :: 0, 0, 1, 0;
         0,  0, 0, 1, 0 :: 0, 1, 1, 0;
         0,  0, 1, 1, 0 :: 0, 1, 1, 1;
         0,  0, 1, 1, 1 :: 0, 1, 0, 1;
         0,  0, 1, 0, 1 :: 0, 1, 0, 0;
         0,  0, 1, 0, 0 :: 1, 1, 0, 0;
         0,  1, 1, 0, 0 :: 1, 1, 0, 1;
         0,  1, 1, 0, 1 :: 1, 1, 1, 1;
         0,  1, 1, 1, 1 :: 1, 1, 1, 0;
         0,  1, 1, 1, 0 :: 1, 0, 1, 0;
         0,  1, 0, 1, 0 :: 1, 0, 1, 1;
         0,  1, 0, 1, 1 :: 1, 0, 0, 1;
```

```
        0,   1, 0, 0, 1 :: 1, 0, 0, 0;
        0,   1, 0, 0, 0 :: 0, 0, 0, 0;
END TRUTH_TABLE;
```

.stm (Stimulus) File for Example 3

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray3.src'.

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q3,q2,q1,q0];

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;
```

```
        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

END SYSTEM_TEST;
```

Example 4: Synchronously Reset Gray Code Counter Using a Truth Table and IF Construct (AMD MACH)

This is the same gray code counter design. The handling of the synchronous reset has been pulled out of the truth table and place in a parent IF statement. The signals representing the counter value are now the array members `q[3]`, `q[2]`, `q[1]`, and `q[0]`.

```
" GRAY4
" AMD

INPUT clock, reset;
OUTPUT q[4] CLOCKED_BY clock;

IF reset THEN
        q = 0;
ELSE

        TRUTH_TABLE
                q       :: q;
                "--------------
                0000b :: 0001b;
                0001b :: 0011b;
                0011b :: 0010b;
                0010b :: 0110b;
                0110b :: 0111b;
                0111b :: 0101b;
                0101b :: 0100b;
                0100b :: 1100b;
                1100b :: 1101b;
                1101b :: 1111b;
                1111b :: 1110b;
```

```
                1110b :: 1010b;
                1010b :: 1011b;
                1011b :: 1001b;
                1001b :: 1000b;
                1000b :: 0000b;
        END TRUTH_TABLE;
END IF;
```

.*cst* (Constraint) File for Example 4

```
WEIGHT PRICE 10 ;

TEMPLATE = MACH110 OR MACH120 OR MACH130 OR MACH210 OR
MACH215 OR MACH220 OR MACH230 OR MACH435 OR MACH465 ;
```

.*stm* (Stimulus) File for Example 4

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray4.src'."

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q[3],q[2],q[1],q[0]];
        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;
```

```
        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

END SYSTEM_TEST;
```

Example 5: Synchronously Reset Gray Code Counter Using CASE Statement

This is the same gray code counter design using a CASE statement.

```
" GRAY5
" AMD

INPUT clock, reset;
OUTPUT q[4] CLOCKED_BY clock;
IF reset THEN
        q = 0;
ELSE
        CASE q
                WHEN 0000b=>
                        q = 0001b;
                WHEN 0001b=>
                        q = 0011b;
                WHEN 0011b=>
                        q = 0010b;
                WHEN 0010b=>
                        q = 0110b;
                WHEN 0110b=>
                        q = 0111b;
```

```
                WHEN 0111b=>
                        q = 0101b;
                WHEN 0101b=>
                        q = 0100b;
                WHEN 0100b=>
                        q = 1100b;
                WHEN 1100b=>
                        q = 1101b;
                WHEN 1101b=>
                        q = 1111b;
                WHEN 1111b=>
                        q = 1110b;
                WHEN 1110b=>
                        q = 1010b;
                WHEN 1010b=>
                        q = 1011b;
                WHEN 1011b=>
                        q = 1001b;
                WHEN 1001b=>
                        q = 1000b;
                WHEN 1000b=>
                        q = 0000b;
        END CASE;
END IF;
```

.stm (Stimulus) File for Example 5

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray5.src'."

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', " JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q[3],q[2],q[1],q[0]];

        MESSAGE('RESET...');
        SET reset=1;
```

```
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

END SYSTEM_TEST;
```

Example 6: Synchronously Reset Gray Code Counter Using IF Statement

This is the same gray code counter design using IF statements.

```
" GRAY6
" AMD

INPUT clock, reset;
OUTPUT q[4] CLOCKED_BY clock;
IF reset THEN
        q = 0;
```

```
ELSE
        IF q = 0000b THEN
                q = 0001b;
        ELSIF q = 0001b THEN
                q = 0011b;
        ELSIF q = 0011b THEN
                q = 0010b;
        ELSIF q = 0010b THEN
                q = 0110b;
        ELSIF q = 0110b THEN
                q = 0111b;
        ELSIF q = 0111b THEN
                q = 0101b;
        ELSIF q = 0101b THEN
                q = 0100b;
        ELSIF q = 0100b THEN
                q = 1100b;
        ELSIF q = 1100b THEN
                q = 1101b;
        ELSIF q = 1101b THEN
                q = 1111b;
        ELSIF q = 1111b THEN
                q = 1110b;
        ELSIF q = 1110b THEN
                q = 1010b;
        ELSIF q = 1010b THEN
                q = 1011b;
        ELSIF q = 1011b THEN
                q = 1001b;
        ELSIF q = 1001b THEN
                q = 1000b;
        ELSIF q = 1000b THEN
                q = 0000b;
        END IF;
END IF;
```

.stm (Stimulus) File for Example 6

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray6.src'.
```

```
" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q[3],q[2],q[1],q[0]];

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;
END SYSTEM_TEST;
```

Example 7: Synchronously Reset Gray Code Counter Using a State Machine

This is the same gray code counter design using a STATE_MACHINE construct with explicit state values.

```
" GRAY7
AMD

INPUT clock, reset;
OUTPUT q[4] CLOCKED_BY clock;

IF reset THEN
        q = 0;
ELSE
        STATE_MACHINE gray STATE_BITS q;
                STATE s1[0000b]:
                        GOTO s2;
                STATE s2[0001b]:
                        GOTO s3;
                STATE s3[0011b]:
                        GOTO s4;
                STATE s4[0010b]:
                        GOTO s5;
                STATE s5[0110b]:
                        GOTO s6;
                STATE s6[0111b]:
                        GOTO s7;
                STATE s7[0101b]:
                        GOTO s8;
                STATE s8[0100b]:
                        GOTO s9;
                STATE s9[1100b]:
                        GOTO s10;
                STATE s10[1101b]:
                        GOTO s11;
                STATE s11[1111b]:
                        GOTO s12;
                STATE s12[1110b]:
                        GOTO s13;
```

```
                STATE s13[1010b]:
                        GOTO s14;
                STATE s14[1011b]:
                        GOTO s15;
                STATE s15[1001b]:
                        GOTO s16;
                STATE s16[1000b]:
                        GOTO s1;
        END gray;
END IF;
```

.stm (Stimulus) File for Example 7

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray7.src'.

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q[3],q[2],q[1],q[0]];

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
```

```
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;
END SYSTEM_TEST;
```

Example 8: Synchronously Reset Gray Code Counter Using a State Machine

This is a synchronous reset gray code counter design using the STATE_MACHINE's built in gray code state value assignment capability. This example takes advantage of gray code assignment to make a gray-code counter. Normally, this built-in capability would be used as part of a more involved state machine design.

```
" GRAY8
" AMD

INPUT clock, reset;
OUTPUT q[4] CLOCKED_BY clock;
IF reset THEN
        q = 0;
ELSE
        STATE_MACHINE gray STATE_BITS q STATE_VALUES
GRAY_CODE;

                STATE s1:
                        GOTO s2;
                STATE s2:
                        GOTO s3;
                STATE s3:
                        GOTO s4;
                STATE s4:
                        GOTO s5;
                STATE s5:
                        GOTO s6;
```

```
                STATE s6:
                        GOTO s7;
                STATE s7:
                        GOTO s8;
                STATE s8:
                        GOTO s9;
                STATE s9:
                        GOTO s10;
                STATE s10:
                        GOTO s11;
                STATE s11:
                        GOTO s12;
                STATE s12:
                        GOTO s13;
                STATE s13:
                        GOTO s14;
                STATE s14:
                        GOTO s15;
                STATE s15:
                        GOTO s16;
                STATE s16:
                        GOTO s1;
        END gray;
END IF;
```

.stm (Stimulus) File for Example 8

```
" This is the stimulus source for the 4-bit gray code counter
" with 'synchronous reset' in the file 'gray8.src'.

" By using the keyword 'SYSTEM_TEST' instead of
" 'SIMULATION', JEDEC test vectors are produced when
" PLD/CPLD devices are targeted.

SYSTEM_TEST;

        VAR I;
        TRACE reset, clock, [q[3],q[2],q[1],q[0]];
        MESSAGE('RESET...');
        SET reset=1;
```

```
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;

        FOR i = 0 TO 6 DO
                CLOCKF clock;
        END FOR;

        MESSAGE('RESET...');
        SET reset=1;
        CLOCKF clock;
        SET reset=0;

        MESSAGE('START COUNT...');
        FOR i = 0 TO 15 DO
                CLOCKF clock;
        END FOR;
END SYSTEM_TEST;
```

Drink Machine Examples

The following examples implement the coin counting needs of a drink machine. The purpose of these examples is to demonstrate some of the capabilities of the STATE_MACHINE construct and give an introduction to the effect of the DEFAULT_TO expression.

Example 1: Drink Machine Using a State Machine

This implementation of a drink machine demonstrates the use of state machines. It accepts inputs indicating the insertion of nickels, dimes, and quarters. Its outputs are a dime coin return, nickel coin return, and a signal to dispense a drink. A drink costs 30 cents. A drink will be automatically dispensed when the correct total or greater is reached.

```
" DRINK1
" AMD

INPUT nickel, dime, quarter, clock;
OUTPUT return_dime, return_nickel, dispense_drink;

" This state machine, by default, uses D_FLOPs to represent " the
current state.

" The CLOCKED_BY expression causes state transitions to
" occur when the 'clock' signal transitions and the conditions
" for a particular GOTO are met.

STATE_MACHINE drink_machine CLOCKED_BY clock;
        STATE Zero:
                IF nickel THEN
                        GOTO Five;
                ELSIF dime THEN
                        GOTO Ten;
```

```
        ELSIF quarter THEN
                GOTO TwentyFive;
        ELSE
                GOTO Zero;
        END IF;
        dispense_drink = 0;
        return_dime = 0;
        return_nickel = 0;
STATE Five:
        IF nickel THEN
                dispense_drink = 0;
                GOTO Ten;
        ELSIF dime THEN
                dispense_drink = 0;
                GOTO Fifteen;
        ELSIF quarter THEN
                dispense_drink = 1;
                GOTO Zero;
        ELSE
                dispense_drink = 0;
                GOTO Five;
        END IF;
        return_dime = 0;
        return_nickel = 0;
STATE Ten:
        IF nickel THEN
                dispense_drink = 0;
                return_nickel = 0;
                GOTO Fifteen;
        ELSIF dime THEN
                dispense_drink = 0;
                return_nickel = 0;
                GOTO Twenty;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_nickel = 1;
                GOTO Zero;
```

```
        ELSE
                dispense_drink = 0;
                return_nickel = 0;
                GOTO Ten;
        END IF;
        return_dime = 0;
STATE Fifteen:
        IF nickel THEN
                dispense_drink = 0;
                return_dime = 0;
                GOTO Twenty;
        ELSIF dime THEN
                dispense_drink = 0;
                return_dime = 0;
                GOTO TwentyFive;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_dime = 1;
                GOTO Zero;
        ELSE
                dispense_drink = 0;
                return_dime = 0;
                GOTO Fifteen;
        END IF;
        return_nickel = 0;
STATE Twenty:
        IF nickel THEN
                dispense_drink = 0;
                return_nickel = 0;
                return_dime = 0;
                GOTO TwentyFive;
        ELSIF dime THEN
                dispense_drink = 1;
                return_nickel = 0;
                return_dime = 0;
                GOTO Zero;
```

```
        ELSIF quarter THEN
                dispense_drink = 1;
                return_dime = 1;
                return_nickel = 1;
                GOTO Zero;
        ELSE
                dispense_drink = 0;
                return_nickel = 0;
                return_dime = 0;
                GOTO Twenty;
        END IF;
STATE TwentyFive:
        IF nickel THEN
                dispense_drink = 1;
                return_nickel = 0;
                return_dime = 0;
                GOTO Zero;
        ELSIF dime THEN
                dispense_drink = 1;
                return_nickel = 1;
                return_dime = 0;
                GOTO Zero;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_nickel = 0;
                return_dime = 1;
                GOTO OweDime;
        ELSE
                dispense_drink = 0;
                return_nickel = 0;
                return_dime = 0;
                GOTO TwentyFive;
        END IF;
```

```
        STATE OweDime:
                " This state causes a wait of a clock cycle
                " before trying to return a second dime.
                dispense_drink = 0;
                return_nickel = 0;
                return_dime = 1;
                GOTO Zero;
END drink_machine;
```

Example 2: Drink Machine Using a State Machine and Default Values

This is the same drink machine design as in Example 1 with some changes:

The numerous assignments to the outputs (*return_dime, return_nickel, dispense_drink*) cluttered up the previous design. This design has a DEFAULT_TO 0 on the outputs causing each signal to have a 0 value except where they are explicitly assigned a different value. Without this DEFAULT_TO expression, the compiler would assume the design does not care what value these signals have when they are not assigned to explicitly.

The design recommends to the fitting tools the signals representing the current state be implemented with JK flip-flops rather than D flip-flops.

A DEFAULT_TO LAST_VALUE has been added to the state machine declaration which makes the state not change unless an explicit GOTO is given.

The state machine now has an ELSE clause making the machine more robust. This handles cases when the signals representing the current state get an undefined combination of values.

```
" DRINK2
" AMD

INPUT nickel, dime, quarter, clock, reset;
OUTPUT return_dime, return_nickel, dispense_drink DEFAULT_TO
0;

" The CLOCKED_BY expression causes state transitions to
" occur when the 'clock'signal transitions and the conditions
" for a particular GOTO are met.

" The RESET_BY expression causes the state machine to
" transition back to the first state (Zero) whenever the 'reset'
" signal is true.

" The DEFAULT_TO LAST_VALUE causes each state to
" transition to itself by default. So, any GOTO from a state to " itself
is unnecessary.

STATE_MACHINE drink_machine
CLOCKED_BY clock RESET_BY reset DEFAULT_TO
LAST_VALUE;
        STATE Zero:
                IF nickel THEN
                        GOTO Five;
                ELSIF dime THEN
                        GOTO Ten;
                ELSIF quarter THEN
                        GOTO TwentyFive;
                END IF;
        STATE Five:
                IF nickel THEN
                        GOTO Ten;
                ELSIF dime THEN
                        GOTO Fifteen;
                ELSIF quarter THEN
                        dispense_drink = 1;
                        GOTO Zero;
                END IF;
```

```
STATE Ten:
        IF nickel THEN
                GOTO Fifteen;
        ELSIF dime THEN
                GOTO Twenty;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_nickel = 1;
                GOTO Zero;
        END IF;
STATE Fifteen:
        IF nickel THEN
                GOTO Twenty;
        ELSIF dime THEN
                GOTO TwentyFive;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_dime = 1;
                GOTO Zero;
        END IF;
STATE Twenty:
        IF nickel THEN
                GOTO TwentyFive;
        ELSIF dime THEN
                dispense_drink = 1;
                GOTO Zero;
        ELSIF quarter THEN
                dispense_drink = 1;
                return_nickel = 1;
                return_dime = 1;
                GOTO Zero;
        END IF;
STATE TwentyFive:
        IF nickel THEN
                dispense_drink = 1;
                GOTO Zero;
```

```
                ELSIF dime THEN
                        return_nickel = 1;
                        dispense_drink = 1;
                        GOTO Zero;
                ELSIF quarter THEN
                        dispense_drink = 1;
                        return_dime = 1;
                        GOTO OweDime;
                END IF;
        STATE OweDime:
                " This state causes a wait of a clock cycle
                " before trying to return a second dime.
                return_dime = 1;
                GOTO Zero;
        ELSE
                " This ELSE makes sure that the state machine
                " resets itself if it somehow gets into an
                " undefined state.
                GOTO Zero;
END drink_machine;
```

Seven-Segment Display Handler Example

The following example creates a design taking a binary number and displaying it as two decimal digits on a 7-segment LED display. This example demonstrates the use of the TRUTH_TABLE statement and the CASE statement. It also introduces the use of PROCEDUREs and the concept of local versus system level signals.

```
" SEGMENT
" AMD

" This procedure takes a 4-bit number and creates the 7
" signals needed to display its decimal image on a 7-segment
" digit display.  The numbering of the segments is as follows:

"  0-> ---
"  5-> | | <-1
"  6-> ---
"  4-> | | <-2
"  3-> ---

" For values from 0-9 the corresponding digit is displayed. For
" values from 10-15 an 'E' is displayed indicating an
" erroneous value.

PROCEDURE display_digit(INPUT number[4]; OUTPUT digit[7]);

        TRUTH_TABLE
                number :: digit;
                                " exactly the same as:
                                " number[3..0] :: digit[6..0]

                        "--------------
                           0:: 0111111b;
                           1:: 0000110b;
                           2:: 1011011b;
                           3:: 1001111b;
```

```
                    4:: 1100110b;
                    5:: 1101101b;
                    6:: 1111101b;
                    7:: 0000111b;
                    8:: 1111111b;
                    9:: 1101111b;
                ELSE:: 1111100b;  " This creates the pattern
                                  " for an 'E'
        END TRUTH_TABLE;
END display_digit;

" This procedure takes a 7 bit binary input value and
" generates two decimal digit values to represent the value in
" decimal.Values greater than 99 should not occur.

" Note the CASE statements have no ELSEs.  This is
" because values greater than 99 won't be passed to this
" procedure.  The compiler will assume the design doesn't
" care what values 'high' and 'low' have when 'value' is
" greater than 99.  It will take advantage of this assumption to "
generate the smallest possible equations which guarantee
" 'high' and 'low' have the specified values when 'value' is
" less than or equal to 99.

PROCEDURE make_decimal(INPUT value[7]; OUTPUT high[4],
low[4]);
        " Create the low order digit.
        CASE value
                WHEN 0,10,20,30,40,50,60,70,80,90=>
                        low = 0;
                WHEN 1,11,21,31,41,51,61,71,81,91=>
                        low = 1;
                WHEN 2,12,22,32,42,52,62,72,82,92=>
                        low = 2;
                WHEN 3,13,23,33,43,53,63,73,83,93=>
                        low = 3;
                WHEN 4,14,24,34,44,54,64,74,84,94=>
                        low = 4;
```

```
                WHEN 5,15,25,35,45,55,65,75,85,95=>
                        low = 5;
                WHEN 6,16,26,36,46,56,66,76,86,96=>
                        low = 6;
                WHEN 7,17,27,37,47,57,67,77,87,97=>
                        low = 7;
                WHEN 8,18,28,38,48,58,68,78,88,98=>
                        low = 8;
                WHEN 9,19,29,39,49,59,69,79,89,99=>
                        low = 9;
        END CASE;

        " Create the high order digit.
        CASE value
                WHEN 0..9=>
                        high = 0;
                WHEN 10..19=>
                        high = 1;
                WHEN 20..29=>
                        high = 2;
                WHEN 30..39=>
                        high = 3;
                WHEN 40..49=>
                        high = 4;
                WHEN 50..59=>
                        high = 5;
                WHEN 60..69=>
                        high = 6;
                WHEN 70..79=>
                        high = 7;
                WHEN 80..89=>
                        high = 8;
                WHEN 90..99=>
                        high = 9;
        END CASE;
END make_decimal;
```

```
" The following portion of the design is outside of any
" PROCEDURE or FUNCTION.  It is the portion of the design
" specified at this level resulting in a real implementation.
" The above PROCEDUREs only impact the real
" implementation because they are called here.  Note all
" PROCEDUREs and FUNCTIONs must always come before
" the system level portion of the design.

" This is the 7-bit input value.
INPUT value[7];

" These are intermediate values of the two decimal digits.
 NODE high_val[4], low_val[4];

" These are the 7 segment values for the two digits.
OUTPUT digit_high[7], digit_low[7];

" These procedures calls create real instances of the
" procedures described above.  Note 'display_digit' is
" called twice and will create two separate instances of the
" logic described in that procedure.

make_decimal(value, high_val, low_val);
display_digit(high_val, digit_high);
display_digit(low_val, digit_low);
```

Adders and Multipliers

The following examples implement 1-bit, 2-bit, 4-bit, and 8-bit adders and a 4x4 multiplier. These examples demonstrate the use of FUNCTIONs and PROCEDUREs, the behavior of arrays and groups, and the concept of libraries and the USE clause.

Example 1: 1-Bit, 2-Bit, 4-Bit and 8-Bit Adder Procedures

This example consists of four adder PROCEDUREs. They implement 1-bit, 2-bit, 4-bit, and 8-bit adders, each with carry in and carry out. Since this example contains only PROCEDUREs and no system level code, it does not result in a real implementation of hardware.

```
" ADDER1
" AMD

" This example consists of four adder PROCEDUREs. They
" implement 1-bit, 2-bit, 4-bit, and 8-bit adders each with
" carry in and carry out. Since this example contains only
" PROCEDUREs and no system level code, it does not
" result in a real design.

" This example is intended to demonstrate PROCEDUREs.
" For real designs, the build in addition operator '.+.' can
" be used to perform addition at any width.

" This PROCEDURE implements a 1-bit adder. The inputs 'a'
" and 'b' are the signals to be added. The 'result' is the 1-bit
" result of the addition. There is also a 1-bit carry in and a
" 1-bit carry out.

"
```

```
PROCEDURE add1(INPUT a, b, carry_in; OUTPUT result, carry);
        result = a(+)b(+)carry_in;
        carry = a*b+a*carry_in+b*carry_in;
END add1;

" This PROCEDURE implements a 2-bit adder using the 1-bit
" adder above.  The inputs 'a' and 'b' are the 2-bit arrays to be
" added.  The 'result' is the 2-bit result of the addition.
" There is also a 1-bit carry in and a 1-bit carry out.

" Each call of add1 creates a separate instance of the logic
" described in add1.  The NODE is used to hold the carry out
" from addition of the low order bits.  It is used as the carry in
" to the addition of the high order bits.

PROCEDURE add2(INPUT a[2], b[2], carry_in; OUTPUT result[2],
carry);
        NODE low_carry;
        add1(a[0], b[0], carry_in, result[0], low_carry);
        add1(a[1], b[1], low_carry, result[1], carry);
END add2;

" This PROCEDURE implements a 4-bit adder using the 2-bit
" adder above.  The inputs 'a' and 'b' are the 4-bit arrays to be
" added.  The 'result' is the 4-bit result of the addition.  There
" is also a 1-bit carry in and a 1-bit carry out.

PROCEDURE add4(INPUT a[4], b[4], carry_in; OUTPUT result[4],
carry);
        NODE low_carry;
        add2(a[1..0], b[1..0], carry_in, result[1..0], low_carry);
        add2(a[3..2], b[3..2], low_carry, result[3..2], carry);
END add4;



" This PROCEDURE implements an 8-bit adder using the 4-
" bit adder above.  The inputs 'a' and 'b' are the 8-bit arrays to
" be added.  The 'result' is the 8-bit result of the addition.
```

```
" There is also a 1-bit carry in and a 1-bit carry out.
"

PROCEDURE add8(INPUT a[8], b[8], carry_in; OUTPUT result[8],
carry);
        NODE low_carry;

        add4(a[3..0], b[3..0], carry_in, result[3..0], low_carry);
        add4(a[7..4], b[7..4], low_carry, result[7..4], carry);
END add8;
```

Example 2: 1-Bit, 2-Bit, 4-Bit and 8-Bit Adder Functions

This example is very similar to the previous adder example except that it implements four adder FUNCTIONs instead of PROCEDUREs. The return value of each FUNCTION is an array 1 bit wider than the width of the arrays being added.

```
" ADDER2
" AMD

" This example is very similar to the previous adder example
" except it implements four adder FUNCTIONs instead of
" PROCEDUREs. The return value of each FUNCTION is an
" array 1 bit wider than the width of the arrays being added.
" This example is intended to demonstrate PROCEDUREs.
" For real designs, the built-in addition operator '.+.' can be
" used to perform addition at any width.

" This FUNCTION implements a 1-bit adder. The
" RETURN instruction makes the associated 2-bit array be
" the RETURN value of the FUNCTION

FUNCTION add1(a, b, carry_in)[2];
        RETURN [a*b+a*carry_in+b*carry_in, a(+)b(+)carry_in];
END add1;
```

```
" This FUNCTION implements a 2-bit adder.

FUNCTION add2(a[2], b[2], carry_in)[3];
        NODE low_carry, carry, result[2];

        [low_carry, result[0]] = add1(a[0], b[0], carry_in);
        [carry, result[1]] = add1(a[1], b[1], low_carry);
        RETURN [carry, result];
END add2;

" This FUNCTION implements a 4-bit adder.  Note that the
" groups being assigned to consist of a 1-bit signal and a 2-bit
" array reference which combine to make a 3-bit group.

FUNCTION add4(a[4], b[4], carry_in)[5];
        NODE low_carry, carry, result[4];

        [low_carry, result[1..0]] = add2(a[1..0],b[1..0],carry_in);
        [carry, result[3..2]] = add2(a[3..2],b[3..2],low_carry);
        RETURN [carry, result];
END add4;


" This FUNCTION implements an 8-bit adder.

FUNCTION add8(a[8], b[8], carry_in)[9];
        NODE low_carry, carry, result[8];

        [low_carry, result[3..0]] = add4(a[3..0],b[3..0],carry_in);
        [carry, result[7..4]] = add4(a[7..4],b[7..4],low_carry);
        RETURN [carry, result];
END add8;
```

Example 3: Combinatorial 4x4 Multiplier Function

This example is a combinatorial 4x4 multiplier implemented in a FUNCTION. It uses the 4-bit adder FUNCTION from the previous example.

```
" MULT1
" AMD

" This USE clause causes the definition of the 'add4'
" FUNCTION to be brought in from library 'adder2' (assume
" the previous adder FUNCTION example was in file
" 'adder2.src'). This is functionally the same as if the
" 'add4'FUNCTION were written here. However, USEing a
" PROCEDURE or FUNCTION compiles faster than writing it
" again since is has already been compiled.

USE 'adder2'.add4;

" 4-bit multiplier implemented with combinatorial logic.

FUNCTION mult(x[4], y[4])[8];
        " These arrays are used to hold the intermediate
        " results.The bit numbering corresponds to bit
        " positions in the 8-bit product.

        NODE temp0[4..0], temp1[5..1], temp2[6..2], temp3[7..3];

        " This implements a simple shift-add multiply scheme,
        " although the entire multiply is done combinatorially.

        IF y[0] THEN
                temp0 = [0, x];
        ELSE
                temp0 = 0;
        END IF;
        IF y[1] THEN
                temp1 = add4(temp0[4..1], x, 0);
        ELSE
                temp1 = [0, temp0[4..1]];
        END IF;
```

```
        IF y[2] THEN
                temp2 = add4(temp1[5..2], x, 0);
        ELSE
                temp2 = [0, temp1[5..2]];
        END IF;
        IF y[3] THEN
                temp3 = add4(temp2[6..3], x, 0);
        ELSE
                temp3 = [0, temp2[6..3]];
        END IF;
        RETURN [temp3, temp2[2], temp1[1], temp0[0]];
END mult;
```

Example 4: Combinatorial 4x4 Multiplier Functions

This example is another implementation of a 4x4 multiplier. It uses the built in addition operator.

```
" MULT2
" AMD

" This example is another implementation of a 4x4 multiplier.
" It uses the built in addition operator.

" 4-bit multiplier implemented with combinatorial logic.

FUNCTION mult1(a[1], b[1])[1];
        RETURN a*b;
END mult1;

FUNCTION mult2(a[2],b[2])[4];
        RETURN [0, 0, 0, mult1(a[0], b[0])]
          .+. [0, 0, mult1(a[1], b[0]), 0]
          .+. [0, 0, mult1(a[0], b[1]), 0]
          .+. [0, mult1(a[1], b[1]), 0, 0];
END mult2;
```

```
FUNCTION mult4(a[4],b[4])[8];
        RETURN [0, 0, 0, 0, mult2(a[1..0], b[1..0])]
          .+. [0, 0, mult2(a[3..2], b[1..0]), 0, 0]
          .+. [0, 0, mult2(a[1..0], b[3..2]), 0, 0]
          .+. [mult2(a[3..2], b[3..2]), 0, 0, 0, 0];
END mult4;
```

4-Bit ALU Example

The following example implements the 7C901 4-bit arithmetic logic unit. This example demonstrates a large real design that takes advantage of several of the constructs covered in the previous examples.

```
"AMD

" reg_file"

" This procedure implements the 16x4 register file with 2
" read ports and one write port.
"    reg_op -- operation to perform -- either no store (Nop) or
"              store b_in to addressed register (f_to_b)
"    a_addr -- address of the register that appears on the
"    a_out port
"    b_addr -- address of the regsiter on the a_out port and also
"              the address to write into from the b_in port
"    b_in   -- input port for writing data
"    cp     -- clock signal for the registers
"    a_out  -- a output port
"    b_out  -- b output port
"

MACRO Nop   0;
MACRO f_to_b 1;

PROCEDURE reg_file( INPUT reg_op, a_addr[4], b_addr[4], b_in[4],
                          cp;
   OUTPUT a_out[4], b_out[4] );

   NODE reg0[4], reg1[4], reg2[4], reg3[4] CLOCKED_BY cp;
   NODE reg4[4], reg5[4], reg6[4], reg7[4] CLOCKED_BY cp;
   NODE reg8[4], reg9[4], reg10[4], reg11[4] CLOCKED_BY cp;
   NODE reg12[4], reg13[4], reg14[4], reg15[4] CLOCKED_BY cp;

   CASE a_addr
     WHEN 0=> a_out = reg0;
```

```
    WHEN 1=> a_out = reg1;
    WHEN 2=> a_out = reg2;
    WHEN 3=> a_out = reg3;
    WHEN 4=> a_out = reg4;
    WHEN 5=> a_out = reg5;
    WHEN 6=> a_out = reg6;
    WHEN 7=> a_out = reg7;
    WHEN 8=> a_out = reg8;
    WHEN 9=> a_out = reg9;
    WHEN 10=> a_out = reg10;
    WHEN 11=> a_out = reg11;
    WHEN 12=> a_out = reg12;
    WHEN 13=> a_out = reg13;
    WHEN 14=> a_out = reg14;
    WHEN 15=> a_out = reg15;
  END CASE;

  CASE b_addr
    WHEN 0=> b_out = reg0;
    WHEN 1=> b_out = reg1;
    WHEN 2=> b_out = reg2;
    WHEN 3=> b_out = reg3;
    WHEN 4=> b_out = reg4;
    WHEN 5=> b_out = reg5;
    WHEN 6=> b_out = reg6;
    WHEN 7=> b_out = reg7;
    WHEN 8=> b_out = reg8;
    WHEN 9=> b_out = reg9;
    WHEN 10=> b_out = reg10;
    WHEN 11=> b_out = reg11;
    WHEN 12=> b_out = reg12;
    WHEN 13=> b_out = reg13;
    WHEN 14=> b_out = reg14;
    WHEN 15=> b_out = reg15;
  END CASE;

  IF reg_op = f_to_b THEN
    CASE b_addr
      WHEN 0=> [reg0..reg15] = [b_in,reg1..reg15];
      WHEN 1=> [reg0..reg15] = [reg0,b_in,reg2..reg15];
      WHEN 2=> [reg0..reg15] = [reg0,reg1,b_in,reg3..reg15];
```

```
        WHEN 3=> [reg0..reg15] = [reg0..reg2,b_in,reg4..reg15];
        WHEN 4=> [reg0..reg15] = [reg0..reg3,b_in,reg5..reg15];
        WHEN 5=> [reg0..reg15] = [reg0..reg4,b_in,reg6..reg15];
        WHEN 6=> [reg0..reg15] = [reg0..reg5,b_in,reg7..reg15];
        WHEN 7=> [reg0..reg15] = [reg0..reg6,b_in, reg8..reg15];
        WHEN 8=> [reg0..reg15] = [reg0..reg7,b_in,reg9..reg15];
        WHEN 9=> [reg0..reg15] = [reg0..reg8,b_in,reg10..reg15];
        WHEN 10=> [reg0..reg15] = [reg0..reg9,b_in,reg11..reg15];
        WHEN 11=> [reg0..reg15] =
                  [reg0..reg10,b_in,reg12..reg15];
        WHEN 12=> [reg0..reg15] =
                   [reg0..reg11,b_in,reg13..reg15];
        WHEN 13=> [reg0..reg15] =
                   [reg0..reg12,b_in,reg14..reg15];
        WHEN 14=> [reg0..reg15] = [reg0..reg13,b_in,reg15];
        WHEN 15=> [reg0..reg15] = [reg0..reg14,b_in];
      END CASE;
    ELSE " must be Nop
      [reg0..reg15] = [reg0..reg15];
    END IF;
END reg_file;

"
" The shifter appears in two places, as the input to the
" register file and as part of the Q-register loop.
"
"   dir  -- direction of shift:
"             Pass -- no shift
"             Up   -- shift left
"             Down -- shift right
"   in   -- 4-bit input value
"   in_l  -- low bit to shift in if left shift
"   in_h  -- high bit to shift in if right shift
"   out   -- 4-bit output value
"   out_l -- output of low bit if right shift
"   out_h -- output of high bit if left shift
"

MACRO Pass 0;
MACRO Up   1;
MACRO Down 2;
```

```
PROCEDURE shifter( INPUT dir[2], in[4], in_l, in_h;
  OUTPUT out[4], out_l, out_h );

  CASE dir
    WHEN Pass=>
      out = in;
      out_l = in_l;
      out_h = in_h;
    WHEN Up=>
      [out_h,out,out_l] = [in,in_l,in_l];
    WHEN Down=>
      [out_h,out,out_l] = [in_h,in_h,in];
  ELSE
    [out_h,out,out_l] = [in_h,in,in_l];
  END CASE;
END shifter;

"
" The Q-register is a temporary 4-bit register
"
"  cp    -- clock signal for registers
"  q_op  -- operation select
"           Nop   -- no operation on Q
"           q_to_q -- update Q from Q via the shifter
"           f_to_q -- store the f input into Q
"  f     -- 4-bit input register
"  q_in  -- 4-bit input from the Q shifter
"  q_out -- 4-bit output to select mux and Q shifter
"

" MACRO Nop  0;
MACRO q_to_q  1;
MACRO f_to_q  2;

PROCEDURE Q_reg( INPUT cp, q_op[2], f[4], q_in[4]; OUTPUT q_out[4] );
```

```
    NODE q[4] CLOCKED_BY cp;

    q_out = q;
    CASE q_op
      WHEN f_to_q=>  q = f;
      WHEN q_to_q=>  q = q_in;
    ELSE
      q = q;
    END CASE;

END Q_reg;

"
" Alu data sel selects the data for the alu A and B inputs
"
"   sel    -- select input, one of the following=>
"              AQ --  r <- A, s <- Q
"              AB --  r <- A, s <- B
"              ZQ --  r <- 0, s <- Q
"              ZB --  r <- 0, s <- B
"              ZA --  r <- 0, s <- A
"              DA --  r <- D, s <- A
"              DQ --  r <- D, s <- Q
"              DZ --  r <- D, s <- 0
"   a      -- 4-bit a input
"   b      -- 4-bit b input
"   q      -- 4-bit q input
"   d      -- 4-bit d input
"   r      -- 4-bit r output
"   s      -- 4-bit s output
"

MACRO AQ 0;
MACRO AB 1;
MACRO ZQ 2;
MACRO ZB 3;
MACRO ZA 4;
MACRO DA 5;
MACRO DQ 6;
MACRO DZ 7;
```

```
PROCEDURE Alu_data_sel( INPUT sel[3], a[4], b[4], q[4], d[4];
  OUTPUT r[4], s[4] );

  CASE sel
    WHEN AQ=> r = a; s = q;
    WHEN AB=> r = a; s = b;
    WHEN ZQ=> r = 0; s = q;
    WHEN ZB=> r = 0; s = b;
    WHEN ZA=> r = 0; s = a;
    WHEN DA=> r = d; s = a;
    WHEN DQ=> r = d; s = q;
    WHEN DZ=> r = d; s = 0;
  END CASE;
END Alu_data_sel;

"
" Alu implements a 4-bit 8-function alu
"
"  Alu opcodes:
"
"   xADD   ADD    r+s
"   xSUBR  SUBR   r-s
"   xSUBS  SUBS   s-r
"   xOR    OR     r | s
"   xAND   AND    r & s
"   xNOTRS NOTRS  ~r & s
"   xEXOR  EXOR   r ^ s
"   xEXNOR EXNOR  ~(r ^ s)
"

MACRO xADD  0;
MACRO xSUBR 1;
MACRO xSUBS 2;
MACRO xOR   3;
MACRO xAND  4;
MACRO xNOTRS 5;
MACRO xEXOR 6;
MACRO xEXNOR 7;

"
" add_op -- implement the add operator
```

```
"
"   Cin   -- carry in
"   r     -- 4-bit r input
"   s     -- 4-bit s input
"   f     -- 4-bit sum
"   Cout  -- Carry out
"   G     -- generate output
"   P     -- propagate output
"   Ov    -- overflow output
"

PROCEDURE add_op( INPUT Cin, r[4], s[4];
   OUTPUT f[4], Cout, G, P, Ov );

   NODE g0,g1,g2,g3;
   NODE p0,p1,p2,p3;
   NODE c4,c3,c2,c1;
   NODE f3,f2,f1,f0;
   NODE gx, px, o;

   g0 = r[0]*s[0]; g1 = r[1]*s[1]; g2 = r[2]*s[2]; g3 = r[3]*s[3];
   p0 = r[0]+s[0]; p1 = r[1]+s[1]; p2 = r[2]+s[2]; p3 = r[3]+s[3];
   c1 = r[0]*s[0] + r[0]*Cin + s[0]*Cin;
   c2 = r[1]*s[1] + r[1]*c1 + s[1]*c1;
   c3 = r[2]*s[2] + r[2]*c2 + s[2]*c2;
   c4 = r[3]*s[3] + r[3]*c3 + s[3]*c3;
   f0 = r[0] (+) s[0] (+) Cin;
   f1 = r[1] (+) s[1] (+) c1;
   f2 = r[2] (+) s[2] (+) c2;
   f3 = r[3] (+) s[3] (+) c3;

   f = [f3,f2,f1,f0];
   gx = /+(g3,p3*g2,p3*p2*g1,p3*p2*p1*g0);
   px = /*(p3,p2,p1,p0);
   Cout = c4;
   o = c3 (+) c4;
   G = gx;
   P = px;
   Ov = o;
END add_op;
```

```
"
" or_op -- implement the logical or operator
"
"   Cin   -- carry in
"   r     -- 4-bit r input
"   s     -- 4-bit s input
"   f     -- 4-bit logical sum
"   Cout  -- Carry out
"   G     -- generate output
"   P     -- propagate output
"   Ov    -- overflow output
"

PROCEDURE or_op( INPUT Cin, r[4], s[4];
  OUTPUT f[4], Cout, G, P, Ov );

  NODE p0,p1,p2,p3;
  NODE f3,f2,f1,f0;
  NODE gx, px, o, c;

  p0 = r[0]+s[0]; p1 = r[1]+s[1]; p2 = r[2]+s[2]; p3 = r[3]+s[3];
  f0 = r[0] + s[0];
  f1 = r[1] + s[1];
  f2 = r[2] + s[2];
  f3 = r[3] + s[3];

  f = [f3,f2,f1,f0];
  gx = p3*p2*p1*p0;
  px = 0;
  c = /*(p3,p2,p1,p0) + Cin;
  o = /*(p3,p2,p1,p0) + Cin;
  Cout = c;
  G = gx;
  P = px;
  Ov = o;
END or_op;

"
" and_op -- implement the logical and operator
"
"   Cin   -- carry in
```

```
"   r     -- 4-bit r input
"   s     -- 4-bit s input
"   f     -- 4-bit logical product
"   Cout  -- Carry out
"   G     -- generate output
"   P     -- propagate output
"   Ov    -- overflow output
"

PROCEDURE and_op( INPUT Cin, r[4], s[4];
   OUTPUT f[4], Cout, G, P, Ov );

   NODE g0,g1,g2,g3;
   NODE f3,f2,f1,f0;
   NODE gx, px, o, c;

   g0 = r[0]*s[0]; g1 = r[1]*s[1]; g2 = r[2]*s[2]; g3 = r[3]*s[3];
   f0 = r[0] * s[0];
   f1 = r[1] * s[1];
   f2 = r[2] * s[2];
   f3 = r[3] * s[3];

   f = [f3,f2,f1,f0];
   gx = /(g0 + g1 + g2 + g3);
   px = 0;
   c = g3 + g2 + g1 + g0 + Cin;
   o = g3 + g2 + g1 + g0 + Cin;
   G = gx;
   P = px;
   Ov = o;
   Cout = c;
END and_op;

"
" xnor_op -- implement the logical xnor (equivalence) operator
"
"   Cin   -- carry in
"   r     -- 4-bit r input
"   s     -- 4-bit s input
"   f     -- 4-bit logical equivalence
"   Cout  -- Carry out
```

```
"   G     -- generate output
"   P     -- propagate output
"   Ov    -- overflow output
"

PROCEDURE xnor_op( INPUT Cin, r[4], s[4];
   OUTPUT f[4], Cout, G, P, Ov );

   NODE g0,g1,g2,g3;
   NODE p0,p1,p2,p3;
   NODE f3,f2,f1,f0;
   NODE ov1, ov2;
   NODE gx, px, o, c;

   g0 = r[0]*s[0]; g1 = r[1]*s[1]; g2 = r[2]*s[2]; g3 = r[3]*s[3];
   p0 = r[0]+s[0]; p1 = r[1]+s[1]; p2 = r[2]+s[2]; p3 = r[3]+s[3];
   f0 = /(r[0] (+) s[0]);
   f1 = /(r[1] (+) s[1]);
   f2 = /(r[2] (+) s[2]);
   f3 = /(r[3] (+) s[3]);

   f = [f3,f2,f1,f0];
   gx = g3 + p3*g2 + p3*p2*g1 + p3*p2*p1*g0;
   px = g3 + g2 + g1 + g0;
   c = /+( g3, p3*g2, p3*p2*g1, p3*p2*p1*p0*(g0+Cin) );
   ov1 = p2 + g2*p1 + /g2*/g1*/p0 + /g2*/g1*/g0*Cin;
   ov2 = /p3 + /g3*/p2 + /g3*/g2*/p1 + /g3*/g2*/g1*/p0 +
         /g3*/g2*/g1*/g0*Cin;
   o = ov1 (+) ov2;
   G = gx;
   P = px;
   Ov = o;
   Cout = c;
END xnor_op;

"
" alu -- implement alu
"
"  op    -- operation to perform -- see opcodes above
"  Cin   -- carry in
"  r     -- 4-bit r input
```

```
"  s     -- 4-bit s input
"  f     -- 4-bit function result output
"  Cout  -- carry out
"  G_    -- generate output
"  P_    -- propagate output
"  sign  -- sign of the result -- f[3]
"  Ov    -- overflow output
"  Zero  -- asserted if f[0]..f[3] all zero
"

PROCEDURE alu( INPUT op[3], Cin, r[4], s[4];
   OUTPUT f[4], Cout, G_, P_, sign, Ov, Zero );

   NODE fx[4];
   NODE g, p, o;

   CASE op
      WHEN xADD=>   add_op( Cin, r, s, fx, Cout, g, p, o );
      WHEN xSUBR=>  add_op( Cin, r, /s, fx, Cout, g, p, o );
      WHEN xSUBS=>  add_op( Cin, /r, s, fx, Cout, g, p, o );
      WHEN xOR=>    or_op( Cin, r, s, fx, Cout, g, p, o );
      WHEN xAND=>   and_op( Cin, r, s, fx, Cout, g, p, o );
      WHEN xNOTRS=> and_op( Cin, /r, s, fx, Cout, g, p, o );
      WHEN xEXOR=>  xnor_op( Cin, /r, s, fx, Cout, g, p, o );
      WHEN xEXNOR=> xnor_op( Cin, r, s, fx, Cout, g, p, o );
   END CASE;

   sign = fx[3];
   Zero = /(fx[3]+fx[2]+fx[1]+fx[0]);
   f = fx;
   G_ = g;
   P_ = p;
   Ov = o;

END alu;

"
" Output Data Selector
"
"   outsel  -- output selection:
"               a_to_y => y <- a
```

```
"              f_to_y => y <- f
"   a       -- 4-bit a input
"   f       -- 4-bit f input
"   y       -- 4-bit y output
"

MACRO a_to_y 0;
MACRO f_to_y 1;

PROCEDURE Out_select( INPUT outsel, a[4], f[4]; OUTPUT y[4] );

   CASE outsel
      WHEN a_to_y=> y = a;
      WHEN f_to_y=> y = f;
   END CASE;
END Out_select;

"
" Destination decode
"
"  Decode the destination information and generate control signals
"  for the various components related to destination control.
"
"
"  op     -- destination opcode:
"              xQREG    move f to q and y, no store or shift
"              xNOP     move f to y, no store or shift
"              xRAMA    move a to y, store a in reg_file, no shift, Q nop
"              xRAMF    move f to y, store f in reg_file, no shift, Q nop
"              xRAMQD   f to reg_file shifted right, and Q shifted right
"              xRAMD    f to reg_file shifted right, Q nop
"              xRAMQU   f to reg_file shifted left, and Q shifted left
"              xRAMU    f to reg_file shifted left, Q nop
"  Rop    -- reg_file control signal
"  Rshift -- reg_file shifter direction control
"  Qop    -- Q register control signals
"  Qshift -- Q register shifter direction control
"  Yop    -- Y output mux selection control
"

MACRO xQREG  0;
```

```
MACRO xNOP  1;
MACRO xRAMA  2;
MACRO xRAMF  3;
MACRO xRAMQD 4;
MACRO xRAMD  5;
MACRO xRAMQU 6;
MACRO xRAMU  7;

PROCEDURE DestDecode(INPUT op[3];
                     OUTPUT Rop, Rshift[2], Qop[2], Qshift[2], Yop);
  TRUTH_TABLE
     op    ::   Rop,  Rshift,     Qop,  Qshift,    Yop;
    xQREG  ::   Nop,   Pass,    f_to_q,   Pass,  f_to_y;
    xNOP   ::   Nop,   Pass,     Nop,     Pass,  f_to_y;
    xRAMA  :: f_to_b,  Pass,     Nop,     Pass,  a_to_y;
    xRAMF  :: f_to_b,  Pass,     Nop,     Pass,  f_to_y;
    xRAMQD :: f_to_b,  Down,    q_to_q,   Down,  f_to_y;
    xRAMD  :: f_to_b,  Down,     Nop,     Pass,  f_to_y;
    xRAMQU :: f_to_b,   Up,     q_to_q,    Up,   f_to_y;
    xRAMU  :: f_to_b,   Up,      Nop,     Pass,  f_to_y;
  END TRUTH_TABLE;
END DestDecode;

"
" The following instantiations connect the components together to
" form the 7C901
"
"   uop     -- 901 opcode:
"              ------------------------------------------
"              | i8 | i7 | i6 | i5 | i4 | i3 | i2 | i1 | i0 |
"              ------------------------------------------
"              |          |          |          |
"              | dst control  | alu function |  alu source  |
"
"   a_addr  -- reg_file a address (a output)
"   b_addr  -- reg_file b address (b output and b_in store)
"   Rin0    -- reg_file shifter low-order in bit
"   Rin3    -- reg_file shifter high-order in bit
"   Qin0    -- Q register low-order in bit
"   Qin3    -- Q register high_order in bit
"   Cin     -- carry in from previous stage
```

```
"   OE      -- Y-output three-state control
"   d       -- direct data input (4-bits)
"   cp      -- synchronous clocking signal
"   y_out   -- 4-bit output (three-state controlled)
"   Rout0   -- reg_file shifter low-order output bit
"   Rout3   -- reg_file shifter high-order output bit
"   Qout0   -- Q register shifter low-order output bit
"   Qout3   -- Q register shifter high-order output bit
"   G_      -- carry generate output
"   P_      -- carry propagate output
"   Cout    -- carry out
"   Sign    -- sign of the operation result (f[3])
"   Ov      -- overflow result
"   Zero    -- operation result is zero
"
" Note 1: in the device, the various shift in and out bits are combined
"   on bidirectional pins, if that were desired here, we could
"   create biput pins and connect the pins from the component to
"   the appropriate connections on the biputs.
"
" Note 2: Some of the signals on the device are specified as low-true.
"   If this component were to be completely internal to a device,
"   this would not matter, but if this were to be the only component
"   in a package and the pins were to have the same polarity,
"   at the top level low-true signals can be declared for the
"   pins and the internally high-true signals connected to them,
"   and the external package will show the correct behavior.
"

PROCEDURE CY7C901( INPUT uop[9], a_addr[4], b_addr[4], Rin0, Rin3,
                       Qin0, Qin3, Cin, OE, d[4], cp;
                   OUTPUT y_out[4], Rout0, Rout3, Qout0, Qout3,
                       G_, P_, Cout, Sign, Ov, Zero);

  NODE B_in[4], Qin[4];
  NODE f[4], a[4], b[4], q[4], r[4], s[4];
  NODE Rop, Rshift[2], Qop[2], Qshift[2], Yop;
  NODE y[4] ENABLED_BY oe;

  Shifter( Rshift, f, Rin0, Rin3, B_in, Rout0, Rout3 );
  Reg_File( Rop, a_addr, b_addr, B_in, cp, a, b );
```

```
    Q_reg( cp, Qop, f, Qin, q );
    Shifter( Qshift, q, Qin0, Qin3, Qin, Qout0, Qout3 );
    Alu_data_sel( uop[2..0], a, b, q, d, r, s );
    Alu( uop[5..3], Cin, r, s, f, Cout, G_, P_, sign, Ov, Zero );
    Out_select( Yop, a, f, y );
    y_out = y;
    DestDecode( uop[8..6], Rop, Rshift, Qop, Qshift, Yop );

END CY7C901;


INPUT uop[9], a_addr[4], b_addr[4], Rin0, Rin3, Qin0, Qin3, Cin,        OE,
d[4], cp;
OUTPUT y_out[4], Rout0, Rout3, Qout0, Qout3, G_, P_, Cout, Sign,        Ov,
Zero;

CY7C901(uop, a_addr, b_addr, Rin0, Rin3, Qin0, Qin3, Cin, OE, d,        cp,
y_out, Rout0, Rout3, Qout0, Qout3, G_, P_, Cout, Sign,  Ov, Zero);
```

C MACHXL Warning and Error Messages

Contents

Introduction

This appendix is an alphabetical listing of errors and warnings used by MACHXL during compiling, partitioning and optimizing, along with an explanation of each. A listing of any errors in a design can be found in *filename.err*.

'SYMBOL_NAME' cannot be LOW_TRUE.

Only INPUTs, OUTPUTs, and NODEs can be declared to be LOW_TRUE.

'SYMBOL_NAME' cannot be a flip-flop.

Only OUTPUTs and NODEs can be declared as flip flops.

'SYMBOL_NAME' cannot have RESET_BY or PRESET_BY without CLOCKED_BY.

Only a clocked register can be given RESET_BY and PRESET_BY constructs. To create an SR_LATCH make the CLOCKED_BY expression = 0;

'SYMBOL_NAME' cannot have a DEFAULT_TO.

Only OUTPUTs and NODEs can be given a DEFAULT_TO.

'SYMBOL_NAME' cannot have control information.

Only OUTPUTs and NODEs can be given CLOCKED_BY, ENABLED_BY, RESET_BY, or PRESET_BY.

'SYMBOL_NAME' cannot have two DEFAULT_TO expressions.

Only JK_FLOPs and SR_FLOPs can be given two DEFAULT_TO expressions.

'SYMBOL_NAME' has CLOCK_ENABLED_BY without CLOCKED_BY.

'SYMBOL_NAME' needs two DEFAULT_TO expressions.

JK_FLOPs and SR_FLOPs must be given two DEFAULT_TO expressions separated by a comma. The first expression is for the J or S. The second expression is for the K or R.

.pi target device did not pass plscan. Please run plscan.

A BIT_WIDTH bit number cannot be represented in BIT_WIDTH bits.

A WHEN clause needs a GOTO in STATE 'STATE_NAME'.

An example that would produce this warning is:

```
CASE a
  WHEN 1:
    GOTO st1;
  WHEN 2:
    x = 1;
  ELSE
    GOTO st2;
END CASE;
```

The above code says to go to `st1` when `a` is 1 and to go to `st2` when `a` is not 1 or 2, but it doesn't say where to go to when `a` is 2. The compiler will fall back on the DEFAULT_TO values of the state bits to determine the state transition when `a` is 2.

Access denied to product

Array 'SYMBOL_NAME' cannot be forward referenced.

Array 'SYMBOL_NAME' cannot have >VALUE members.

Array 'SYMBOL_NAME' cannot have zero size.

Array 'SYMBOL_NAME' was not renamed since an element clashes with another symbol name

The period character is not a valid symbol in the HDL. Internally generated names may contain periods however. In attempting to recreate a *.src* file, all periods are converted to underscores. In this design, renaming the specified array caused a conflict between one of the array elements and an existing signal name. For this reason, the array was NOT renamed.

Array index VALUE out of range.

Only elements within the range of the array declared in the .src file may be referenced in the pi file. An array declared without a left bound will be zero-based (e.g. OUTPUT `a[4]` will have the elements `a[0]`, `a[1]`, `a[2]`, `a[3]`). An array declared with left and right bounds will have elements between and including the bounds (e.g.OUTPUT `a[1..4]` will have the elements `a[1]`, `a[2]`, `a[3]`, and `a[4]`).

Assuming 'DEFAULT_AVAILABLE_FILE' as the available file.

At TIME ns: Expected value 'SIM_VALUE' does not match pin value 'SIM_VALUE' for signal SYMBOL_NAME.

At TIME ns: both R and S set on RS flip flop SYMBOL_NAME -- result unknown.

At TIME ns: both preset and reset set for SYMBOL_NAME -- result unknown.

At TIME ns: more than one message defined during this step.

At TIME ns: signal SYMBOL_NAME is unstable.

At TIME ns: signal value for SYMBOL_NAME in expression is Z.

The current value associated with the named signal is Z, and the signal is being used as an input to another function. In this situation, the value of the signal is assumed to be X.

Attempting to find any device pin that can fit the following signals:

The fitter will attempt to find AT LEAST ONE place a signal can fit.

Attempting to find at least one part that can fit any output signal.

The fitter will attempt screen the parts which cannot fit ANY signals. If no parts can fit any signals, then this is a fatal error.

Attempting to fit a reduced partition.

Identifies an attempt to fit into an AMD Mach device after removing one or more functions from the prior fit attempt. The fitter may repeat fitting attempts at reduced partitions until a fit is achieved.

Attempting to fit at UTILIZATION_VALUE percent utilization.

Identifies an attempt to fit into an AMD Mach device at the indicated utilization. The fitter may repeat fitting attempts at lower utilizations until a fit is achieved.

BLOWN and INTACT are not allowed at the global level.

Fuse assignments may only be given inside a fixed group.

BLOWN and INTACT are not allowed in subgroups.

Fuse assignments may appear in a fixed group, but not in subgroups of a fixed group, fixed subgroups of a fixed group, or at the global level.

BLOWN and INTACT fuse lists overlap.

A fuse cannot be assigned to be both blown and intact.

Bad .afb file.

The .afb file is unreadable to the optimizer. Remove the .afb file and rerun plcomp to regenerate it.

Bad database version in line 1.

Bad file format in 'FILE_NAME.afb'.

The .afb file has become corrupted. Remove it and recompile the .src file to recreate the .afb file.

Bad file format in .afb file.

The .afb file has become corrupted. Remove it and recompile the .src file to recreate the .afb file.

Bad flip-flop type in .fb file.

Bad mode 'SYMBOL_NAME' in set_db_access_mode().

Biput instance 'INSTANCE_NAME' not driven by a tri-state device.

All biput ports are required to be driven by a tri-state device.

Biputs-as-inputs exceed pal block limits.

The sum of inputs and outputs/biputs exceed the device/pal block limits.

Build of SYMBOL_NAME is complete.

CHECK 'LOG_FILE_NAME' FOR ERRORS.

CHECK 'LOG_FILE_NAME' FOR WARNINGS.

Can only multiply, divide, and modulo with constants.

Can only use LATCHED_BY with D_LATCH.

Can't assign INPUT signal SYMBOL_NAME to OUTPUT pin PIN_NAME on device DEVICE_NAME
An attempt was made to place an INPUT signal on an OUTPUT pin

Can't assign signal SYMBOL_NAME to NC pin PIN_NAME on device DEVICE_NAME
An attempt was made to assign a signal to a NO_CONNECT pin of the device.

Can't assign signal SYMBOL_NAME to RESET pin PIN_NAME on device DEVICE_NAME

Can't assign signal SYMBOL_NAME to ground pin PIN_NAME on device DEVICE_NAME
An attempt was made to assign a signal to a GND pin of the device.

Can't assign signal SYMBOL_NAME to hidden pin PIN_NAME on device DEVICE_NAME

Can't assign signal SYMBOL_NAME to power pin PIN_NAME on device DEVICE_NAME
An attempt was made to assign a signal to a POWER pin of the device.

Can't open FILE_TYPE file FILE_NAME

Can't open NPI file FILE_NAME

Can't open error file ERROR_FILE_NAME

Can't open file SYMBOL_NAME

Can't open group file FILENAME

In attempting to generate a .npi file, the back annotation software could not open the specified file for writing. Check existing file and directory permissions.

Can't open manufacturer info database DATABASE_NAME

Can't open new "src" file 'FILENAME' for writing

When using the PLDoc option to generate the reduced equations in *.src* HDL format, the file *design.src* is created to hold the information. When PLDoc attempted to open this file in write mode, it was unable. Most probable cause of the error is file or directory permissions.

Cannot RETURN .Z. from FUNCTION.

Cannot assign 'SIGNAL_NAME' as an input signal on the output pin 'PIN_NAME'.

A signal given as an input signal in the pi file may not be assigned to an output pin.

Cannot assign 'SIGNAL_NAME' as an output signal on the non-output pin 'PIN_NAME'.

A signal given as an output signal in the pi file may not be assigned to a non-output pin; i.e. a pin that is not an output pin, biput pin, or a node.

Cannot assign 'SIGNAL_NAME' to ground pin 'PIN_NAME' of template 'SYMBOL_NAME'.

No signals may be assigned to ground pins of the device.

Cannot assign 'SIGNAL_NAME' to no-connect pin 'PIN_NAME' of template 'SYMBOL_NAME'.

No signals may be assigned to pins of the device that are not connected.

Cannot assign 'SIGNAL_NAME' to power pin 'PIN_NAME' of template 'SYMBOL_NAME'.

No signals may be assigned to power pins of the device.

Cannot assign .C., .S., .X., or .Z. to VAR 'SYMBOL_NAME'.

Cannot assign .S. or .X. to INPUT 'SYMBOL_NAME'.

Cannot assign .Z. to 'SYMBOL_NAME'.

The .Z. can only be assigned to OUTPUTs and NODEs since only these can have an ENABLED_BY expression.

Cannot assign clock enable

Cannot assign the clock enable equation for this output.

Cannot assign clock to register

Cannot assign the clock expression for this output to this macrocell.

Cannot assign multiple array or range signals to a single pin.

Only one signal can be assigned at a time to a pin. A signal declared as an array in the .src file implies multiple signal array elements in the pi file, so a single symbol assigned to a pin may be an illegal assignment if the symbol is declared as an array.

Cannot assign output enable

Cannot assign the output enable expression for this output to this macrocell.

Cannot assign preset

Cannot assign the preset equation for this output.

Cannot assign reset

Cannot assign the reset equation for this output.

Cannot assign to 'SYMBOL_NAME'.

Cannot determine default footprint of template 'SYMBOL_NAME'.

Cannot find 'SYMBOL_NAME' in 'FILE_NAME'.

This means that a USE construct has specified a FUNCTION or PROCEDURE that cannot be found in the given file.

Cannot fit accumulated inputs on the device

The output signals in a group from the PI file require a certain set of inputs. The complete set of inputs, however, could not be fit.

Cannot fit auxiliary signals needed for fixed signal group

After fitting the original group of required signals from a group in the PI file, other output signals were needed on this device. These other output signals must be fit either as output or brought in on input pins. The limits of this device, however, prevented the fitter from resolving the need for these auxiliary outputs.

Cannot fit signal group due to fixed output signals

The output signals, unassigned to a pin, from a group in the PI file could NOT all be fit on this device.

Cannot give state values to ALGORITHM_TYPE state machine 'STATE_MACHINE_NAME'.

A state machine with a specified STATE_VALUES algorithm cannot have the state value specified for individual states since the specified STATE_VALUES algorithm will assign state values to all states.

Cannot have DONT CARE digits in this constant.

Cannot have NO_REDUCE on VIRTUAL 'SYMBOL_NAME'.

Cannot have OUTPUT parameters in FUNCTION 'FUNCTION_NAME'.

Cannot have a HIDDEN INPUT.

Cannot have more than MAX_MACRO_PARAMS macro parameters.

Cannot index non-array 'SYMBOL_NAME'.

Cannot invoke FUNCTION_OR_PROCEDURE 'SYMBOL_NAME' as a PROCEDURE_OR_FUNCTION.

Cannot make high-true output

The macrocell does not have the capability to assign a high-true output signal

to this pin.

Cannot make low-true output

The macrocell does not have the capability to assign a low-true output signal to this pin.

Cannot make output combinatorial

The output macrocell cannot be made combinatorial. An example of a macrocell that cannot be combinatorial is an output of the P16R8.

Cannot make output hidden

The hidden node of this device cannot accept a node signal, or a signal which for some reason must be visible.

Cannot make output registered

The output macrocell cannot be made registered. An example of a macrocell that cannot be registered is an output of the P16L8.

Cannot negate a constant.

Cannot open 'SYMBOL_NAME' for error output.

Cannot open 'SYMBOL_NAME'.

Cannot open .log file 'SYMBOL_NAME' for writing.

Cannot open EDIF 2.0.0 file 'EDIF_FILE'

Cannot open simulation listing file 'FILE_NAME'.

Cannot open test vector file 'FILE_NAME'.

Cannot operate on number with DONT CAREs.

Cannot reference WIRED_BUS member 'SYMBOL_NAME'.

Any signal feeding a WIRED_BUS cannot be referenced. Only the WIRED_BUS signal itself can be referenced. This WIRED_BUS signal and all signals feeding the WIRED_BUS will have the same value since they are wired together.

Cannot represent all STATE values with given STATE_BITS.

This occurs when the state value given to a STATE is to large to be represented with the number of bits given in the STATE_BITS construct.

Cannot resolve OE requirements of macrocells.

The fitter cannot satisfy pal block output enable requirements.

Cannot set up XOR

There are not XOR rows available on this pin to fit this signal.

Cannot set up feedback

The feedback required to properly fit this signal is not available.

Cannot use .Z. in this context.

Cannot use DONT CARE digits in this constant.

Cannot use group in this context.

Character 'OPTION_DELIMITER' used in filename.

Clock pin needed for clock.

An input is assigned to a clock pin that must be reserved for a clock signal.

Collapsing 'SYMBOL_NAME'.

Collapsing declared VIRTUALs.

This is the optimization phase where signals declared to be VIRTUAL in the .src file or in the .pi file are collapsed.

Collapsing equations.

This is the optimization phase where NODEs are either collapsed or made into PHYSICAL NODEs. This is controlled by the optimizer equation properties specified in the .pi file.

Combinatorial feedback collapsing 'SYMBOL_NAME'.

The collapsing of signals involved in combinatorial feedback is postponed until this phase.

Combinatorial signal 'SYMBOL_NAME' cannot DEFAULT_TO LAST_VALUE.

Only registers can have DEFAULT_TO LAST_VALUE. Only registers hold a value from the previous clock cycle to default to.

Conflict between properties 'PROPERTY_NAME' and 'PROPERTY_NAME'.

The two properties' definitions or effects conflict, so the properties cannot both be given on the same signal, in the same group, in the same fixed group, or at the global level.

Constant too large.

Copying default .pi file 'DEFAULT_PI_FILE_NAME' to 'PI_FILE_NAME'.

There is no physical information file for your specific design, so the system will attempt to use a default physical information file.

Copying default cost file 'SYMBOL_NAME' to 'SYMBOL_NAME'.

Cost VALUE larger than 1000.

Could not copy default pi file 'PI_FILE_NAME' to file 'PI_FILE_NAME'.

The system attempted to use the default physical information file, but was unable to copy the default pi file to a pi file for your specific design. Read permissions on the default pi file or write permissions on the current directory may have been denied.

Could not find default pi file 'PI_FILE_NAME'.

There is no physical information file for your specific design, so the system attempted to use a default physical information file, but was unable to find the file.

Could not open 'SYMBOL_NAME' file

Could not open available file - SYMBOL_NAME

Could not open default .cst file 'SYMBOL_NAME'.

Could not open device library: SYMBOL_NAME

Could not open file 'FILENAME' for reading.

Could not open part library: SYMBOL_NAME

Could not open text file: SYMBOL_NAME

D_LATCH 'SYMBOL_NAME' needs LATCHED_BY expression.

Declaration has low true symbol '/' in addition to LOW_TRUE keyword.

Design SYMBOL_NAME is up to date.

Design SYMBOL_NAME not found

Design SYMBOL_NAME not found.

Design construct not followed by CELLREF design name

Design has no input ports defined.

Design has no nets.

Design has no output ports defined.

Device 'DEVICE_NAME' - I/O PAD count exceeds device limit of NUMPINS

The total number of INPUT/OUTPUT signals to be placed on the device exceeds the total number of usable pins on the device

Device 'DEVICE_NAME' is not targeted in the PI file, and is therefore unusable.

This device architecture must be the target of a PI file fixed group to be considered as part of a solution. Since it is not targeted, this device architecture is thrown from the list of possible devices.

Device DEVICENAME used for annotation -- update .pi accordingly

The back annotation software found a different device in the post place and route netlist file than in the original pi file.

Device file missing: SYMBOL_NAME (SYMBOL_NAME)

Device library SYMBOL_NAME not found

Different suffixes on identifiers with '..' operator.

Directory SYMBOL_NAME not found

Divide by zero.

Division by 0.

Drawfield - illegal field number VALUE

Due to the above warnings, the PI and NPI files may be out of synch

Due to a mismatch between the original pi file and what actually happened during place and route, the PI and NPI files are most likely out of synch. To assure the pin placement the next time through, manually merge the pi and npi files.

Duplicate STATE name 'STATE_NAME'.

Duplicate WHEN values.

The CASE statement has the same value in two different WHEN clauses.

Duplicate value for STATE 'STATE_NAME'.

The STATE_MACHINE has two different STATES with overlapping state values. A dont care digit will overlap with any other digit value.

EDIF 2.0.0 file 'EDIF_FILE' empty

EDIF 2.0.0 file 'EDIF_FILE' is incomplete

ELSE of CASE needs a GOTO in STATE 'STATE_NAME'.

An example that would produce this warning is:

```
CASE a
   WHEN 1:
     GOTO st1;
   WHEN 2:
     GOTO st2;
 END CASE;
```

The above code says to go to st1 when 'a' is 1 and to go to st2 when 'a' is 2, but it doesn't say where to go to when 'a' has some other value. The compiler will fall back on the DEFAULT_TO values of the state bits to determine the state transition for other values of 'a'.

END name 'SYMBOL_NAME' does not match 'SYMBOL_NAME'.

END name 'SYMBOL_NAME' does not match STATE_MACHINE name 'SYMBOL_NAME'.

Edit aborted

Edit canceled

Encountered COMP_ON without preceding COMP_OFF.

Equation reduction.

Equation too large for symbol 'SYMBOL_NAME'.

This indicates that the design was written in a way that caused an equation to exceed the internal equation size limit. The design should

probably be modified to use NODEs to hold intermediate equation values to avoid generating such large equations. There are options available to raise the internal equation size limit and to have the compiler automatically generate NODEs for some equations.

Equation too large.

Error in cost file SYMBOL_NAME.

Error opening source file 'HDL_SOURCE_FILE'.

PLSchematic was unable to open the output source file.

Error writing database SYMBOL_NAME in replace_index_maker

Error writing database SYMBOL_NAME in replace_maker

Errors found by the pterm string parser.

Errors that must be corrected were found while parsing the pterm string.

Errors found in netlist unable to continue

This means that serious and/or fatal errors were encountered in the input netlist. Processing is discontinued and no output files will be generated.

Errors found in pin assignments.

Errors were found in the pin assignments in the physical information file.

Errors found while comprehending fixed groups.

Errors that must be corrected were found while the system was understanding the fixed groups in the physical information file.

Errors found while creating internal feedback groups.

Errors that must be corrected were found while the system was creating groups of signals that depend on internal feedback.

Errors found while verifying available devices.

Errors in simulation vectors, error vectors ignored

Errors occurred while building the output and equation lists.

Errors occurred while comprehending the fit_with properties.

Errors that must be corrected were found while implementing therequirements of the FIT_WITH property.

Exceeded maximum of VALUE range identifiers.

Exceeds pal block enable limit.

The combined pin assignments exceed the number of enable terms for a PAL block.

Exceeds pal block pterm allocation capabilities.

The combined pin assignments exceed the ability to assign product terms.

Expecting '(' for parameter list of MACRO 'MACRO_NAME'.

Expecting '(expression list)' after unary 'OPERATOR'.

Expecting '.'.

Expecting ':' after STATE value.

Expecting ':' after WHEN number.

Expecting ':' or '[value]:' after STATE name.

Expecting ';'.

Expecting '=' after 'FOR VAR_NAME'.

Expecting '=' in INITIAL statement.

Expecting '=' in SET statement.

Expecting '=' to follow assignment expression.

Expecting '=>' after WHEN number.

Expecting ']'.

Expecting '}'.

Expecting argument name for MACRO 'MACRO_NAME'.

Expecting array name or '[signals]' to follow STATE_BITS.

Expecting constant expression.

Expecting first WHEN clause after CASE expression.

Expecting function or procedure name after USE filename.

The syntax of a USE construct is "USE 'filename'.name;" the filename must be in single quotes and the .name must be outside of the quotes. The .name is optional.

Expecting fuse number or list of numbers.

Expecting identifier to follow '.'.

Expecting integer value for switch SWITCH

Expecting name after KEYWORD.

Expecting number after '['.

Expecting number after KEYWORD.

Expecting number or DONT CARE in truth table entry.

The TRUTH_TABLE entries to the left of the :: must be either constants
or .X.

Expecting number or identifier to follow '-'.

Expecting number to follow '..'.

Expecting pin name or number.

Expecting procedure name or number tag to follow '.'.

Expecting property name.

Expecting quoted file name after KEYWORD.

Expecting quoted group name.

Expecting quoted target string.

Expecting second identifier after '..'.

Expecting second number after '..' in range.

Expecting signal name.

Expecting single quoted filename name after KEYWORD.

Expecting string value for switch SWITCH

Expecting symbol list after 'KEYWORD'.

Expecting symbol list in declaration.

Expecting the keyword 'GROUP' to follow the keyword 'FIXED'.

Expecting variable after FOR.

FBINCLUDE not supported in SIMULATION.

Failed to close available file.

Failed to create database: SYMBOL_NAME in create_acro

Failed to create database: SYMBOL_NAME in create_dlib

Failed to create database: SYMBOL_NAME in create_pinmp

Failed to find 'SIGNAL_NAME' for fitting with 'SIGNAL_NAME'.

The FIT_WITH property indicated two signals should be fit together. However, the second signal could not be found as an output or node signal. The signal must exist in the original design as a output or non-virtual node.

Failed to find database version in first line.

Failed to find fit_with signal 'SIGNAL_NAME'.

The FIT_WITH property indicated two signals should be fit together. However, the second signal could not be found. The signal must exist in the original design.

Failed to find suitable node assignment and signal routing: MACH_PART:DEVICE#.

The current partition could not be placed and routed by the mach fitter. It could be routed with no placement considerations or placed with no routing considerations but no valid combination of placements and routings could be found.

Failed to fit design. See SYMBOL_NAME

Your design cannot be fit. Refer to the .log file for the reasons for this failure. The .log file will state which functions could NEVER be fit, and also how many device attempts occurred. There are a large number of factors which contribute to a design being unfittable, and often these factors compound. Areas to examine are how signals are fit as a group, how much I/O is required for the design, and also how FPGAs are being used.

Failed to fit fixed group SYMBOL_NAME from PI on SYMBOL_NAME. See SYMBOL_NAME

The fixed groups must be fit successfully before any solution can be given. The fixed group indicated, however, could not be fit. Refer to the .log file.

Failed to generate fuse map: MACH_PART:DEVICE#.

A problem occurred in assigning pterm rows within the mach part.

Failed to open SYMBOL_NAME database in get_pinout_array().

Failed to read database TYPE in first line of text file.

Fast critical net fanout limit (12) exceeded -- check after optimization

Fast critical net fanout limit (8) warning -- check after optimization

File 'FILENAME' not found.

File 'FILE_NAME.afb' does not exist.

File 'SYMBOL_NAME' not found in path, File generation ignored.

File 'SYMBOL_NAME' not found in path. File modification ignored.

File - 'SYMBOL_NAME' not found

File FILENAME - The DEVICE string does not contain a package designator

On the "DEVICE IS EPMXXXXYY" line, the XXXX is the template name and the first Y is the package designator. The specified file does not contain a package designator on this line.

File FILENAME - the package specified (PACKAGE_STR) is not supported

On the "DEVICE IS EPMXXXXYY" line, the XXXX is the template name and the first Y is the package designator where (D is CDIP, P is DIP, J is CLCC, L is LCC, G is PGA, S is SOD, W is CQFP, and Q is PQFP.

File FILENAME -- SYMBOL_NAME does not contain either an "lcc", "qfp", or "pga" package

The VAR DDFPACKAGE line should contain a string of the form : "/minc/als/data/a1000/a1010/lcc44.ddf". The package (lcc or pga) in this .pin file is not of this format.

File FILENAME contains an invalid VAR DFFPACKAGE line

The VAR DDFPACKAGE line should contain a string of the form : "/minc/als/data/a1000/a1010/lcc44.ddf". The line in the .pin file is not of this format.

File FILENAME does not contain a template name

The .mpn (MINC pin) file must contain a template name on line 2.

File FILENAME does not contain a valid device name

The .mpn (MINC pin) file must contain a device name of the form "MANUF PARTNAME" on line 1.

File FILENAME does not contain a valid fusemap file name

The .mpn file must contain a fuse map file name on line 3.

File FILENAME does not contain a valid number of pins value

The .mpn file must contain the number of pins on the device on line 4.

File FILENAME does not contain a valid template name

The template name specified is not a valid MINC template name

File FILENAME specified a different device than the .pi file

The back annotation software found a different device in the post place and route netlist file than in the original pi file.

File FILE_NAME does not exist - rerun The fitter with netlisting ON

File PINFILE_NAME can not be opened in write mode

File SYMBOL_NAME can not be opened for reading

File SYMBOL_NAME can not be opened for writing

File SYMBOL_NAME can not be opened to rename file contents

File SYMBOL_NAME does not define number of pins on VAR DFFPACKAGE line

The VAR DDFPACKAGE line should contain a string of the form : "/minc/als/data/a1000/a1010/lcc44.ddf" where 44 in the number of pins in the package. The line in this .pin file is not of this format.

File devlib.dbf can not be opened

File does not begin with EDIF keyword

File is not EDIF version 2.0.0

File minclib.lib not found in path for reading

Fitting signal 'SIGNAL_NAME' on virtual pin PIN_NUMBER implies fitting an unfittable group of signals

The signal being fit has some feedback internal to the part, either because it is a node or there is pre-enable feedback. The signals needing to use this feedback must therefore be fit on this same part. There are not enough device resources, however, to have all those signals fit on this part.

Fixed grouping for MACH_PART:DEVICE# exceeds limits for:

Precedes one or more device constraints violated by a function group.

Fixed grouping for MACH_PART:DEVICE# pal block PAL_BLOCK_ID(OPTIONAL)exceeds limits for:

Precedes one or more pal block constraints violated by a function group.

Flip-flop 'SYMBOL_NAME' needs CLOCKED_BY expression.

Format error in library file line LINENO -- line skipped

Format error on line LINENO of newnames.txt -- line skipped

Fs Could not locate device library 'SYMBOL_NAME'.

Fs Could not open device library 'SYMBOL_NAME'.

Function FUNCTION_ID cannot fit due to grouping constraints.

Signal in user specified grouping or pin assignment violates Mach constraints.

Function SIGNAL cannot fit on pin PIN# because:

Mach function pin assignment cannot be satisfied for the reason(s) listed.

Function SIGNAL_ID cannot fit on pin PIN# due to buried register fanout constraints.

User pin assignments violate restrictions on Mach230 buried macrocell fanouts. Mach230 buried register fanouts must be within pal block pairs (A-H, B-G, C-F, D-D).

Function is not a unary function

The function does not qualify as a unary function for fitting on a unary node of this device.

Functions FUNCTION_ID and FUNCTION_ID use the same macrocell.

The named functions are assigned so that they require the same macrocell.

Fuse assignment in a fixed group with footprint target 'TARGET_STRING'.

A fixed group targeted toward a footprint will potentially be fit into many different device architectures. Since fuse numbers and fuse configurations depend on the device architecture, a fuse assignment in a fixed group targeted toward a footprint may have radically different and unexpected effects when implemented in different devices.

Fusemap file 'FUSEFILE' in file 'FILENAME' conflicts with an existing fusemap file

The fusemap filename found on line 4 of the .mpin file conflicts with an already existing fusemap filename for this design.

GLOBAL not supported.

GOTO STATE_NAME goes to nonexistent state.

GOTO STATE_NAME is outside of STATE_MACHINE.

Generation of NODEs for equation minimization is on.

By default, the compiler generates NODEs to break up the logic for complicated operators such as the arithmetic operators. This gives the optimizer greater flexibility to generate efficient equations for the target hardware. This node generation can pose a problem for designers trying to fix the pinout of a portion of a design while changing the functionality of another portion. This node generation can be controlled from the menu. See the Optimizing a Design chapter for more on optimization.

Group name must be quoted with single quotes (").

Hardware locking device NOT installed.

Hit end of file with unmatched COMP_OFF.

IF has GOTO in only one half in STATE 'STATE_NAME'.

An example that would produce this warning is:

```
IF a THEN
  GOTO st1;
END IF;
```

The above code says to go to st1 when 'a' is asserted, but it doesn't say where to go to when 'a' is not asserted. The compiler will fall back on the DEFAULT_TO values of the state bits to determine the state transition when 'a' is not asserted.

IS not supported.

Illegal DONT CARE digit in decimal constant.

Illegal argument 'STRING' for demorgan property.
The legal arguments for the DEMORGAN_SYNTH property are AUTO, FORCE, and OFF.

Illegal argument 'STRING' for flipflop synthesis property.
The legal arguments for the FF_SYNTH property are AUTO and OFF.

Illegal argument 'STRING' for xor synthesis property.
The legal arguments for the XOR_TO_SOP_SYNTH property are AUTO, FORCE, and OFF.

Illegal character ASCII_VALUE in source.
An illegal character appeared in the file. Legal characters include all alphanumeric characters, spaces, tabs, newlines, carriage returns, formfeeds, vertical tabs and the punctuation characters indicated in the manual for building each operator.

Illegal command line switch SWITCH found

Illegal command line switches

Illegal config file SYMBOL_NAME specified

Illegal digit in base NUMERICAL_BASE constant.
The digit is not a valid digit in the numerical base of the constant.

Illegal file name 'FILE_NAME'.

Illegal operation on .X.

Illegal operation on .Z.

Illegal operation on DONT CARE.

Illegal operation on constant and .X.

Illegal pin name 'PIN_NAME' for template 'TEMPLATE_NAME'.

The pin name given for a pin assignment in the physical informationfile was not a valid pin name for the device.

Illegal solution number, using 1

Improper nesting of ')' in parameter for MACRO 'MACRO_NAME'.

Improper nesting of ']' in parameter for MACRO 'MACRO_NAME'.

Incompatible suffix for flip-flop 'SYMBOL_NAME'.

Inconsistent member sizes in WIRED_BUS declaration.

Incorrect target string formatting: 'SYMBOL_NAME'.

Initial routing of signals through switch matrix failed: MACH_PART:DEVICE#.

The current partition could not be routed by the mach fitter.

Input SIGNAL_ID cannot fit on pin PIN# because:

Mach input pin assignment cannot be satisfied for the reason(s) listed.

Input has wrong ff/latch type

The input pin has the wrong configuration for fitting this unary signal.

Input pin 'PIN_NAME' of instance 'INSTANCE_NAME' is not connected.

PLSchematic checks each component instance to verify that all input pins are connected. Each unconnected input pin is connected to either VCC or GND.

Input pin 'VALUE' of instance 'SYMBOL_NAME' is not connected.

Input signal 'SYMBOL_NAME' may be given only once without a pin assignment.

An input signal without a pin assignment may appear only once per group or DEVICE.

Input signal SYMBOL_NAME is assigned to more than one pin

A signal can only be placed on ONE input pin of the device. The .pi file specifies that the signal in error be placed on TWO or more pins of the device.

Input signals are not allowed at the global level.

Input signals may only be given inside a group or DEVICE.

Inputs within an unfixed group are ignored.

Since The fitter will automatically fit all of the input signals that a group requires, there is no need to mention input signals in an unfixed group.

Instance 'INSTANCE_NAME' of type 'COMPONENT_TYPE' is not connected.

The named component instance is unconnected. This will not affect the final results since the equations for this device will be optimized out but is flagged because it is likely that the user either inadvertently left the component or forgot to connect it to the rest of the design.

Instance keyword not followed by instance name

Instance not followed by component name

Instance or net keyword found before contents

Instance rename not followed by 2 instance names

Interface not followed by PORT keyword

Internally fit WIRED_BUS signal 'SYMBOL_NAME' is needed on another device

In this design, the WIRED_BUS signal was fit inside of the Xilinx device. This signal was, however, needed as an INPUT on another device. You must either move all signals that need the WIRED_BUS onto this Xilinx device OR implement the WIRED_BUS external to the Xilinx device (by fitting each of the signals comprising the WIRED_BUS on BIPUT pins.

Invalid PORT syntax

Invalid character 'VALUE'.

Invalid device #VALUE specified for solution VALUE (valid values - 1 to VALUE)

Invalid pin label 'LABEL_NAME'.

Invalid solution #SOLUTION_NUMBER specified assuming solution #1.

Invalid solution switch value -- VALUE

Invocation of undeclared FUNCTION_OR_PROCEDURE 'SYMBOL_NAME'.

Joined or portref statement not preceded by net construct

Left parenthesis found in expression

Library Locking Error.

Line VALUE does not contain a line of the form "signal:pin [INPUT|OUTPUT|BIPUT]"

Line too long.
A single line in the physical information file may be no longer than1024 characters.

List of SYMBOL_NAME signals already placed :

Locking device NOT installed or illegal authorization file

Lost connection to server for SYMBOL_NAME, exiting...

MACH PARTITION_LEVEL partitioning exceeds limits
The Mach partition cannot be reduced to the current limit due to user specified fixed groups or internal feedback grouping.

MACH failed PARTITION_LEVEL partitioning
The partitioner cannot divide the functions into the required number of partitions while remaining within the current limits.

MACH failed PARTITION_LEVEL pre-partitioning
The partitioner cannot divide the functions into the required number of partitions while remaining within the current limits.

MACRO 'MACRO_NAME' never terminates.

The macro is missing the terminating }.

MACRO 'MACRO_NAME' spans multiple lines.

Macro expansion too large.

Making 'SYMBOL_NAME' be PHYSICAL NODE.

Manufacturer name 'SYMBOL_NAME' is too long -- must be VALUE or less characters

Max # of update entries (NUMENTRIES) exceeded -- processing terminated at line LINENO of library

Max # of update entries (NUMENTRIES) exceeded -- processing terminated at line LINENO of newnames.txt

Medium critical net fanout limit (12) exceeded -- check after optimization

Medium critical net fanout limit (8) warning -- check after optimization

Memory exhausted, VALUE of the VALUE solutions are saved.

When The fitter detects the system no longer has memory available, The fitter frees some memory to get room to work, and then saves in order as many solutions as possible.

Mismatched group sizes.

This means that an operation has been performed on two arrays or groups of signals that have a different number of bits from each other.

Mismatched range identifiers 'SYMBOL_NAME' and 'SYMBOL_NAME'.

Missing '(' after 'KEYWORD'.

Missing ')' for argument list of MACRO 'MACRO_NAME'.

Missing ')' in invocation of MACRO 'MACRO_NAME'.

Missing ';'.

Missing 'KEYWORD'.

Missing KEYWORD1 after KEYWORD2.

Missing TEST_VECTORS keyword on simulation vector table.

Missing name after END.

Missing quote on string.
A string must be enclosed by a pair of single quotes ('), but one of the quotes on either the left or right side of the string was missing.

Missing right quote on string.

Mixed INPUT and BIPUT declarations for 'SYMBOL_NAME'.

Mixed use of NO_REDUCE on members of assigned group.

Modulo by 0.

Multiple NODE declarations for 'SYMBOL_NAME' in separate PROCEDUREs.

Multiple TARGET constructs in fixed group.

A fixed group may be targeted toward only one device.

Multiple cell constructs found in file 'EDIF_FILE'

Multiple conflicting assignments to 'SYMBOL_NAME'.

A signal can only be assigned one value for any condition. An example that would produce this error is:

```
IF a THEN
  x = 1;
END IF;
IF b THEN
  x = 0;
END IF;
```

This example says that 'x' should be 1 when 'a' is asserted. It also says that 'x' should be 0 when 'b' is asserted. This is a problem if both 'a' and 'b' are asserted.

Multiple interface constructs found in file 'EDIF_FILE'

Multiple library constructs found with no design construct

Multiple port assignments for a single net

Must pass assignable expression to OUTPUT 'SYMBOL_NAME'.

Must specify TARGET in order to specify fuse assignments.

The target device must be known for fuse assignments to be meaningful.

Must specify TARGET in order to specify no-connect pin assignments.

The target device must be known for pin assignments to be meaningful.

Must specify TARGET in order to specify pin assignments.

The target device must be known for pin assignments to be meaningful.

N bit number cannot be represented in M bits.

NAME is only allowed within a group or fixed group.

NO_CONNECT is not allowed at global level.

No-connect pin assignments may only appear inside a fixed group.

NO_CONNECT is not allowed in subgroups.

No-connect pin assignments may appear in a fixed group, but not in subgroups of the fixed group, fixed subgroups of the fixed group, or at the global level.

Need suffix for flip-flop 'SYMBOL_NAME'.

The signal name for a JK_FLOP, SR_FLOP, or T_FLOP must have a suffix appended to indicate which input of the flop is driven by the assigned expression.

Nested assignment to 'SYMBOL_NAME' which has NO_REDUCE.

A symbol declared with NO_REDUCE must be given its equation outside of any IF, CASE, TRUTH_TABLE, or STATE_MACHINE statements to guarantee that the equation will be implemented as given without any reduction.

Nesting of subgroups too deep.

The level of nesting of groups and fixed groups is restricted to two. Fixed groups may have subgroups or fixed subgroups, but the subgroups and fixed subgroups may not have subgroups or fixed subgroups.

Nesting too deep.

This is caused by some construct or combination of constructs in the source file being nested too deeply. Make sure you have an END on all constructs requiring ENDs.

Net 'NET_NAME' does not drive any instance input pins.

Net 'NET_NAME' is driven by multiple instance output pins.

A net can be driven by only one device output pin.

Net 'NET_NAME' is not driven by an instance output pin.

Net 'NET_NAME' is unconnected.

Net construct found before previous net complete

No GOTO to STATE 'STATE_NAME'.

If there is no GOTO to a particular STATE then the state may not be reachable and may not be necessary to the design.

No Pins Assigned Before Auto Pin Placement

No clock expression

This signal is registered, but the clock expression is unavailable, probably because this equation earlier was deemed too large.

No device supports the construct of a latch with a CLOCK_ENABLED_BY (SIGNAL_NAME).

There is no device known to The fitter which has a clock enable for a latch.

No devices available that fit an output. See LOG_FILE_NAME

The fitter will attempt screen the parts which cannot fit ANY signals. No parts can fit any signals, and therefore The fitter could not continue.

No devices match scan criteria...

The criteria you supply remove device architectures from consideration. These criteria, along with you authorization, combine to create an available list of devices. In this case, there are no devices available.

No equations in the fb file...

No functions in design fit into target device 'SYMBOL_NAME'.

The device was targeted to fit a group of functions in the physical information file, but was not able to fit any of the functions in the design.

No input pins connected to instance 'INSTANCE_NAME' of type 'COMPONENT_TYPE'.

All of the input pins on the named instance are unconnected. This will not affect the final results since the equations for this device will be optimized out.

No library construct found in file 'EDIF_FILE'

No more than one DEFAULT construct per file is allowed.

Only one fixed group may be specified as the default fixed group. All signals in the design that were not mentioned in the pi file will be placed into the default fixed group.

No output pins connected to instance 'INSTANCE_NAME' of type 'COMPONENT_TYPE'.

All of the output pins on the named instance are unconnected. This will not affect the final results since the equations for this device will be optimized out.

No outputs in design.

No remaining data equations for output signal 'SIGNAL_NAME'.

The DEMORGAN_SYNTH, XOR_TO_SOP_SYNTH, and FF_SYNTH properties remove equations from consideration. If other equations were already removed due to their size, then there may be no equations left to implement the output signal's functionality.

No solution has been selected in the fb

No solution information in fb

No solution selected during the fitter assuming solution #1.

No solutions in .fb file.

No solutions were found for specified design

No stimulus file.

No system area in design.

No templates match criteria after execution of PLScan.

The scanner, after screening out parts based on your criteria andauthorization, has left NO parts for The fitter to use.

No valid devices

There are no device architectures available for fitting.

Noncritical net fanout limit (12) exceeded -- check after optimization

Noncritical net fanout limit (8) warning -- check after optimization

Not enough columns for this output signal

This device, probably a PLA, does not have enough AND columns. The various data equations are all too large given the remaining number of columns available.

Not enough inputs available on device

The number of inputs required to fit this output exceeds the available number of input and biputs of this device.

Not enough rows feeding the OR gate for this output signal

This device does not have enough AND rows feeding the OR gate. The various data equations are all too large given the number of AND rows available.

Number of inputs in truth table entry does not match header.

Number of outputs in truth table entry does not match header.

Number of test vector entries does not match header.

Number too large.

Numbers out of order in range.

Old VENDOR_NAME netlist file 'NETFILE_NAME' can not be removed

Old VENDOR_NAME pinout file 'PINFILE_NAME' can not be removed

On bus SYMBOL_NAME: signals SYMBOL_NAME and SYMBOL_NAME are driving the bus in different directions.

On bus SYMBOL_NAME: signals SYMBOL_NAME and SYMBOL_NAME are driving the bus in the same direction.

Operation produced negative number.

Operation would result in negative constant.

Out of memory

Out of memory before any solution could be found.
The fitter has detected there is no remaining memory, and the solution search was interrupted before any solution could be found.

Out of memory in make_set_element

Out of memory in newstr()

Out of memory, and NO alternative memory actions available.
After attempting alternative memory actions, there still is no memory available. The fitter can no longer proceed.

Out of memory, attempting to save solutions ...
When The fitter detects the system no longer has memory available, The fitter frees some memory to get room to work, and then attempts to save solution in order.

Out of memory.

Output file 'SYMBOL_NAME' already exists.

Output pin 'PIN_NAME' of instance 'INSTANCE_NAME' is not connected.
PLSchematic checks each component instance to verify that all output pins are connected. The user is warned of this condition and processing continues.

Output pin 'VALUE' of instance 'SYMBOL_NAME' is not connected.

Output signal 'SIGNAL_NAME' cannot be fit with 'SIGNAL_NAME'.

The FIT_WITH property indicated these two signals should be fit together, but for some reason they CANNOT fit together. The feeding signal must be the only signal in the other signals equation. The feeding signal cannot be inverted. The feeding signal must be a node. The receiving signal cannot be registered or latched. Both signals cannot be enabled.

Output signal 'SYMBOL_NAME' can not ALSO be placed on an input pin

A signal that was specified as an OUTPUT to this device was also specified as an INPUT to this device. For the specified architecture, this capability is not allowed.

Overlapping TRUTH_TABLE entries.

The TRUTH_TABLE statement has two different entries with an overlapping set of conditions. At least one of the input values of each entry must be different from the corresponding input value of the other entries. A .X. overlaps with any other input value.

PI Footprint: 'FOOTPRINT_NAME' not in .avl file or has been eliminated by constraints.

PI demorganization property for 'SIGNAL_NAME' conflicts with the NOREDUCE option.

Demorganization is turned off if the NOREDUCE option is set, so the only legal argument for the DEMORGAN_SYNTH property is OFF.

PI target STRING1 STRING2 not in library ; please check spelling

PI target STRING1 STRING2 rejected, by lock: 'TEMPLATE_NAME', check constraints.

PI target STRING1 STRING2 rejected, family:'FAMILY_NAME', check constraints.

PI target STRING1 STRING2 rejected, fmax = FMAX_VALUE Mhz, check constraints.

PI target STRING1 STRING2 rejected, icc = ICC_VALUE ma, check constraints.

PI target STRING1 STRING2 rejected, manufacturer:'MANUFACTURER_NAME', check constraints.

PI target STRING1 STRING2 rejected, package:'PACKAGE_NAME', check constraints.

PI target STRING1 STRING2 rejected, price = PRICE, check constraints.

PI target STRING1 STRING2 rejected, temp_range:'TEMP_VALUE', check constraints.

PI target STRING1 STRING2 rejected, template: 'TEMPLATE_NAME', check constraints.

PI target STRING1 STRING2 rejected, tpd = TPD_VALUE ns, check constraints.

PI target STRING1 STRING2 rejected, user1 = 'USER1_VALUE', check constraints.

PI target STRING1 STRING2 rejected, user2 = 'USER2_VALUE', check constraints.

PI target Template: 'TEMPLATE_NAME' Footprint: 'FOOTPRINT_NAME' is not in .avl file or has been eliminated by constraints.

PI target error(s) detected.

PI target template: 'STRING' is not a valid template name.

PLA device DEVICE_NAME does not exist in database -- no annotation performed
The back annotation software attempts to find a device in its database with the template and numpins specified in the .mpn file. No part exists to match the specified criteria.

PLDoc - file open error

PLDoc usage: pldoc design_name

The fitter has not been run

The fitter usage: the fitter design_name

PLFuse usage: plfuse design_name

PLOpt has not been run

PLOpt usage: plopt design_name

PLScan has not been run

PLScan usage: plscan design_name

PLSchematic usage: plsch design_name reader_flag

PLSim usage: plsim design_name [stimulus_file]

PORTHID is obsolete and has been replaced by NODE.

The PORTHID primitive is no longer supported and will be replace internally by the NODE primitive. This primitive will not be included in subsequent releases of the product.

Pal block PAL_BLOCK_ID is not valid for device SYMBOL_NAME

Part name 'SYMBOL_NAME' is too long -- must be VALUE or less characters

Pass 1 error checking.

During the first pass of compilation the compiler finds all errors that can be found without generating equations. This is done to quickly report errors and terminate compilation before the slower equation generation is performed.

Pass 2 equation generation.

During the second pass of compilation the compiler generates the equations for all of the signals in the design. Some errors may be reported during this pass if they are discovered as a result of equation generation.

Pin PIN_NAME (signal SYMBOL_NAME) exceeds the maximum pins for the device

The pin name found in the post-place and route netlist is not a recognized pin name.

Pin PIN_NAME is not a valid pin for device DEVICE_NAME

There is no such pin for the specified device.

Pin PIN_NAME_OR_NUMBER already assigned.

A signal was assigned to a pin that already had a signal assigned to it.

Pin assignments are not allowed at global level.

Pin assignments may be given in fixed groups and fixed subgroups of fixed groups.

Pin assignments are not allowed in unfixed groups.

Pin assignments may be given in fixed groups and fixed subgroups of fixed groups.

Pin is not an output pin

The pin in this configuration is not an output pin. You cannot assign an output signal to this type of pin.

Pin is not available

This pin has already been assigned or is otherwise unavailable.

Pin number must be numeric

Pin number must be preceded by an ampersand

Pinlabel 'SYMBOL_NAME' not in correct format

PrintDevices usage: pr_devs design_name [#devices]

Procedure/function SYMBOL_NAME not in design.

Property 'PROPERTY_NAME' takes NUM_ARGUMENTS argument(s).

The property must be given with the correct number of arguments.

Property 'PROPERTY_NAME' takes argument 'TRUE' or 'FALSE'.

The property takes one argument that can only be the word 'TRUE' or the word 'FALSE'.

Property 'PROPERTY_NAME' takes one argument.

The property takes one argument which may consist of either a number, a word consisting of alphanumeric characters and '_', or a string (a sequence of characters enclosed in single quotes).

Property 'PROPERTY_NAME' takes one numeric argument.

The property takes one argument which may consist of a number.

Property 'PROPERTY_NAME' was given twice.

The same property can appear only once in a given context. A context can be one of: a signal, a group, a fixed group, or the global level.

Property PROPERTY_NAME is not accepted globally, on a group, or on a fixed group.

The named property was given at the global level, in a group, or in a fixed group, but is not allowed to appear in these contexts.

Property PROPERTY_NAME is not accepted on input signals.

The property may not be given on input signals.

Property PROPERTY_NAME is not accepted on output signals.

The property may not be given on output signals.

Property PROPNAME is an invalid property name -- property ignored

When reading the FB, an invalid PI property was found. This property will be ignored during the read.

RETURN must be inside of FUNCTION.

Ran out of dynamic memory.

Range identifier 'IDENT' needs to end with a number.

Range identifier 'SYMBOL_NAME' needs ending number.

Range identifiers 'IDENT1' and 'IDENT2' not identical before end numbers.

Ranging over more that VALUE identifiers.

Reason undisclosed
The reason for not fitting the signal is unstated, but could be a number of contributing problems or a problem that cannot be easily described.

Recursive USE of SYMBOL_NAME

Recursive invocation of FUNCTION_OR_PROCEDURE 'SYMBOL_NAME'.
A FUNCTION or PROCEDURE cannot invoke itself.

Recursive macro definition for MACRO 'MACRO_NAME'.

Redeclaration of 'SYMBOL_NAME'.

Redeclaration of CONTROL_KEYWORD expression.

Redeclaration of MACRO 'MACRO_NAME'.

Redeclaration of STATE_BITS.

Redeclaration of STATE_VALUES.

Redeclaration of VAR 'VAR_NAME'.

Redeclaration of symbol 'SYMBOL_NAME'.

Redeclared step size, new value used.

Reducing 'SYMBOL_NAME'.

Reference of undeclared VAR 'SYMBOL_NAME'.

Removing 'SYMBOL_NAME'.

Removing special JEDEC character '*' from header.
Jedec files consider an asterisk to be a special character. Asterisks are replaced by spaces in header strings to avoid creating a bad JEDEC file.

Removing unused NODEs.
This is the optimization phase where signals that do not contribute to any OUTPUT signal are removed since they are not needed in the design.

Renaming array signal OLDNAME caused a clash with signal EXISTING_SIGNAL -- new name is NEWNAME

Repeated results of last partition.
A particular MACH partition attempt which is reduced or partitioned at a lower utilization is discarded because it exactly repeats the prior failed partition.

SIMULATION must be placed in the .stm file.

SPECIAL not supported.

STATE 'STATE_NAME' needs a GOTO.

If no GOTO is given for a STATE then the compiler will fall back on the DEFAULT_TO values of the state bits to determine the state transition.

STATE_BIT 'SYMBOL_NAME' must be a NODE or OUTPUT.

STATE_BITs have incompatible CLOCKED_BY expressions.

The signals given in the STATE_BITS construct must all have been declared with the same CLOCKED_BY expression. If a CLOCKED_BY is given in the STATE_MACHINE header then it must also match the CLOCKED_BY expression of each state bit.

STATE_BITs have multiple DEFAULT_TO expressions.

The signals given in the STATE_BITS construct must all have been declared with the same DEFAULT_TO expression. If a DEFAULT_TO is given in the STATE_MACHINE header then it must also match the DEFAULT_TO expression of each state bit.

STATE_MACHINE must default to 0, .X., or LAST_VALUE.

SYMBOL_NAME

SYMBOL_NAME is an illegal config file

Scan not necessary: files are up to date.

Schematic design error(s) found in 'INPUT_NETLIST' - unable to continue.

PLSchematic has a built-in rules checker to verify that the input netlist is consistent. If it is not the error is printed and processing is discontinued before the output files are generated.

Should not have '.' in library name 'FILE_NAME'.

The syntax of a USE construct is "USE 'filename'.name;" the filename must be in single quotes and the .name must be outside of the quotes. The .name is optional.

Signal 'SIGNAL_NAME' is an input through internal feedback to 'SIGNAL_NAME', hence they must be fit together, but cannot.

The two signals must be fit together because one signal uses the internal feedback of the other signal, but they cannot be fit so that the internal feedback is visible to the signal that uses it.

Signal 'SIGNAL_NAME' on virtual pin PIN_NUMBER of device 'DEVICE_NAME' failed:

The device fitter/partitioner could not place the output signal on the output or biput pin. The reason for the failure follows on the next line.

Signal 'SYMBOL_NAME' already mentioned as an output in the .pi file.

A signal may be mentioned as an output in the .pi file only once. A signal may be mentioned as an input in the .pi file multiple times.

Signal 'SYMBOL_NAME' is an input and can't be VIRTUAL.

Only node signals can be marked as virtual signals.

Signal 'SYMBOL_NAME' is not an output signal.

The signal is not an output signal, but was marked as an output with OUTPUT.

Signal 'SYMBOL_NAME' is physical but used as a virtual signal.

Only node signals can be marked as virtual signals.

Signal 'SYMBOL_NAME' is virtual but used as a physical signal.

The signal was declared in the .src file as a virtual node signal but was used in the pi file outside of the virtual construct.

Signal 'SYMBOL_NAME' was not renamed due to a conflict with signal 'SYMBOL_NAME'

The period character is not a valid symbol in the HDL. Internally generated names may contain periods however. In attempting to recreate a .src file, all periods are converted to underscores. In this design, renaming the signal caused a conflict with an existing signal name. For this reason, the signal was NOT renamed.

Signal OLD_SYMBOL_NAME was placed on pin PIN_NAME in pi - signal NEW_SYMBOL_NAME there after place and route

In the original PI file, the user specified a signal be placed on the the specified pin.. After the place and route software was run, however, a DIFFERENT signal was placed on that pin.

Signal OLD_SYMBOL_NAME was specified in pi on pin PIN_NAME as an INPUT_OR_OUTPUT but after place and route is an INPUT_OR_OUTPUT.

In the original PI file, the user specified a signal be placed on the the specified pin. After the place and route software was run, however, the signal usage on the pin, input or output, was different.

Signal SIGNAL_ID cannot fit due to invalid pin type.

User specified pin assignment places signal on invalid pin.

Signal SIGNAL_ID is assigned to multiple pins.

The mach pre-partitioner could not implement the fixed grouping of the .pi file.

Signal SYMBOL_NAME -- DEMORGAN_SYNTH FORCE was specified -- no OFFSET equation exists

In the .pi file, the specified signal has the DEMORGAN_SYNTH FORCE property. This signal does not have an OFFSET representation (probably due to the equation size) and cannot, therefore, be used as specified.

Signal SYMBOL_NAME cannot fit as a registered input as required by pin VALUE.

The signal does not meet the qualifications for a registered input (unary) signal, but the pin specified requires that the signal be fit that way.

Signal SYMBOL_NAME has neither a D equation or an alternate flip-flop equation

The requested synthesized equation is not available for the specified signal

Signal SYMBOL_NAME has no INPUT/OUTPUT/BIPUT specification -- skipped

During back-annotation, the specified signal did not have a valid input/output/biput specifier attached to it. For this reason, the signal will be ignored during annotation.

Signal SYMBOL_NAME is an EXT signal in the .xnf yet was not found in the .lca file

The EXT lines of the .xnf files indicate INPUT/OUTPUT/BIPUT status of the signals used in the design. The only allowed designators are I/O/B.

Signal SYMBOL_NAME is in a WIRED_BUS but is not an enabled, non_clocked NODE

Signal SYMBOL_NAME was specified in pi on pin PIN_NAME but not found after place and route

In the original PI file, the user specified a signal be placed on the the specified pin. After the place and route software was run, however, the signal was NOT placed on that pin.

Signal SYMBOL_NAMEA (line LINENO) is not a valid signal name

The specified signal is not a valid signal name for this design

Signal may require split pin; split pin limit is exhausted.

The fitter budgets split pins (biput converted to node and input) based on the output (non-split pin) count for each pal block. The fitter has detected that there are no more split pins available.

Signal name 'SYMBOL_NAME' does not end with a number.

The signal is understood to be part of a range of signals, such as 'a1..a10', but its name does not end with a number.

Signal range names 'SYMBOL_NAME' and 'SYMBOL_NAME' have different bases.

The non-numerical prefixes on the signal range names are different, so cannot form a range.

Signals SIGNAL_ID and SIGNAL_ID use the same pin.

The named signals are assigned to the same pin.

Signals cannot fit into any device. See LOG_FILE_NAME

The fitter will attempt to find AT LEAST ONE place a signal can fit. In this case, signals could not fit anywhere on any device.

Signals cannot fit into the targeted devices. See LOG_FILE_NAME

Signals were targeted to certain devices, but some signal could not be fit in their targeted device.

Simulating.

Single level device targeted, but fixed subgroups were given.

Fitting a single level device such as a P22V10 implies that the fixed group information must also be single level; i.e. no fixed subgroups (which the system cannot merge into a single-level

description) may be present. Fixed subgroups of a fixed group are useful in fitting a multi-level device.

Skipped - Invalid format

Skipped - Invalid manufacturer: MANUFACTURER_NAME

Skipped - Invalid part number: PART_NUMBER

Skipped - Invalid status 'STATUS'

Solution switch value VALUE exceeds # of solutions in fb(VALUE)

Source file 'SYMBOL_NAME' has been modified since PLComp has been run

Source line too long.

State values must be given to all states or none.

String length cannot exceed 1024 characters

Switch table not initialized

Symbol 'SIGNAL_NAME', set in the PI file, cannot fit on pin 'PIN_NAME'. Either you

The signal cannot be fit on the pin that it is assigned to in the physical information file. This can happen if the signal type (input or output) does not match the pin type or if the signal has an equation that requires device resources that are unavailable. See the .log file for possible fitter error output.

Symbol 'SIGNAL_NAME', set in the PI file, cannot fit on virtual biput pin PIN_NUMBER

Symbol 'SIGNAL_NAME', set in the PI file, cannot fit on virtual input pin PIN_NUMBER

Symbol 'SYMBOL_NAME' -- The OUTFFT signal MUST be an OUTPUT

Symbol 'SYMBOL_NAME' is not an array.

A symbol not declared as an array in the .src file cannot be indexed like an array.

Symbol 'SYMBOL_NAME' not used by any OUTPUT.

The means that a signal is not needed to drive the value of any OUTPUTs in the design. This signal is not necessary to the design.

Symbol SYMBOL_NAME - 'PRIMITIVE_NAME' INPUT can not be used in any other equations

Symbol SYMBOL_NAME - 'SYMBOL_NAME' primitive requires single signal input to NODE

Symbol SYMBOL_NAME - OUTFFT requires an ENABLED output fed by a D flip-flop node

Symbol SYMBOL_NAME - The INFF input signal must feed a D flip-flop NODE signal

Symbol SYMBOL_NAME - The INLAT input signal must feed a D latch NODE signal

Symbol SYMBOL_NAME - The OUTFF property only applies to CLOCKED D flip-flop outputs

Symbol SYMBOL_NAME - The OUTFF signal can not have a RESET or PRESET equation

Symbol SYMBOL_NAME - The OUTFFT property only applies to ENABLED outputs

Symbol SYMBOL_NAME - The PRIMITIVE_NAME NODE can not have a RESET or PRESET equation

Symbol SYMBOL_NAME - The PROPERTY_NAME property cannot be used on tristate signals with feedback

Symbol SYMBOL_NAME - The PROPERTY_NAME property must be used on a buffer

Symbol SYMBOL_NAME - The PROPERTY_NAME property must be used on a inverter

Symbol SYMBOL_NAME - Use XOR was specified in .pi, but does not exist. Property ignored.

In the .pi file, the signal had the XOR_TO_SOP_SYNTH OFF property. The specified signal does not contain an XOR representation however. The software will ignore the XOR request and represent this signal in sum of products form.

Symbol SYMBOL_NAME is an internal node to the OUTFFT primitive and can't be used in the design

Symbol SYMBOL_NAME is lowtrue with OUTFFT primitive and can't be used in the design

Symbol SYMBOL_NAME not in this simulation section.

Symbol SYMBOL_NAME, when used in an OUTFFT primitive, can't have a RESET or PRESET equation

Syntax error in DECLARATION_TYPE declaration.

Syntax error in MESSAGE. (May have used " instead of ')

Syntax error in SET statement.

Syntax error in expression list.

This is a syntax error that occurs when the compiler is attempting to process a list of expressions. In this case, the compiler cannot determine a more helpful description of what is wrong.

Syntax error in pterm string parser near 'STRING'.

The syntax for a pterm string is a series of signal names, optionally inverted with a '/', and separated with '*'s. White space is allowed.

Syntax error in simulation statement.

Syntax error in statement.

This is a syntax error that occurs when the compiler is attempting to process a statement. In this case, the compiler cannot determine a more helpful description of what is wrong.

Syntax error in target string 'TARGET_STRING'.

Syntax error while parsing the .cst file near SYMBOL_NAME at VALUE.

Syntax error.

This is a syntax error that occurs in a context where the system cannot determine a more helpful description of what is wrong. It indicates that a construct has not been legally constructed. See the manual for the proper syntax of each construct. Synthesizing and reducing. This is the phase where register synthesis and equation reduction takes place. For signal by signal reporting of activity, turn on the verbose option.

TARGET is only allowed within a fixed group.

Only fixed groups and fixed subgroups of fixed groups may be targeted.

Target string 'TARGET_STRING' too long.

The target string cannot be more than 160 characters long.

Target string must be quoted with single quotes (").

Targeted subgroups of an untargeted group are not allowed.

A fixed group must have a TARGET construct in order for subgroups of a fixed group to have a TARGET.

Template TEMPLATE_NAME - the device's input signal limit was exceeded placing signal The number of signals specified as input to the device in the fixed group

Exceeds the number of total input pins on the device.

Template TEMPLATE_NAME - the device's output signal limit was exceeded placing signal 'SYMBOL_NAME'

The number of output signals specified in the fixed group exceeds the number of total output/biput pins on the device.

Template: 'TEMPLATE_NAME' in .pi file is not in .avl file or has been eliminated by constraints.

Text file format error - line VALUE

The '[+]' operator is no longer supported. Use '(+)'.

The compiler now automatically generates equations for hardware exclusive-or. There is no longer the need for the designer to specify this via the hardware-exclusive-or operator.

The HEADER_TYPE header is already given.

The NO_COLLAPSE property on 'SIGNAL_NAME' conflicts with the FIT_WITH property on 'SIGNAL_NAME'.

If two signals are to be fit together, neither of them can have the NO_COLLAPSE property.

The cost file (.cst) has been modified. Please run plscan.

The design has no system level symbols.

Only system level symbols (those declared outside PROCEDUREs/FUNCTIONs) will become actual signals in the target devices. If a design consists solely of PROCEDUREs/FUNCTIONs then there are no actual signals to implement in the target devices. The PROCUDUREs/FUNCTIONs must be invoked at the system level to create an implementable design.

The optimizer generates this message when no system level signals exist in the design.

The group consists of the following functions

Precedes a listing of a group of functions which violates Mach constraints.

The specified .fb file is from a previous major release (VERSION_NUMBER.X) - please recompile

An attempt has been made to use a .FB file from a previous release of the software. This file is not binary compatible from release to MAJOR release. It will be necessary to recompile the .src file and run PLOpt to create a

new .FB file.

The third parameter is optional. If used, it must be a positive integer.

The user1, user2, and price fields were all negative

There were errors in 'PI_FILENAME'.

There were errors which must be corrected in the physical information file.

This part will not be included in PLDPRIMS in subsequent releases.

Too few parameters to 'SYMBOL_NAME'.

Too few pins for label 'LABEL_NAME'.

Too few pins for label SYMBOL_NAME

Too many entries in cmd file -- 4000 maximum

Too many equations

Too many equations in design.

Too many parameters to 'SYMBOL_NAME'.

Too many symbols in design.

Two GOTOs active under same condition in STATE 'STATE_NAME'.

There can only be one STATE to go to for any condition. An example of this error is:

```
IF a THEN
  GOTO st1;
END IF;
GOTO st2;
```

The above code says to go to st1 when 'a' is asserted, but it also says to go to st2 regardless of the value of 'a'.

Unable to find component library 'LIBRARY_NAME'

The named component library was not found. The location of the library has to specified either explicitly by prepending an absolute path or by adding the location of the library to the environment variable, MINC_PATH.

Unable to load OrCAD TTL library

Plschematic was unable to access the ORCAD TTL libraries. This is either because they do not exist, they are not located in the MINC distribution directory in the subdirectory orcadttl, or the environment variable, MINC_PATH does not include the path to the MINC distribution directory.

Unable to load base component library 'LIBRARY_NAME'

PLSchematic was unable to access the requested library file. Check the file permissions.

Unable to load extended component library 'LIBRARY_NAME'

PLSchematic was unable to access the requested library file. Check the file permissions.

Unable to open file 'FILE_NAME'.

The given filename either does not exist in the current directory or the file does not have read permission.

Unable to open file minclib.lib for reading

Unable to validate product authorization.

The code which prevents unauthorized use of the product is not allowing access to the product. On a PC, either the hardware lock is not properly installed or the authorization code given via auth.exe do not correspond to the hardware lock and the product being run. On a workstation, the authorization codes given in the license.dat file do not correspond to the host machine and the product being run. See the Installation chapter of the manual for more on software locking.

Unassigned input signal 'SYMBOL_NAME' from .pi file unneeded and ignored

An input signal was specified in the .pi file but was not needed on the device

Unassigned node signal 'SYMBOL_NAME' from .pi file unneeded and ignored

A node signal was specified in the .pi file but was not needed on the device

Undeclared array member 'SYMBOL_NAME'.

Undeclared symbol 'SYMBOL_NAME'.

Undefined reader flag specified - 'NETLIST_READER'

Undetermined reason.
The fitter cannot detect the specific reason why the signal cannot fit with other signals assigned to the device or pal_block.

Unexpected EOF

Unexpected end of file.
This means the end of a source file was encountered before a construct that was being processing had terminated. This is often caused by a missing END on a construct requiring an END.

Unexpected token 'STRING' found by pterm string parser.
Legal tokens in a pterm string are '*', '/', and legal signal names.

Unknown STATE_VALUES method 'ALGORITHM_TYPE'.

Unknown database type = VALUE, in open_write_txt_db()

Unknown error message: SYMBOL_NAME

Unknown family: 'SYMBOL_NAME' in .cst file.

Unknown header type 'HEADER_TYPE'.
The '#' that initiates a header must be followed by one of the legal header types specified in the manual section on headers.

Unknown manufacturer: 'SYMBOL_NAME' in .cst file.

Unknown package: 'SYMBOL_NAME' in .cst file.

Unknown signal 'SIGNAL_NAME' found by the pterm string parser.

Only signals declared in the design are allowed in this pterm string.

Unknown signal 'SYMBOL_NAME' in range.

The signal appeared as part of a range of signals in the pi file, but was never declared in the .src file.

Unknown symbol 'SYMBOL_NAME'.

The symbol was not an INPUT, NODE or OUTPUT signal in the .src file.

Unknown template: 'SYMBOL_NAME' in .cst file.

Unknown temprange: 'SYMBOL_NAME' in .cst file.

Unneeded input signal 'SYMBOL_NAME' was ignored.

Since The fitter will automatically fit all of the input signals that output signals require, an input signal in a fixed group that is not assigned to a pin will only be fit on the device if one of the functions that is fit in the device requires it.

Unrecogized database TYPE in text file.

Unrecognized property name 'PROPERTY_NAME'.

Updatelpf usage: updipf design_name

Usage: plbld design_name

User terminated with <CTRL><BREAK>

Virtual signal 'SYMBOL_NAME' was given a physical assignment.

Virtual signals cannot be used anywhere in the pi file except in the virtual construct.

Warnings during simulation, see FILE_NAME.

Weight VALUE larger than 100.

Word 'ARGUMENT' in target string too long.

No individual word in the target string may be more than 80 characters long.

Wrong number of params to MACRO 'MACRO_NAME'.

Wrong output register type

The register type of this output macrocell does not match the types of equations available.

Wrong version of The fitter run

Wrong version of PLScan run

XORSOFT is obsolete and has been replaced by XOR.

The XORSOFT primitive is no longer needed now that exclusive-or synthesis is supported. If a device has a hardware exclusive-or it will be used. Otherwise, it will be synthesized.

Yacc stack overflow while parsing the .cst file.

Yacc stack overflow.

D AMD MACH Support Supplement

Contents

Introduction

This appendix contains a series of application notes specific to using MACHXL with the MACH family of devices.

The MACH Family Data Book from AMD provides detailed device-specific information. This appendix assumes you know how to use MACHXL and are familiar with the MACH devices.

For the most part, this User's Guide assumes the design process flows smoothly from beginning to end. In the real world this is seldom the case. The information presented here focuses on things that can go wrong in the design flow and steps you can take to remedy those problems.

In this section we first define a generic design process as a framework for discussion. We then detail each step of the process for MACH-specific issues you may encounter. We focus on optimizing the design for MACH devices, and on covering what can go wrong in the design phase. Specific attention is paid to interpreting output from MACHXL in cases where a design fails to fit or test correctly.

Issues and questions are raised in each discussion of the design flow, and the answers and technical information are presented following each of these discussions.

Overview of the Design Process

In this appendix we will use a generic design process consisting of five steps:

- Design Conception
- Design Expression
- Design Implementation
- Design Testing
- Design Integration

Design **Conception** is the user's responsibility. In this stage you have a well defined problem and develop a basic idea of how a solution is achieved.

Design **Expression** involves producing a functional description of the solution design in a form MACHXL understands, i.e., some combination of HDL source files and schematics.

Design **Implementation** places the design in an electronic device. Since most designs go through iterations, there will probably be several implementations before the final form is reached. But each implementation is a functional realization of the current design.

Design **Testing** verifies the implementation works as intended. This meaning generating test stimulus, simulating the design, and using the resulting vectors to test the circuit(s).

The integration step insures the design is ready for manufacturing. This includes achieving a suitable pinout, and making sure the pinout can be reproduced.

MACH Issues in the Design Flow

This application note goes through the steps in the design flow presented in the prior section and discusses device-specific enhancements for MACH designs. It also discusses problems occurring when using MACH devices.

Discussions of device specific issues and problems are followed by references to application notes providing techniques and information necessary for resolution.

Design Conception

If your design requires:

- predictable speed
- capacity up to 10,000 gates (manufacturers specifications)
- larger designs with device partitioning (optional),

the MACH family of devices and MACHXL are the path to your solution.

See the section later in this appendix entitled "***Summary of MACH Family of Devices***" for more information.

Design Expression

Although MACHXL's Design Synthesis Language is device independent, there are some practices that will help the designs to fit into MACH parts. These practices also aid in selection of MACH 1xx and 2xx parts appropriate to the design.

The synchronous MACH parts, MACH 110, 120, 130, 210, 220 and 230, have provisions for clocking by a signal on a pin. The asynchronous parts, MACH 215 and 4xx, have provisions for clocking by either a pin or a single product term. If your design needs a clock which is more complex than these options provide, it is possible to clock by a complex logic function using the MACHXL fitter. This function will be wired to a clock pin or used internally on an asynchronous device.

See the application note later in this appendix entitled "***MACH Designs With Complex Clock Functions***".

Although both the MACH215 and MACH4xx support asynchronous functions, some functions or groups of functions can fit only in the 215. Functions that are clocked by a pterm and have a reset and preset can only fit in the 215. Groups of functions that have more than eight distinct pairs of reset and preset equations can only fit on the MACH215.

See the application note later in this appendix entitled "***Fitting Asynchronous Function in MACH Devices***" for more information on these restrictions.

When combining XOR equations with T-Flops, you may need to insert a node ahead of the register.

See the application note later in this appendix entitled "***XOR T-equations on the MACH4xx***".

Design Implementation

The design implementation phase consists of optimizing and fitting the design (MACHXL takes care of these). In this phase you will have to set optimizing parameters that give the best implementation. You may also cover designs not fitting on the first attempt. Occasionally the design may fit, but may not meet your speed or size requirements. In these cases there are things you can do to help MACHXL fit the specific MACH part while meeting the design requirements.

When implementing a design, the best results are achieved when you select optimizing parameters tuned to the specific MACH part used. These parameters control MACHXL's node collapsing. These include tradeoffs in equation size, feedback passes through the array, and routing requirements.

For more information on selecting optimizing parameters see the application note entitled "***Guidelines for MACH Specific Optimization***".

If a design fails to fit, there are several tools to help you find the problem(s). These include the *.log* file, the *.rpt* file and MACHXL's ability to partition your design.

Each time the MACHXL fitter runs it produces a *.log* file. If the run succeeds, the *.log* file simply records the time of execution. If a fitting run fails, the *.log* file will contain information that explains (to the degree possible) why the design did not fit. If you are using group and pin assignments in the *.pi* (physical information) file, the log file will contain any messages regarding the validity of these assignments. The *.log* file is the first place to look when you have fitting problems.

See the application note later in this appendix ***entitled "Understanding the .log File Messages***" for information on the contents of the *.log* file.

When you specify a MACH device in the *.pi* (physical information) file, the MACH fitter generates a device-specific *.rpt* file. The *.rpt* file is generated whether the fitter succeeds in fitting or not. If the fitter fails, the *.rpt* file may be incomplete, but will contain valuable information showing which resources presented the most problems in fitting. This may help to change the design or the *.pi* file to make the design more fittable.

See the application note "***Reading the MACH .rpt File***" later in this appendix for more information.

Even if you are trying to fit a design into one device, it may be better to let the partitioner in MACHXL use multiple MACH devices. This is especially true for designs that fit all but one or all but a few functions. By letting MACHXL's partitioner fit the design into two devices, it's easier to determine which functions are causing problems.

Some techniques described in the application note ***entitled "MACH and the NUMDEVS Parameter"*** later in this appendix can help complete the fitting.

If your design fits, you may still want to adjust it to take advantage of certain speed/space tradeoffs available in the MACH4xx. These include the use of input registers, and controlling use of the MACH4xx asynchronous mode and T-flip-flop synthesis.

See the following application notes later in this appendix for more information.

"***Using MACH Input Registers***"

"***Control of the Asynchronous Mode in the MACH4xx***"

"***Control of T-Flop Synthesis in the MACH4xx***"

Design Testing

In the design testing phase, you are burning parts and testing them with vectors in the JEDEC file. These vectors are generated by the simulator using the *.stm* file.

Errors reported by the tester can usually be tracked down to a few sources. These are discussed in the following sections in this appendix. If you can't track the errors from these sections, try a different part. The test vectors are intended to weed out bad parts, however the electronically erasable MACH parts are fully tested by the manufacturer and seldom faulty.

For more information see the following application notes in this appendix.

"***Analyzing Test Vector Errors***"

"***MACH4xx Power-On Reset***"

"***Hazard Free Combinatorial Latches***" .

Design Integration

The most important aspect of the integration step it to produce a pinout suitable for board layout and duplicated in the event of design changes. This can be difficult with complex programmable devices. We have developed some techniques at AMD PLD Applications (SW) (call us during regular business hours at 1 800 222-4323 within the U.S. or 1 408 749-5703 internationally) which can assist with this process.

The first note is simply on MACH Pin Identification. This will help with reading and manipulating *.pi* files, the source file for all pin assignment information.

See the application note entitled "***MACH Pin and Node Identification***".

In cases where you are not committed to a specific pinout you can guide the fitter to a suitable pinout while letting the fitter have some flexibility in partitioning the design for ease of routing. This will leave more resources for future design changes.

See the application note entitled ***"Achieving Satisfactory Pinouts with MACH Devices***".

If you need to duplicate a pinout to which you are already committed, MINC has developed helpful techniques that may be of help.

See the application note later in this appendix entitled ***"Refitting into MACH Devices***".

The MACH fitter generally leaves unused I/O pins floating, leaving it to the user to tie them to VCC or GND. In some integration environments it is necessary to drive any unused pins from within the device rather than tying the pin to a constant voltage. An application note describes how to force these pins to be driven. See ***"Forcing Unused MACH Outputs to be Driven***".

Application Note:

Summary of MACH Family Devices

This application note provides an overview of the device capabilities in the MACH family. The AMD MACH Data Book is the authoritative source of this and similar information.

MACH Family of Devices

Device	Pins	Macro cells	PAL Blks	Inputs/ Blk	Max Pterms/ Macrocell	Output Macrocells	Buried Macrocells
MACH110	44	32	2	22	12	32	0
MACH111	44	32	2	26	12	32	0
MACH120	68	48	4	26	12	48	0
MACH130	84	64	4	26	12	64	0
MACH131	84	64	4	26	12	64	0
MACH210	44	64	4	22	16	32	32
MACH211	44	64	4	26	16	32	32
MACH215	44	64	4	22	12	32	0
MACH220	68	96	8	26	16	48	48
MACH221	68	96	8	26	16	48	48
MACH230	84	128	8	26	16	64	64
MACH231	84	128	8	32	16	64	64
MACH355	144	96	6	33	20	96	0
MACH435	84	128	8	33	20	128	0
MACH445	100	128	8	33	20	128	0
MACH465	208	256	16	34	20	256	0

Device	Input Reg	Max Inputs	Max Outputs	Clks	Speed ns
MACH110	0	38	32	2	12,15,20
MACH111	0	38	32	4	7, 10, 12, 15, 20
MACH120	0	56	48	4	15,20
MACH130	0	70	64	4	15,20
MACH131	0	70	64	4	7, 10, 12, 15, 20
MACH210	0	38	32	2	7, 10, 12, 15, 20
MACH211	0	38	32	4	7, 10, 12, 15, 20
MACH215	32	38	48	2+32	12, 15, 20
MACH220	0	56	48	4	12, 15, 20
MACH221	0	56	48	4	7, 10, 12, 15, 20
MACH230	0	70	64	4	10, 15, 20
MACH231	0	70	64	4	7, 10, 12, 15, 20
MACH355	0	102	96	4	15,20
MACH435	64	70	64	4	12, 15,20
MACH445	64	70	64	4	12, 15,20
MACH465	128	146	128	4	12, 15,20

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 can select its OE from either of the two available pterms or select either 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 can select its OE from either of the two available pterms or select either constant '1' or '0'.

MACH 215, MACH 4xx

These devices have an OE pterm per output. They can be programmed independently to '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 that in the MACHXL system, 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, on an individual function basis.

Clock Functions

MACH 1xx, MACH 2xx

These devices support pin clock only.

MACH 215

This device supports pin clock or clock by pterm. The output macrocells can be clocked by pin 13 or by a local pterm or by the inverse of either of those signals. The input registers can be clocked by either pin 13 or pin 35 or by the inverse of either of those signals.

MACH 4xx

This device supports clock by pin or clock by pterm. The pin clock mode can select from any of four clock pins or the inverse of those signals. Not all possible clock signal and inverse combinations are available in a given block. See device manufacturer literature for specifics.

For all MACH devices the clock signals are also signal inputs to the switch matrix and can be routed to the blocks.

Packaging

All like pin-count packages are pin compatible. Therefore, when a MACH110 design, for example, exceeds the capacity of the device, a MACH210 can generally be substituted.

Low Power Mode

The MACH211 has a low-power mode selectable pin-by-pin or for the whole device. This mode lowers power consumption, but also limits the speed of the device.

Application Note:

MACH Designs With Complex Clock Functions

Devices: All MACH

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

MACH Clock Limitations

The synchronous MACH parts (MACH1x0 and MACH2x0) can only be clocked by pin.

The synchronous MACH parts (MACH215 and MACH 3 & 4 families) can clock by single pterms, and invert clock signals in most cases.

In either case, the fitter allows the user to generate and use a more complex clock than the part supports directly. This could be the sum of two or more pterms, or a single pterm on an asynchronous part. The user must describe an output that has the clock function as its data equation, and clocks by that output signal.

MACH 1 and 2

The complex clock output in the MACH 1 & 2 families can be used internally or externally as the clock. The only exception is if the MACH215 clock pin is unavailable. Then the clock signal is routed to the PAL blocks where it is needed and connected using the clock pterm.

MACH 3 and 4

A function generated in the MACH 3 & 4 family parts can be used internally or externally as the clock. The fitter will default to using the clock signal

internally to save the pins used in external routing. In this case you can declare the clock function to be a node to prevent the clock from taking an I/O pin.

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;
```

Application Note:

Fitting Asynchronous Functions in MACH Devices

Devices: MACH215 MACH4xx

Both the MACH215 and MACH4xx devices support asynchronous functions, but they have different capabilities, making some functions or groups of functions suitable for the MACH215 which will not fit in the MACH4xx.

Pterm Clock and Reset and Preset

Any equation requiring the MACH4xx to be in asynchronous mode must have at most one reset or preset equation. This is encountered specifically when the clock expression is a product term (pterm).

Functions of this type can fit only on the MACH215, 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 MACH4xx, 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 pairs of RESET and PRESET equations it cannot fit in one MACH4xx, but may fit in one MACH215. The following set of functions can fit only in a MACH215:

```
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;
```

Application Note:

XOR T-Equations on the MACH4xx

Devices: MACH4xx

Although the MACH4xx supports XOR and hardware TFF registers, if you are fitting an XOR T-equation you may need to insert a node between the equation and the T-register.

XOR-TFF Problem Defined

When a function requires both a TFF register and an XOR equation, it may not fit in MACHXL. Within the compiler, the XOR equation must be expanded to be placed as a T-equation. If the size of the expanded XOR equation is greater than 20 pterms, it must be placed on a node as an XOR equation where it can be fit.

Example

This design will not fit because equation *o2*.T expands to 24 terms.

```
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, the design will fit 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;
```

Application Note:

Guidelines for MACH-Specific Optimization

Devices: All MACH

There are specific optimizing parameters suitable for the MACH devices. Within this range of suitable parameters there are tradeoffs on equation size and speed.

Suitable Optimizing Parameters for MACH Devices

The following parameters are used in the *.pi* file for MACH designs.

For the MACH4xx:

```
{
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,
MAX_XOR_PTERMS            0
}
```

Optimizing Adjustments

The MAX_PTERMS (and MAX_XOR_PTERMS for MACH4xx) parameters are the most critical values affecting fitting and speed. We suggest selecting values from the list following those parameters. For MACH4xx, 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. The MACH 1xx/2xx devices do not support XOR.

		⇦ Larger ⇦ Faster		Smaller ⇨ Slower ⇨	
MACH4xx					
	MAX_PTERMS	20	15	10	5
	MAX_XOR_PTERMS	19	14	9	4
MACH 1xx/2xx					
	MAX_PTERMS	16	12	8	4

The Effect of MAX_PTERMS and MAX_XOR_PTERMS

The effect of changing the optimizing parameters can be checked by the nodes in the *.doc* file after optimizing. The number of nodes will generally decrease as the MAX_PTERMS parameter increases.

The effect of changing the parameters is summarized here:

Higher MAX_PTERMS

- More node collapsing
- Larger functions
- Faster implementation
- May increase routing requirements

Lower MAX_PTERMS

- Less node collapsing
- Smaller functions
- Slower implementation
- May increase routing requirements

Note that either High or Low MAX_PTERMS 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 allows 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.

In critical fitting cases, it may be necessary to try several optimizing values for satisfactory results.

Application Note:

Understanding the *.log* File Messages

Devices: All MACH

The *.log* file, *design_name.log*, stores error messages and status information emitted from the fitting system. When a design fails to fit, it is the first place to check for information on what caused the failure. The types of messages appearing in the *.log* file are described.

The *.log* File

The *.log* file from the fitter contains messages generated in the process of fitting a design. Some of these messages also appeared on the screen as the fitting was in progress, but many of them appear only in the *.log* file. Messages in the *.log* file appear in generally chronological order and range from informational to immediately fatal.

Information Messages

These messages track the progress of the fitter but do not provide information on specific signals that do or do not fit. The fitter may make several attempts at fitting different combinations of signals before it succeeds or fails. These messages delimit the attempts. Generally, the first attempt will have the most specific information on why a design fails to fit.

"Attempting to fit at <UTILIZATION_VALUE> percent utilization."

Identifies an attempt to fit into an AMD MACH device at the indicated utilization. The fitter may repeat fitting attempts at lower utilizations until a fit is achieved.

"Attempting to fit a reduced partition."

Identifies an attempt to fit into an AMD MACH device after removing one or more functions from the prior attempt to fit. The fitter may repeat attempts at reduced partitions until a fit is achieved.

"Repeated results of last partition."

A particular MACH partition attempt reduced or partitioned at a lower utilization is discarded because it exactly repeats the prior failed partition.

General Failure Messages

The following general failure messages indicate the fitting point where an error occurred.

"Initial routing of signals through switch matrix failed: <PART#:DEV#>."

The current partitioning could not be routed by the MACH fitter.

"Failed to find suitable node assignment and signal routing: <PART#:DEV#>."

The current partitioning could not be placed and routed by the MACH fitter. It could be routed with no placement considerations or placed with no routing considerations but no valid combination of placements and routings could be found.

"Failed to generate fuse map: <PART#:DEV#>."

A problem occurred in assigning pterm rows within the MACH part.

"Cannot resolve OE requirements of macrocells."

The fitter cannot satisfy PAL block output enable requirements.

"MACH <DEVICE | PAL_BLOCK> partitioning exceeds limits"

The MACH partition cannot be reduced to the current limit due to a group of functions placed in a *.pi* file DEVICE or SECTION or because of internal feedback grouping.

"DEVICE number <DEV#> in the PI file contains functions which cannot be fit by <PART#>."

The DEVICE contains functions which are not fittable by this MACH part.

"MACH failed <DEVICE | PAL_BLOCK> pre-partitioning."

The partitioner cannot divide the functions into the required number of partitions while remaining within the current limits. This is a failure in applying *.pi* file DEVICE or SECTION groups.

"MACH failed <DEVICE | PAL_BLOCK> partitioning."

The partitioner cannot divide the functions into the required number of partitions while remaining within the current limits. This is a failure during automatic partitioning.

"MACH failed to route clock signals to PAL blocks."

The partitioner could not accomodate the combined clock requirements of the PAL blocks as partitioned (check clock polarity and clock pin assignments).

"No functions are remaining which can fit into a <PART#>."

The fitter has run out of functions which can fit into a MACH device of type <PART#>.

Pin Assignment Messages

When the *.pi* file specifies pins for specific signals, two classes of errors are possible. The first is invalid pin assignments. These errors are listed here. The second is invalid groupings, where no single pin assignment is in error, but some combination of assignments in a given device or PAL block is in error.

"MACH clock signal <SIGNAL_ID> must be assigned to a clock pin"

The clock signal has been placed on a pin other than a clock pin. The signal needs to be on a clock pin in order to clock one or more functions.

"Signal <SIGNAL_ID> cannot fit due to invalid pin type."

User-specified pin assignment places signal on invalid pin.

"Signal <SIGNAL_ID> is assigned to multiple pins."

The MACH pre-partitioner could not implement the multiple pin assignments of the *.pi* file.

"Function <SIGNAL_ID> cannot fit on pin <PIN#> because:"

MACH function pin assignment cannot be satisfied for the reason(s) listed. The possible reasons are shown below:

"Functions <SIGNAL_ID> and <SIGNAL_ID> use the same macrocell."

The named functions are assigned so that they require the same macrocell.

"Signal <SIGNAL_ID> cannot fit as a registered input as required by pin <PIN#>."

The signal does not meet the qualifications for a registered input (unary) signal, but the pin specified requires the signal be fit that way.

"Exceeds PAL block enable limit."

The combined pin assignments exceed the number of enable terms for a PAL block.

"Exceeds PAL block pterm allocation capabilities."

The combined pin assignments exceed the ability to assign product terms.

"Signal may require split pin; split pin limit is exhausted."

The fitter budgets split pins (biput converted to node and input) based on the output (non-split pin) count for each PAL block. The fitter has detected no more split pins are available.

"Undetermined reason."

The fitter cannot determine why the signal did not fit with other signals assigned to the device or PAL_block.

"Input <SIGNAL_ID> cannot fit on pin <PIN#> because:

MACH input pin assignment cannot be satisfied for the reason(s) listed:

"Clock pin needed for clock."

An input is assigned to a clock pin that must be reserved for a clock signal.

"Biputs-as-inputs exceed PAL block limits."

The sum of inputs and outputs/biputs exceed the device/PAL block limits.

"Signals <SIGNAL_ID> and <SIGNAL_ID> use the same pin."

The named signals are assigned to the same pin.

"Node assigned to buried logic has fanouts assigned to another device."

The function cannot be placed on a buried macrocell because a fanout of the function is on another device.

Grouping Messages

Signals may be placed in groups by using a GROUP or SECTION statement in the *.pi* file, or by assigning signals to pins in the same PAL block which

implicitly groups them. Some combination of signals cannot be groups because of device constraints. The following messages address these issues.

"PAL block <BLOCK_ID> is not valid for device <PART#>"

A *.pi* file SECTION contains a reference to an invalid PAL block name.

"PAL block <BLOCK_ID> cannot satisfy reset/preset requirements of all functions"

The PAL block partition contains functions which are interdependent due to reset/preset requirements. The inverted form of one function must be fit with the true form of the other, or vice-versa, however they cannot both fit in an acceptable form due to cluster availability constraints.

"Function <SIGNAL_ID> cannot fit on pin <PIN#> due to buried register fanout constraints."

User pin assignments violate restrictions on MACH230 buried macrocell fanouts. MACH230 buried register fanouts must be within PAL block pairs (A-H, B-G, C-F, D-E).

"Function <SIGNAL_ID> cannot fit due to grouping constraints."

Signal in user specified grouping or pin assignment violates MACH group constraints. This is usually a conflict between pin assignments and PAL block targets in the *.pi* file.

"Function grouping in .pi file DEVICE for <PART#:DEV#> exceeds limits for:"

"Function grouping in .pi file SECTION for <PART#:DEV#> PAL block <BLOCK_ID> exceeds limits for:"

"--- Number of functions"

"--- Number of signals"

"--- Number of clocks"

"--- Number of output enables"
"--- Number of reset and presets"
"--- Number of pterm clusters"
"--- Number of inputs"
"--- Number of pins"
"--- Number of outputs"
"--- Number of feedback paths"
"--- Number of input registers (or invalid assignment)"

Either of the first two message lines is followed by one or more of the resource constraints on the following lines. This indicates the DEVICE or SECTION group violated the indicated resource limits for the MACH part.

"The group consists of the following functions"

"--- <SIGNAL_ID>"

"--- <SIGNAL_ID>" ...

This message contains a listing of functions in a group which violates MACH constraints. It may follow any other grouping error message.

Examples

***.log* File of a Successful Fit:**

```
PLFit V3.1 - Patent (5,140,526) - Copyright MINC
Incorporated 1987-1993

checking: ...
fitting ...
Elapsed time: 00:00:49
```

***.log* file with a specific violation, in this case, "Number of Clocks":**

```
PLFit V3.1 - Patent (5,140,526) - Copyright MINC
Incorporated 1987-1993
checking: ...
fitting ...
Attempting device 1, template MACH435 ...
Warning:    Function grouping in PI file DEVICE for
MACH435:1 exceeds limits for:
Warning:          --- Number of clocks
Warning:    The group consists of the following functions
Warning:          --- BIT_1___p14
Warning:          --- BIT_2___p15
Warning:          --- BIT_3___p16
Warning:          --- BIT_4___p17
Warning:          --- BIT_5___p18
Warning:          --- BIT_6___p19
Warning:          --- BIT_7___p20
Warning:          --- BIT_8___p21
Attempting to fit at 100 percent utilization.
MACH device partitioning exceeds limits
Elapsed time: 00:00:03
-----------------------------------------------------------
```

***.log* file where place and route failed without a specific cause:**

```
PLFit V3.1 - Patent (5,140,526) - Copyright MINC
Incorporated 1987-1993

checking: ...
fitting ...
Attempting device 1, template MACH435 ...
Attempting to fit at 100 percent utilization.
Failed to find suitable node assignment and signal
routing: MACH435:1.
Attempting to fit a reduced partition.
Attempting to fit at 97 percent utilization.
Repeated results of last partition.
```

```
Attempting to fit at 94 percent utilization.
Repeated results of last partition.
Attempting to fit at 91 percent utilization.
Repeated results of last partition.
Attempting to fit at 88 percent utilization.
Repeated results of last partition.
Attempting to fit at 85 percent utilization.
MACH device partitioning exceeds limits
Elapsed time: 00:37:28
```

Application Note:

Understanding the *.rpt* File

Devices: All MACH

A report file, *design_name.rpt*, is generated if and only if a MACH device is targeted in the *.pi* file. There will be one *.rpt* file for each MACH device in a TARGET statement. The contents of the *.rpt* file is described here. It is useful both to aid in understanding cases that do not fit and to determine how a design was fit and the resources it used.

Obtaining a *.rpt* File

To obtain a *.rpt* file, you must place both DEVICE and TARGET statements in the *.pi* file. No further specifications are required. If you use internal groupings or pin assignments, that also goes in the DEVICE statement but is strictly optional.

The simplest *.pi* file which will generate a *.rpt* file is as follows:

```
DEVICE TARGET
      'part_number amd <part number>';
END DEVICE;
```

Contents of the Report File

The report file contains device specific fitting information regarding the internal resources of the MACH device. It shows which macrocells and routing paths are used by each signal.

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 cannot 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 the user understand why the fit attempt failed, how far the fitting proceeded and what aspect of the fitting caused problems.

The MACH 1xx and 2xx fitter produces a slightly different format *.rpt* file output than the 435 fitter (and future members of the MACH 3 and 4 family). In either case, the report file has the same structure. The sections of the report file are listed here with a brief description of each.

Heading

Before any information section, the report file gives the date the design was run through the fitter, the part type and device number, the design name and user supplied design information.

Failure Disclaimers

If the design fails in partitioning or place and route, a disclaimer is printed immediately following the heading. This alerts the user the design did not fit successfully and the information may be missing or inconsistent.

Summary Statistics

This section summarizes the design in terms of number of inputs nodes and outputs which fall into certain catagories.

Device Resource Utilization

This section provides utilization statistics for the different resource types of the device and its PAL blocks.

Partitioner Report

This section shows how the design is partitioned into pal blocks.

Clock Assignments

In the MACH 3 and 4 family, this clock assignment section shows which pin clocks are used in which pal blocks.

Signal Directory

Here all inputs, outputs and nodes on the part are listed with specific assignment information for each signal.

Resource Assignment Map

Here we go through the device in physical order by pin and macrocell and show which resources are used by which signal.

An example of each section is shown below.

Heading

To identify the *.rpt* file by design and fitter run, the header contains the date and time the design was run through the fitter and the users information from the source file.

```
DATE: Thu May 6 20:40:18 1993        -- Date design was
                                     -- run

DESIGN: f215g12                  -- Design name Part name
DEVICE: MACH435:1                -- and position in PI file
                                 -- DEVICE statement list.


TITLE: FILE f215g12.src          -- User supplied
                                 -- information from
                                 -- .src file.
COMPANY: AMD, SANTA CLARA
PROJECT: MACH Certification
REVISION: 001
COMMENT: Mon Sep 14 14:08:26 1992
```

Failure Disclaimers

If the design fails in partitioning or place and route, a disclaimer is printed immediately following the heading. This alerts the user the design did not fit successfully and the information may be missing or inconsistent.

There are different disclaimers depending on where the fitting failed and the device type being fit.

If a MACH435 or MACH 1xx or 2xx family device design fails in 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 in 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.  Fanouts ending in '--' represent signals which
could not be routed.
```

The SIGNAL DIRECTORY information indicates how far the fitting process proceeded before it was abandoned. The un-routed and/or un-placed signals should point to the cause of fitting problems. You may need to modify the design or manually direct the partitioner to achieve a fit in the selected design.

If a MACH4xx design fails in 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 accruate except for
pin and macrocell designations.  The RESOURCE ASSIGNMENT
```

```
MAP may have missing or redundant signals and conflicting
resource assignments.
```

The relative conflict levels for each resource type are listed here. This indicates the reason for failure in fitting.

```
Pins         3
Input Regs   0
Macrocells   0
Pterms       352
Feedbacks    0
Fanouts      0
```

This disclaimer includes statistics showing which resource proved most troublesome during the fit operation, in this case the fitter had trouble assigning product terms. This gives you, the designer, key information on where to modify your design to attempt another fit.

Summary Statistics

This section summarizes the design in terms of number of inputs nodes and outputs which fall into certain catagories. It breaks out the nodes and outputs both by PAL block (how many function per block). Because the MACH 3 and 4 family have more ways to fit a function, the fitter provides more statistics for these designs.

MACH 1 and 2 statistics look like this:

```
5  Inputs
0  Registered/Latched Inputs
11 Outputs
0  Tri-states
0  Nodes

Functions by block ( 8, 3, 0, 0 )
```

MACH 3 and 4 statistics look like this:

```
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
Asynchronous Equations          0
Single-Pterm Equations          32
Total Pterms Required           32
```

Note the total of 'Outputs', 'Tri-states' and 'Nodes' should equal the total of 'Function by block' and the total of the 'Macrocells' and 'Input Registers/Latches' statistics. The numbers from XOR Equations' down are not mutually exclusive nor should they match the total number of functions.

Device Resource Utilization

This section provides utilization statistics for the different resource types of the device and its pal blocks. First, the global resource utilization is presented, then resource statistics for each PAL block are listed.

Because of the different architectures of the MACH 1 and 2 family and the MACH 3 and 4 family, each has a slightly different set of resource statistics. These examples show the global statistics and one PAL block statistic set for each device family.

The MACH 1 and 2 family resource statistics look like this:

```
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
Product Terms:     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
Product Terms:      64          12          16          75
```

The MACH 3 and 4 family resource statistics look like this:

```
Resource          Available     Used        Remaining    %
    Clocks:           4           1           3          25
      Pins:          70          67           3          95
Input Regs:          64           0          64           0
Macrocells:         128          96          32          75
    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 resources referenced in these tables are defined here.

Clocks	Clock pins used for clock signals
Pins	Input and I/O pins used in any capacity
Input Lines	Array inputs
I/O Macro	Output Macrocells
Total Macro	Output and Buried Macrocells
Product Terms	AND array rows used in equation generation
Input Regs	Dedicated input registers

Macrocells	Macrocells without output/buried distinction
Feedbacks	Inputs to the Switch matrix
Fanouts	Inputs to the AND Array(s)

Partitioner Report

This section shows which functions (outputs and nodes) are assigned to which pal block. 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 set/reset equations are required for the assigned functions.

Clock Assignments

In the MACH 3 and 4 family, the clocks signals may vary from one pal block to another. This clock assignment sections shows which clocks are required in which pal blocks, and which phase (true or inverted) is needed.

The CLOCK ASSIGNMENT section may have zero to four clock pins listed depending on how many clocks are used in the design. Here is an example with two clocks:

```
CLOCK ASSIGNMENTS:

Notes:      block usage 'H' indicates used in TRUE sense.

            block usage 'L' indicates used in INVERSE
            sense.

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
```

This design uses CLK0 on pin 62 and it 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'.

Signal Directory

Here all inputs, outputs and nodes on the part are listed with specific assignment information for each signal. The format of this section is different for the MACH 1 and 2 family and for the MACH 3 and 4 family.

The MACH 1 and 2 signal directory looks like this:

```
SIGNAL DIRECTORY:

Notes:        Leading '-' indicates signal not
              assigned.
              Trailing '+' indicates feedback path is from
              pin.
              Functions with '0' Clusters are input
              registered.

Signal              Source       PalBlk       Pal Block Inputs
# Name              Type           Clusters
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 +
```

This example is cut short due to space. Every input, output and node is listed in this directory. The data columns are defined here:

Signal

The index number used to reference the signal

Signal Name

The user Identifyer for the signal

Source Type

{Input | Hidden | Output | Biput | Internal} with register type qualifiers

PalBlk
PAL block where output or node is assigned

Clusters
Number of Pterm Clusters used to generate function

Pal Block Inputs
Array input lines for Signal Fanouts

The MACH 3 and 4 signal directory looks like this:

```
SIGNAL DIRECTORY:
Notes: Register type suffix '_X' indicates XOR used;
       Register type suffix '_A' indicates Asychronous 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 has the signal index, signal type and signal name. The signal index, always in brackets, is used in the RESOURCE ASSIGNMENT MAP to identify the signal since there is not always enough room for the full signal name. The Signal type is one of {Input | Reg. Input | Reg. Feedback | Node | Tri-state | Output}.

The second line contains the assignment information for the signal. If the signal appears on a pin, the pin number and type is provided. Function and Inputs on I/O pins provide the block number of the pin and/or macrocell. Functions provide macrocell assignment information along with specifics on how the function is fit. This includes the number of pterms the functions require, the register type used to implement the function, and notes if the function implementation has any notable characteristics. These are noted in the 'Notes:' section at the top.

Resource Assignment Map

This section follows the physical layout of the device and shows where each signal is assigned. As with the SIGNAL DIRECTORY, the format of this section is different for the two families of MACH devices.

The MACH 1 and 2 family is simpler to represent since there is a one-to-one relationship between pins, macrocells, and array inputs. The format of the RESOURCE ASSIGNMENT MAP looks like this:

```
RESOURCE ASSIGNMENT MAP:
MINC          Node          Pin/Macro     Signal
Node#         Type              ID        (###)        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  I0                   ( 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
```

This example is cut short due to space. Every input, output and node is listed in this directory. The data columns are defined here:

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 identifyer
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 family is more complex to represent since the paths between pins, macrocells, and array inputs are programmable. The format of the RESOURCE ASSIGNMENT MAP looks like this:

```
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 [---] -
```

This example is cut short due to space. Every input, output and node is listed in this directory. The data columns are defined here:

PINOUT	Signals on physical pins
Pin	Physical pin number
[Sig]	Signal index of pin signal
PLACEMENT	Resources used to generate nodes and outputs
InReg/Mcell	Input Register (IR) or Macrocell (MC) identifier
[Sig]	Signal index of node or output
Pterms EAS	Pterm steering (See below)
ROUTING	Signals into and out of Switch Matrix
Feedback	ID Identifyer of Switch Matrix input
Feedback [Sig]	Signal index of feedback signal
Feedback Src	Source directed to Switch Matrix { Pin | IR | MC }
Fanout	PAL block inputs assigned to signal

Pterm steering is indicated for three pterm clusters per macrocell. The three clusters, designated 'E', 'A' and 'S', are the 'equation', 'asynchronous' and 'single' clusters. The E cluster consists of the two pterms which are always part of the data equation. The A cluster is the two pterms which are either used as part of the data equation or used as asychronous clock and reset. The S cluster is the single pterm which is either part of the data equation or half of the XOR equation. The steering of these clusters is designated by the characters '=', 'u', 'd' and 'U', which mean 'local macrocell', 'up one macrocell', 'down one macrocell', or 'Up two macrocells' respectively. 'Up' and 'down' are not necessarily physically up or down the printout. Up is to a lower numbered macrocell, while down is 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.

Application Note:

MACH and the Number of Devices Constraint

DEVICES: ALL MACH

In cases which fail to fit into a single MACH device, the "Number of Devices" constraint (NUMBER_DEVICES in the *.cst* file) should be removed to investigate the problem. Two alternative approaches to investigation are provided here.

The Problem

When you want a single device solution, it is natural to set the "Number of Devices" constraint to 1. In the case where the design does not immediately fit into one device, this provides minimal information on why the design does not fit.

Using 'default' in the .pi File Entry

One alternative is to force all of the design into the first part. It may appear setting NUMDEVS to 1 would do this, but it is really a much weaker statement, saying only "quit after one device is filled". This is reasonable in the context of automatic device selection and a device limit greater than one.

To force the entire design into one part, a MACH210 for example, use the 'default' signal reference in a DEVICE statement int the *.pi* file. The default reference is the same as naming all signals in the design not mentioned elsewhere in the *.pi* file.

Example

```
DEVICE
      TARGET 'part_number AMD MACH210-15JC';
      default;
END DEVICE;
```

When you do this, the design may fit the first time. If it does not, the *.log* file may contain valuable information about why the circuit cannot be fit.

It may exceed device limits such as Reset/Preset constraints. In this case, you may need to adjust the design to the limits of the device, or use another part or parts with greater resources.

The fitter may not be able to find a suitable partition. In this case the *.log* file indicates that this is the case. The *.rpt* file shows you the best partition the system could produce and why it is not valid for the device. At this point, you may see a better partition, or you may again need to adjust the design or implementation specifications.

Finally, the partitioner may be able to assign functions to PAL blocks, but the fitter may fail at place and route. Again, the *.rpt* file will show how far the fitter proceeded and what areas proved troublesome. The designer may be able to assist the fitter by adjusting the design and/or providing some direction to the fitting process through the *.pi* file.

In general, look for resources which are in high utilization. If macrocells are in high demand, more node collapsing may relieve the problem. If pterms are in high demand, you might try extracting some common factors into a common node. At any rate, knowing why a design doesn't fit is the first step to solving a fitting problem.

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 are being left out of the first device.

If you generate fusemaps for the two device solution, the .npi file may allow you to work one or two functions back into the first device. To do this, edit the *.npi* file to make a new *.pi* file. In the process, take the functions assigned to the second device and include them in the DEVICE statement for the first device but without any pin assignments. The fitter may be able to work them in even when it could not find a solution on the first pass.

If that does not work, you may be able to adjust the design by node collapsing or factoring to allow room for the left out functions, or you may conclude the design requires a larger device.

Application Note:

Using MACH Input Registers

DEVICES: MACH2xx, MACH4xx

The MACH 2xx and 4xx devices are capable of registering signals between the I/O pin and the switch matrix. In the MACH215 and MACH4xx, there is a dedicated register for each I/O pin. The other MACH2xx devices use the buried macrocell adjacent to the pin to perform the registration. The MACH fitters attempt to use these registers as often as possible because their use saves both routing resources and propagation delay. This application note describes how to detect when these resources are used and how to direct the fitter to use them fitting your design.

Input Register Pin Names

The MACH4xx and MACH215 have dedicated hardware for the input register function. These are called Unary pins in MACHXL because they support a function of exactly one signal. In both devices, the pin is designated as 'UNARY_OF_##', where '##' is the associated physical pin number.

In the MACH 2x0 devices, the pin signal is registered by routing it 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 designated as 'BURIED_OF_##' where '##' is the associated physical pin number. To force use of the input register mode, it is not enough to assign a signal to that pin. The assignment is ambiguous and will be interpreted by MACHXL as an internal node assignment. Later in this application note we will discuss how to make an input register assignment on these devices.

MACH 2xx vs MACH4xx

The MACH4xx devices have separate input register resources. Because this much simplifies the fitting of unary functions, these assignments are simple

and direct. Any unary function can be manually assigned to UNARY_OF_<pin>, or can be placed by the fitter in automatic fitting. Further, the MACH4xx is able to automatically use these resources to register the feedback of an output function. Because of the simplicity of this mechanism, the remainder of this application note covers the MACH2xx series parts.

The MACH215 does have separate hardware for input registers, but because of its general architecture, it is handled by the MACH 1xx/2xx fitter and shares the restrictions of that fitter.

Input Registration

The input register configuration has several advantages over the conventional routing where the input goes into the switch matrix, is brought to the PAL block array and fit as any other node. 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 MACHXL Version 3.0, there is no "INPUT CLOCKED_BY ..." construct, so fitters look for nodes that have a single signal as the D equation. These functions are referred to as 'unary' functions because they are functions of one signal. Fitters for devices with input registers automatically fit unaries on input registers whenever possible.

The MACH 2xx user may 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;
```

Detection

To detect signals which fit as unaries, the user must inspect the *.rpt* file. In the signal list section of the *.rpt* file is a column with the number of clusters used for each function. A function with zero ('0') clusters was fit as a unary.

In the example shown above, the function 'u' is fit as a unary as shown in this extract from it's *.rpt* file:

```
Signal      Source            PalBlk    Pal Block Inputs
# Name      Type                Clusters
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 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, use the *.pi* file and simply place the input signal on an I/O pin and the function on the adjacent BURIED macrocell.

The following *.pi* statements when used with the above example source file, will 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 :BURIED_OF_4;
END DEVICE;
```

Preventing a Function From Being Fit as Unary

To prevent a function from being fit as a unary, the user should fix either the input or the function signal to a pin. The pin may be the same pin which was previously fit automatically as a unary. The fact that one but not both signals is fixed is sufficient to prevent the unary configuration.

Application Note:

Control of the Asynchronous Mode in the MACH4xx

Devices: MACH4xx

This application note explains how the MACH4xx fitter uses the asynchronous macrocell feature of this device. It explains how the designer can manually control the implementation of asynchronous clocking of functions.

The MACHXL fitter operates on the assumption that asynchronous fitting is an available option to it, but at a resource and timing cost. The fitter tries to fit designs without resorting to asynchronous mode if this does not require leaving PAL-blocks underutilized, or using extra devices.

If a design is to be fit using asynchronous mode, the fitter will select the block reset and preset, and the block clock signals so as to minimize the number of macorcells that are fit in asynchronous mode.

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. Therefore, an algorithm minimizing the number of asynchronous resets should be adequate.

On the other hand, this algorithm may not be enough to select the functions using asynchronous clocking. The difference in timing between the pin clock and an array pterm generated clock signal may be of overriding importance to the designer.

By using manual grouping and selecting the signals are placed on the clock pins, the designer can control which functions are clocked asynchronously.

Application Note

Control of T-Flop Synthesis in the MACH4xx

DEVICES: MACH4xx

This application note covers the implementation of equations in the MACH4xx device. For some equations, the T-Flop may have a smaller equation, but has slightly greater delay. For speed-sensitive circuits, the designer may wish to use D-flops exclusively. The XOR in the MACH4xx provides for relatively efficient implementation of T equations using the D register.

Normal Operation

Unless otherwise directed, the fitter will fit the smallest equation of D, T, or XOR, or their complements.

DFF Only Fitting

If the designer does not want the TFF mode of the macrocell to be used, he must first design his circuit in terms of DFF equations. This is the default that will be generated if the designer does not reference T_FLOP or other register types.

The next step in the procedure is to assert the .pi file option "{ FF_SYNTH OFF }" in the designs *.pi* file. This will restrict the design to fitting only DFF equations.

This option can be applied to specific signals or to the entire device or design. This is controlled by the scope of the option placement.

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 the MINC language is

```
T_FLOP OUTPUT F CLOCKED_BY clk ...;
F = (T);
```

while the DFF equivalent function is

```
OUTPUT F CLOCKED_BY clk ...;
F = (T) (+) F;
```

Application Note:

Analyzing Test Vector Errors

DEVICES: All MACH

When a programmed device fails the JEDEC test vectors, there are some common areas to investigate to determine the source of the problem. This application note discusses some common sources of test vector problems (for additional information see the next two application notes entitled "*MACH Power-On Reset*" and "*Hazard Free Combinatorial Latches*".

Simulator Warnings

The simulator portion of MACHXL is the authority on exactly what functionality the source language defined. Its output file, *design_name.sim*, may contain warnings indicating the circuit functionality conflicts with the test vectors defined in your *.stm* file. These conflicts should be resolved before you can expect to pass test vectors.

It is often easiest to let the simulator determine the output value. You can use the '.S.' value in the *.stm* file test language to select this choice.

Initial States

A recurring problem in test vectors is conflicts during the initial states. The 'INITIAL' and 'INITIAL_TO' statements in the test language can be used to tell the simulator what assumptions to make regarding the initial state of signals.

It is good general practice to put a reset in the early steps of the test to place the device in a known state.

Glitches in Control Logic

If a register is clocked or set using a product of signals, it is important to realize the tester does not change all the inputs at one time. The tester may

generate glitches in the control equations which cause unexpected changes of state in a register.

If, for example, an output is clocked by (clk1 * clk2) and on a particular step clk1 goes from 1 to 0 while clk2 goes from 0 to 1, there is a 50/50 chance that clk2 will transition first producing a momentary pulse on the clock line. This can cause a change of state in the register which is not reflected in the simulation.

In such cases, it is best to use two test steps to insure that clk1 goes low before clk2 goes high.

Application Note

MACH Power-On Reset

Devices: MACH4xx

The MACH4xx has a built in power-on reset feature that sets all registers to a known state when power is applied to the part. This application note discusses how the user can determine the state of the registers, and steps the user can take to manage the power-on feature.

MACHXL DSL Reset Definition

MACHXL's 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 will go the the low-voltage state when it is reset. If the signal is a LOW_TRUE sense, then the a reset will cause the signal to go to the high voltage state. In both cases, the signal is in its unasserted condition. This is a *logical* reset.

Nominal Case

Most applications of the MACH4xx will perform a *logical* reset on power-up. Registered signals will go to the unasserted state.

Exception Cases

Cases violating the power-on logical reset are flagged in the *.rpt* file signal directory with the string "RS_SWAP". These signals will receive a logical preset at power-on. This condition can be caused by one of two things.

1. Macrocells in asynchronous mode having a preset equation will perform a power-on logical preset. Functions which are fit using an asynchronous macrocell are flagged in the *.rpt* file signal directory with the string "ASYNC".

2. A function will perform 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).

Manual partitioning can prevent this "out-of-phase" condition. Manual partitioning may allow a function with a preset equation fit in an asynchronous macrocell to be fit in a synchronous macrocell if the function is not inherently asynchronous, (i.e., if it does not have a clock which is a product of multiple signals.)

Application Note:

Hazard-Free Combinatorial Latches

Devices: All MACH

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 logical design for a latch may be subject to hazard conditions that may cause the latch to fail.. This application note describes how to protect against hazard conditions by inserting redundancy into the latch equation.

Basic Latch Circuit

The basic transparent D-Latch expressed in MACHXL looks like the follwing:

```
INPUT Data;
INPUT LatchEnable;
NODE Dlatch;
DLatch = LatchEnable * Data
          + /LatchEnable * DLatch;
```

Hazard Term

A Karnaugh map will reveal a potential hazard when the LatchEnable goes from 1 to 0 while Data is asserted. In the MACH devices, it is possible to lose the data during this transition. To protect against this hazard condition, a "Cover Term" must be added to the DLatch equation. In addition, steps must be taken to prevent the Cover Term from being reduced out.

Hazard Free Latch

We suggest encapsulating the combinatorial latch function in a DSL procedure which adds the hazard Cover Term and the 'NO_REDUCE' option to the output. Here is the source for this procedure:

```
PROCEDURE DLatch(INPUT Data, LatchEnable; OUTPUT
DLatchOut NO_REDUCE);
DLatchOut = LatchEnable * Data
            + /LatchEnable * DLatchOut
            + Data * DLatchOut;           "Cover Term
END Dlatch;
```

Application Note:

MACH Pin and Node Identification

Devices: All MACH

This application note describes the naming convention for pins of MACH devices. It provides a table of which pins are on which device. As a companion to this information, see the table at the end of this appendix, *Complete List of MACH Pin Names*.

Naming Convention

MACH devices have both physical pins (the ones on the device package), and virtual pins (i.e., node locations within the device).

Physical pins are referenced by the pin number in the package diagram.

MACH virtual pins are named according to their characteristics and their location in the device. The virtual pin names are derived from the following base names which imply the listed characteristics.

BURIED_OF_
Buried pins are nodes internal to the device which can never be made visible to a pin. In the MACH 2xx parts, these are the odd numbered macrocells.

SHADOW_OF_
Shadow pins are biput macrocells that can be disconnected from an I/O pin. The macrocell is then used as a buried node, and the pin as an input. In the 1xx and 2xx parts all I/O pins have corresponding shadow pins.

UNARY_OF_
Unary pins are nodes which register a single signal. Most often they are input registers. In the MACH215 and MACH4xx input registers are available on all I/O pins.

MACROCELL_
This designator is used in the MACH4xx for all logic macrocells. This is because any of the macrocells can be buried or tied to an I/O pin. In fact, any of the macrocells can be tied to any one of four I/O pins. Since the macrocells

are neither truly buried as defined above, nor are they directly associated with an identifiable I/O pin, they are simply designated as macrocells.

The suffix of the virtual pin names indicates its location in the device. The first three pin base names are suffixed with an I/O pin number. This is the pin adjacent to the buried pin, tied to the shadow pin, or input to the unary pin.

The last base name, MACROCELL_, is suffixed with the PAL Block designator and location index within the block.

Pin Name Tables

This table shows the physical pin numbers according to PAL Block. It then provides the names of related virtual pins. In each case '##' refers to the associated physical pin number.

Device PAL Block	I/O Pins	Virtual Pins	
MACH110/111			
'A'	2-9, 14-21	SHADOW_OF_##	
'B'	24-31, 36-43	SHADOW_OF_##	
MACH210/211			
'A'	2-9	SHADOW_OF_##	BURIED_OF_##
'B'	14-21	SHADOW_OF_##	BURIED_OF_##
'C'	24-31	SHADOW_OF_##	BURIED_OF_##
'D'	36-43	SHADOW_OF_##	BURIED_OF_##
MACH215			
'A'	2-9	SHADOW_OF_##	UNARY_OF_##
'B'	14-21	SHADOW_OF_##	UNARY_OF_##
'C'	24-31	SHADOW_OF_##	UNARY_OF_##
'D'	36-43	SHADOW_OF_##	UNARY_OF_##
MACH120			
'A'	2-7, 9-14	SHADOW_OF_##	
'B'	21-26, 28-33	SHADOW_OF_##	
'C'	36-41, 43-48	SHADOW_OF_##	
'D'	55-60, 62-67	SHADOW_OF_##	

Device PAL Block	I/O Pins	Virtual Pins	
MACH220/221			
'A'	2-7	SHADOW_OF_##	BURIED_OF_##
'B'	9-14	SHADOW_OF_##	BURIED_OF_##
'C'	21-26	SHADOW_OF_##	BURIED_OF_##
'D'	28-33	SHADOW_OF_##	BURIED_OF_##
'E'	36-41	SHADOW_OF_##	BURIED_OF_##
'F'	43-48	SHADOW_OF_##	BURIED_OF_##
'G'	55-60	SHADOW_OF_##	BURIED_OF_##
'H'	62-67	SHADOW_OF_##	BURIED_OF_##
MACH130/131			
'A'	3-10, 12-19	SHADOW_OF_##	
'B'	24-31, 33-40	SHADOW_OF_##	
'C'	45-52, 54-61	SHADOW_OF_##	
'D'	66-73, 75-82	SHADOW_OF_##	
MACH230			
'A'	3-10	SHADOW_OF_##	BURIED_OF_##
'B'	12-19	SHADOW_OF_##	BURIED_OF_##
'C'	24-31	SHADOW_OF_##	BURIED_OF_##
'D'	33-40	SHADOW_OF_##	BURIED_OF_##
'E'	45-52	SHADOW_OF_##	BURIED_OF_##
'F'	54-61	SHADOW_OF_##	BURIED_OF_##
'G'	66-73	SHADOW_OF_##	BURIED_OF_##
'H'	75-82	SHADOW_OF_##	BURIED_OF_##
MACH435			
'A'	3-10 UNARY_OF_##	MACROCELL_A00 - MACROCELL_A15,	
'B'	12-19 UNARY_OF_##	MACROCELL_B00 - MACROCELL_B15,	
'C'	24-31 UNARY_OF_##	MACROCELL_C00 - MACROCELL_C15,	
'D'	33-40 UNARY_OF_##	MACROCELL_D00 - MACROCELL_D15,	
'E'	45-52 UNARY_OF_##	MACROCELL_E00 - MACROCELL_E15,	

Device PAL Block	I/O Pins	Virtual Pins
'F'	54-61	MACROCELL_F00 - MACROCELL_F15, UNARY_OF_##
'G'	66-73	MACROCELL_G00 - MACROCELL_G15, UNARY_OF_##
'H'	75-82	MACROCELL_H00 - MACROCELL_H15, UNARY_OF_##
MACH465		
'A'	190-197	MACROCELL_A00 - MACROCELL_A15, UNARY_OF_##
'B'	200-207	MACROCELL_B00 - MACROCELL_B15, UNARY_OF_##
'C'	3-10	MACROCELL_C00 - MACROCELL_C15, UNARY_OF_##
'D'	13-20	MACROCELL_D00 - MACROCELL_D15, UNARY_OF_##
'E'	32-39	MACROCELL_E00 - MACROCELL_E15, UNARY_OF_##
'F'	42-49	MACROCELL_F00 - MACROCELL_F15, UNARY_OF_##
'G'	54-61	MACROCELL_G00 - MACROCELL_G15, UNARY_OF_##
'H'	64-71	MACROCELL_H00 - MACROCELL_H15, UNARY_OF_##
'I'	86-93	MACROCELL_I00 - MACROCELL_I15, UNARY_OF_##
'J'	96-103	MACROCELL_J00 - MACROCELL_J15, UNARY_OF_##
'K'	107-114	MACROCELL_K00 - MACROCELL_K15, UNARY_OF_##
'L'	117-124	MACROCELL_L00 - MACROCELL_L15, UNARY_OF_##
'M'	136-143	MACROCELL_M00 - MACROCELL_M15, UNARY_OF_##
'N'	146-153	MACROCELL_N00 - MACROCELL_N15, UNARY_OF_##

'O'	158-165	MACROCELL_O00 - MACROCELL_O15, UNARY_OF_##
'P'	168-175	MACROCELL_P00 - MACROCELL_P15, UNARY_OF_##

Application Note:

Achieving Satisfactory Pinouts with MACH Devices

Devices: All MACH

This application note gives general guidelines for shaping the pinout configuration of a design fit into MACH devices.

Procedure

The general approach is to first fit the problem unconstrained to prove there is a solution; then mold that solution into a pin-out meeting the board layout requirements. The steps needed to do this are listed below:

1. Generate an unconstrained solution. Run the fitter to get an *.npi* file.

2. Copy the *.npi* file to the *.pi* file. Strip the pin assignments and take out the NO_CONNECT information.

3. Determine which sets of signals must be fit together on localized or sequential pins. The group statement will fit those signals in the same PAL block. Group those signals within the fixed group of the device. Leave the INPUT signals for later. Not every function must be in a group. It may help to sort the *.pi* file first to get signals with like names together, as they often are grouped together.

The *.pi* file will look like this:

```
DEVICE
TARGET 'PART_NUMBER AMD MACH130-15JC';
INPUT B20M;
INPUT NACKIO;
```

```
INPUT NACKI1;
 ...

TXC;

GROUP
      COL1;
      CRS1;
END GROUP;

GROUP
      COL2;
      CRS2;
END GROUP;

GROUP
      NACKO0;
      NACKO1;
      NACKO2;
END GROUP;

      ...
END DEVICE;
```

4. Run the fitter on the grouped *.pi* file. This shows which groups go best with other groups due to similar signal, OE, and RESET requirements. If this fails to fit the most likely cause is a group which violates the constraints of a PAL block. This will be noted in the *.log* file. Find the offending group and either dissolve it or divide it into two groups.

5. When the local groups are fit into PAL blocks, generate another *.npi* file and make that the current *.pi* file.

6. It may be necessary at this time to swap the contents between two PAL blocks. If the PAL block assignments are satisfactory, go to step 7. Remember that if you have outputs in different PAL blocks that must be adjacent, you can have them span the boundary of adjacent PAL blocks or the wrap-around between the last PAL

block and the first. Pal block assignment can be done by using targeted groups within the device groups. The PAL blocks are named 'A' through 'H', 'D' or 'B' depending on how whether there are eight, four or two PAL blocks in the device.

7. The first step is to target PAL blocks. The *.pi* file will show the pins of each signal but not the PAL block, refer to a pin-out table for the device and determine where the PAL block divisions occur. Divide the current *.pi* file into PAL block groups using the fixed group construct with target statements. Again, save the inputs for later. Then strip the pin numbers and reassign the groups as required.

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;
```

8. Now fit and generate an *.npi* file. If the fit fails, consult the *.log* file and make adjustments as required. One thing to try would be rotating the PAL block assignments ('A' to 'B', 'B' to 'C', ... 'H' to 'A').

9. When the PAL block assignments are satisfactory, generate another *.npi* file and make that the current *.pi* file.

10. The last step is to find suitable pin assignments within the PAL blocks. First, add comments to the *.pi* file to show the PAL blocks limits. Then separate all the inputs and strip off their pin numbers. Again, they will be handled last unless there are overriding reasons to place them earlier. The reasoning here is inputs have only routing constraints, while functions have routing, pterm allocation, and control function constraints and are generally the harder assignments.

11. The key here is to take small steps. Work on one PAL block at a time. Strip off the pin numbers. Pick one group of signals and assign it the desired pin assignment. Then run fit. If it fails, be sure to check the *.log* file, although often it will indicate routing could not be achieved. Try shifting 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. When you finish one PAL block, go on to the next. Be sure to leave room for sequential assignments of input groups. It may by helpful to leave biputs available adjacent to the dedicated input pins so input groups can fit across dedicated inputs and onto the biputs. Remember clock signals must go on clock/input pins.

Application Note:

Refitting into MACH Devices

Devices: All MACH

The first step to successfully refitting a MACH design is to plan from the beginning of design implementation to allow for refitting. 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 will undoubtedly change and changes may not refit the way the original design was fit. As much as possible, try to work with what the fitter would prefer to do, especially in terms of partitioning into PAL blocks, rather than forcing a pre-conceived pinout.

This application note describes the best method currently available to recreate a specific pinout for a MACH design. We assume you have fit a design, and produced a JEDEC map and a *.npi* file. The objective is to make a *.pi* file which reproduces the pinout.

For more information on these subjects, see the application notes entitled ***"Achieving Satisfactory Pin-outs with the MACH Fitter"*** and ***"MACH Pin and Node Identification"***.

Concept

The procedure described below preserves the PAL block partitioning of all the functions while allowing the 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 device pins. To do this, we convert the flat *.npi* file to a two level file adding PAL block SECTIONs within a DEVICE construct. In the PAL block groups we place all the outputs and all the buried nodes which were in one PAL block. In general, we leave the pin assignments on the outputs, but not on the nodes. Special attention must be paid to nodes which were brought to pins for routing purposes (i.e., omit pin assignment) and functions fit as registered inputs (i.e., preserve buried pin assignment).

The *.pi* file property FLOAT_NODES can be used to release the nodes from their pin assignments, while keeping them in the PAL block to which they were assigned. This is useful when trying to recreate a pinout. The FLOAT_NODES *.pi* file property can be used in replacement of the procedure described below.

Procedure

1) Fit the design using a template statement in the *.pi* file (the fitter). This *.pi* file produces the *design_name.rpt* file which will be needed later. The *.pi* file can be as simple as:

```
DEVICE TARGET 'PART_NUMBER amd mach210-15jc'; END DEVICE;
```

2) Document the design . This produces the *design_name.doc* file.

3) Implement the design. This produces the JEDEC file and also the *design_name.npi* file.

4) Copy *design_name.npi* to *design_name.pi*.

5) In the *.pi* file, move all inputs to the top (or bottom) of the file, preserve pin assignments.

6) Set up two, four or eight fixed groups depending on the device. See tables following for grouping information.

7) Segregate all outputs and nodes into sections according to which PAL block they were originally fit.

8) For MACH2xx parts, check the *.rpt* file for nodes fit using zero clusters (SIGNAL DIRECTORY section). These nodes are fit with input registers and the pin assignment must be preserved. See the previous section entitled ***"Using MACH Input Registers"***.

9) For MACH435, preserve any pin assignment to UNARY_OF_##. This is an input register assignment.

10)Check for nodes which have been fit on I/O pins and are not required on another device. The *.doc* file lists all nodes (at the top), and the wire list (at the bottom) shows which are wired to another device. You can drop the pin assignment on nodes which are not needed on another device.

11) Except as indicated in steps 7) and 8), drop all pin assignments for buried logic, and preserve all pin assignments for I/O pins.

12) Rerun the fitter. If the design fits successfully, you have a repeatable solution.

Example

A design was fit into a MACH230. The *.rpt* file contains the following lines in the signal directory indicating that 'df_reg[1]' and 'df_reg[2]' are fit on input registers:

```
Signal              Source              PalBlk     Pal Block Inputs
# Name              Type                  Clusters
68 df_reg[2]        DFF Hidden            A  0  A13
69 df_reg[1]        DFF Hidden            A  0  A12
```

In addition, we see node *df_reg[0]* is placed on a pin. This is done for routing purposes, since the signal is not needed outside the device.

Using the procedure above, we generate the *.pi* file below from the *.npi* file produced by the fuse mapper.

```
------------------- .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]:BURIED_OF_7;
df_reg[2]:BURIED_OF_8;
s0:SHADOW_OF_19;
s2:SHADOW_OF_18;
dcnt[0]:SHADOW_OF_17;
s1:SHADOW_OF_14;
dval:SHADOW_OF_13;
dcnt[2]:BURIED_OF_38;
dcnt[4]:SHADOW_OF_36;
prep_done:SHADOW_OF_35;
dcnt[5]:SHADOW_OF_33;
dcnt[3]:BURIED_OF_50;
dcnt[1]:SHADOW_OF_71;
dv_lvl0:BURIED_OF_80;
dv_lvl1:SHADOW_OF_76;
END DEVICE;

 ------------------- NEW .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]:BURIED_OF_7; "Part of input register assignment
df_reg[2]:BURIED_OF_8; "Part of input register assignment
END SECTION;

SECTION
frame:16;
s0;          ":SHADOW_OF_19;
s2;          ":SHADOW_OF_18;
dcnt [0];    ":SHADOW_OF_17;
s1;          ":SHADOW_OF_14;
dval;        ":SHADOW_OF_13;
END SECTION;

SECTION
dout[9]:24;
dout[8]:25;
dout[4]:26;
dout[3]:27;
END SECTION;

SECTION
dcnt[2];     ":BURIED_OF_38;
dcnt[4];     ":SHADOW_OF_36;
prep_done;   ":SHADOW_OF_35;
dcnt[5];     ":SHADOW_OF_33;
END SECTION;

SECTION
dout[18]:45;
dcnt[3];     ":BURIED_OF_50;
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];    ":SHADOW_OF_71;
END SECTION;

SECTION
df_reg[0];  ":75; This is a node on a pin
dout[13]:78;
dout[10]:79;
dout[0]:80;
dv_lvl0;    ":BURIED_OF_80;
dv_lvl1;    ":SHADOW_OF_76;
END SECTION;
END DEVICE;
```

Application Note:

Forcing Unused MACH Outputs to be Driven or Floating

DEVICES: MACH 1xx and 2xx

For MACH 1xx and 2xx devices, I/O pins having neither input or output signals may be driven or floating, depending on whether the associated macrocell (shadow pin) is used. If a hidden function is placed on the macrocell, the pin is placed in the high impedance or 'floating' state. If the macrocell is not used, the pin is placed in a driven state and a constant value is placed on the pin. This application note describes a method to configure these pins, should you need to do so.

This does not apply to the MACH435 because these outputs have built in pull ups on the outputs, providing a default input when left unconnected.

Some of the 1xx and 2xx family of devices are available with pull ups as well. Consult the AMD Data Book for this information. Also see the table at the end of this appendix, ***Fuse Commands for Forcing Outputs to be Driven.***

Forcing Outputs Driven

To force an output to be driven, the user must first assign all outputs to pins so that the unused pins are known. Then, in the *.pi* file, the user places fuses statements ('INTACT <fuse no.>' and 'BLOWN <fuse no.>' to modify the implementation.

The table, ***Fuse Commands for Forcing Outputs to be Driven,*** contains fuse assignment statements to assert the tri-state enable for unused pins on all MACH devices.

If a node was placed on the corresponding shadow pin, its signal is present on the pin. Otherwise, the pin will be asserted either high or low depending on how other unused internal resources are dispensed.

Example

An example *.pi* file would look like this. For space, we consider only PAL block 'A'. We have outputs on pins 2-9, and wish 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;
```

Forcing Outputs Floating

To force an output to float, the user must first assign all outputs to pins so the unused pins are known. Then, in the *.pi* file, the user places fuse statements ('INTACT <fuse no.>' and 'INTACT <fuse no.>' to modify the implementation.

The table, ***Fuse Commands for Forcing Outputs to be Driven,*** can be used to configure floating outputs. Just replace the 'BLOWN' keyword with the 'INTACT' keyword.

If a node has been placed on the corresponding shadow pin, its signal is present on the pin. Otherwise, the pin is asserted either high or low depending on how other unused internal resources are dispensed with.

Example

An example *.pi* file would look like this. For space, we only consider PAL block 'A'. We have outputs on pins 2-9, and wish 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;
```

Application Note:

Possible Pin Incompatibility Between MACH230 and MACH435

Devices: MACH230 and MACH435

In rare cases, designs fitting in a MACH230 are not pin compatible with the MACH435. We say 'rare' because it happens only when 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 3 and 4 families, this sense is reversed to provide the more common funtionality of **transparent high, latch low**.

Since the MACH435 can select clock polarity, this change is seldom a problem for the 435. Not all combinations of clock polarities for all clock pins are availible within a single PAL block. This means 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) allowing:

- both clocks TRUE
- both clocks inverted
- both phases of one of the clock pair.

A given PAL block cannot have one clock of the pair with true sense and the other with inverted sense.

Consider a MACH230 design with a register and latch in the same PAL block. Assume 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 MACH230 and the MACH435 mean the 435 must invert the latch enable to achieve the same functionality. This means the PAL block needs exactly the same clock polarity (but can't have it), having true sense of one member and inverted sense of the other.

If one of the functions is a node, it may be possible to move it to another block. It may also be possible to force one of the clocks to be asynchronous (clocking by pterm row) by using an internal node to produce the clock signal. The point is to be aware there may be problems in these rare cases with ported designs.

Application Note:

Complete List of MACH Pin Names

Devices: All MACH

This table gives a complete list of MACH device pin names and the internal pin numbers associated with each.

Pin Numbering

The internal pin numbers refer to the physical pins followed by the virtual pins. Internal pin numbers start with zero, so the first pin of a 44-pin package is 0 and the last is 43. The first virtual pin is 44. A similar numbering scheme is used for the 68 and 84 pin packages.

44-Pin Packages

These devices use 0 through 43 for the physical pins. Virtual pins begin at 44 and go on as required by the device. The MACH110 has 32 virtual pins, the MACH210 and MACH215 each have 64 virtual pins.

MACH110

SHADOW_OF_2,	SHADOW_OF_3,	SHADOW_OF_4,	SHADOW_OF_5,
SHADOW_OF_6,	SHADOW_OF_7,	SHADOW_OF_8,	SHADOW_OF_9,
SHADOW_OF_14,	SHADOW_OF_15,	SHADOW_OF_16,	SHADOW_OF_17,
SHADOW_OF_18,	SHADOW_OF_19,	SHADOW_OF_20,	SHADOW_OF_21,
SHADOW_OF_24,	SHADOW_OF_25,	SHADOW_OF_26,	SHADOW_OF_27,
SHADOW_OF_28,	SHADOW_OF_29,	SHADOW_OF_30,	SHADOW_OF_31,
SHADOW_OF_36,	SHADOW_OF_37,	SHADOW_OF_38,	SHADOW_OF_39,
SHADOW_OF_40,	SHADOW_OF_41,	SHADOW_OF_42,	SHADOW_OF_43

MACH210

SHADOW_OF_2,	BURIED_OF_2,	SHADOW_OF_3,	BURIED_OF_3,
SHADOW_OF_4,	BURIED_OF_4,	SHADOW_OF_5,	BURIED_OF_5,
SHADOW_OF_6,	BURIED_OF_6,	SHADOW_OF_7,	BURIED_OF_7,
SHADOW_OF_8,	BURIED_OF_8,	SHADOW_OF_9,	BURIED_OF_9,
SHADOW_OF_21,	BURIED_OF_21,	SHADOW_OF_20,	BURIED_OF_20,
SHADOW_OF_19,	BURIED_OF_19,	SHADOW_OF_18,	BURIED_OF_18,
SHADOW_OF_17,	BURIED_OF_17,	SHADOW_OF_16,	BURIED_OF_16,
SHADOW_OF_15,	BURIED_OF_15,	SHADOW_OF_14,	BURIED_OF_14,
SHADOW_OF_24,	BURIED_OF_24,	SHADOW_OF_25,	BURIED_OF_25,

SHADOW_OF_26, BURIED_OF_26, SHADOW_OF_27, BURIED_OF_27,
SHADOW_OF_28, BURIED_OF_28, SHADOW_OF_29, BURIED_OF_29,
SHADOW_OF_30, BURIED_OF_30, SHADOW_OF_31, BURIED_OF_31,
SHADOW_OF_43, BURIED_OF_43, SHADOW_OF_42, BURIED_OF_42,
SHADOW_OF_41, BURIED_OF_41, SHADOW_OF_40, BURIED_OF_40,
SHADOW_OF_39, BURIED_OF_39, SHADOW_OF_38, BURIED_OF_38,
SHADOW_OF_37, BURIED_OF_37, SHADOW_OF_36, BURIED_OF_36

MACH215

SHADOW_OF_2, UNARY_OF_2, SHADOW_OF_3, UNARY_OF_3,
SHADOW_OF_4, UNARY_OF_4, SHADOW_OF_5, UNARY_OF_5,
SHADOW_OF_6, UNARY_OF_6, SHADOW_OF_7, UNARY_OF_7,
SHADOW_OF_8, UNARY_OF_8, SHADOW_OF_9, UNARY_OF_9,
SHADOW_OF_21, UNARY_OF_21, SHADOW_OF_20, UNARY_OF_20,
SHADOW_OF_19, UNARY_OF_19, SHADOW_OF_18, UNARY_OF_18,
SHADOW_OF_17, UNARY_OF_17, SHADOW_OF_16, UNARY_OF_16,
SHADOW_OF_15, UNARY_OF_15, SHADOW_OF_14, UNARY_OF_14,
SHADOW_OF_24, UNARY_OF_24, SHADOW_OF_25, UNARY_OF_25,
SHADOW_OF_26, UNARY_OF_26, SHADOW_OF_27, UNARY_OF_27,
SHADOW_OF_28, UNARY_OF_28, SHADOW_OF_29, UNARY_OF_29,
SHADOW_OF_30, UNARY_OF_30, SHADOW_OF_31, UNARY_OF_31,
SHADOW_OF_43, UNARY_OF_43, SHADOW_OF_42, UNARY_OF_42,
SHADOW_OF_41, UNARY_OF_41, SHADOW_OF_40, UNARY_OF_40,
SHADOW_OF_39, UNARY_OF_39, SHADOW_OF_38, UNARY_OF_38,
SHADOW_OF_37, UNARY_OF_37, SHADOW_OF_36, UNARY_OF_36

68-Pin Packages

These devices use 0 through 67 for the physical pins. Virtual pins begin at 68 and go on as required by the device. The MACH120 has 48 virtual pins, the MACH220 has 96 virtual pins.

MACH120

SHADOW_OF_2, SHADOW_OF_3, SHADOW_OF_4, SHADOW_OF_5,
SHADOW_OF_6, SHADOW_OF_7, SHADOW_OF_9, SHADOW_OF_10,
SHADOW_OF_11, SHADOW_OF_12, SHADOW_OF_13, SHADOW_OF_14,
SHADOW_OF_33, SHADOW_OF_32, SHADOW_OF_31, SHADOW_OF_30,
SHADOW_OF_29, SHADOW_OF_28, SHADOW_OF_26, SHADOW_OF_25,
SHADOW_OF_24, SHADOW_OF_23, SHADOW_OF_22, SHADOW_OF_21,
SHADOW_OF_36, SHADOW_OF_37, SHADOW_OF_38, SHADOW_OF_39,
SHADOW_OF_40, SHADOW_OF_41, SHADOW_OF_43, SHADOW_OF_44,
SHADOW_OF_45, SHADOW_OF_46, SHADOW_OF_47, SHADOW_OF_48,
SHADOW_OF_67, SHADOW_OF_66, SHADOW_OF_65, SHADOW_OF_64,
SHADOW_OF_63, SHADOW_OF_62, SHADOW_OF_60, SHADOW_OF_59,
SHADOW_OF_58, SHADOW_OF_57, SHADOW_OF_56, SHADOW_OF_55

MACH220

SHADOW_OF_2, BURIED_OF_2, SHADOW_OF_3, BURIED_OF_3,
SHADOW_OF_4, BURIED_OF_4, SHADOW_OF_5, BURIED_OF_5,
SHADOW_OF_6, BURIED_OF_6, SHADOW_OF_7, BURIED_OF_7,

MACH220 (con't)

SHADOW_OF_14, BURIED_OF_14, SHADOW_OF_13, BURIED_OF_13
SHADOW_OF_12, BURIED_OF_12, SHADOW_OF_11, BURIED_OF_11,
SHADOW_OF_10, BURIED_OF_10, SHADOW_OF_9, BURIED_OF_9,
SHADOW_OF_21, BURIED_OF_21, SHADOW_OF_22, BURIED_OF_22,
SHADOW_OF_23, BURIED_OF_23, SHADOW_OF_24, BURIED_OF_24,
SHADOW_OF_25, BURIED_OF_25, SHADOW_OF_26, BURIED_OF_26,
SHADOW_OF_33, BURIED_OF_33, SHADOW_OF_32, BURIED_OF_32,
SHADOW_OF_31, BURIED_OF_31, SHADOW_OF_30, BURIED_OF_30,
SHADOW_OF_29, BURIED_OF_29, SHADOW_OF_28, BURIED_OF_28,
SHADOW_OF_36, BURIED_OF_36, SHADOW_OF_37, BURIED_OF_37,
SHADOW_OF_38, BURIED_OF_38, SHADOW_OF_39, BURIED_OF_39,
SHADOW_OF_40, BURIED_OF_40, SHADOW_OF_41, BURIED_OF_41,
SHADOW_OF_48, BURIED_OF_48, SHADOW_OF_47, BURIED_OF_47,
SHADOW_OF_46, BURIED_OF_46, SHADOW_OF_45, BURIED_OF_45,
SHADOW_OF_44, BURIED_OF_44, SHADOW_OF_43, BURIED_OF_43,
SHADOW_OF_55, BURIED_OF_55, SHADOW_OF_56, BURIED_OF_56,
SHADOW_OF_57, BURIED_OF_57, SHADOW_OF_58, BURIED_OF_58,
SHADOW_OF_59, BURIED_OF_59, SHADOW_OF_60, BURIED_OF_60,
SHADOW_OF_67, BURIED_OF_67, SHADOW_OF_66, BURIED_OF_66,
SHADOW_OF_65, BURIED_OF_65, SHADOW_OF_64, BURIED_OF_64,
SHADOW_OF_63, BURIED_OF_63, SHADOW_OF_62, BURIED_OF_62

84-Pin Packages

These devices use 0 through 83 for the physical pins. Virtual pins begin at 84 and go on as required by the device. The MACH130 has 64 virtual pins, the MACH230 has 128 virtual pins, and the MACH435 has 192 virtual pins.

MACH130

SHADOW_OF_3, SHADOW_OF_4, SHADOW_OF_5, SHADOW_OF_6,
SHADOW_OF_7, SHADOW_OF_8, SHADOW_OF_9, SHADOW_OF_10,
SHADOW_OF_12, SHADOW_OF_13, SHADOW_OF_14, SHADOW_OF_15,
SHADOW_OF_16, SHADOW_OF_17, SHADOW_OF_18, SHADOW_OF_19,
SHADOW_OF_40, SHADOW_OF_39, SHADOW_OF_38, SHADOW_OF_37,
SHADOW_OF_36, SHADOW_OF_35, SHADOW_OF_34, SHADOW_OF_33,
SHADOW_OF_31, SHADOW_OF_30, SHADOW_OF_29, SHADOW_OF_28,
SHADOW_OF_27, SHADOW_OF_26, SHADOW_OF_25, SHADOW_OF_24,
SHADOW_OF_45, SHADOW_OF_46, SHADOW_OF_47, SHADOW_OF_48,
SHADOW_OF_49, SHADOW_OF_50, SHADOW_OF_51, SHADOW_OF_52,
SHADOW_OF_54, SHADOW_OF_55, SHADOW_OF_56, SHADOW_OF_57,
SHADOW_OF_58, SHADOW_OF_59, SHADOW_OF_60, SHADOW_OF_61,
SHADOW_OF_82, SHADOW_OF_81, SHADOW_OF_80, SHADOW_OF_79,
SHADOW_OF_78, SHADOW_OF_77, SHADOW_OF_76, SHADOW_OF_75,
SHADOW_OF_73, SHADOW_OF_72, SHADOW_OF_71, SHADOW_OF_70,
SHADOW_OF_69, SHADOW_OF_68, SHADOW_OF_67, SHADOW_OF_66

MACH230

SHADOW_OF_3,	BURIED_OF_3,	SHADOW_OF_4,	BURIED_OF_4,
SHADOW_OF_5,	BURIED_OF_5,	SHADOW_OF_6,	BURIED_OF_6,
SHADOW_OF_7,	BURIED_OF_7,	SHADOW_OF_8,	BURIED_OF_8,
SHADOW_OF_9,	BURIED_OF_9,	SHADOW_OF_10,	BURIED_OF_10,
SHADOW_OF_19,	BURIED_OF_19,	SHADOW_OF_18,	BURIED_OF_18,
SHADOW_OF_17,	BURIED_OF_17,	SHADOW_OF_16,	BURIED_OF_16,
SHADOW_OF_15,	BURIED_OF_15,	SHADOW_OF_14,	BURIED_OF_14,
SHADOW_OF_13,	BURIED_OF_13,	SHADOW_OF_12,	BURIED_OF_12,
SHADOW_OF_24,	BURIED_OF_24,	SHADOW_OF_25,	BURIED_OF_25,
SHADOW_OF_26,	BURIED_OF_26,	SHADOW_OF_27,	BURIED_OF_27
SHADOW_OF_28,	BURIED_OF_28,	SHADOW_OF_29,	BURIED_OF_29,
SHADOW_OF_30,	BURIED_OF_30,	SHADOW_OF_31,	BURIED_OF_31,
SHADOW_OF_40,	BURIED_OF_40,	SHADOW_OF_39,	BURIED_OF_39,
SHADOW_OF_38,	BURIED_OF_38,	SHADOW_OF_37,	BURIED_OF_37,
SHADOW_OF_36,	BURIED_OF_36,	SHADOW_OF_35,	BURIED_OF_35,
SHADOW_OF_34,	BURIED_OF_34,	SHADOW_OF_33,	BURIED_OF_33,
SHADOW_OF_45,	BURIED_OF_45,	SHADOW_OF_46,	BURIED_OF_46,
SHADOW_OF_47,	BURIED_OF_47,	SHADOW_OF_48,	BURIED_OF_48,
SHADOW_OF_49,	BURIED_OF_49,	SHADOW_OF_50,	BURIED_OF_50,
SHADOW_OF_51,	BURIED_OF_51,	SHADOW_OF_52,	BURIED_OF_52,
SHADOW_OF_61,	BURIED_OF_61,	SHADOW_OF_60,	BURIED_OF_60,
SHADOW_OF_59,	BURIED_OF_59,	SHADOW_OF_58,	BURIED_OF_58,
SHADOW_OF_57,	BURIED_OF_57,	SHADOW_OF_56,	BURIED_OF_56,
SHADOW_OF_55,	BURIED_OF_55,	SHADOW_OF_54,	BURIED_OF_54,
SHADOW_OF_66,	BURIED_OF_66,	SHADOW_OF_67,	BURIED_OF_67,
SHADOW_OF_68,	BURIED_OF_68,	SHADOW_OF_69,	BURIED_OF_69,
SHADOW_OF_70,	BURIED_OF_70,	SHADOW_OF_71,	BURIED_OF_71,
SHADOW_OF_72,	BURIED_OF_72,	SHADOW_OF_73,	BURIED_OF_73,
SHADOW_OF_82,	BURIED_OF_82,	SHADOW_OF_81,	BURIED_OF_81,
SHADOW_OF_80,	BURIED_OF_80,	SHADOW_OF_79,	BURIED_OF_79,
SHADOW_OF_78,	BURIED_OF_78,	SHADOW_OF_77,	BURIED_OF_77,
SHADOW_OF_76,	BURIED_OF_76,	SHADOW_OF_75,	BURIED_OF_75

MACH435

UNARY_OF_3,	UNARY_OF_4,	UNARY_OF_5,	UNARY_OF_6,
UNARY_OF_7,	UNARY_OF_8,	UNARY_OF_9,	UNARY_OF_10,
UNARY_OF_19,	UNARY_OF_18,	UNARY_OF_17,	UNARY_OF_16,
UNARY_OF_15,	UNARY_OF_14,	UNARY_OF_13,	UNARY_OF_12,
UNARY_OF_24,	UNARY_OF_25,	UNARY_OF_26,	UNARY_OF_27,
UNARY_OF_28,	UNARY_OF_29,	UNARY_OF_30,	UNARY_OF_31,
UNARY_OF_40,	UNARY_OF_39,	UNARY_OF_38,	UNARY_OF_37,
UNARY_OF_36,	UNARY_OF_35,	UNARY_OF_34,	UNARY_OF_33,
UNARY_OF_45,	UNARY_OF_46,	UNARY_OF_47,	UNARY_OF_48,
UNARY_OF_49,	UNARY_OF_50,	UNARY_OF_51,	UNARY_OF_52,
UNARY_OF_61,	UNARY_OF_60	UNARY_OF_59,	UNARY_OF_58,
UNARY_OF_57,	UNARY_OF_56,	UNARY_OF_55,	UNARY_OF_54,
UNARY_OF_66,	UNARY_OF_67,	UNARY_OF_68,	UNARY_OF_69,
UNARY_OF_70,	UNARY_OF_71,	UNARY_OF_72,	UNARY_OF_73,
UNARY_OF_82,	UNARY_OF_81,	UNARY_OF_80,	UNARY_OF_79,
UNARY_OF_78,	UNARY_OF_77,	UNARY_OF_76,	UNARY_OF_75,

MACH435 (con't)

MACROCELL_A00,	MACROCELL_A01,	MACROCELL_A02,	MACROCELL_A03,
MACROCELL_A04,	MACROCELL_A05,	MACROCELL_A06,	MACROCELL_A07,
MACROCELL_A08,	MACROCELL_A09,	MACROCELL_A10,	MACROCELL_A11,
MACROCELL_A12,	MACROCELL_A13,	MACROCELL_A14,	MACROCELL_A15,
MACROCELL_B00,	MACROCELL_B01,	MACROCELL_B02,	MACROCELL_B03,
MACROCELL_B04,	MACROCELL_B05,	MACROCELL_B06,	MACROCELL_B07,
MACROCELL_B08,	MACROCELL_B09,	MACROCELL_B10,	MACROCELL_B11,
MACROCELL_B12,	MACROCELL_B13,	MACROCELL_B14,	MACROCELL_B15,
MACROCELL_C00,	MACROCELL_C01,	MACROCELL_C02,	MACROCELL_C03,
MACROCELL_C04,	MACROCELL_C05,	MACROCELL_C06,	MACROCELL_C07,
MACROCELL_C08,	MACROCELL_C09,	MACROCELL_C10,	MACROCELL_C11,
MACROCELL_C12,	MACROCELL_C13,	MACROCELL_C14,	MACROCELL_C15,
MACROCELL_D00,	MACROCELL_D01,	MACROCELL_D02,	MACROCELL_D03,
MACROCELL_D04,	MACROCELL_D05,	MACROCELL_D06,	MACROCELL_D07,
MACROCELL_D08,	MACROCELL_D09,	MACROCELL_D10,	MACROCELL_D11,
MACROCELL_D12,	MACROCELL_D13,	MACROCELL_D14,	MACROCELL_D15,
MACROCELL_E00,	MACROCELL_E01,	MACROCELL_E02,	MACROCELL_E03,
MACROCELL_E04,	MACROCELL_E05,	MACROCELL_E06,	MACROCELL_E07,
MACROCELL_E08,	MACROCELL_E09,	MACROCELL_E10,	MACROCELL_E11,
MACROCELL_E12,	MACROCELL_E13,	MACROCELL_E14,	MACROCELL_E15,
MACROCELL_F00,	MACROCELL_F01,	MACROCELL_F02,	MACROCELL_F03,
MACROCELL_F04,	MACROCELL_F05,	MACROCELL_F06,	MACROCELL_F07,
MACROCELL_F08,	MACROCELL_F09,	MACROCELL_F10,	MACROCELL_F11,
MACROCELL_F12,	MACROCELL_F13,	MACROCELL_F14,	MACROCELL_F15,
MACROCELL_G00,	MACROCELL_G01,	MACROCELL_G02,	MACROCELL_G03,
MACROCELL_G04,	MACROCELL_G05,	MACROCELL_G06,	MACROCELL_G07,
MACROCELL_G08,	MACROCELL_G09,	MACROCELL_G10,	MACROCELL_G11,
MACROCELL_G12,	MACROCELL_G13,	MACROCELL_G14,	MACROCELL_G15,
MACROCELL_H00,	MACROCELL_H01,	MACROCELL_H02	MACROCELL_H03,
MACROCELL_H04,	MACROCELL_H05,	MACROCELL_H06,	MACROCELL_H07,
MACROCELL_H08,	MACROCELL_H09,	MACROCELL_H10,	MACROCELL_H11,
MACROCELL_H12,	MACROCELL_H13,	MACROCELL_H14,	MACROCELL_H15,

MACH465

UNARY_OF_190,	UNARY_OF_191,	UNARY_OF_192,	UNARY_OF_193,
UNARY_OF_194,	UNARY_OF_195,	UNARY_OF_196,	UNARY_OF_197,
UNARY_OF_200,	UNARY_OF_201,	UNARY_OF_202,	UNARY_OF_203,
UNARY_OF_204,	UNARY_OF_205,	UNARY_OF_206,	UNARY_OF_207,
UNARY_OF_20,	UNARY_OF_19,	UNARY_OF_18,	UNARY_OF_17,
UNARY_OF_16,	UNARY_OF_15,	UNARY_OF_14,	UNARY_OF_13,
UNARY_OF_10,	UNARY_OF_9,	UNARY_OF_8,	UNARY_OF_7,
UNARY_OF_6,	UNARY_OF_5,	UNARY_OF_4,	UNARY_OF_3,
UNARY_OF_32,	UNARY_OF_33,	UNARY_OF_34,	UNARY_OF_35,
UNARY_OF_36,	UNARY_OF_37,	UNARY_OF_38,	UNARY_OF_39,
UNARY_OF_42,	UNARY_OF_43,	UNARY_OF_44,	UNARY_OF_45,
UNARY_OF_46,	UNARY_OF_47,	UNARY_OF_48,	UNARY_OF_49,
UNARY_OF_71,	UNARY_OF_70,	UNARY_OF_69,	UNARY_OF_68,
UNARY_OF_67,	UNARY_OF_66,	UNARY_OF_65,	UNARY_OF_64,
UNARY_OF_61,	UNARY_OF_60,	UNARY_OF_59,	UNARY_OF_58,
UNARY_OF_57,	UNARY_OF_56,	UNARY_OF_55,	UNARY_OF_54,

MACH465 (con't)

UNARY_OF_86,	UNARY_OF_87,	UNARY_OF_88,	UNARY_OF_89,
UNARY_OF_90,	UNARY_OF_91,	UNARY_OF_92,	UNARY_OF_93,
UNARY_OF_96,	UNARY_OF_97,	UNARY_OF_98,	UNARY_OF_99,
UNARY_OF_100,	UNARY_OF_101,	UNARY_OF_102,	UNARY_OF_103,
UNARY_OF_124,	UNARY_OF_123,	UNARY_OF_122,	UNARY_OF_121,
UNARY_OF_120,	UNARY_OF_119,	UNARY_OF_118,	UNARY_OF_117,
UNARY_OF_114,	UNARY_OF_113,	UNARY_OF_112,	UNARY_OF_111,
UNARY_OF_110,	UNARY_OF_109,	UNARY_OF_108,	UNARY_OF_107,
UNARY_OF_136,	UNARY_OF_137,	UNARY_OF_138,	UNARY_OF_139,
UNARY_OF_140,	UNARY_OF_141,	UNARY_OF_142,	UNARY_OF_143,
UNARY_OF_146,	UNARY_OF_147,	UNARY_OF_148,	UNARY_OF_149,
UNARY_OF_150,	UNARY_OF_151,	UNARY_OF_152,	UNARY_OF_153,
UNARY_OF_175,	UNARY_OF_174,	UNARY_OF_173,	UNARY_OF_172,
UNARY_OF_171,	UNARY_OF_170,	UNARY_OF_169,	UNARY_OF_168,
UNARY_OF_165,	UNARY_OF_164,	UNARY_OF_163,	UNARY_OF_162,
UNARY_OF_161,	UNARY_OF_160,	UNARY_OF_159,	UNARY_OF_158,
MACROCELL_A00,	MACROCELL_A01,	MACROCELL_A02,	MACROCELL_A03,
MACROCELL_A04,	MACROCELL_A05,	MACROCELL_A06,	MACROCELL_A07,
MACROCELL_A08,	MACROCELL_A09,	MACROCELL_A10,	MACROCELL_A11,
MACROCELL_A12,	MACROCELL_A13,	MACROCELL_A14,	MACROCELL_A15,
MACROCELL_B00,	MACROCELL_B01,	MACROCELL_B02,	MACROCELL_B03,
MACROCELL_B04,	MACROCELL_B05,	MACROCELL_B06,	MACROCELL_B07,
MACROCELL_B08,	MACROCELL_B09,	MACROCELL_B10,	MACROCELL_B11,
MACROCELL_B12,	MACROCELL_B13,	MACROCELL_B14,	MACROCELL_B15,
MACROCELL_C15,	MACROCELL_C14,	MACROCELL_C13,	MACROCELL_C12,
MACROCELL_C11,	MACROCELL_C10,	MACROCELL_C09,	MACROCELL_C08,
MACROCELL_C07,	MACROCELL_C06,	MACROCELL_C05,	MACROCELL_C04,
MACROCELL_C03,	MACROCELL_C02,	MACROCELL_C01,	MACROCELL_C00,
MACROCELL_D15,	MACROCELL_D14,	MACROCELL_D13,	MACROCELL_D12,
MACROCELL_D11,	MACROCELL_D10,	MACROCELL_D09,	MACROCELL_D08,
MACROCELL_D07,	MACROCELL_D06,	MACROCELL_D05,	MACROCELL_D04,
MACROCELL_D03,	MACROCELL_D02,	MACROCELL_D01,	MACROCELL_D00,
MACROCELL_E00,	MACROCELL_E01,	MACROCELL_E02,	MACROCELL_E03,
MACROCELL_E04,	MACROCELL_E05,	MACROCELL_E06,	MACROCELL_E07,
MACROCELL_E08,	MACROCELL_E09,	MACROCELL_E10,	MACROCELL_E11,
MACROCELL_E12,	MACROCELL_E13,	MACROCELL_E14,	MACROCELL_E15,
MACROCELL_F00,	MACROCELL_F01,	MACROCELL_F02,	MACROCELL_F03,
MACROCELL_F04,	MACROCELL_F05,	MACROCELL_F06,	MACROCELL_F07,
MACROCELL_F08,	MACROCELL_F09,	MACROCELL_F10,	MACROCELL_F11,
MACROCELL_F12,	MACROCELL_F13,	MACROCELL_F14,	MACROCELL_F15,
MACROCELL_G15,	MACROCELL_G14,	MACROCELL_G13,	MACROCELL_G12,
MACROCELL_G11,	MACROCELL_G10,	MACROCELL_G09,	MACROCELL_G08,
MACROCELL_G07,	MACROCELL_G06,	MACROCELL_G05,	MACROCELL_G04,
MACROCELL_G03,	MACROCELL_G02,	MACROCELL_G01,	MACROCELL_G00,
MACROCELL_H15,	MACROCELL_H14,	MACROCELL_H13,	MACROCELL_H12,
MACROCELL_H11,	MACROCELL_H10,	MACROCELL_H09,	MACROCELL_H08,
MACROCELL_H07,	MACROCELL_H06,	MACROCELL_H05,	MACROCELL_H04,
MACROCELL_H03,	MACROCELL_H02,	MACROCELL_H01,	MACROCELL_H00,
MACROCELL_I00,	MACROCELL_I01,	MACROCELL_I02,	MACROCELL_I03,
MACROCELL_I04,	MACROCELL_I05,	MACROCELL_I06,	MACROCELL_I07,

MACH465 (con't)

MACROCELL_I08, MACROCELL_I09, MACROCELL_I10, MACROCELL_I11,
MACROCELL_I12, MACROCELL_I13, MACROCELL_I14, MACROCELL_I15,
MACROCELL_J00, MACROCELL_J01, MACROCELL_J02, MACROCELL_J03,
MACROCELL_J04, MACROCELL_J05, MACROCELL_J06, MACROCELL_J07,
MACROCELL_J08, MACROCELL_J09, MACROCELL_J10, MACROCELL_J11,
MACROCELL_J12, MACROCELL_J13, MACROCELL_J14, MACROCELL_J15,
MACROCELL_K15, MACROCELL_K14, MACROCELL_K13, MACROCELL_K12,
MACROCELL_K11, MACROCELL_K10, MACROCELL_K09, MACROCELL_K08,
MACROCELL_K07, MACROCELL_K06, MACROCELL_K05, MACROCELL_K04,
MACROCELL_K03, MACROCELL_K02, MACROCELL_K01, MACROCELL_K00,
MACROCELL_L15, MACROCELL_L14, MACROCELL_L13, MACROCELL_L12,
MACROCELL_L11, MACROCELL_L10, MACROCELL_L09, MACROCELL_L08,
MACROCELL_L07, MACROCELL_L06, MACROCELL_L05, MACROCELL_L04,
MACROCELL_L03, MACROCELL_L02, MACROCELL_L01, MACROCELL_L00,
MACROCELL_M00, MACROCELL_M01, MACROCELL_M02, MACROCELL_M03,
MACROCELL_M04, MACROCELL_M05, MACROCELL_M06, MACROCELL_M07,
MACROCELL_M08, MACROCELL_M09, MACROCELL_M10, MACROCELL_M11,
MACROCELL_M12, MACROCELL_M13, MACROCELL_M14, MACROCELL_M15,
MACROCELL_N00, MACROCELL_N01, MACROCELL_N02, MACROCELL_N03,
MACROCELL_N04, MACROCELL_N05, MACROCELL_N06, MACROCELL_N07,
MACROCELL_N08, MACROCELL_N09, MACROCELL_N10, MACROCELL_N11,
MACROCELL_N12, MACROCELL_N13, MACROCELL_N14, MACROCELL_N15,
MACROCELL_O15, MACROCELL_O14, MACROCELL_O13, MACROCELL_O12,
MACROCELL_O11, MACROCELL_O10, MACROCELL_O09, MACROCELL_O08,
MACROCELL_O07, MACROCELL_O06, MACROCELL_O05, MACROCELL_O04,
MACROCELL_O03, MACROCELL_O02, MACROCELL_O01, MACROCELL_O00,
MACROCELL_P15, MACROCELL_P14, MACROCELL_P13, MACROCELL_P12,
MACROCELL_P11, MACROCELL_P10, MACROCELL_P09, MACROCELL_P08,
MACROCELL_P07, MACROCELL_P06, MACROCELL_P05, MACROCELL_P04,
MACROCELL_P03, MACROCELL_P02, MACROCELL_P01, MACROCELL_P00

Application Note:

Fuse Commands for Forcing Outputs to be Driven

Devices: MACH 1xx/2xx

The following table gives the fuse commands for the *.pi* file to force the named pin to be driven.

MACH110 OE ASSERTION

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 ;

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 ;

MACH120 OE ASSERTION

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 ;
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 ;

MACH120 OE ASSERTION (con't)

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 ;

MACH130 OE ASSERTION

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 ;
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 ;

MACH130 OE ASSERTION (con't)

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 ;
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 ;

MACH130 OE ASSERTION (con't)

Pin 82:	INTACT 15414 ;	BLOWN 15415 ;

MACH210 OE ASSERTION

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 ;
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 ;

MACH215 OE ASSERTION

Pin 02:	BLOWN 88 .. 131 ;
Pin 03:	BLOWN 440 .. 483 ;
Pin 04:	BLOWN 792 .. 835 ;

MACH215 OE ASSERTION (con't)

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 ;
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 ;

MACH220 OE ASSERTION

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 ;

MACH220 OE ASSERTION (con't)

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 ;
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 ;
Pin 63:	INTACT 23206 ;	BLOWN 23207 ;
Pin 64:	INTACT 23190 ;	BLOWN 23191 ;
Pin 65:	INTACT 23174 ;	BLOWN 23175 ;
Pin 66:	INTACT 23158 ;	BLOWN 23159 ;
Pin 67:	INTACT 23142 ;	BLOWN 23143 ;

MACH230 OE ASSERTION

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 ;
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 ;

MACH230 OE ASSERTION (con't)

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

'

*

.

[

A

B

C

D

E

F

G

K

L

M

N

Q

R

S

T

U

V

W

X

Z

Sales Offices

North American

ALABAMA		(205) 830-9192
ARIZONA		(602) 242-4400
CALIFORNIA,		
Calabasas		(818) 878-9988
Irvine		(714) 450-7500
Sacramento (Roseville)		(916) 786-6700
San Diego		(619) 560-7030
San Jose		(408) 922-0300
CANADA, Ontario,		
Kanata		(613) 592-0060
Willowdale		(416) 222-7800
COLORADO		(303) 741-2900
CONNECTICUT		(203) 264-7800
FLORIDA,		
Boca Raton		(407) 361-0050
Clearwater		(813) 530-9971
Orlando (Longwood)		(407) 862-9292
GEORGIA		(404) 449-7920
IDAHO		(208) 377-0393
ILLINOIS, Chicago (Itasca)		(708) 773-4422
KENTUCKY		(606) 224-1353
MARYLAND		(410) 381-3790
MASSACHUSETTS		(617) 273-3970
MINNESOTA		(612) 938-0001
NEW JERSEY,		
Cherry Hill		(609) 662-2900
Parsippany		(201) 299-0002
NEW YORK,		
Brewster		(914) 279-8323
Rochester		(716) 425-8050
NORTH CAROLINA,		
Charlotte		(704) 875-3091
Raleigh		(919) 878-8111
OHIO,		
Columbus (Westerville)		(614) 891-6455
Dayton		(513) 439-0268
OREGON		(503) 245-0080
PENNSYLVANIA		(610) 398-8006
TEXAS,		
Austin		(512) 346-7830
Dallas		(214) 934-9099
Houston		(713) 376-8084

International

BELGIUM, Antwerpen	TEL	(03) 248-4300
	FAX	(03) 248-4642
CHINA,		
Beijing	TEL	(861) 465-1251
	FAX	(861) 465-1291
Shanghai	TEL	(8621) 267-8857
	TEL	(8621) 267-9883
	FAX	(8621) 267-8110
FINLAND, Helesinki	TEL	(358) 0 881 3117
	FAX	(358) 0 804 1110
FRANCE, Paris	TEL	(1) 49-75-1010
	FAX	(1) 49-75-1013
GERMANY,		
Bad Homburg	TEL	(06172)-92670
	FAX	(06172)-23195
München	TEL	(089) 450530
	FAX	(089) 406490
HONG KONG, Kowloon	TEL	(852) 2956-0388
	FAX	(852) 2956-0588
ITALY, Milano	TEL	(02) 339-0541
	FAX	(02) 3810-3458
JAPAN,		
Osaka	TEL	(06) 243-3250
	FAX	(06) 243-3253
Tokyo	TEL	(03) 3346-7600
	FAX	(03) 3346-5197
KOREA, Seoul	TEL	(82) 2784-0030
	FAX	(82) 2784-8014
SINGAPORE, Singapore	TEL	(65) 348-1188
	FAX	(65) 348-0161
SWEDEN,		
Stockholm area	TEL	(08) 98-6180
(Bromma)	FAX	(08) 98-0906
TAIWAN, Taipei	TEL	(886) 2715-3536
	FAX	(886) 2712-2182
UNITED KINGDOM,		
London area	TEL	(01483) 74-0440
(Woking)	FAX	(01483) 75-6196
Manchester area	TEL	(01925) 83-0380
(Warrington)	FAX	(01925) 83-0204

North American Representatives

ARIZONA	
THORSON DESERT STATES, INC.	(602) 998-2444
CALIFORNIA	
Chula Vista – SONIKA ELECTRONICA	(619) 498-8340
CANADA,	
Burnaby, B.C. – DAVETEK MARKETING	(604) 430-3680
Kanata, Ontario – VITEL ELECTRONICS	(613) 592-0090
Mississauga, Ontario – VITEL ELECTRONICS	(905) 564-9720
Lachine, Quebec – VITEL ELECTRONICS	(514) 636-5951
ILLINOIS	
Skokie – INDUSTRIAL REPS, INC.	(708) 967-8430
INDIANA	
Kokomo – SCHILLINGER ASSOC.	(317) 457-7241
IOWA	
LORENZ SALES	(319) 377-4666
KANSAS,	
Merriam – LORENZ SALES	(913) 469-1312
Wichita – LORENZ SALES	(316) 721-0500
MEXICO,	
Guadalajara – SONIKA ELECTRONICA	(523) 647-4250
Mexico City – SONIKA ELECTRONICA	(525) 754-6480
Monterrey – SONIKA ELECTRONICA	(528) 358-9280
MICHIGAN,	
Brighton – COM-TEK SALES, INC	(810) 227-0007
Holland – COM-TEK SALES, INC	(616) 335-8418
MINNESOTA	
MEL FOSTER TECH. SALES, INC	(612) 941-9790
MISSOURI	
LORENZ SALES	(314) 997-4558
NEBRASKA	
LORENZ SALES	(402) 475-4660
NEW YORK,	
Hauppauge – COMPONENT CONSULTANTS, INC	(516) 273-5050
East Syracuse – NYCOM	(315) 437-8343
Fairport – NYCOM	(716) 425-5120
OHIO,	
Centerville – DOLFUSS ROOT & CO	(513) 433-6776
Westlake – DOLFUSS ROOT & CO	(216) 899-9370
PUERTO RICO	
COMP REP ASSOC, INC	(809) 746-6550
UTAH	
FRONT RANGE MARKETING	(801) 288-2500
WASHINGTON	
ELECTRA TECHNICAL SALES	(206) 821-7442
WISCONSIN	
Brookfield – Industrial Representatives, Inc.	(414) 574-9393

Advanced Micro Devices, Inc. One AMD Place, P.O. Box 3453, Sunnyvale, CA 94088-3453, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450

APPLICATIONS HOTLINE & LITERATURE ORDERING • TOLL FREE: (800) 222-9323 • (408) 749-5703
• UK & Europe 44-0-256-811101 • France 0590-8621 • Germany 0130-813875 • Italy 1678-77224

BAN-8M-7/95-0 6/01/95
17703D Printed in USA

ADVANCED MICRO DEVICES, INC.
One AMD Place
P.O. Box 3453
Sunnyvale,
California 94088-3453
(408) 732-2400
(800) 538-8450
TWX: 910-339-9280
TELEX: 34-6306

APPLICATIONS HOTLINE &
LITERATURE ORDERING
USA (408) 749-5703
JAPAN (03)-3346-7600
UK & EUROPE 44-(0)256-811101
TOLL FREE
USA (800) 222-9323
FRANCE 0590-8621
GERMANY 0130-813875
ITALY 1678-77224

http://www.amd.com

RECYCLED &
RECYCLABLE

Printed in USA
BAN-8M-7/95-0
17703D