CUPL USERS Guide

Copyright

Logical Devices, Inc.
1221 S Clarkson St. Suite 200
Denver, CO 80210
Technical Support Telephone: (303) 722-6868

Website: www.logicaldevices.com

Table Of Contents

1. User Guide Overview

This manual is designed to serve as a learning aid and as a reference manual for CUPL, the programmable logic compiler from Logical Devices, Inc. It is divided into five sections. The Reference section, the Language Reference section, the Simulator Reference section, the Design example section, and the Appendices. The Reference section provides specifc information about the programs that make up the CUPL package. The Appendices contain a variety of information including error messages and contacting Logical Devices.

2. Conventions Used In This Manual

This manual gives step-by-step procedures and examples. To make it easy to follow these procedures, the following conventions are used.

LDI software is not case sensitive. It doesn't matter whether upper or lower case characters are typed.

Return — Return is the key that must be pressed to execute a command or accept an option. This key is called different names on different systems. For example:

Enter, Enter, ⏎, ENTER, RETURN

⬭—⬭ — Connected keys indicate the keys must be pressed simultaneously. For example:

Press Ctrl —Alt —Del

* — An asterisk in a filename indicates any characters can occupy that position and all remaining positions.

Boldface — Boldface is used for two purposes. First, it is used to highlight menu or file names within text, and, second, it indicates characters that must be typed from the keyboard.These characters are usually designated as “Enter the following:” or are set aside by line spacing. For example:

del pcprint.cfg

Italics — Italics represent variable names. For example:

***filename*.SCH**

< > Variable names are indicated by angle brackets. For example:

<filename>.SCH

[ ] Square brackets indicate the enclosed item is optional. For example:

prepack filename.fil [filename.lib]

When shown on the screen, square brackets indicate the name of a key. For example:

Press [Return] to accept

3. Wincupl User Interface

This chapter will show the features available in WinCUPL. These include graphical waveform simulation, highlighted text editor, bubble entry to CUPL source, schematic to CUPL source, macro insertion, and table wizard. For details on features and usage of each package, reference that modules users manual.

Overview

The WinCUPL package is comprised of four different modules.

WinCUPL A powerful front end and user interface for all of the WinCUPL tools including the compiler.

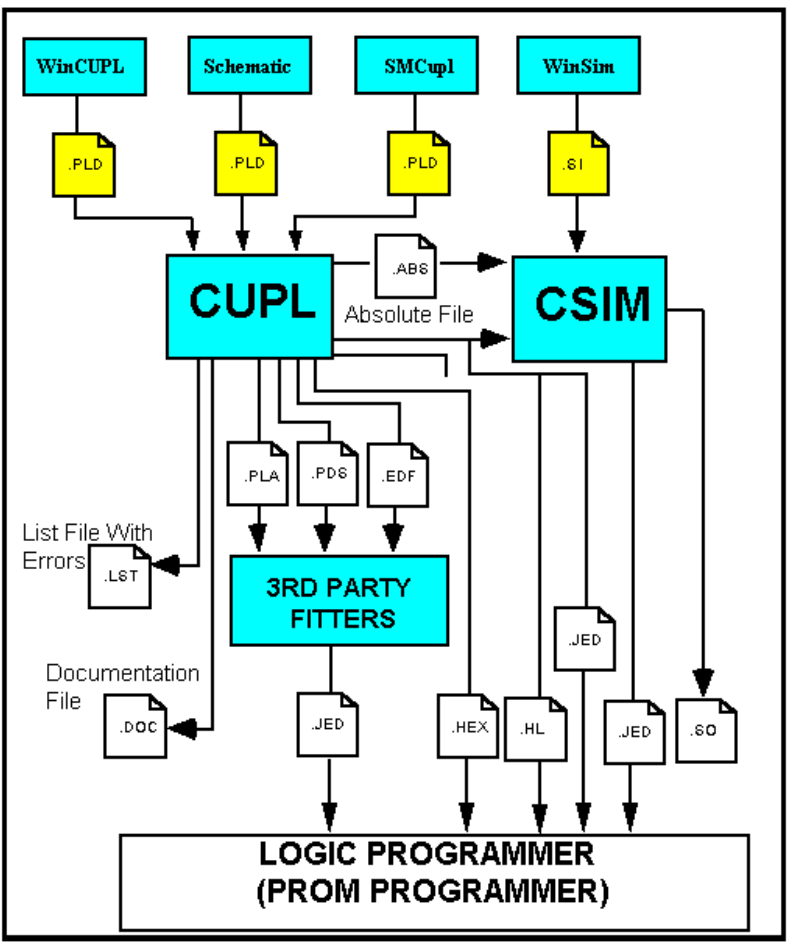
WinSim Designs can be graphically simulated with WinSim to test the design with user defined inputs to verify the design. Both the simulation inputs and the results of the simulation can be graphically viewed and modified with WinSim.

Schematic Schematic is a tool used for creating schematic diagrams for initial design analysis. Once the diagram has been created, the diagram is validated to determine if all of the components are connected by wires to other components or to grounds, ports or power ports. In addition, several tests are performed to insure that inputs and outputs are not tied together, and that all components have been named. If the diagram passes the validation process, CUPL source code describing the behavior of the drawing is generated which can be can be compiled and the design simulated using the CUPL Compiler and simulators.

SMCupl SMCupl is a tool used for creating State Diagrams for initial design analysis. Once the state diagram has been created, the diagram is validated to determine if all of the states and transitions are meaningful and can be reached. The validation process also check the usage of all variables within the diagram. If the state diagram passes the validation process, a CUPL language source file can be generated that can be compiled and the design simulated using the CUPL Compiler and simulators

CUPL Data Flow

The following diagram illustrates the data flow for creating a design and implementing the design using CUPL.

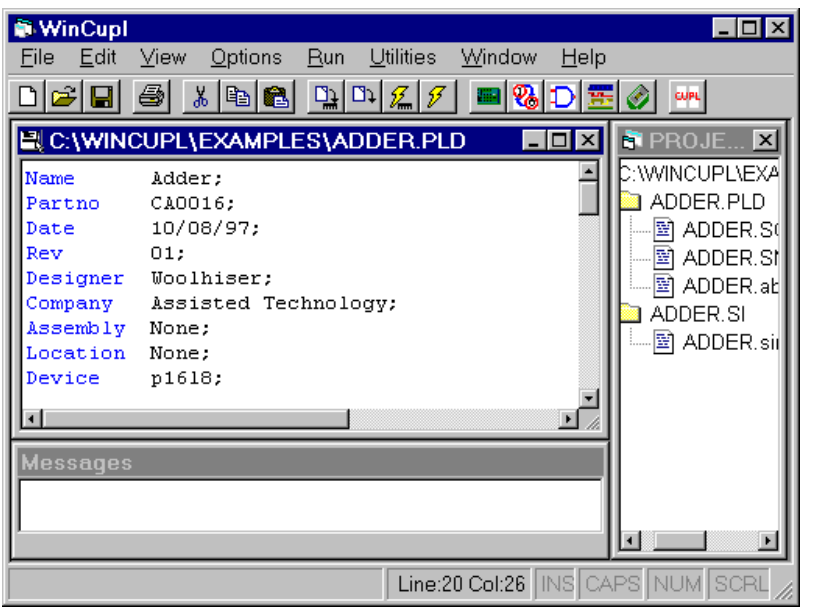


CUPL Data Flow

First, a logic description is created using the CUPL language which may be generated from Schematic, SMCupl or manually created using the WinCUPL source editor. Then, the design is compiled to create a fusemap file for downloading to a device programmer. Optionally, a test specification file may be created to verify the design. CSIM is executed to compare the expected values in the test file to the actual values in the absolute file created by CUPL. When simulation is complete without any errors, the verified test vectors can be appended to the download file generated by CUPL.

WinCUPL

WinCUPL provides a powerful integrated development environment (IDE) for developing designs using the CUPL compiler and tools. Key features of the IDE include:



WinCUPL Editor

Syntax highlighting in the WinCUPL editor that can be completely customized for any language using color and other text attributes.

User customizable tool bar and menus for seamless integration of all WinCUPL tools and any other tools and programs you desire.

Easy navigation of all WinCUPL tools using the toolbar

Truth table editor that allows you to enter binary truth tables into your CUPL source code graphically.

Automated tools for generating and managing CUPL macro's and for referencing CUPL macro's within source files.

Complete 32 bit version of the CUPL compiler and supporting tools.

Powerful file import features_import capabilities.

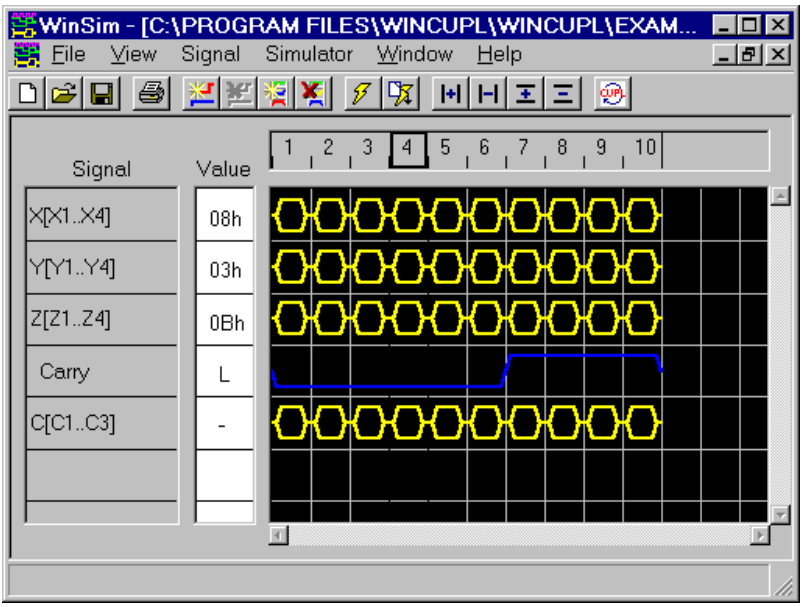
Integrated Backpin utility

Integrated PlPartition utility

Integrated ISP utility

WinSim

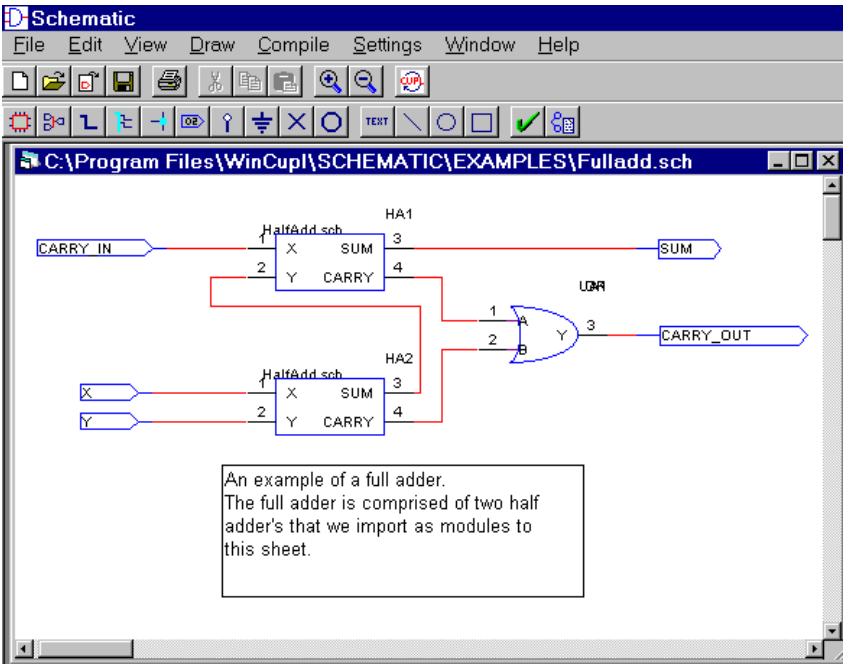
WinSim is a graphical based tool for creating and editing simulator (CSIM) input files and for displaying the results of the simulator. Signal names from the design file are loaded with the simulation file to remove errors in signal names. Vectors can be set up so that the simulator determines the outputs and WinSim will assign the results to the simulation source file automatically. Vector types can be assigned colors to easily distinguish between input, output, bus and clock signals.



WinSim User Interface

Schematic/PCB

Schematic/PCB is a tool used for creating schematic diagrams for initial design analysis. The PCB feature of Schematic/PCB allows you to generate a Printed Circuit Board design for the schematic. The PCB features may be used in conjunction with the schematic drawing or independently.



Schematic User Interface

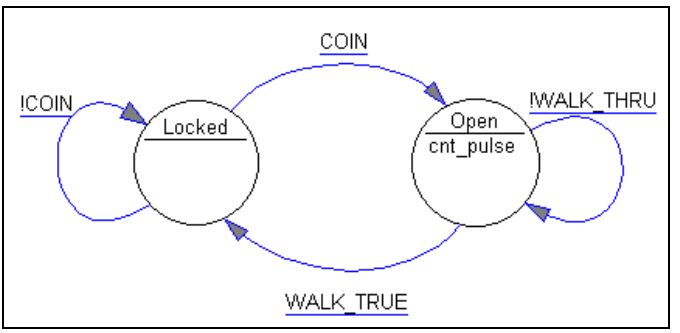
Once the a schematic diagram has been created, the diagram is validated to determine if all of the components are connected by wires to other components or to grounds, ports or power ports. In addition, several tests are performed to insure that inputs and outputs are not tied together, and that all components have been named.

If the diagram passes the validation process, an HDL description of the drawing is generated.

After validation of the schematic design, the schematic can be loaded into the PCB editor and a circuit board can be laid out using the PCB editor.

SMCupl

SMCupl is a tool used for creating State Diagrams for initial design analysis. The following drawing is an example of a subway turnstile state diagram



SMCUPL Turnstile Example

Once the state diagram has been created, the diagram is validated to determine if all of the states and transitions are meaningful and can be reached. The validation process also check the usage of all variables within the diagram.

If the state diagram passes the validation process, a PLD language source file can be generated that can be compiled and the design simulated using the CUPL PLD Language Compiler and simulators.

4. System Overview

Key Features

The key features of the CUPL package include:

- **Universal applicability**. CUPL supports products from all manufacturers of PLDs, enabling a user to put the same functional logic into physically different parts, to create a second source at the socket. CUPL produces a standard type of file called JEDEC. This is a download file that is compatible with any logic programmer that uses JEDEC files.

- **A high-level language**. Expression substitution for equations, shorthand notation for lists, address ranges, and bit fields are available to save design time.

  CUPL simplifies Boolean expressions by the distributive property and DeMorgan's Theorem.

  State machine syntax provides a powerful means of implementing any synchronous application using either Mealy or Moore state machine models.

  Truth table syntax provides a way to clearly express certain logic descriptions.

  User-defined functions allow the creation of keywords for use by CUPL.

- **Flexible documentation**. CUPL provides a template file for standard "fill in the blanks" documentation and allows the placement of free-form comments throughout a design.

  CUPL's comprehensive error-checking capability generates detailed error messages designed to lead to the source of any problems.

- **Powerful minimizer and simulation programs**. CUPL contains the fastest and most powerful minimizer offered for programmable logic equation reduction, featuring four levels of minimization.

  The CUPL simulation program enables logic to be simulated prior to using a PLD. This feature prevents blown devices and helps debug system-level problems. Test vectors verified by **CSIM** can be downloaded to a logic programmer.

CUPL (Universal Compiler for Programmable Logic) is a set of programs that provides tools for designing with PLDs. The CUPL system consists of the following modules: CUPLX, CUPLA, CUPLB, CUPLM, and CUPLC. A brief description of each of the program modules follows.

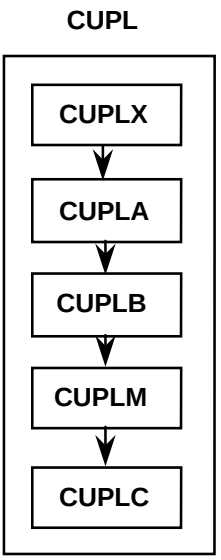


Figure 3-1. CUPL Execution Flow

CUPLX: CUPL Language Preprocessor

Scans the **.PLD** (input) file, processing the preprocessor directives, i.e. $DEFINE. Generates an intermediate file with all preprocessor directives expanded.

CUPLA: CUPL Language Parser

Scans and parses the intermediate file generated by CUPLX. Utilizes a table driven parser to generate a symbol table and expanded equations. Expands state machine, truth table, and user defined function syntax into boolean equations. Also, performs simple logic reduction while processing range statements.

CUPLB: CUPL Design to Target Device Linker

Resolves links between design and the target device model. Expands the parsed equations according to signal polarity and physical characteristics, via DeMorgan's Theorem. Builds the final symbol table, containing device model links, and bit mapped representation of the logic design.

CUPLM: CUPL Logic Minimizer

Executes logic minimization algorithms on the bit mapped logic generated in CUPLB. Processes only the equations for which reduction has been requested. If no reduction is requested then the intermediate file generated by CUPLB is renamed for use by CUPLC. Performs multiple output minimization for FPLA architecture.

CUPLC: CUPL Design Fitter

Determines if the design fits the target device architecture and builds a fuse map. The fuse map and symbol table are used to generate the documentation and JEDEC files.

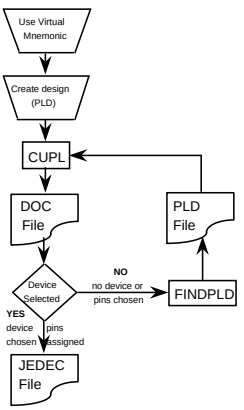


Figure 3-2. Device Independent Design Flow

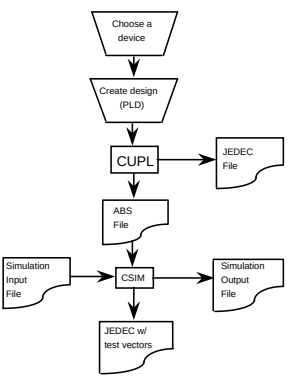


Figure 3-3. Device Specific Design Flow

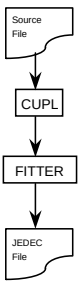


Figure 3-4. CUPL Device Fitting

Minimization Techniques

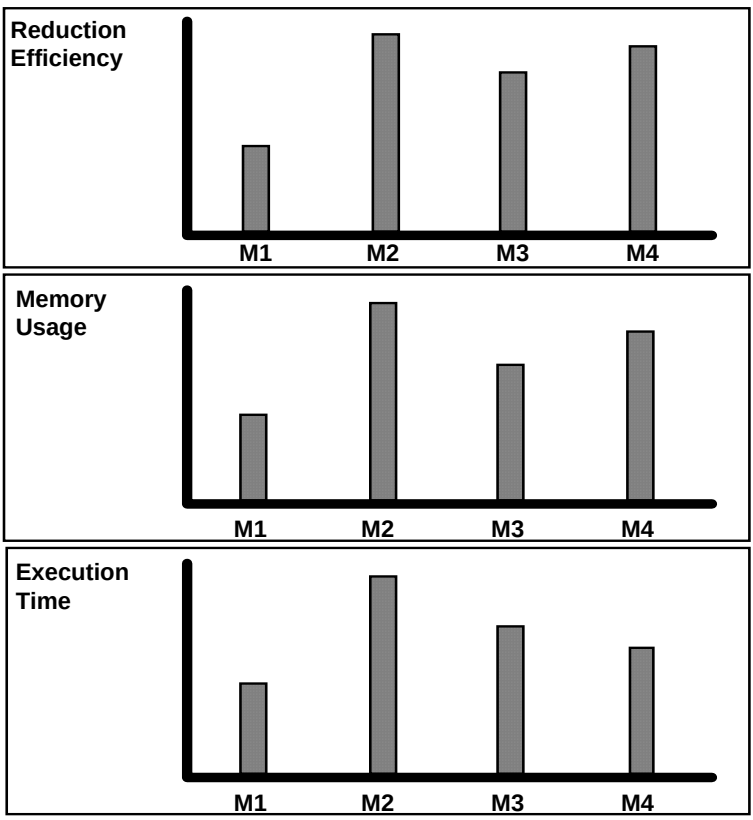


Figure 3-4. Logic Minimization Levels

Flag	Minimization Description
M1	Quick Minimization
M2	Quine-McCluskey Minimization
M3	Presto Minimization
M4	Expresso Minimization

CUPL Command Line

Run **CUPL** using the following command line format:

cupl [-flags] [library] [device] source

where

-**flags** is the following set of compiler options:

-j	JEDEC download format
-h	ASCII-HEX download format
-i	HL download format
-n	use input filename for output file
-a	create absolute file
-l	create listing file
-e	create expanded macro definition file
-x	create expanded product-terms in documentation file
-f	create fuse plot/chip diagram in documentation file
-p	create PDIF database interchange format file
-b	create Berkeley PLA format file
-c	create PALASM format file
-d	deactivate unused OR terms
-r	disable product term merging
-g	program security fuse
-o	treat all state machines as “one-hot”
-u	use specified library for compilation
-s	perform logic simulation after compilation
-w	perform simulation with waveform output (MS-DOS only)
-m0	no minimization
-m1	quick minimization (default)
-m2	Quine McCluskey
-m3	Presto
-m4	Expresso
-q	MIcrosoft format for error messages
-zq	QuickLogic’s QDIF file
-kb	Optimize product term usage for pin or pinnode variables. This overrides the DEMORGAN statement if it appears in the source file
-kd	DeMorganize all pin and pinnode variables. This overrides the DEMORGAN statement if it appears in the source file

-ks	Force product term sharing during minimization. This is also referred to as group reduction
-kx	Do not expand XOR to AND-OR equations. This is used for device independent designs or designs targeted for fitter-supported devices where the fitter supports XOR gates

Table 4-1. Compiler Option Flags

library is the path name and library name used with the **-u** flag to specify a library other than the default library.

device is the device mnemonic for the type of part to be used in the compilation. Use the **CBLD** program to list available devices (see Chapter 3, "Using CBLD").

source is the user-created ASCII logic description file (***filename*.PLD**). The **.PLD** extension is assumed for the source file and may be omitted when giving the **CUPL** command.

The square brackets indicate optional items.

CUPL Option Flags

Multiple option flags can be specified when running **CUPL**. A hyphen must be typed before the first flag entered, but is optional for additional flags. Spaces also can be put between the option flags. For example, the following two **CUPL** command lines are equivalent:

cupl -a -l -j p16r4 waitgen Return

cupl -alj p16r4 waitgen Return

Type **CUPL** without any flags to see the command line format and a list of the option flags. Table 4-2 lists descriptions of the **CUPL** option flags and output files. An introductory example will be presented in the next chapter**.**

Table 4-2. Compiler Option Flags

Option Flag	Description
j	Generates a JEDEC-compatible ASCII download file (**filename.JED**). The filename is not necessarily the same as the logic description filename input to **CUPL**. The **NAME** statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
h	Generates an ASCII-hex download file (**filename.HEX**). This format is available only for PROMs. The filename is not necessarily the same as the logic description filename input to **CUPL**. The **NAME** statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
i	Generates an HL download file (**filename.HL**). This format is available only for the Signetics IFL devices. The filename is not necessarily the same as the logic description filename input to **CUPL**. The **NAME** statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
a	Generates an absolute file (**filename.ABS**) for use by the **CSIM** logic simulation program.
n	Allows the source filename to be used as the JEDEC filename instead of using the name in the **NAME** field of the source file.
l	Generates an error listing file (**filename.LST**). Each line in the original source file is numbered. Error messages are listed at the end of the file and use the line numbers for reference.
x	Generates a documentation file (**filename.DOC**) which contains an expanded listing of the logic terms in sum-of-products format and a symbol table of all variables used in the source file. It includes the total number of product terms and the number available for each output.

f	Generates a fuse plot in the documentation file. For PAL devices, each output pin is listed and the associated product term rows are shown with the starting JEDEC fuse number. Fuses present are denoted with "**x**". Fuses blown are denoted with "**-**". For IFL devices, the HL download format is used, showing JEDEC fuse numbers with input terms denoted as "**H**," "**L**," "**0**," or "**-**".
b	Generates a Berkeley PLA file (**filename.PLA**) for use by the Berkeley PLA tools, such as PLEASURE, or other PLA layout tools which use the Berkeley PLA format. The compiler
d	In IFL devices, the OR-gate output array is driven by each of the AND-gate product terms. Normally, unused OR-gate inputs are left connected to the product term array so that new terms may be added. However, with this option, the unused OR-gate inputs are removed (deactivated) from the product term array. The result is reduced propagation delay from input to output.
r	In IFL devices, each product term from the AND- gate array may be shared among any number of OR- gate outputs. This option defeats this capability, forcing identical product terms to be generated for each output OR-array when required. The result is reduced propagation delay from input to output. This option will also force minimization to be performed on each output individually (as opposed to minimization on all outputs at once) when level **m2** or **m4** minimization is chosen.
g	Adds the necessary code in the JEDEC download file to automatically allow the device programmer to blow the security fuse when programming. Not all programmers support this option.
u	Overrides the default device library specified in the environment. Specify the complete path and filename for the library. Use this option on systems that may have special libraries created for unique or custom devices.
s	Creates the absolute file and automatically runs the **CSIM** logic simulator. **CSIM** is run with the **-l** option that creates a list file. If the **-j** flag was specified for **CUPL**, it will be passed to **CSIM**, creating a JEDEC download file with test vectors.

e	Generates an expanded macro definition file (**filename.MX**) which contains an expanded listing of all macros used in the source file. It also contains the expanded expressions that use the **REPEAT** command.
w	(MS-DOS only) Creates the absolute file and automatically runs the **CSIM** logic simulator with waveform output. **CSIM** is executed with the **-w** option that displays the output in wave form.
m0	Defeats all logic minimization during a **CUPL** compilation. It is useful when working with PROMs, to keep contained product terms from being eliminated.
m1 - m4	CUPL provides four minimization levels: **-m1**, **-m2**, **-m3**, and **-m4**. The default minimization level is **m1**. Figure U4-1 shows the relative memory usage, speed, and efficiency of the four minimization levels. Minimization levels **m2** and **m4** will perform multiple output minimization in IFL devices. This maximizes product term sharing in these types of devices.
zq	QuickLogic's QDIF file
kb	Optimize product term usage for pin or pinnode variables. This overrides the DEMORGAN statement if it appears in the source file
kd	DeMorganize all pin and pinnode variables. This overrides the DEMORGAN statement if it appears in the source file
ks	Force product term sharing during minimization. This is also referred to as group reduction
kx	Do not expand XOR to AND-OR equations. This is used for device independent designs or designs targeted for fitter-supported devices where the fitter supports XOR gates
q	Selects the Microsoft format for error messages. This applies only to the error messages displayed on the screen. (It does not affect the error format in the error listing file..) The reason for the alternate format is to allow CUPL to be executed within a text editor which has this feature (e.g. MULTI-EDIT) and once an error has been encountered, the file designated by the error message is brought to the screen with the cursor prompting at the line containing the error.

Boolean Logic

Table 4-3 shows the Boolean Logic rules for eliminating excess product terms from the expanded equations, used by the logic reduction algorithms built into the **CUPL** compiler.

Expression	Result
!0	= 1
!1	= 0
A & 0	= 0
A & 1	= A
A & A	= A
A & !A	= 0
A # 0	= A
A # 1	= 1
A # A	= A
A # !A	= 1
A & (A # B)	= A
A # (A & B)	= A

Table 4-3. Boolean Logic Rules

CSIM Command Line

Use the following comman line for running CSIM

csim [-flags] [library] [device] source

where

-flags is the following set of simulator options:

-l create listing file.

-j append test vectors to JEDEC file.

-n use source filename for JEDEC file.

-v display simulation results to terminal.

-u use specified library for simulation.

library is the library name and path name if the **-u** flag is being used to specify a library other than the default library.

device must be the same device mnemonic as was used in the CUPL compilation. Specifying the device is optional; if a device is not specified, **CSIM** uses the device CUPL compiled (contained in the **.ABS** file).

source is the user-created ASCII test specification file (***filename*.SI**). The extension **.SI** is assumed for the source file and may be omitted when giving the **CSIM** command.

The square brackets indicate optional items.

CSIM Option Flags

Multiple option flags can be specified when running **CSIM**. A hyphen must be used before the first flag entered, but can be omitted for subsequent flags. Spaces may also be placed between the flags. For example, the following two **CSIM** command lines are equivalent:

csim -l -v -j p16r4 waitgen Return

csim -lvj p16r4 waitgen Return

CSIM can be typed without any flags, to see the command line format and a list of the option flags.

Table 4-3 lists descriptions of the **CSIM** option flags.

Table 4-3. Simulator Option Flags

Option Flag	Description
-j	Appends the structured test vectors generated by the simulation onto the existing JEDEC download file.
-l	Generates a simulation listing file (*filename*.SO.) The input and output values for each variable are listed. Error messages are listed following each vector, with the signal name in error displayed.

-n	Allows the source filename to be used as the JEDEC filename instead of using the name in the NAME field of the source file.
-v	Displays the contents of the listing file to the screen. When the simulation data begins to appear on the screen, type Ctrl-S to stop the display (and any key to start it again) or Ctrl-C to cancel the simulation.
-u	Overrides the default device library specified in the environment. Specify the complete path and library name. This option is of particular use on systems that have special libraries created for unique or custom devices.

5. PLD Guide

This first chapter is intened as an introduction to programmable logic. You may consider skipping this section.

What Is Programmable Logic?

Programmable logic, as the name implies, is a family of components that contains arrays of logic elements (AND, OR, INVERT, LATCH, FLIP-FLOP) that may be configured into any logical function that the user desires and the component supports. There are several classes of programmable logic devices: ASICs, FPGAs, PLAs, PROMs, PALs, GALs, and complex PLDs.

ASICs

ASICs are *Application Specific Integrated Circuits* that are mentioned here because they are user definable devices. ASICs, unlike other devices, may contain analog, digital, and combinations of analog and digital functions. In general, they are mask programmable and not user programmable. This means that manufacturers will configure the device to the user specifications. They are used for combining a large amount of logic functions into one device. However, these devices have a high initial cost, therefore they are mainly used where high quantities are needed. Due to the nature of ASICs, CUPL and other programmable logic languages cannot fully support these devices.

Basic Architecture

First, a *user programmable device* is one that contains a pre-defined general architecture in which a user can program a design into the device using a set of development tools. The general architectures may vary but normally consists of one or more arrays of AND and OR terms for implementing logic functions. Many devices also contain combinations of flip-flops and latches which may be used as storage elements for inputs and outputs of a device. More complex devices contain *macrocells.* Macrocells allow the user to configure the type of inputs and outputs that are needed for a design.

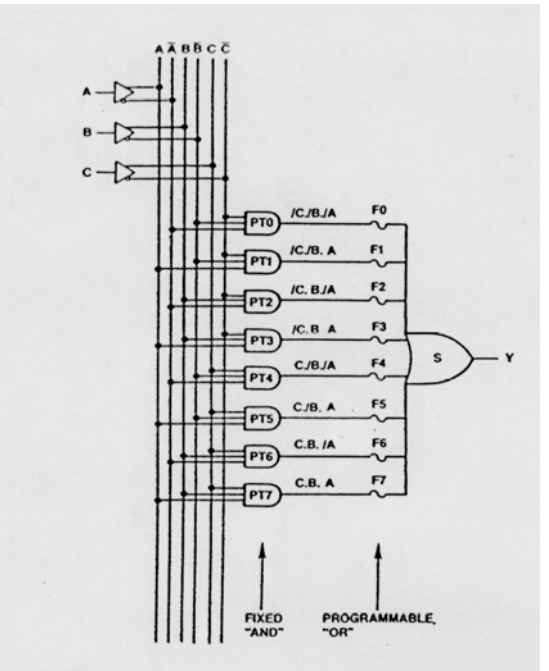


Figure 5-1 Elementary PROM Architecture

PROMs

PROMs are *Programmable Read Only Memories.* Even though the name does not imply programmable logic, PROMs, are in fact logic. The architecture of most PROMs typically consists of a fixed number of AND array terms that feeds a programmable OR array. They are mainly used for decoding specific input combinations into output functions, such as memory mapping in microprocessor environments.

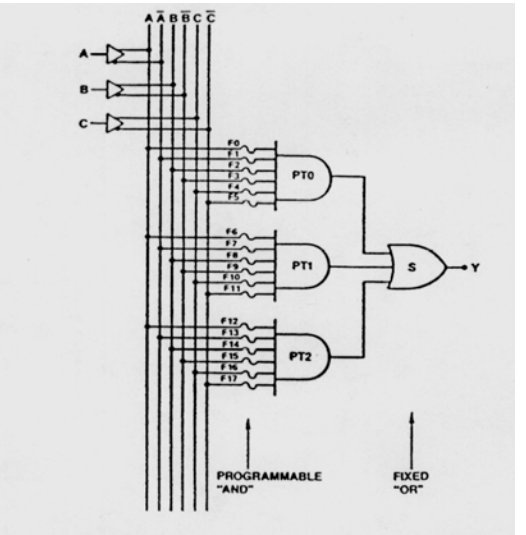


Figure 5-2 Elementary PAL architecture

PALs

PALs are *Programmable Array Logic* devices. The internal architecture consists of programmable AND terms feeding fixed OR terms. All inputs to the array can be ANDed together, but specific AND terms are dedicated to specific OR terms. PALs have a very popular architecture and are probably the most widely used type of user programmable device. If a device contains macrocells, it will usually have a PAL architecture. Typical macrocells may be programmed as inputs, outputs, or input/output (I/O) using a tri-state enable. They normally have output registers, which may or may not be used in conjunction with the associated I/O pin. Other macrocells have more than one register, various type of feedback into the arrays, and occasionally feedback between macrocells. These devices are mainly used to replace multiple TTL logic functions commonly referred to as *glue logic.*

GALs

GALs are *Generic Array Logic* devices. They are designed to emulate many common PALs thought the use of macrocells. If a user has a design that is implemented using several common PALs, he may configure several of the same GALs to emulate each of

the other devices. This will reduce the number of different devices in stock and increase the quantity purchased. Usually, a large quantity of the same device lowers the individual device cost. Also, these devices are electrically erasable, which makes them very useful for design engineers.

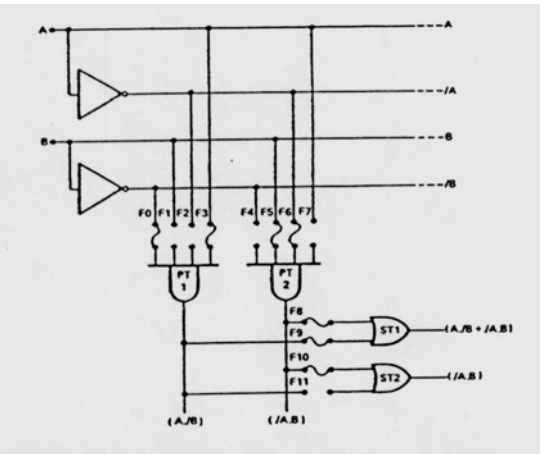


Figure 5-3 Elementary PLA architecture

PLAs

PLAs are *Programmable Logic Arrays.* These devices contain both programmable AND and OR terms which allow any AND term to feed any OR term. PLAs probably have the greatest flexibility of the other devices with regard to logic functionality. They typically have feedback from the OR array back into the AND array which may be used to implement asynchronous state machines. Most state machines, however, are implemented as synchronous machines. With this in mind, manufacturers created a type of PLA called a *Sequencer* which has registered feedback from the output of the OR array into the AND array.

Complex PLDs

Complex PLD's are what the name implies, *Complex Programmable Logic Devices.* They are considered very large PALs that have some characteristics of PLAs. The basic architecture is very much like a PAL with the capability to increase the amount of AND

terms for any fixed OR term. This is accomplished by either stealing adjacent AND terms, or using AND terms from an expander array. This allows for most any design to be implemented within these devices.

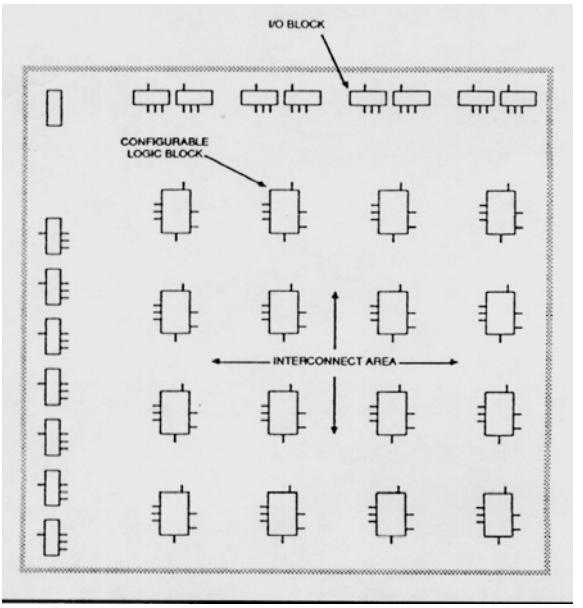


Figure 5-4 Elementary FPGA architecture

FPGAs

FPGAs are *Field Programmable Gate Arrays.* Simply put, they are electrically programmable gate array ICs that contain multiple levels of logic. FPGAs feature high gate densities, high performance, a large number of user-definable inputs and outputs, a flexible interconnect scheme, and a gate-array-like design environment. They are not constrained to the typical AND-OR array. Instead, they contain an interior matrix of configurable logic clocks (CLBs) and a surrounding ring of I/O blocks (IOBs). Each CLB contains programmable combinatorial logic and storage registers. The combinatorial logic section of the block is capable of implementing any Boolean function of its input variables. Each IOC can be programmed independently to be an input, and output with tri-state control or a bi-directional pin. It also contains flip-flops that can be used to buffer inputs and outputs. The interconnection resources are a network of lines that run horizontally and vertically in the rows and columns between the CLBs.

Programmable switches connect the inputs and outputs of IOBs and CLBs to nearby lines. Long lines run the entire length or breadth of the device, bypassing interchanges to provide distribution of critical signals with minimum delay or skew. Designers using FPGAs can define logic functions of a circuit and revise these functions as necessary. Thus FPGAs can be designed and verified in a few days, as opposed to several weeks for custom gate arrays.

Device Technologies And Packaging

Device Technologies

Some of the technologies available are CMOS (Complimentary Metal Oxide Semiconductor), bipolar TTL, GaAs (Gallium Arsenide), and ECL (Emitter Coupled Logic) as well as combination fabrications like BiCMOS and ECL/bipolar. The two fastest semiconductor technologies are ECL and GaAs. However, these are also the most power hungry. Generally speed is proportional to power consumption.

Device Packaging

The packaging for devices fall into two categories: erasability and physical configuration. Certain devices have the capability of being erased and reprogrammed. These devices are erased by either applying UV light, or a high voltage to re-fuse the cross-connection link. An UV erasable device will have a ”window” in the middle of the device that allows the UV light to enter inside. An electrically erasable device usually needs to have a high voltage applied to certain pins to erase the device. A device that cannot be erased is called One Time Programmable (OTP). As the name suggests, these devices can only be programmed once. Recent advances allow reprogramming without the use of high voltages

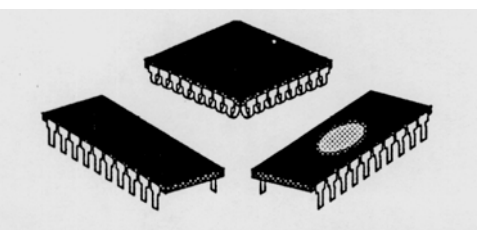


Figure 5-5 Picture of DIP and LCC devices

Programmable devices come in many shapes and sizes. Most devices come in the following physical configurations: DIP (Dual Inline Package), SKINNY-DIP, LCC (Leaded Chip Carrier), PLCC (Plastic Leaded Chip Carrier), QFP (Quad Flat Pack), BGA (Ball Grid Array), SOIC (Small Outline I.C.), TSOP (Thin Small Outline), and PGA (Pin Grid Array). These devices can be rectangular with pins on two sides, square with pins on all sides, or square with pins on the underside. It is important for the hardware and software development tools to fully support as many device types as possible to take full advantage of the myriad of devices on the market.

Programming Logic Devices

Programmable logic devices are programmed by either shorting or opening connections within a device array, thus connecting or disconnecting inputs to a gate. Most hardware programmers receive a fuse information file from a software development package in ASCII format. The ASCII file could either be in JEDEC format for PLDs or HEX format for PROMs. This file contains the information necessary for the programmer to program the device. The JEDEC file contains fuse connections that are represented by an address followed by a series of 1's and 0's where a "1" indicates a disconnected state and a "0" indicates a connected state. The JEDEC file can also contain information that allows the hardware programmer the ability to perform a functional test on the device.

Functionally Testing Logic Devices

A functional test may be performed after programming a device, provided that the hardware and software development package can support the generation and use of test vectors. Test vectors consist of a list of pins for the design, input values for each step of the functional test, and a list of expected outputs from the circuit. The programmer sequences through the input values, looks for the predicted outputs, and reports the results to the user. This allows design engineers and production crews the ability to verify that the programmed device works as designers.

6. CUPL Language Reference

This chapter explains CUPL language elements and CUPL language syntax.

Language Elements

This section describes the elements that comprise the CUPL logic description language

Variables

Variables are strings of alphanumeric characters that specify device pins, internal nodes, constants, input signals, output signals, intermediate signals, or sets of signals. This section explains the rules for creating variables.

- Variables can start with a numeric digit, alphabet character, or underscore, but must contain at least one alphabet character.
- Variables are case sensitive; that is, they distinguish between uppercase and lowercase letters.
- Do not use spaces within a variable name. Use the underscore character to separate words.
- Variables can contain up to 31 characters. Longer variables are truncated to 31 characters.
- Variables cannot contain any of the CUPL reserved symbols (see Table 6-2).
- Variables cannot be the same as a CUPL reserved keyword (see Table 6-1).

Examples of some valid variable names are:

a0

A0

8250_ENABLE

Real_time_clock_interrupt

_address

Note how the use of the underscore in the above examples makes the variable names easier to read. Also, note the difference between uppercase and lowercase variable names. The variable A0 is not the same as a0.

Examples of some invalid variable names are:

99	does not contain an alpha character
I/O enable	contains a special character (/)
out 6a	contains a space; the system reads it as two separate variables
tbl-2	contains a dash; the system reads it as two variables.

Indexed Variables

Variable names can be used to represent a group of address lines, data lines, or other sequentially numbered items. For example, the following variable names could be assigned to the eight low order address lines of a microprocessor:

A0 A1 A2 A3 A4 A5 A6 A7

Variable names that end in a number, as shown above, are referred to as indexed variables.

It is best to start indexed variables from zero (0).
e.g. Use X0...4 instead of X1...5.

The index numbers are always decimal numbers between 0 and 31. When used in bit field operations (see the subtopic, Bit Field Declaration Statements in this chapter) the variable with index number 0 is always the lowest order bit.

Variables ending in numbers greater than 31 are not indexed variables

Examples of some valid indexed variable names are as follows:

a23

D07

D7

counter_bit_3

Note the difference between index variables with leading zeroes; the variable **D07** is not the same as **D7**.

Examples of some invalid indexed variable names are as follows:

D0F	index number is not decimal
a36	index number out of range

These are valid variable names, but they are not considered indexed.

Reserved Words and Symbols

CUPL uses certain character strings with predefined meanings called **keywords**. These keywords cannot be used as names in CUPL. Table 6-1 lists these keywords.

Table 6-1 CUPL Reserved Keywords

APPEND	FORMAT	OUT
ASSEMBLY	FUNCTION	PARTNO
ASSY	FUSE	PIN
COMPANY	GROUP	PINNNODE
CONDITION	IF	PRESENT
DATE	JUMP	REV
DEFAULT	LOC	REVISION
DESIGNER	LOCATION	SEQUENCE
DEVICE	MACRO	SEQUENCED
ELSE	MIN	SEQUENCEJK
FIELD	NAME	SEQUENCERS
FLD	NODE	SEQUENCET
TABLE		

CUPL also reserves certain **symbols** for its use that cannot be used in variable names. Table 6-2 lists these reserved symbols.

Table 6-2 CUPL Reserved Symbols

&	#	()	-
*	+	[]	/
:	.	..	/* */
;	,	!	'
=	@	$	^

Numbers

All operations involving numbers in the CUPL compiler are done with 32-bit accuracy. Therefore, the numbers may have a value from 0 to 2^{32} -1. Numbers may be represented in any one of the four common bases: binary, octal, decimal, or hexadecimal. The default base for all numbers used in the source file is hexadecimal, except for device pin numbers and indexed variables, which are always decimal. Numbers for a different base may be used by preceding them with a prefix listed in Table 6-3. Once a base change has occurred, that new base is the default base in the design file until another base is used

Table 6-3. Number Base Prefixes

Base Name		Base	Prefix
Binary	2	'b'	
Octal	8	'o'	
Decimal	10	'd'	
Hexadecimal	16	'h'	

The base letter is enclosed in single quotes and can be either uppercase or lowercase. Some examples of valid number specifications are listed in Table 6-4.

Table 6-4. Sample Base Conversions

Number		Base	Decimal Value
'b'0	Binary	0	
'B'1101	Binary	13	
'O'663	Octal	435	
'D'92	Decimal	92	
'h'BA	Hexadecimal	186	
'O'[300..477]	Octal (range)	192..314	

Binary, octal, and hexadecimal numbers can have don't-care values (“X”) and numerical values. Some examples of valid number specifications with don't-care values are listed in Table 6-5.

Table 6-5. Sample Don't-Care Numbers

Number	Base
'b'1X11	Binary
'O'0X6	Octal
'H'[3FXX..7FFF]	Hexadecimal (range)

Comments

Comments are an important part of the logic description file. They improve the readability of the code and document the intentions, but do not significantly affect the compile time, as they are removed by the preprocessor before any syntax checking is done. Use the symbols **/*** and ***/** to enclose comments; the program ignores everything between these symbols. C style comments // can be used to tell the compiler to ignore everything until the end-of-line marker is reached.

Comments may span multiple lines but cannot be nested. Some examples of valid comments are shown in Figure 6-1.

```
/*****************************************/
/*  This is one way to create a title or   */
/*         an information block            */
/*****************************************/

/* This is another another way to create an information block      */

out1=in1 # in2;       /* A Simple OR Function  */
out2=in1 & in2;       /* A Simple AND Function */
out3=in1 $ in2;       // A Simple XOR Function
```

Figure 6-1. Sample Comments

List Notation

Shorthand notations are an important feature of the CUPL language.

The most frequently used shorthand notation is the list. It is commonly used in pin and node declarations, bit field declarations, logic equations, and set operations. The list format is as follows:

[variable, variable, ... variable]

where

[ ] are brackets used to delimit items in the list as a set of variables.

Two examples of the list notation are as follows:

[UP, DOWN, LEFT, RIGHT]
[A0, A1, A2, A3, A4, A5, A6, A7]

When all the variable names are sequentially numbered, either from lowest to highest or vice versa, the following format may be used:

[variablem..n]

where

m is the first index number in the list of variables.

n is the last number in the list of variables; n can be written without the variable name.

For example, the second line from the example above could be written as follows:

[A0..7]

Index numbers are assumed to be decimal and contiguous. Any leading zeros in the variable index are removed from the variable name that is created. For example:

[A00..07]

is shorthand for:

[A0, A1, A2, A3, A4, A5, A6, A7]

not for:

[A00, A01, A02, A03, A04, A05, A06, A07]

The two forms for the list notation may be mixed in any combination. For example, the following two list notations are equivalent:

[A0..2, A3, A4, A5..7]

[A0, A1, A2, A3, A4, A5, A6, A7]

Template File

When a logic description source file is created using the CUPL language, certain information must be entered, such as header information, pin declarations, and logic equations. For assistance, CUPL provides a template file that contains the proper structure for the source file.

Figure 6-2 shows the contents of the template file.

```
Name     XXXXX;
Partno   XXXXX;
Date     XX/XX/XX;
Revision XX;
Designer XXXXX;
Company  XXXXX;
Assembly XXXXX;
Location XXXXX;
/*********************************************************/
/*                                                       */
/*                                                       */
/*********************************************************/
/*  Allowable Target Device Types:                       */
/*********************************************************/

/**  Inputs  **/
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */

/**  Outputs  **/
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                                */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */
Pin          =           ;      /*                        */

/** Declarations and Intermediate Variable Definitions **/

/**  Logic Equations  **/
```

Figure 6-2. Template File

The template file provides the following sections:

Header Information - Keywords followed by XXXs that are replaced with text to identify the file for archival and revision purposes.

Title Block - Comment symbols that enclose space for describing the function of the design and allowable target devices.

Pin Declaration - Keywords and operators in the proper format for input and output pin declarations and comment space to describe the pin assignments. After pin declarations are made, remove any extra "**pin = ;**" lines. Otherwise, a syntax error will occur during compilation.

The **/* Inputs */** and **/* Outputs */** are comments that provide groupings for readability only. Assign any pin type in any order, no matter how it is used in the logic description file.

Declaration and Intermediate Variable - Space for making declarations, such as bit field declarations (see the subtopics, Bit Field Declaration Statements and Node Declaration Statements in this chapter) and for writing intermediate equations (see the subtopic, Logic Equations in this chapter).

Logic Equation - Space for writing logic equations describing the function of the device (see the subtopic, Logic Equations in this chapter).

Header Information

The header information section of the source file identifies the file for revision and archival purposes. Normally place it at the beginning of the file. CUPL provides 10 keywords to use in header information statements. Begin each statement with a keyword which can be followed by any valid ASCII characters, including spaces and special characters. End each statement with a semicolon. Table 6-6 lists the CUPL header keywords and the information to provide with each keyword.

Table 6-6. Header Information

Keyword	Information
NAME	Normally use the source logic description filename. Use only character strings that are valid for the operating system. The name specified here determines the name for any JEDEC, ASCII - hex, or HL download files. The NAME field accommodates filenames up to 32 characters long. When using systems such as DOS which allow filenames of only eight characters, the filename will be truncated.
PARTNO	Specify a company's proprietary part number (usually issued by manufacturing) for a particular PLD design. The part number is not the type of target PLD. For GAL devices, the first eight characters are encoded using seven-bit ASCII in the User Signature Fuses of the devices' fuse map.
REVISION	Begin with 01 when first creating a file and increment each time a file is altered. REV can be used for an abbreviation.
DATE	Change to the current date each time a source file is altered.
DESIGNER	Specify the designer's name.
COMPANY	Specify the company's name for proper documentation practice and because specifications may be sent to semiconductor manufacturers for high volume PLD orders.
ASSEMBLY	Give the assembly name or number of the PC board on which the PLD will be used. The abbreviation ASSY can be used.
LOCATION	Indicate the PC board reference or coordinate where the PLD is located. The abbreviation LOC can be used.

DEVICE	Set the default device type for the compilation. A device type specified on the command line overrides all device types set in the source file. For multi-device source files, DEVICE must be used with each section if the device types are different.

An example of proper CUPL header information is as follows:

```
Name        WAITGEN ;
Partno      P9000183 ;
Revision    02 ;
Date        1/11/98 ;
Designer    Osann ;
Company     Logical Devices, Inc. ;
Assembly    PC Memory Board ;
Location    U106 ;
Device      Virtual;
```

If any header information is omitted, CUPL issues a warning message, but continues with compilation.

Pin Declaration Statements

Pin declaration statements declare the pin numbers and assign them symbolic variable names. The format for a pin declaration is as follows:

PIN pin_n=[!]var ;

where

PIN is a keyword to declare the pin numbers and assign them variable names.

pin_n is a decimal pin number or a list of pin numbers grouped using the list notation; that is,

[pin_n 1, pin_n 2 ... pin_nn]

! is an optional exclamation point to define the polarity of the input or output signal.

= is the assignment operator.

var is a single variable name or a list of variables grouped using the list notation; that is,

[var, var ... var]

; is a semicolon to mark the end of the pin declaration statement.

The template file provides a section for entering the pin variables individually or in groups using the list notation.

The concept of polarity can often be a confusing one. In any PLD design, the designer is primarily concerned with whether a signal is true or false. The designer should not have to care whether this means that the signal is high or low. For a variety of reasons a board design may require a signal to be considered true when it is logic level 0(low) and false when it is logic 1(high). This signal is considered active-low since it is activated when it is low. This might also be called low-true. If a signal is changed from active-high to active low then the polarity has been changed.

For this reason, CUPL allows you to declare signal polarity in the pin definition and then you do not have to be concerned with it again. When writing equations in CUPL syntax, the designer should not be concerned with the polarity of the signal. The pin declarations declare a translation that will handle the signal polarity.

Suppose that we wanted the following function.

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

What this statement means is that Y will be true when A is true and B is true. We can implement this in a P22V10 device very easily.

```
Pin 2 = A;
Pin 3 = B;
Pin 16 = Y;
Y = A & B;
```

When the device is plugged into a circuit, if a logic 1 is asserted at pins 2 and 3 then the signal at pin 16 will be high.

Let us assume that for some reason we wanted the inputs to read logic 0 as true. We could modify the design to behave this way.

```
Pin 2 = !A;
Pin 3 = !B;
Pin 16 = Y;
Y = A & B;
```

Now even though the ! symbol was placed in the pin declaration to indicate the inverted polarity, the equation still reads as “Y is true when A is true and B is true”. All that has been changed is the translation of true=0 and false=1. So at the design level nothing has changed but in the pin declarations we now map 0 to true and 1 to false.

This promotes the designer to separate the design into layers so as to minimize confusion related to polarity. It is important also that CUPL will modify the feedback signal so that the true/false layer is maintained.

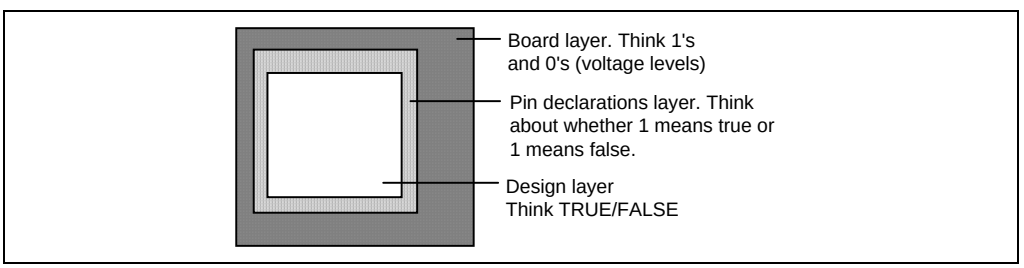


Figure 6-2 Relationship Between Pin Declaration and Signal Polarity.

Use the exclamation point (**!**) to define the polarity of an input or output signal. If an input signal is active-level LO (that is, the asserted TTL signal voltage level is 0 volts), put an exclamation point before the variable name in the pin declaration. The exclamation point informs the compiler to choose the inverted sense of the signal when it is listed as active in the logic equations. The virtual device is an exception to this rule, however. When using the virtual device, CUPL ignores the polarity in the pin declaration. In this case, the equation itself must be negated.

Similarly, if an output signal is active-level LO, define the variable with an exclamation point in the pin declaration and write the logic equation in a logically true form. Use of the exclamation point permits declaring pins without regard to the limitations of the type of target device. With the virtual device, the equation itself must be inverted, since the compiler ignores the polarity in the pin declaration.

If a pin declaration specifying an active-level HI output is compiled for a target device (such as a PAL16L8) that has only inverting outputs, CUPL automatically performs DeMorgan's Theorem on the logic equation to fit the function into the device.

Consider the following example. The logic description file is written for a PAL16L8 device. All output pins are declared as active-HI. The following equation has been written to specify an OR function:

```
c = a # b ;
```

However, because the PAL16L8 contains a fixed inverting buffer on the output pins, CUPL must perform DeMorganization to fit the logic to the device. CUPL generates the following product term in the documentation file:

```
c => ! a & ! b
```

Figure 6-3 shows the process described above.

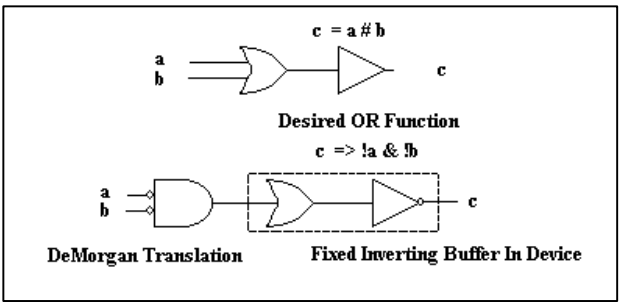


Figure 6-3 Active-HI Pin Declaration for Inverting Buffer

If a design has excessive product terms, CUPL displays an error message and the compilation stops. The documentation file (***filename*.DOC**) lists the number of product terms required to implement the logic function and the number of product terms the device physically has for the particular output pin.

Some examples of valid pin declarations are:

```
pin 1     =     clock;    /*   Register Clock   */
pin 2     =     !enable;  /* Enable I/O Port */
pin [3,4] =     ![stop,go];       /* Control Signals  */
pin [5..7] =    [a0..2];  /* Address Bits 0-2 */
```

The last two lines in the example above are shorthand notations for the following:

```
pin 3     =     !stop;    /* Control Signal */
pin 4     =     !go;            /* Control Signal */
pin 5     =     a0;             /* Address Bit 0   */
pin 6     =     a1;             /* Address Bit 1   */
pin 7     =     a2;             /* Address Bit 2   */
```

For the virtual device, the pin numbers may be left out. This provides a way to do a design without regard for any device related restrictions. The designer can then examine the results and thereby determine the requirements for implementation. The target device can then be chosen. The following are valid pin declarations when using the virtual device.

```
pin     =     !stop;          /* Control Signal */
pin     =     !go;            /* Control Signal */
pin     =     a0;             /* Address Bit 0  */
pin     =     a1;             /* Address Bit 1  */
pin     =     a2;             /* Address Bit 2  */
```

The input, output, or bi-directional nature of a device pin is not specified in the pin declaration. The compiler infers the nature of a pin from the way the pin variable name is used in the logic specification. If the logic specification and the physical characteristics of the target device are incompatible, CUPL displays an error message denoting the improper use of the pin.

Pinnode Declaration Statements

Some devices contain functions that are not available on external pins, but logic equations must be written for these capabilities. For example, the atf1500 contains both buried state registers (flip-flops) and a mechanism for inverting any transition term through a complement array. Before writing equations for these flip-flops (or complement arrays), they must be assigned variable names. Since there are no pins associated with these functions, the **PIN** keyword cannot be used. Use the **PINNODE** keyword to declare variable names for buried functions.

The **PINNODE** keyword is used for explicitly defining buried nodes by assigning a node number to a symbolic variable name. This is similar to the way the pin declaration statements work. The format for a pinnode declaration is as follows:

PINNODE node_n = [!]var;

where

PINNODE is a keyword to declare the node numbers and assign them variable names.

node_n is a decimal node number or a list of node numbers grouped using the list notation; that is,

[node_n1,node_n2 ... node_nn]

! is an optional exclamation point to define the polarity of the internal signal.

= is the assignment operator.

var is a single variable name or list of variables grouped using the list notation; that is,

[var,var ... var]

; is a semicolon used to mark the end of the statement.

For devices that use a fitter, the **node_n** can be ommitted and the fitter will assign the pinnode number. The **Backpin** utilty can be used to place the fitter assigned placement into the source file.

Place pinnode declarations in the “Declarations and Intermediate Variables Definitions” section of the source file provided by the template file.

A list of pinnode numbers for all devices containing internal nodes is included in the Appendix section. Please reference these node numbers for pinnode declarations.

Examples of the use of the **PINNODE** keyword are:

```
PINNODE [29..34] = [State0..5]; /* Internal State Bits */
PINNODE 35 = !Invert;           /* For Complement Array   */
PINNODE 25 = Buried;            /* For Buried register part */
```

Bit Field Declaration Statements

A bit field declaration assigns a single variable name to a group of bits. The format is as follows:

FIELD var = [var, var, ... var] ;

where

FIELD is a keyword.

var is any valid variable name.

[var, var, ... var] is a list of variable names in list notation.

= is the assignment operator.

; is a semicolon used to mark the end of the statement.

The square brackets do not indicate optional items.
They are used to delimit items in a list.

Place bit field declarations in the "Declarations and Intermediate Variable Definitions" section of the source file provided by the template file.

After assigning a variable name to a group of bits, the name can be used in an expression; the operation specified in the expression is applied to each bit in the group. See the subtopic, Set Operations in this chapter for a description of the operations allowed for **FIELD** statements. The example below shows two ways to reference the eight address input bits (A0 through A7) of an I/O decoder as the single variable named ADDRESS.

FIELD ADDRESS = [A7,A6,A5,A4,A3,A2,A1,A0] ;

or

FIELD ADDRESS = [A7..0] ;

When a FIELD statement is used, the compiler generates a single 32-bit field internally. This is used to represent the variables in the bit field. Each bit represents one member of the bit field. **The bit number which represents a member of a bit field is the same as the index number if indexed variables are used**. This means that A0 will always

occupy bit 0 in the bitfield. This also means that the order of appearance of indexed variables in a bit field has no significance. A bit field declared as [A0..7] is exactly the same as a bit field declared as [A7..0]. Because of this mechanism, different indexed variables should not be included in the same bit field. A bit field containing A2 and B2 will assign both of these variables to the same bit position. This will result in the generation of erroneous equations.

Also, bit fields should never contain both indexed and non-indexed variables. This will almost certainly result in erroneous generation of equations.

Do not mix indexed and non-indexed variables in a field statement. The compiler may produce unexpected results.

MIN Declaration Statements

The **MIN** declaration statement overrides, for specified variables, the minimization level specified on the command line when running CUPL. The format is as follows:

MIN var [.ext] = level ;

where

MIN is a keyword to override the command line minimization level.

var is a single variable declared in the file or a list of variables grouped using the list notation; that is,

[var, var, ... var]

.ext is an optional extension that identifies the function of the variable.

level is an integer between 0 and 4.

; is a semicolon to mark the end of the statement.

The levels 0 to 4 correspond to the option flags on the command line, **-m0** through **-m4**.

The **MIN** declaration permits specifying different levels for different outputs in the same design, such as no reduction for outputs requiring redundant or contained product terms (to avoid asynchronous hazard conditions), and maximum reduction for a state machine application.

The following are examples of valid **MIN** declarations.

```
MIN async_out       = 0;    /* no reduction   */
MIN [outa, outb]    = 2;    /* level 2 reduction      */
MIN count.d         = 4;    /* level 4 reduction      */
```

Note that the last declaration in the example above uses the **.d** extension to specify that the registered output variable is the one to be reduced.

FUSE Statement

The **FUSE** statement provides for special cases where it is necessary to blow TURBO or MISER bits. This statement should be used with utmost care, as it can lead to unpredictable results if used incorrectly.

FUSE (fusenumber, x)

where **fusenumber** is the fuse number corresponding to the bit or that must be blown, and x is either 0 or 1. Specify 0 if the bit must not be blown. Specify 1 to blow the bit. **Use this statement with extreme caution**.

In this example, fuse 101 is a MISER Bit or TURBO Bit. This blows fuse number 101.

example:

FUSE(101,1)

DO NOT ATTEMPT TO USE THIS STATEMENT TO BLOW ARBITRARY FUSES!

The fuse statement was designed to blow MISER bits and TURBO Bits only. The exact fuse number for the TURBO or MISER Bit must be specified. Every time this statement is used, CUPL will generate a warning. This is a reminder to double check that the fuse number specified is correct. If a wrong fuse number is specified, disastrous results can occur. Be very careful using this statement. If the **FUSE** statement is used in a design and strange results occur, check the fuse number specified and make sure that it is a correct bit.

Field Comparison Operation

Field comparison operation "==" compares two fields and generates **TRUE** only if the two fields are identical. The two field variables must have the same number of elements (bits).

For example:

```
Field f1 = [a3..0];
Field f2 = [b3..0];
x = f1 == f2;
```

The output x is true only when a3..0 and b3..0 are identical. CUPL implements the field comparison operation by using the following equivalent logic:

```
x = !(a0 $ b0) & !(a1 $ b1) & !(a2 $ b2) & !(a3 $ b3);
```

Extension .CMP

For some devices which have a built-in hardware compare unit, you must specify .CMP extension to the left hand side variable to use the hardware compare function. In this case CUPL does not expand the comparison operation into the low-level equations.

```
x.cmp = f1 == f2;
```

".CMP" extension is used for hardware comparison circuits. It can only be used when the device supports a hardware compare entity such as Intel’s iFX780.

DECLARE

The DECLARE statement is used to declare the attribute of the pins or pinnodes. The attributes can be a hardware property such as a logic entity such as global input buffer and RAM block.

DECLARE <manuf ID> <attrib> <variable list>

CUPL will do the design rule checking for the DECLARE statement.

For example, the following example uses a DECLARE statement to declare a RAM4 block.

```
Pin   =  [A3..0];
Pin   =  WE;
Pin   =  D
Pin   =  O;

DECLARE XILINX RAM4 [A0,A1,A2,A3,WE,D,O];
```

PROPERTY

The PROPERTY statement performs the same function as the DECLARE statement. Unlike the DECLARE statement, CUPL does not do any design rule checking on the PROPERTY statement.

PROPERTY <manuf ID> { property statement };

The following example uses the PROPERTY statement to declare a delay clock. For details on valid property statements, reference the reference the specific fitter.

```
Pin    =   CLK;
Pin    =   X;
Pin    =   Y;
PROPERTY INTEL  { @PIN_ATTRIB X DELAYCLK };
PROPERTY INTEL  { @PIN_ATTRIB Y DELAYCLK };


 [X, Y].ck = CLK;
```

DEMORGAN

You can use the DEMORGAN statement to control the application of DeMorgan's theorem to the equations. It is possible to reduce the number of product terms used.

DEMORGAN [var_list] = Demorgan Option;

Demorgan Option is a number from 0 to 2:

0 standard expression evaluation (default value)

1 force DeMorgan of expression

2 applies DeMorgan's theorem to see if the number of product terms can be reduced.

The DeMorgan statement can only be used with devices that have programmable polarity. That is, any device that has a polarity fuse and/or mux. This statement can also be used with VIRTUAL.

When you select a fixed polarity device such as the P16L8, CUPL will ignore the DEMORGAN statement and evaluate the expression to fit the device. When you select a programmable polarity device or VIRTUAL, CUPL will apply DeMorgan's theorem to the expression depending on the value specified in the DEMORGAN statement. Following are some examples to show how the DEMORGAN statement works.

```
Device  p16l8;

Pin  16  =  !x;

DEMORGAN [x] = 2;  /* best usage of product terms */
x = a # b;
```

Figure 6-4 Fixed polarity device, DEMORGAN statement ignored

CUPL will not apply DeMorgan's theorem on the expression "x = a # b" since pin 16 of the P16L8 has a fixed inverting buffer.

```
Device  p22v10;

Pin 16 = !x;

DEMORGAN [x] = 2; /* best usage of product terms */
x = a # b;
```

Figure 6-5 Choos the best solution for an output in a programmable polarity device

CUPL will generate a DeMorgan equivalent expression for the output (!x = !a & !b) since this version of the expression uses the least number of product terms.

REGISTER_SELECT

The REGISTER_SELECT statement allows the user to convert between different register types automatically. CUPL will generate equivalent logic expressions for the specified register type.

REGISTER_SELECT [var_list] = register_type;

The register_type is a number that indicates the target register:

0 use the specified register

1 D

2 T

3 JK

4 SR

5 select best usage of product terms between D and T

```
REGISTER_SELECT [x] = 1;
x.j = a;
x.k = b;
```

Figure 6-6 Convert a JK registered expression to a D registered expression

CUPL will convert the JK type expression and generate the following D type expression.

```
x.d => a & !x
     # a & !b
     # !b & x;
```

Figure 6-7 Converted D registered expression

Preprocessor Commands

The preprocessor portion of CUPL operates on the source file before it is passed to the parser and other sections of the compiler. The preprocessor commands add file inclusion, conditional compilation, and string substitution capabilities to the source processing features of CUPL. Table 6-7 lists the available preprocessor commands. Each command is described in detail in this section.

Table 6-7. Preprocessor Commands

$DEFINE	$IFDEF	$UNDEF
$ELSE	$IFNDEF	$REPEAT
$ENDIF	$INCLUDE	$REPEND
$MACRO	$MEND	

The dollar sign (**$**) is the first character in all preprocessor commands and must be used in column one of the line. Any combination of uppercase or lowercase letters may be used to type these commands.

$DEFINE

This command replaces a character string by another specified operator, number, or symbol. The format is as follows:

$DEFINE argument1 argument2

where

argument1 is a variable name or special ASCII character.

argument2 is a valid operator, a number, or a variable name.

“Argument1” is replaced by “argument2” at all locations in the source specification after the **$DEFINE** command is given (or until the preprocessor encounters an **$UNDEF** command). The replacement is a literal string substitution made on the input file before being processed by the CUPL compiler. Note that no semicolon or equal sign is used for this command.

The **$DEFINE** command allows numbers or constants to be replaced with symbolic names, for example:

```
$DEFINE     ON          'b'1
$DEFINE     OFF         ‘b'0
$DEFINE     PORTC       'h'3F0
```

The **$DEFINE** command also allows creation of a personal set of logical operators. For example, the following define an alternate set of operators for logic specification:

```
$DEFINE   /     !     Alternate Negation
$DEFINE   *     &     Alternate AND
$DEFINE   +     #     Alternate OR
$DEFINE   :+:   $     Alternate XOR
$DEFINE   {     /*    Alternate Start Comment
$DEFINE   }     */    Alternate End Comment
```

The above definitions are contained in the **PALASM.OPR** file included with the CUPL software package. This file may be included in the source file (see **$INCLUDE** command) to allow logic equations using the PALASM set of logical operator symbols, as well as the standard CUPL operator symbols.

$UNDEF

This command reverses a **$DEFINE** command. The format is as follows:

$UNDEF argument

where

argument is an argument previously used in a **$DEFINE** command.

Before redefining a character string or symbol defined with the **$DEFINE** command, use the **$UNDEF** command to undo the previous definition.

$INCLUDE

This command includes a specified file in the source to be processed by CUPL. The format is as follows:

$INCLUDE filename

where

filename is the name of a file in the current directory.

File inclusion allows standardizing a portion of a commonly used specification. It is also useful for keeping a separate parameter file that defines constants that are commonly used in many source specifications. The files that are included may also contain **$INCLUDE** commands, allowing for "nested" include files. The named file is included at the location of the **$INCLUDE** command.

For example, the following command includes the **PALASM.OPR** file in a source file.

$INCLUDE PALASM.OPR

PALASM.OPR is included with the CUPL software and contains **$DEFINE** commands that specify the following alternate set of logical operators.

```
$DEFINE   /      !      Alternate Negation
$DEFINE   *      &      Alternate AND
$DEFINE   +      #      Alternate OR
$DEFINE   :+:    $      Alternate XOR
$DEFINE   {      /*     Alternate Start Comment
$DEFINE   }      */     Alternate End Comment
```

$IFDEF

This command conditionally compiles sections of a source file. The format is as follows:

$IFDEF argument

where

argument may or may not have previously been defined with a **$DEFINE** command.

When the argument has previously been defined, the source statements following the **$IFDEF** command are compiled until the occurrence of an **$ELSE** or **$ENDIF** command.

When the argument has not previously been defined, the source statements following the **$IFDEF** command are ignored. No additional source statements are compiled until the occurrence of an **$ELSE** or **$ENDIF** command.

One use of **$IFDEF** is to temporarily remove source equations containing comments from the file. It is not possible to "comment out" the equations because comments do not nest. The following example illustrates this technique. NEVER is an undefined argument.

```
$IFDEF NEVER
out1=in1 & in2;      /* A Simple AND Function */
out2=in3 # in4;      /* A Simple OR Function */
$ENDIF
```

Because NEVER is undefined, the equations are ignored during compilation; that is, they function as comments.

$IFNDEF

This command sets conditions for compiling sections of the source file.

```
$IFNDEF argument
```

where

argument may or may not have previously been defined with a **$DEFINE** command.

The **$IFNDEF** command works in the opposite manner of the **$IFDEF** command. When the argument has not previously been defined, the source statements following the **$IFNDEF** command are compiled until the occurrence of an **$ELSE** or **$ENDIF** command.

If the argument has previously been defined, the source statements following the **$IFNDEF** command are ignored. No additional source statements are compiled until the occurrence of an **$ELSE** or **$ENDIF** command.

One use of **$IFNDEF** is to create a single source file containing two mutually exclusive sets of equations. Using an **$IFNDEF** and **$ENDIF** command to set off one of the sets of

equations, quick toggling is possible between the two sets of equations by defining or not defining the argument specified in the **$IFNDEF** command.

For example, some devices contain common output enable pins that directly control all the tri-state buffers, whereas other devices contain single product terms to enable each tri-state buffer individually. In the following example, the argument, **COMMON_OE** has not been defined, so the equations that follow are compiled. Any equations following **$ENDIF** are not compiled.

```
$IFNDEF COMMON_OE
pin 11              = !enable;      /* input pin for OE*/
[q3,q2,q1,q0].oe    = enable;       /* assign tri-state*/
/* equation for     outputs   */
$ENDIF
```

If the device has common output enables, no equations are required to describe it. Therefore, in the above example, for a device with common output enables, define **COMMON_OE** so the compiler skips the equations between **$IFNDEF** and **$ENDIF**.

$ENDIF

This command ends a conditional compilation started with the **$IFDEF** or **$IFNDEF** commands. The format is as follows:

```
$ENDIF
```

The statements following the **$ENDIF** command are compiled in the same way as the statements preceding the **$IFDEF** or **$IFNDEF** commands. Conditional compilation may be nested, and for each level of nesting of the **$IFDEF** or **$IFNDEF** command, an associated **$ENDIF** must be used.

The following example illustrates the use of **$ENDIF** with multiple levels of nesting.

```
$IFDEF      prototype_1
pin 1       = set;          /* Set on pin 1*/
pin 2       = reset;  /* Reset on pin 2*/
$IFDEF      prototype_2
pin 3       = enable;       /* Enable on pin 3*/
pin 4       = disable;      /* Disable on pin 4*/
```

```
$ENDIF
pin 5       = run;    /* Run on pin 5*/
pin 6       = halt;   /* Halt on pin 6*/
$ENDIF
```

$ELSE

This command reverses the state of conditional compilation as defined with **$IFDEF** or **$IFNDEF**. The format is as follows:

```
$ELSE
```

If the tested condition of the **$IFDEF** or **$IFNDEF** commands is true (that is, the statements following the command are compiled), then any source statements between an **$ELSE** and **$ENDIF** command are ignored.

If the tested condition is false, then any source statements between the **$IFDEF** or **$IFNDEF** and **$ELSE** command are ignored, and statements following **$ELSE** are compiled.

For example, many times the production printed circuit board uses a different pinout than does the wire-wrap prototype. In the following example, since Prototype has been defined, the source statements following **$IFDEF** are compiled and the statements following **$ELSE** are ignored.

```
$DEFINE Prototype X       /* define Prototype*/
$IFDEF Prototype
pin 1       = memreq;     /* memory request on  */
                          /* pin 1 of prototype*/
pin 2       = ioreq;      /* I/O request on*/
                          /* pin 2 of prototype*/
$ELSE
pin 1       = ioreq;      /* I/O request on*/
                          /* pin 1 of PCB*/
pin 2       = memreq;     /* memory request on  */
                          /* pin 2 of PCB*/
```

```
$ENDIF
```

To compile the statements following **$ELSE**, remove the definition of Prototype.

$REPEAT

This command is similar to the FOR statement in C language and DO statements in FORTRAN language. It allows the user to duplicate repeat body by index. The format is as follows:

```
$REPEAT index=[number1,number2,…numbern]
    repeat body
$REPEND
```

where **n** can be any number in the range 0 to 1023

In preprocessing, the repeat body will be duplicated from $number_1$ to $number_n$. The index number can be written in short form as [number1..numbern] if the number is consecutive. The repeat body can be any CUPL statement. Arithmetic operations can be performed in the repeat body. The arithmetic expression must be enclosed by braces **{ }**.

For example, design a three to eight decoder.

```
FIELD sel = [in2..0]
$REPEAT i = [0..7]
    !out{i} = sel:'h'{i} & enable;
$REPEND
```

Where index variable i goes from 0 to 7, so the statement “out{i} = sel:'h'{i} &enable;” will be repeated during preprocessing and create the following statements:

```
FIELD sel = [in2..0];
    !out0 = sel:'h'0 & enable;
    !out1 = sel:'h'1 & enable;
    !out2 = sel:'h'2 & enable;
    !out3 = sel:'h'3 & enable;
    !out4 = sel:'h'4 & enable;
    !out5 = sel:'h'5 & enable;
    !out6 = sel:'h'6 & enable;
    !out7 = sel:'h'7 & enable;
```

The following example shows how the arithmetic operation addition (+) and modulus (**%**) are used in the repeat body.

```
/*Design a five bit counter with a control signal advance.
If advance is high, counter is increased by one.*/

FIELD count[out4..0]
SEQUENCE count {
$REPEAT i = [0..31]
    PRESENT S{i}
IF advance & !reset NEXT
S{(i+1)%(32)};
    IF reset NEXT S{0};
    DEFAULT NEXT S{i};
$REPEND
}
```

$REPEND

This command ends a repeat body that was started with **$REPEAT**. The format is as follows:

```
$REPEND
```

The statements following the **$REPEND** command are compiled in the same way as the statements preceding the **$REPEAT** command. For each **$REPEAT** command, an associated **$REPEND** command must be used.

$MACRO

This command creates user-defined macros. The format is as follows:

```
$MACRO name argument1 argument2...argumentn
    macro function body
$MEND
```

The macro function body will not be compiled until the macro name is called. The function is called by stating function name and passing the parameters to the function.

Like the **$REPEAT** command, the arithmetic operation can be used inside the macro function body and must be enclosed in braces.

The following example illustrates how to use the **$MACRO** command.

Use the **$MACRO** command to define a decoder function with an arbitrary number of bits. This example places the macro definition and call in the same file.

```
$MACRO decoder bits  MY_X  MY_Y MY_enable;
    FIELD select = [MY_Y{bits-1}..0];
    $REPEAT i = [0..{2**(bits-1)}]
            !MY_X{i} = select:'h'{i} & MY_enable;
    $REPEND
$MEND
…/* Other statements */
decoder(3, out, in, enable);  /*macro function call*/
```

Calling function decoder will create the following statements by macro expansion.

```
FIELD sel = [in2..0];
    !out0 = sel:'h'0 & enable;
    !out1 = sel:'h'1 & enable;
    !out2 = sel:'h'2 & enable;
    !out3 = sel:'h'3 & enable;
    !out4 = sel:'h'4 & enable;
    !out5 = sel:'h'5 & enable;
    !out6 = sel:'h'6 & enable;
    !out7 = sel:'h'7 & enable;
```

When macros are called, the keyword **NC** is used to represent no connection. Because NC is a keyword, the letters NC should not be used as a variable elsewhere in CUPL.

A macro expansion file can be created by using the **-e** flag when compiling the PLD file. CUPL will create an expanded macro file with the same name as the PLD file, with the extension "**.mx**".

The macro definition can be stored in a separate file with a "**.m**" extension. Using the **$INCLUDE** command, specify the file. All the macro functions in that file will then be accessible. The following example shows the macro definition and calling statement stored in different files.

The macro definition of decoder is stored in the file "macrolib.m**"**

```
$INCLUDE macrolib.m /*specify the macro library */
…/* other statements */
decoder(4, out, in enable);
…/* other statements */
```

More examples can be found in the example directory.

$MEND

This command ends a macro function body started with **$MACRO**. The format is as follows:

$MEND

The statements following the **$MEND** command are compiled in the same way as the statements preceding the **$MACRO** command. For each **$MACRO** command, an associated **$MEND** command.must be used.

Language Syntax

This section describes the CUPL language syntax. It explains how to use logic equations, truth tables, state machine syntax, condition syntax and user-defined functions to create a PLD design.

Logical Operators

CUPL supports the four standard logical operators used for boolean expressions. Table 6-8 lists these operators and their order of precedence, from highest to lowest.

Table 6-8. Precedence of Logical Operators

Operator		Example	Description	Precedence
!	!A	NOT	1	
&	A & B	AND	2	
#	A # B	OR	3	
$	A $ B	XOR	4	

The truth tables in Figure 6-9 list the Boolean Logic rules for each operator.

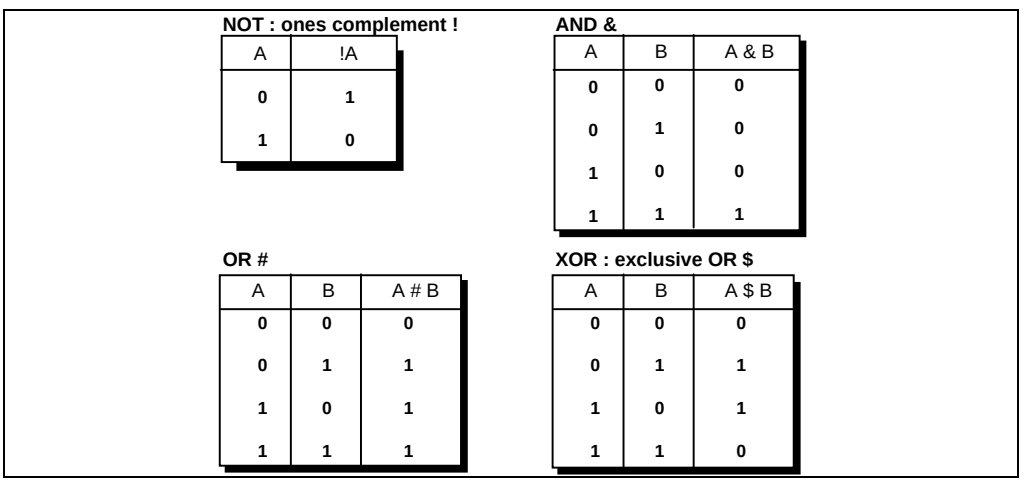


Figure 6-8 Truth Tables

Arithmetic Operators

CUPL supports six standard arithmetic operators used for arithmetic expressions. The arithmetic expressions can only be used in the **$REPEAT** and **$MACRO** commands. Arithmetic expressions must appear in braces **{ }**. Table 6-9 lists these operators and their order of precedence, from highest to lowest.

Table 6-9 Precedence of Arithmetic Operators

Operator	Example		Description	Precedence
**	2**3	Exponentiation		1
*	2*i	Multiplication	2	
/	4/2	Division	2	
%	9%8	Modulus	2	
+	2+4	Addition	3	
-	5-i	Subtraction	3	

CUPL supports one arithmetic function used for arithmetic expressions. The arithmetic expressions can only be used in the **$REPEAT** and **$MACRO** commands. Table 6-10 lists the function.

Table 6-10 Arithmetic Function

Function	Base
LOG2	Binary
LOG8	Octal
LOG16	Hexadecimal
LOG	Decimal

The LOG function returns an integer value. For example:

```
LOG2(32) = 5                    <==>   2**5 = 32
LOG2(33) = ceil(5.0444) = 6     <==>   2**6 = 64
```

Ceil(x) returns the smallest integer not less than x.

Extensions

Extensions can be added to variable names to indicate specific functions associated with the major nodes inside a programmable device, including such capabilities as flip-flop description and programmable three-state enables. Table 6-11 lists the extensions that are supported by CUPL and on which side of the equal sign (=) they are used. The compiler checks the usage of the extension to determine whether it is valid for the specified device and whether its usage conflicts with some other extension used.

Table 6-11 Extensions

Extension Used	Side	Description
.AP	L	Asynchronous preset of flip-flop
.AR	L	Asynchronous reset of flip-flop
.APMUX	L	Asynchronous preset multiplexer selection
.ARMUX	L	Asynchronous reset multiplexer selection
.BYP	L	Programmable register bypass
.CA	L	Complement array
.CE	L	CE input of enabled D-CE type flip-flop
.CK	L	Programmable clock of flip-flop
.CKMUX	L	Clock multiplexer selection
.D	L	D nput of D-type flip-flop
.DFB	R	D registered feedback path selection
.DQ	R	Q output of D-type flip-flop
.IMUX	L	Input multiplexer selection of two pins
.INT	R	Internal feedback path for registered macrocell
.IO	R	Pin feedback path selection
.IOAR	L	Asynchronous reset for pin feedback register
.IOAP	L	Asynchronous preset for pin feedback register
.IOCK	L	Clock for pin feedback register
.IOD	R	Pin feedback path through D register
.IOL	R	Pin feedback path through latch
.IOSP	L	Synchronous preset for pin feedback register

.IOSR	L	Synchronous reset for pin feedback register
.J	L	J input of JK-type output flip-flop
.K	L	K input of JK-type output flip-flop
.L	L	D input of transparent latch
.LE	L	Programmable latch enable
.LEMUX	L	Latch enable multiplexer selection
.LFB	R	Latched feedback path selection
.LQ	R	Q output of transparent input latch
.OBS	L	Programmable observability of buried nodes
.OE	L	Programmable output enable
.OEMUX	L	Tri-state multiplexer selection
.PR	L	Programmable preload
.R	L	R input of SR-type output flip-flop
.S	L	S input of SR-type output flip-flop
.SP	L	Synchronous preset of flip-flop
.SR	L	Synchronous reset of flip-flop
.T	L	T input of toggle output flip-flop
.TEC	L	Technology-dependent fuse selection
.TFB	R	T registered feedback path selection
.T1	L	T1 input of 2-T flip-flop
.T2	L	T2 input of 2-T flip-flop

Each extension provides access to a specific function. For example, to specify an equation for output enable (on a device that has the capability) use the .OE extension. The equation will look as follows:

```
PIN 2 = A;
PIN 3 = B;
PIN 4 = C;
PIN 15 = VARNAME;
VARNAME.OE = A&B;
```

Note that the compiler supports only the flip-flop capabilities that are physically implemented in the device. For example, the compiler does not attempt to emulate a JK-type flip-flop in a device that only has D-type registers. Any attempt to use capabilities not present in a device will cause the compiler to report an error.

For those devices containing bi-directional I/O pins with programmable output enables, CUPL automatically generates the output enable expression according to the usage of the pin. If the variable name is used on the left side of an equation, the pin is assumed to be an output and is assigned binary value 1; that is, the output enable expression is defaulted to the following:

```
PIN_NAME.OE='b'1; Tri-state buffer always ON
```

Those pins that are used only as inputs (that is, the variable name appears only on the right side of an equation) are assigned binary value 0; the output enable expression is defaulted to the following:

```
PIN_NAME.OE = 'b'0;          Tri-state buffer always OFF
```

When the I/O pin is to be used as both an input and output, any new output enable expression that the user specifies overrides the default to enable the tri-state buffer at the desired time.

When using a JK or SR-type flip-flop, an equation must be written for both the J and K (or S and R) inputs. If the design does not require an equation for one of the inputs, use the following construct to turn off the input:

```
COUNT0.J='b'0 ;      /* J input not used          */
```

Control functions such as asynchronous resets and presets are commonly connected to a group (or all) of the registers in a device. When an equation is written for one of these control functions, it is actually being written for all of the registers in the group. For documentation purposes, CUPL checks for the presence of such an equation for each register in the group and generates a warning message for any member of the group that does not have an identical equation. If all the control functions for a given group are defined with different equations, the compiler will generate an error since it cannot decide which equation is the correct one. Remember that this is a device specific issue and it is a good idea to understand the capability of the device being used.

Figure 6-9 shows the use of extensions. Note that this figure does not represent an actual circuit, but shows how to use extensions to write equations for different functions in a circuit.

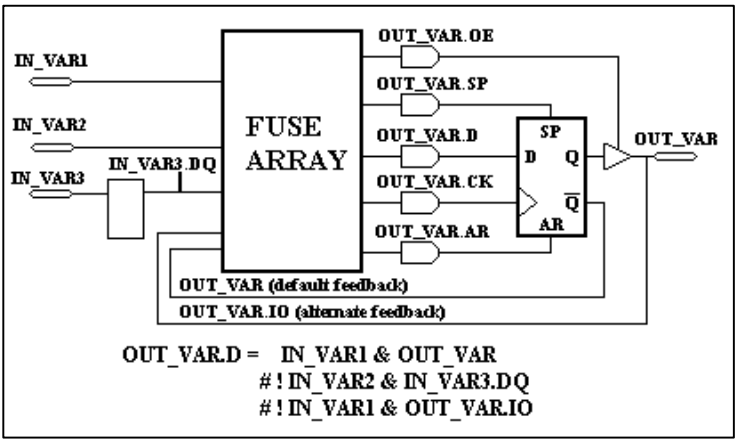


Figure 6-9. Circuit Illustrating Extensions

The figure shows an equation with a **.D** extension that has been written for the output to specify it as a registered output. Note that when feedback (**OUT_VAR**) is used in an equation, it does not have an extension.

The **.DQ** extension is used for input pins only.

Additional equations can be written to specify other types of controls and control points. For example, an equation for the output enable can be written as follows:

```
OUT_VAR.OE = IN_VAR1 # IN_VAR2
```

Feedback Extensions Usage

Certain devices can program the feedback path. For example, the EP300 contains a multiplexer for each output that allows the feedback path to be selected as internal, registered, or pin feedback.

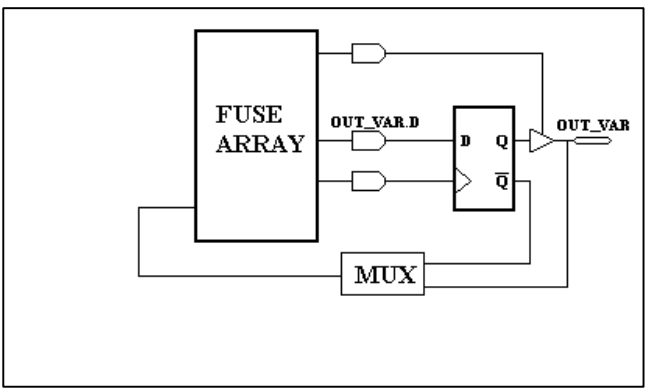


Figure 6-10 shows the EP300 programmable feedback capability.

Figure 6-10. Programmable Feedback

CUPL automatically chooses a default feedback path according to the usage of the output. For example, if the output is used as a registered output, then the default feedback path will be registered, as in Figure 6-11. This default can be overridden by adding an extension to the feedback variables. For example, by adding the **.IO** extension to the feedback variables of a registered output, CUPL will select the pin feedback path.

Figure 6-11 shows a registered output with pin feedback.

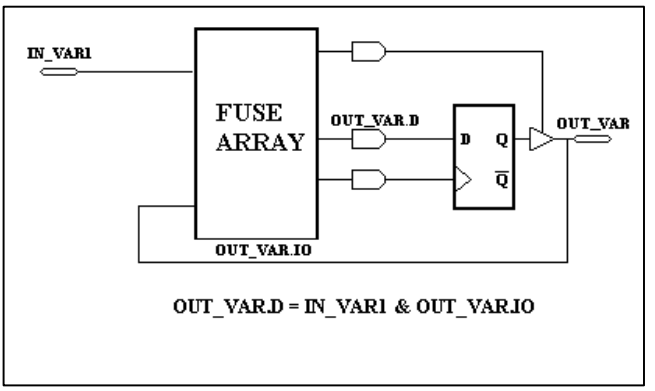


Figure 6-11. Programmable Pin (I/O) Feedback

Figure 6-12 shows a combinatorial output with registered feedback.

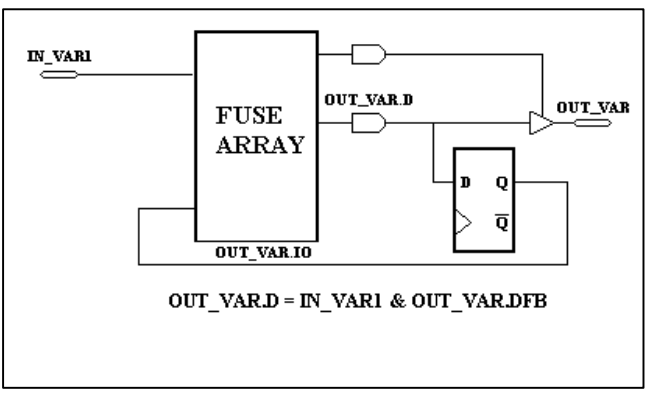


Figure 6-12. Programmable Registered Feedback

Figure 6-13 shows a combinatorial output with internal feedback.

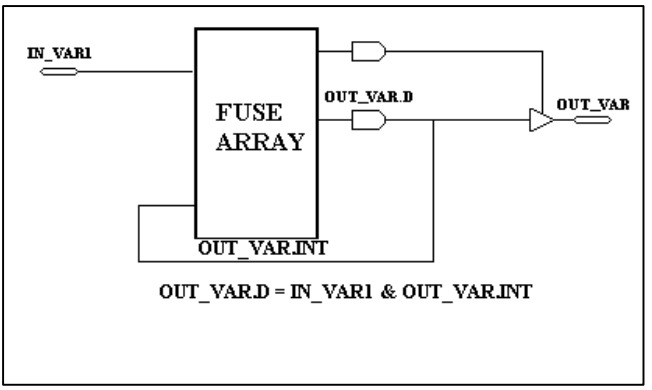


Figure 6-13. Programmable Internal Feedback

Multiplexer Extension Usage

Certain devices allow selection between programmable and common control functions. For example, for each output, the P29MA16 contains multiplexers for selecting between common and product term clocks and output enables.

Figure 6-14 shows the P29MA16 programmable clock and output enable capability.

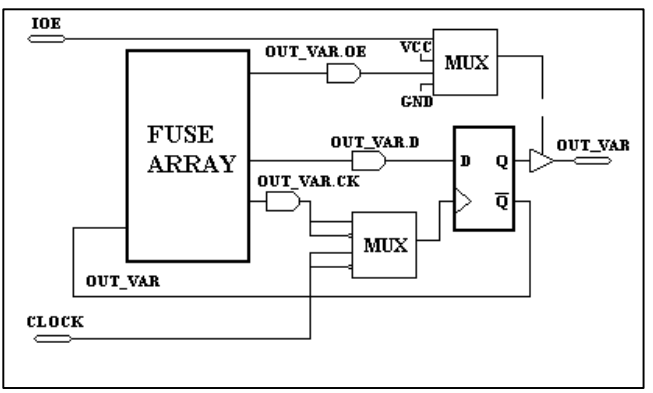


Figure 6-14. Output with Output Enable and Clock Multiplexers

If expressions are written for the **.OE** and **.CK** extensions, the multiplexer outputs are selected as product term output enable and clock, respectively. Otherwise, if expressions are written for the **.OEMUX** and **.CKMUX** extensions, the multiplexer outputs are selected as common output enable and clock, respectively.

Expressions written for the **.OEMUX** and **.CKMUX** extensions can have only one variable and be operated on only by the negation operator, **!**. This is because their inputs are not from the fuse array, but from a common source, such as a clock pin. This is in contrast with expressions written for the **.OE** and **.CK** extensions, which take their inputs from the fuse array.

Figure 6-15 shows a registered output with the output enable multiplexer output selected as Vcc, output enable always enabled, and the clock multiplexer output selected as the common clock pin inverted, negative-edge clock.

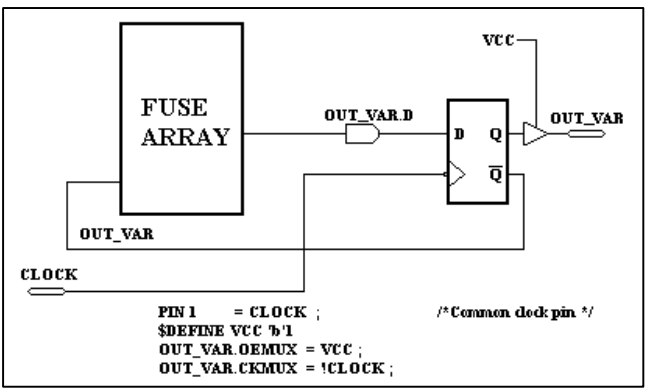


Figure 6-15. Output with Output Enable and Clock Multiplexers Selected

Expressions for the **.OE** and **.OEMUX** extensions are mutually exclusive; that is, only one may be written for each output. Likewise, expressions for the **.CK** and **.CKMUX** extensions are mutually exclusive.

Extension Usage Diagrams

This section contains diagrams and explanations for all the variable extensions.

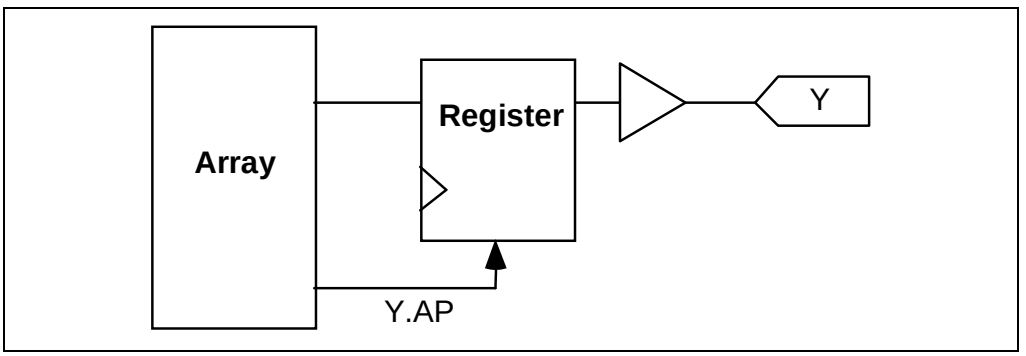


Figure 6-16. .AP Extension

The .AP extension is used to set the Asynchronous Preset of a register to an expression. For example, the equation "Y.AP = A & B;" causes the register to be asynchronously preset when A and B are logically true.

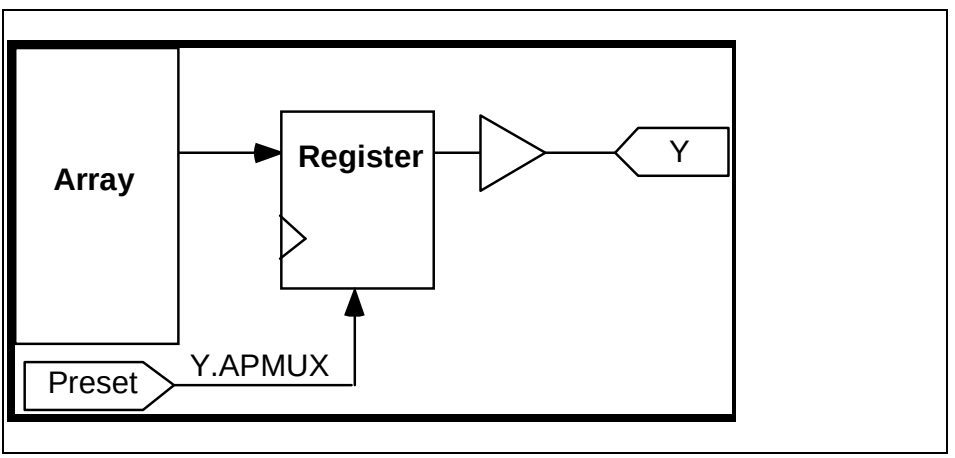


Figure 6-17. .APMUX Extension

Some devices have a multiplexer that enables the Asynchronous Preset to be connected to one of a set of pins. The .APMUX extension is used to connect the Asynchronous Preset directly to one of the pins.

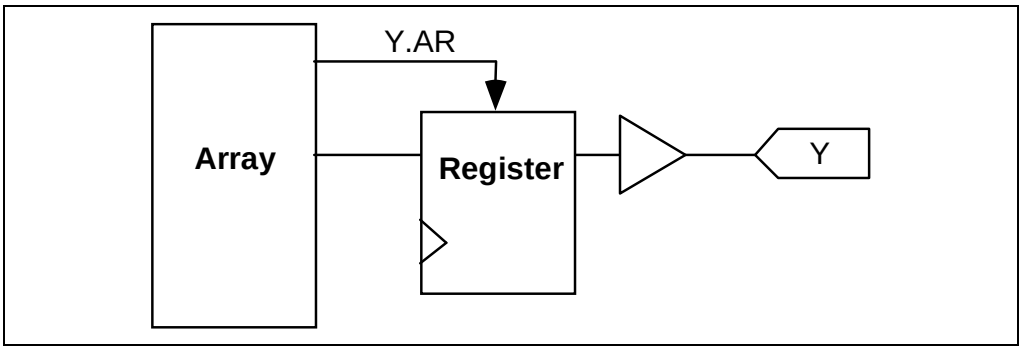


Figure 6-18. .AR Extension

The .AR extension is used to define the expression for Asynchronous Reset for a register. This is used in devices that have one or more product terms connected to the Asynchronous Reset of the register.

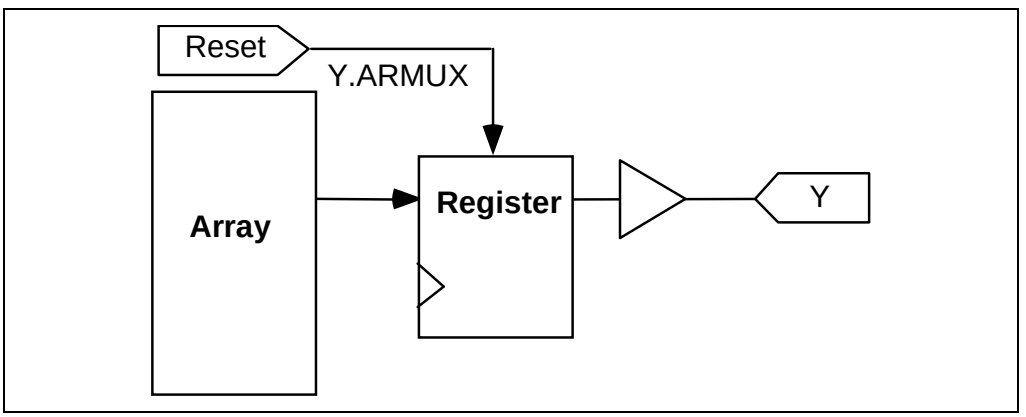


Figure 6-19. .ARMUX Extension

In devices that have a multiplexer for connecting the Asynchronous Reset of a register directly to one or more pins, the .ARMUX extension is used to make the connection. It is possible that a device may have the capability to have Asynchronous Reset connected

either to a pin or to a product term. In this case, the .AR extension is used to select the product term connection, whereas, the .ARMUX extension is used to connect the pin.

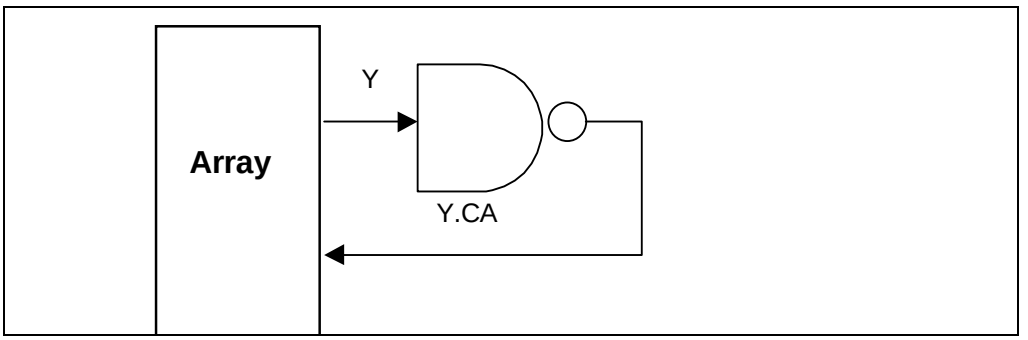


Figure 6-20 CA Extension

The .CA extension is used in a few special cases where devices have complementa array nodes. Devices that have this capability are the F501 and F502.

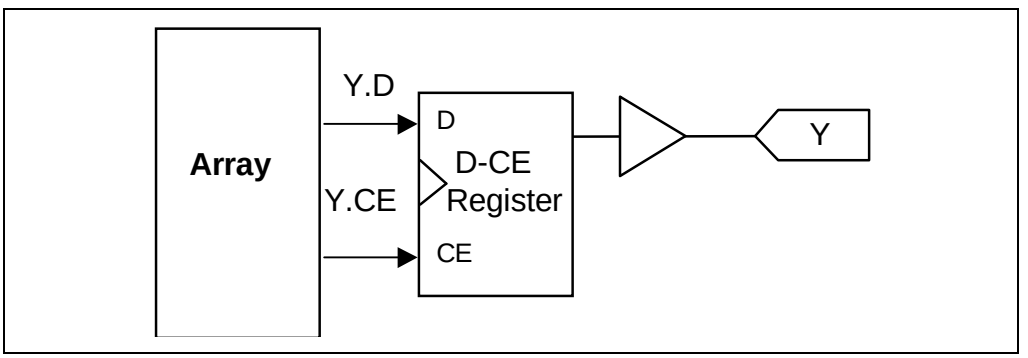


Figure 6-21. .CE Extension

The .CE extension is used for D-CE registers. It serves to specify the input to the CE of the register. In devices that have D-CE registers, and the CE terms are not used, they must be set to binary 1 so that the registers behave the same as D registers. **Failure to enable the CE terms will result in D registers that never change state**.

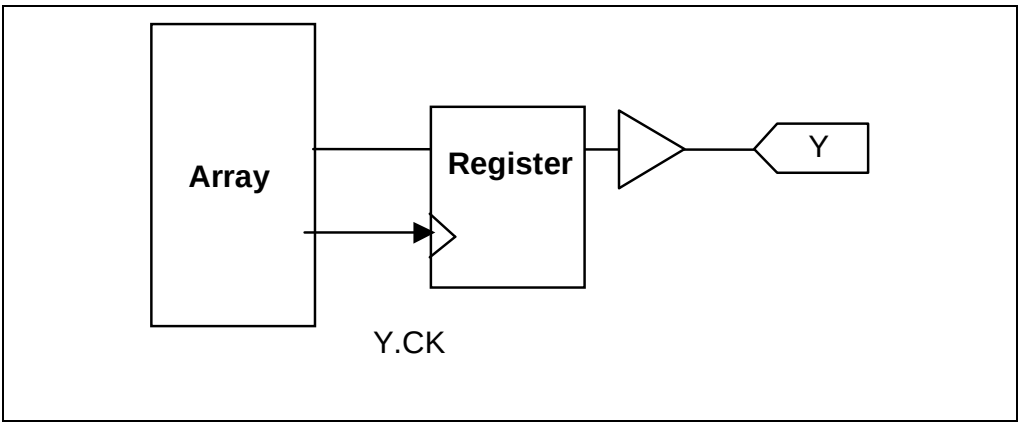


Figure 6-22. .CK Extension

The .CK extension is used to select a **product term driven clock**. Some devices have the capability to connect the clock for a register to one or more pins or to a product term. The .CK extension will select the product term. If the device has multiple **clock pins** us the CKMUX extension.

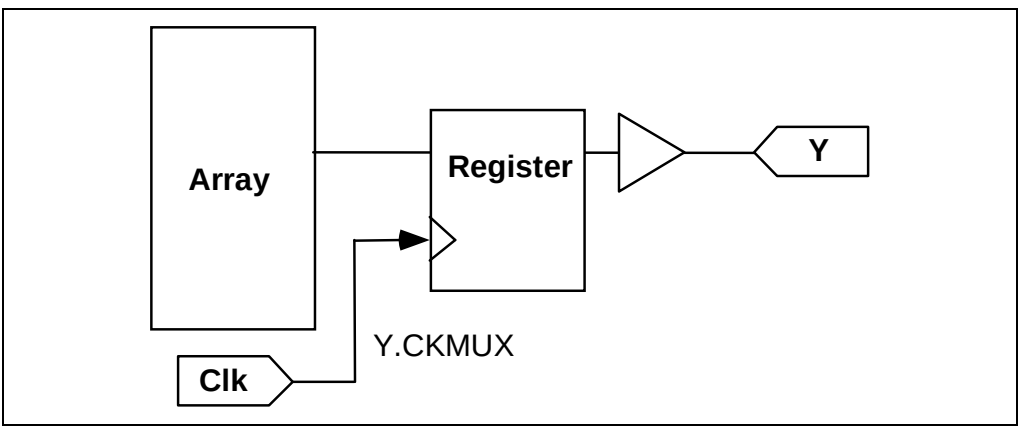


Figure 6-23. .CKMUX Extension

The .CKMUX extension is used to connect the clock input of a register to one of a set of pins. This is needed because some devices have a multiplexer for connecting the clock to one of a set of pins. This does not mean that the clock may be connected to any pin.

Typically, the multiplexer will allow the clock to be connected to one of two pins. Some devices have a multiplexer for connecting to one of four pins.

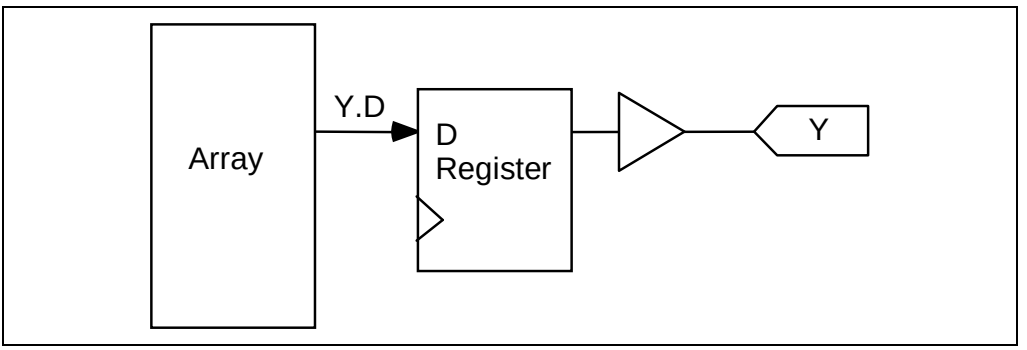


Figure 6-24. .D Extension

The .D extension is used to specify the D input to a D register. The use of the .D register actually causes the compiler to configure programmable macrocells as D registers. For outputs that have only D registered output, the .D extension must be used. If the .D extension is used for an output that does not have true D registers, the compiler will generate an error.

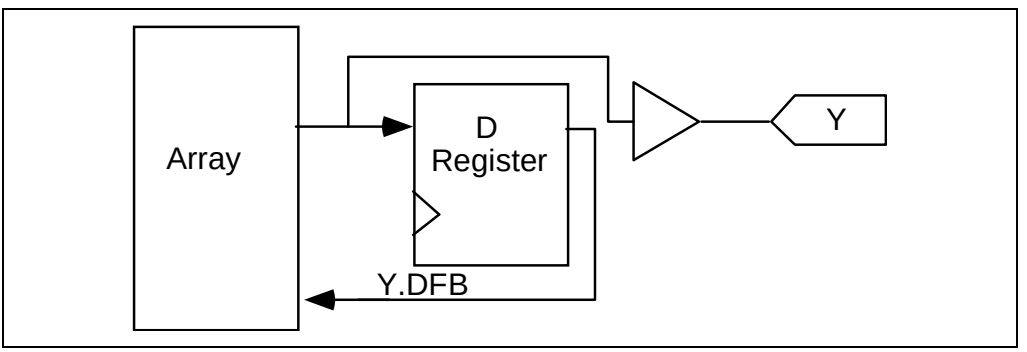


Figure 6-25. .DFB Extension

The .DFB extension is used in special cases where a programmable output macrocell is configured as combinatorial but the D register still remains connected to the output. The .DFB extension provides a means to use the feedback from the register. Under normal conditions, when an output is configured as registered, the feedback from the register is selected by not specifying an extension.

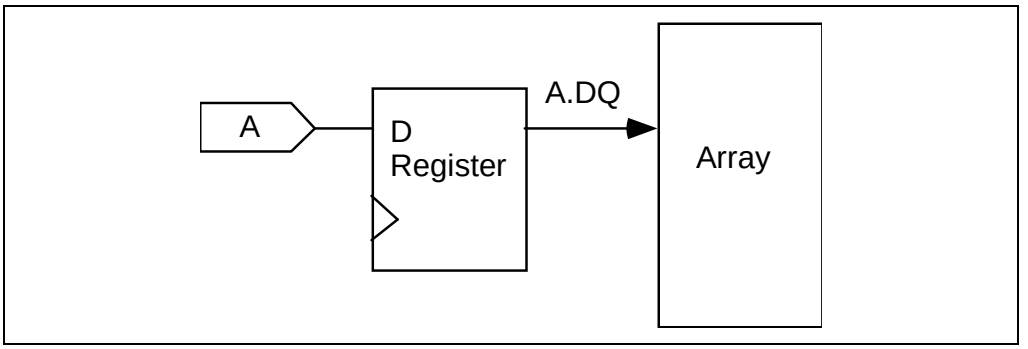


Figure 6-26. .DQ Extension

The .DQ extension is used to specify an input D register. Use of the .DQ extension actually configures the input as registered. The .DQ extension is not used to specify Q output from an output D register.

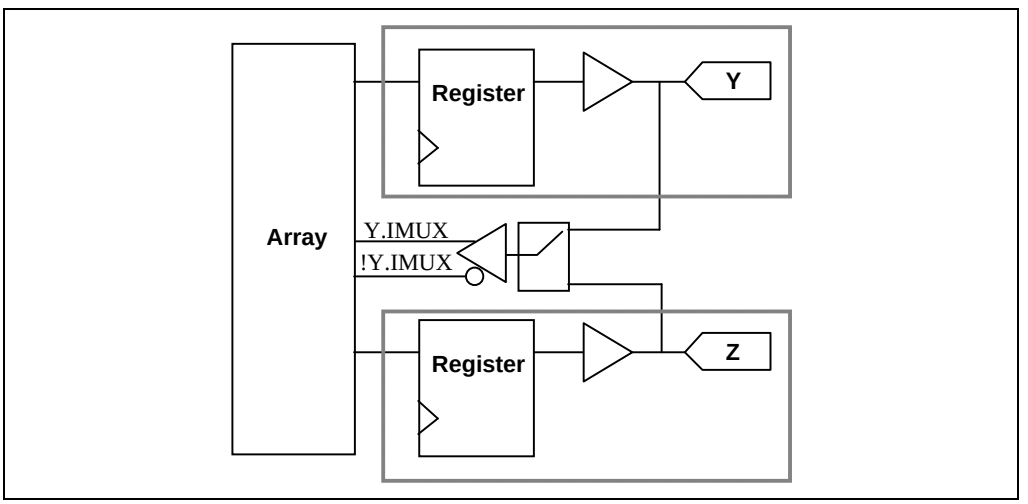


Figure 6-27. .IMUX Extension

The .IMUX extension is an advanced extension which is used to select a feedback path. This is used in devices that have pin feedback from two I/O pins connected to a multiplexer. Only one of the pins may use the feedback path.

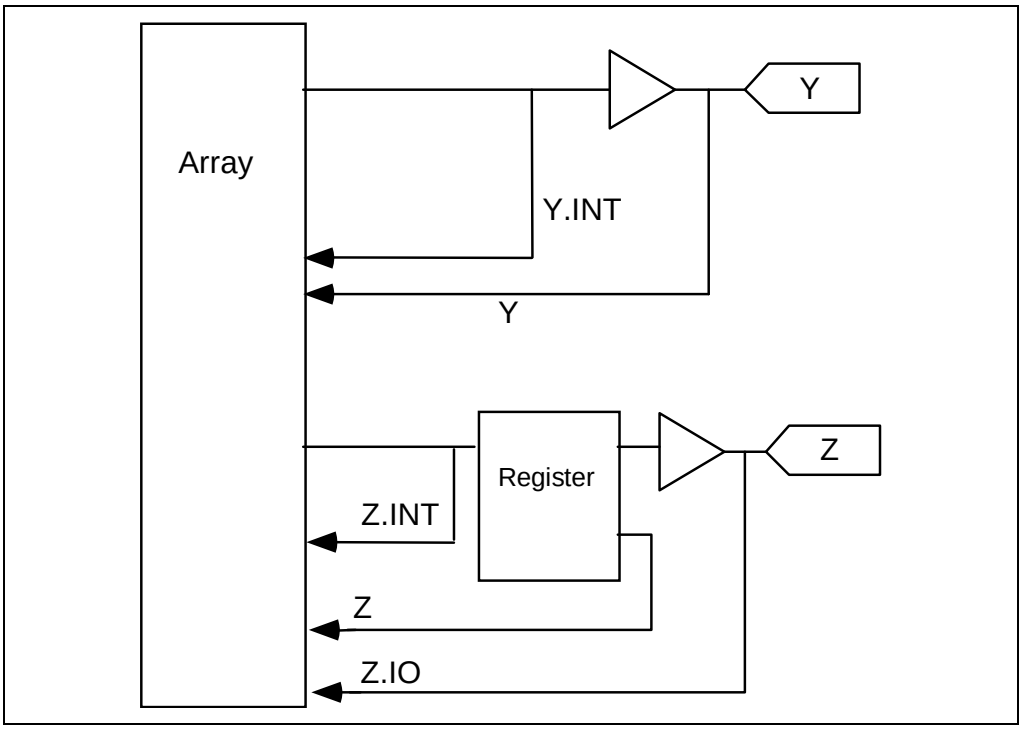


Figure 6-28. .INT Extension

The .INT extension is used for selecting an internal feedback path. This could be used for combinatorial or registered output. The .INT extension provides combinatorial feedback.

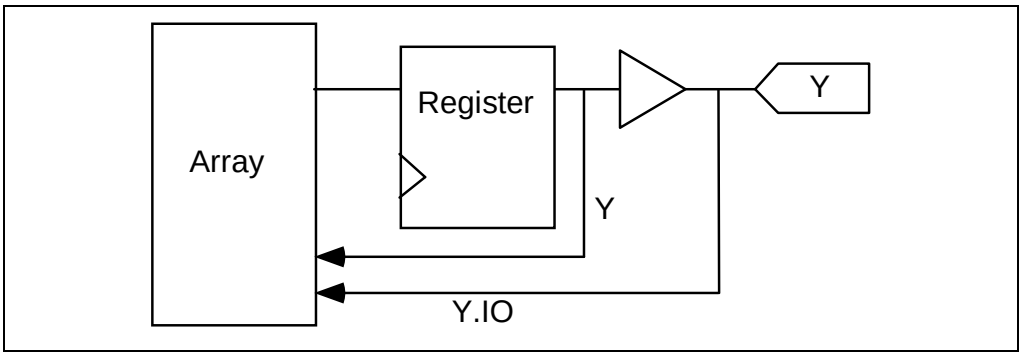


Figure 6-29. .IO Extension

The .IO extension is used to select pin feedback when the macrocell is configured as registered.

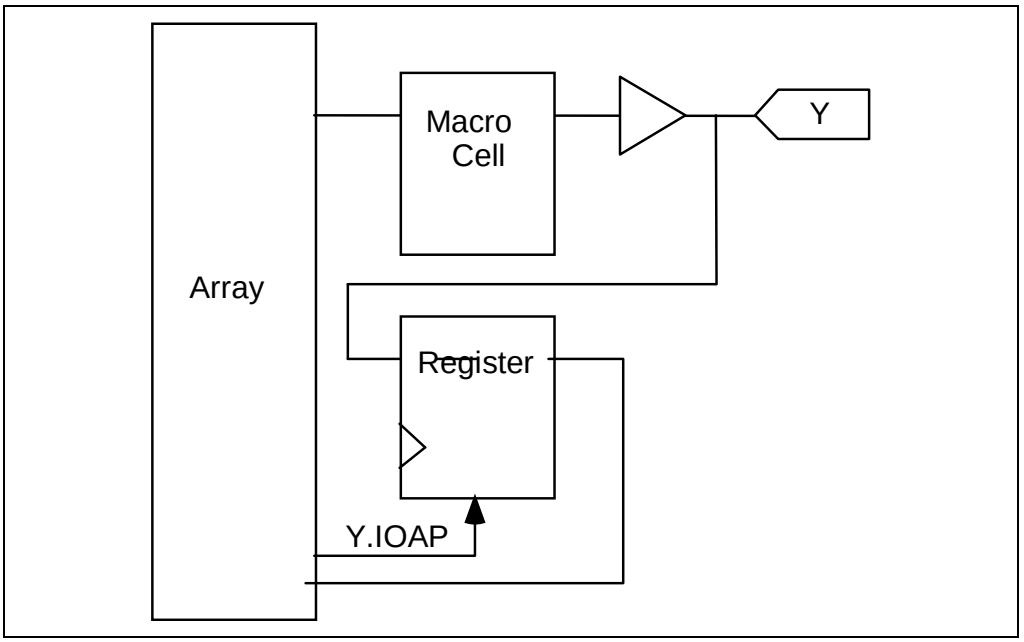


Figure 6-30. .IOAP Extension

The .IOAP extension is used to specify the expression for Asynchronous Preset in cases where there is registered pin feedback from an output macrocell.

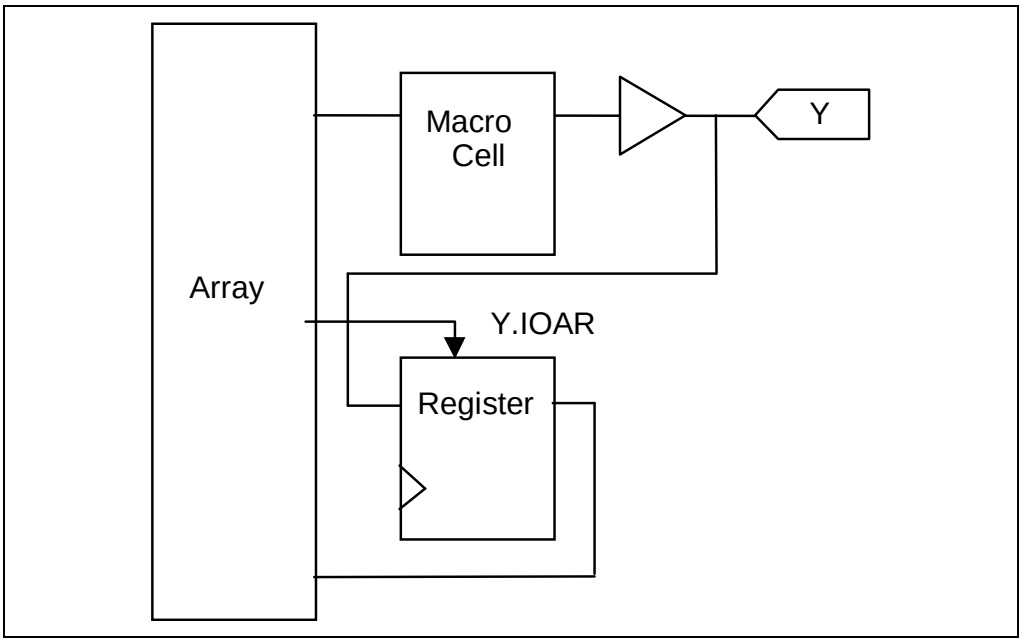


Figure 6-31. .IOAR Extension

The .IOAR extension is used to specify the expression for Asynchronous Reset.in cases where there is registered pin feedback from an output macrocell.

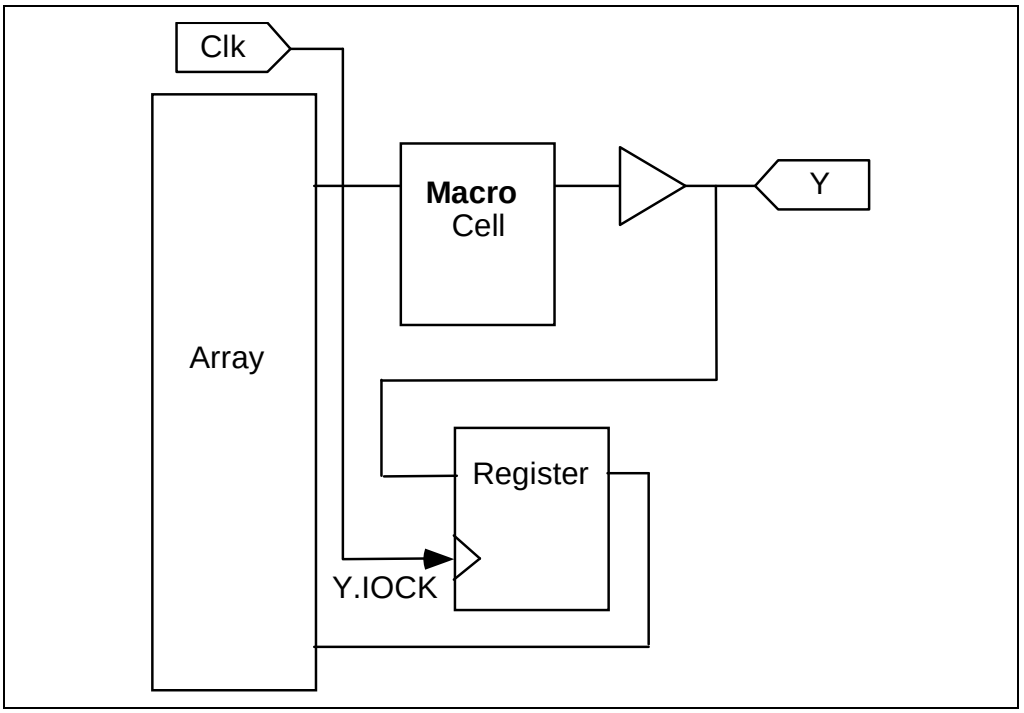


Figure 6-32. .IOCK Extension

The .IOCK extension is used to specify a clock expression for a registered pin feedback that is connected to an output macrocell.

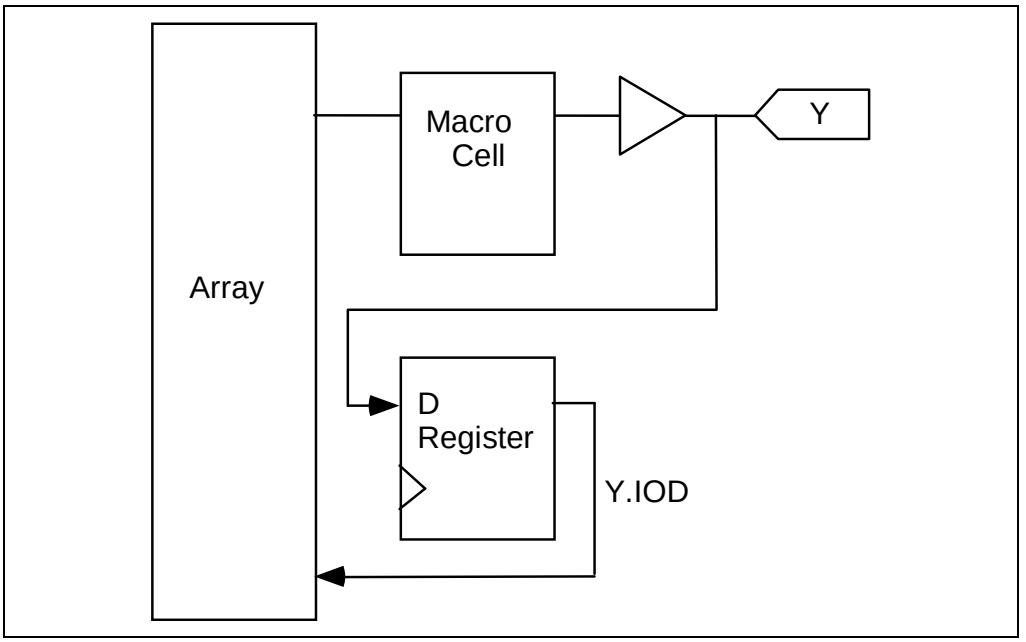


Figure 6-33. .IOD Extension

The .IOD extension is used to specify feedback from a register that is connected to an output macrocell by the pin feedback path.

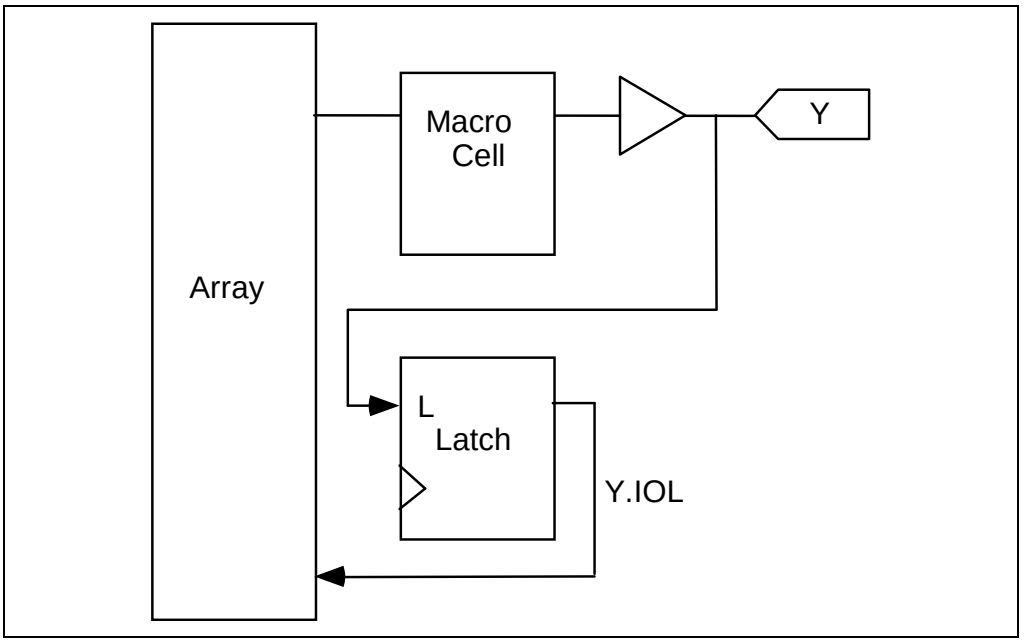


Figure 6-34. .IOL Extension

The .IOL extension is used to specify feedback from a buried latch that is connected to an output macrocell by the pin feedback path.

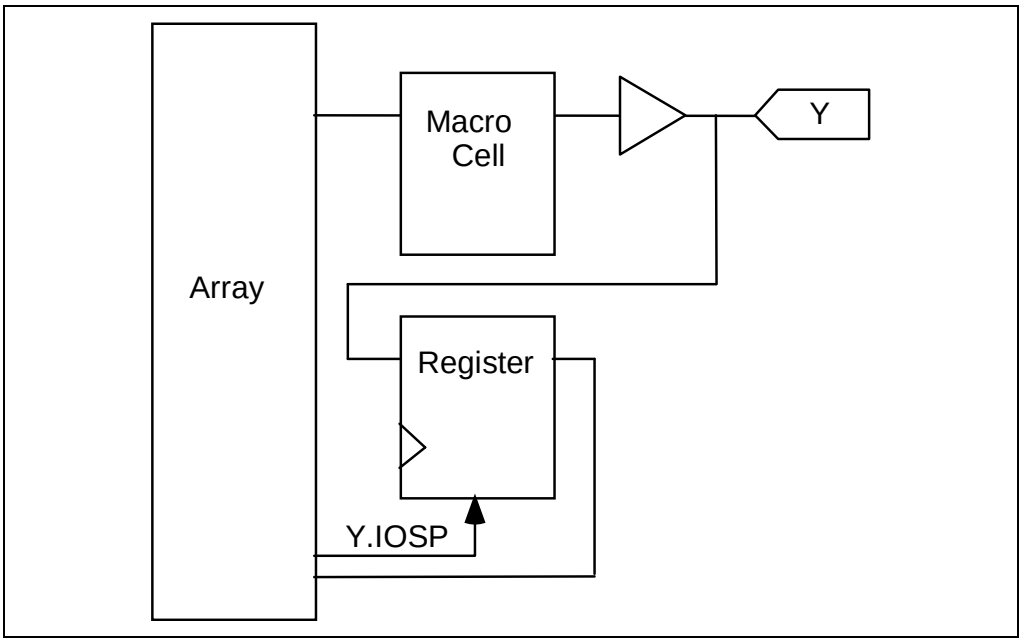


Figure 6-35. .IOSP Extension

The .IOSP extension is used to specify the expression for Synchronous Preset in cases where there is registered pin feedback from an output macrocell.

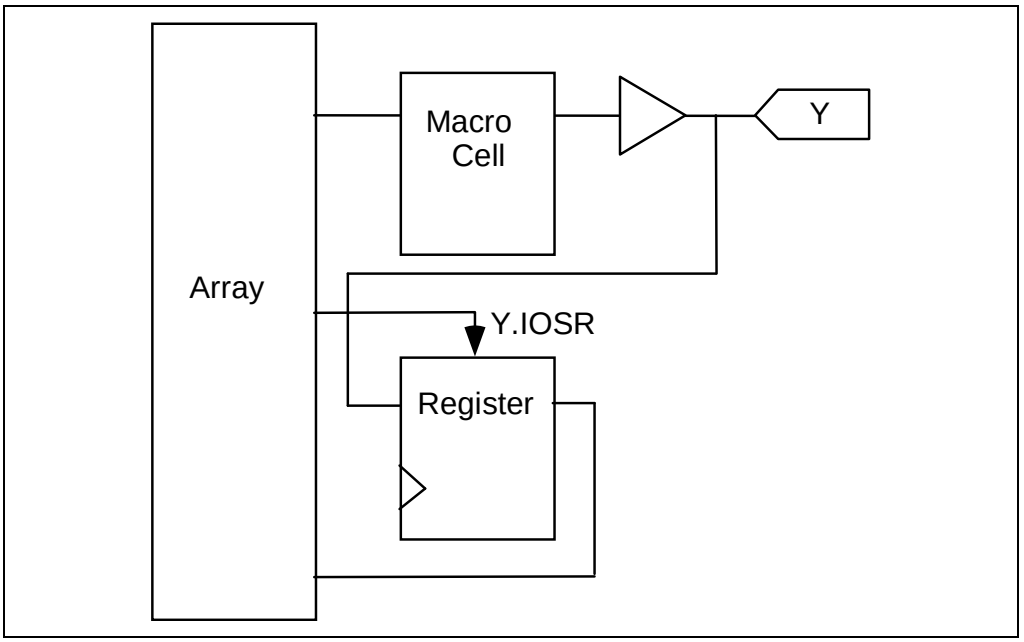


Figure 6-36. .IOSR Extension

The .IOSR extension is used to specify the expression for Synchronous Reset in cases where there is registered pin feedback from an output macrocell.

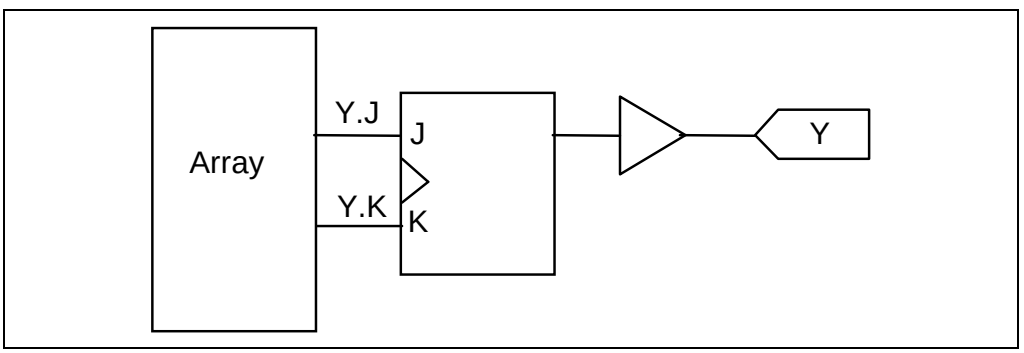


Figure 6-37. .J and .K Extension

The .J and .K extensions are used to specify J and K input to a JK register. The use of the .J and the .K extensions actually cause the compiler to configure the output as JK, if the macrocell is programmable. Equations for both J and K must be specified. If one of the inputs is not used, it must be set to binary 0 to disable it.

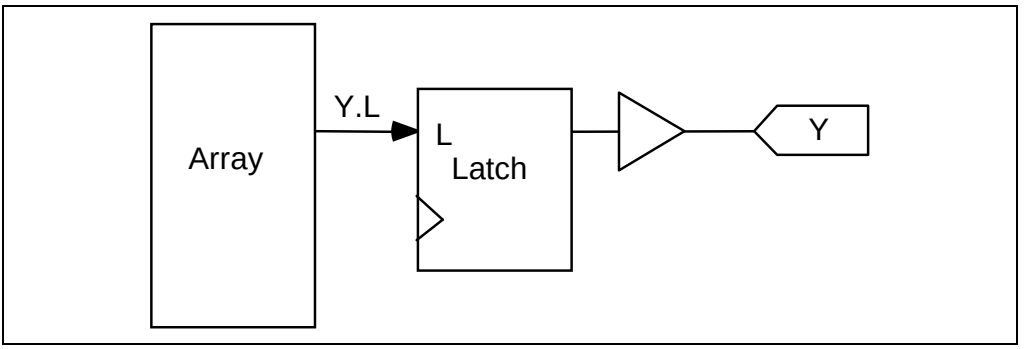


Figure 6-38. .L Extension

The .L extension is used to specify input into a Latch. In devices with programmable macrocells, the use of the .L extension causes the compiler to configure the macrocell as a latched output.

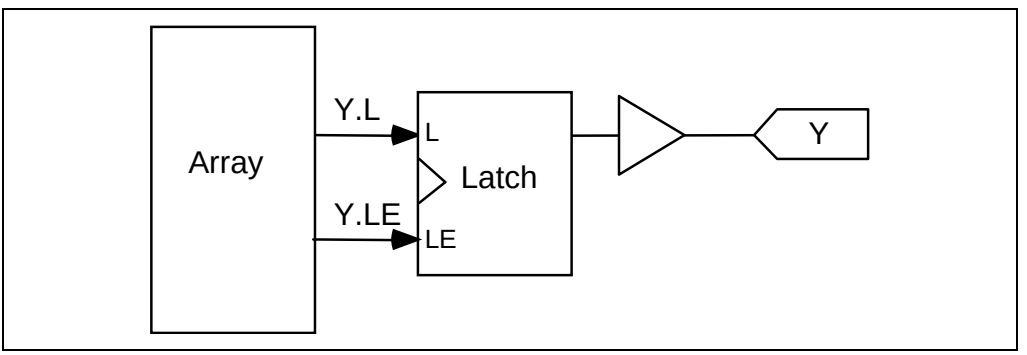


Figure 6-39. .LE Extension

The .LE extension is used to specify the latch enable equation for a latch. The .LE extension causes a product term to be connected to the latch enable.

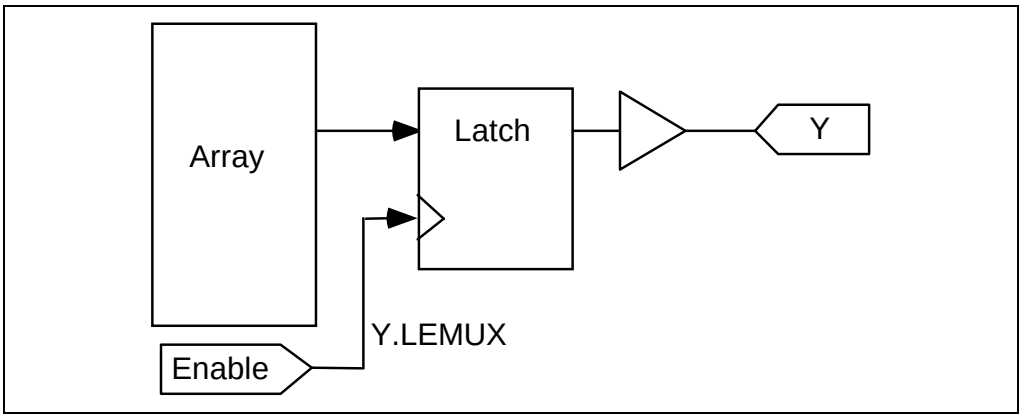


Figure 6-40. .LEMUX Extension

The .LEMUX extension is used to specify a pin connection for the latch enable.

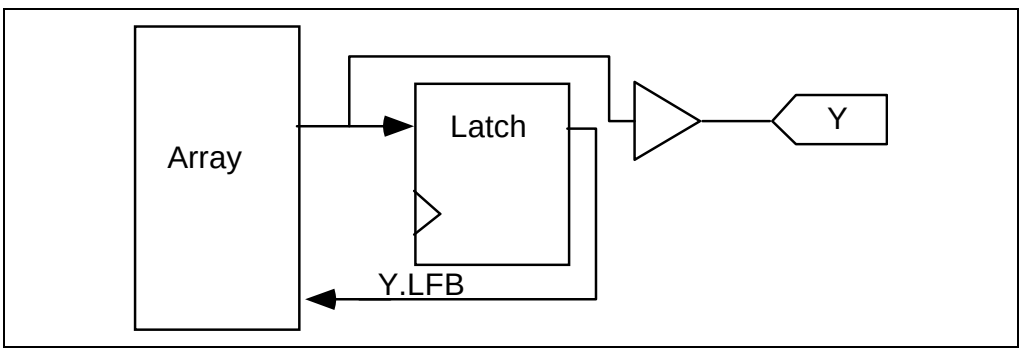


Figure 6-41. .LFB Extension

The .LFB extension is used in special cases where a programmable output macrocell is configured as combinatorial but the latch still remains connected to the output. The .LFB extension provides a means to use the feedback from the latch. Under normal conditions, when an output is configured as latched, the feedback from the latch is selected by using no extension.

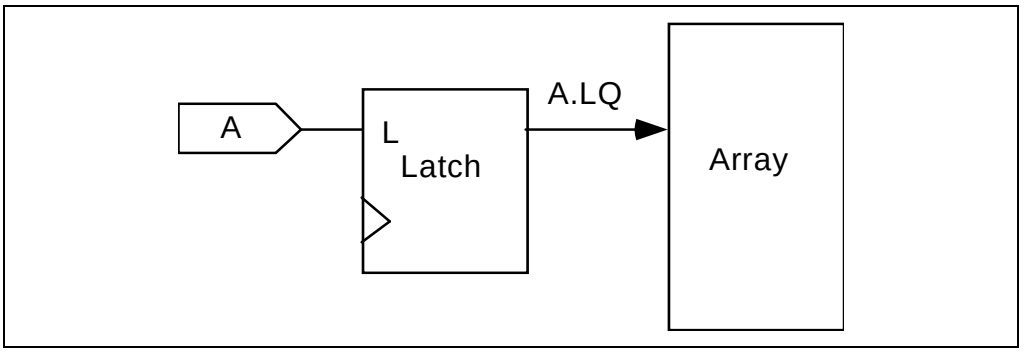


Figure 6-42. .LQ Extension

The .LQ extension is used to specify an input latch. Use of the .LQ extension actually configures the input as latched. The .LQ extension is not used to specify Q output from an output latch.

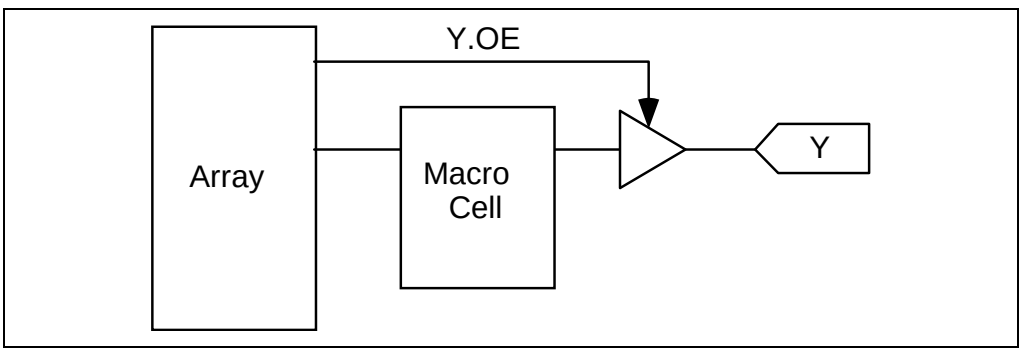


Figure 6-43. .OE Extension

The .OE extension is used to specify a product term driven output enable signal.

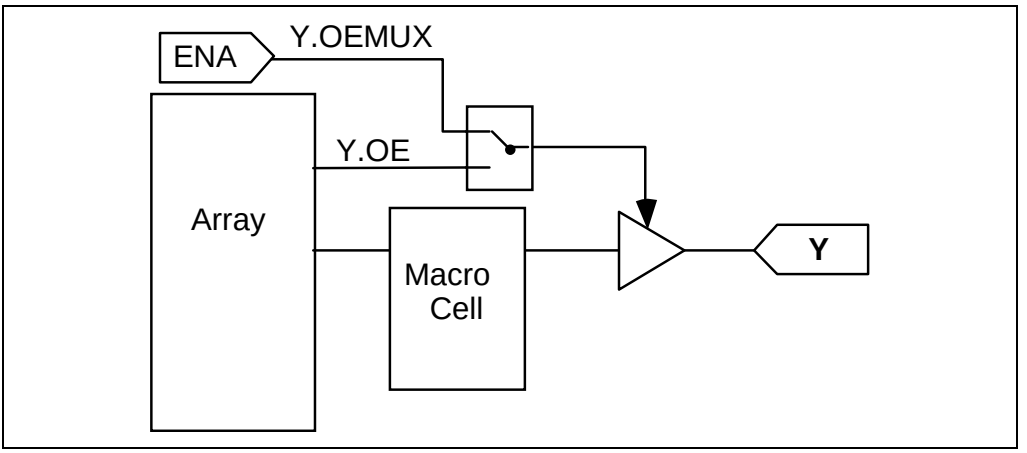


Figure 6-44. .OEMUX Extension

The .OEMUX extension is used to connect the output enable to one of a set of pins. This is needed because some devices have a multiplexer for connecting the output enable to one of a set of pins. This does not mean that the output enable may be connected to any pin. Typically, the multiplexer will allow the output enable to be connected to one of two pins. Some devices have a multiplexer for connecting to one of four pins.

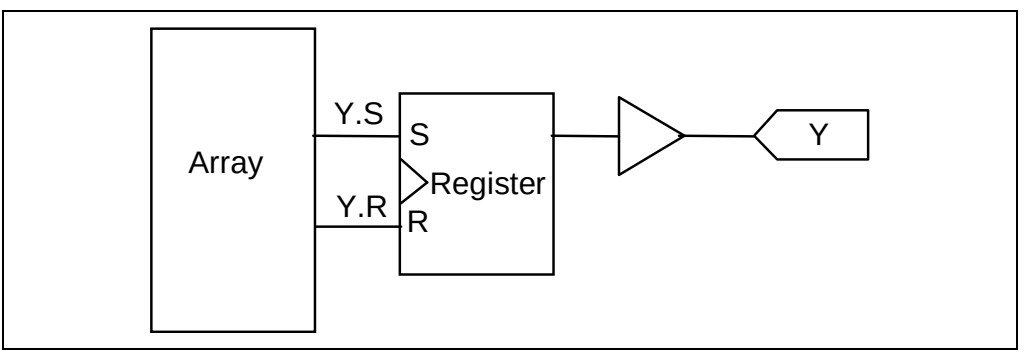


Figure 6-45. .S and .R Extension

The .S and .R extensions are used to specify S and R input to a SR register. The use of the .S and the .R extensions actually cause the compiler to configure the output as SR, if the macrocell is programmable. Equations for both S and R must be specified. If one of the inputs is not used, it must be set to binary 0 to disable it.

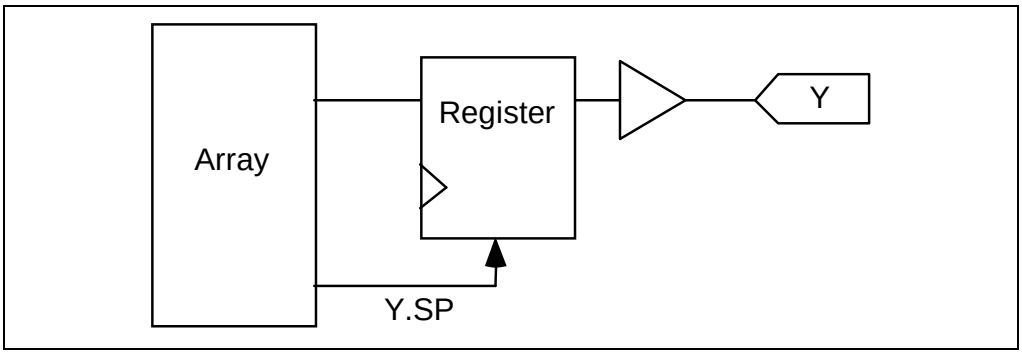


Figure 6-46. .SP Extension

The .SP extension is used to set the Synchronous Preset of a register to an expression. For example, the equation "Y.SP = A & B;" causes the register to be synchronously preset when A and B are logically true.

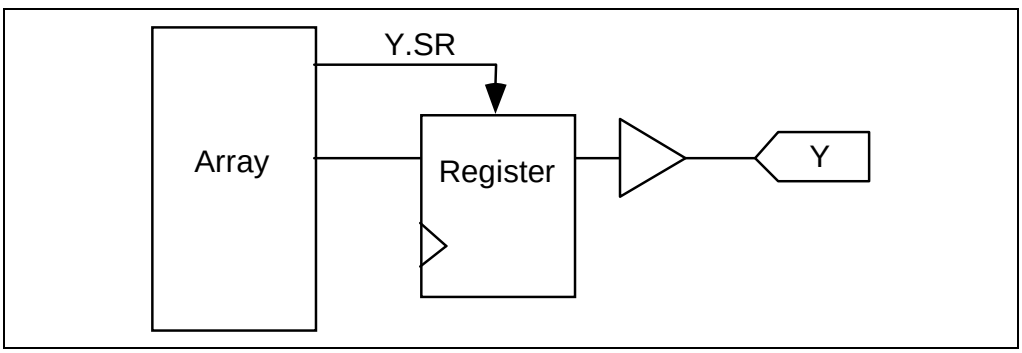


Figure 6-47. .SR Extension

The .SR extension is used to define the expression for Synchronous Reset for a register. This is used in devices that have one or more product terms connected to the Synchronous Reset of the register.

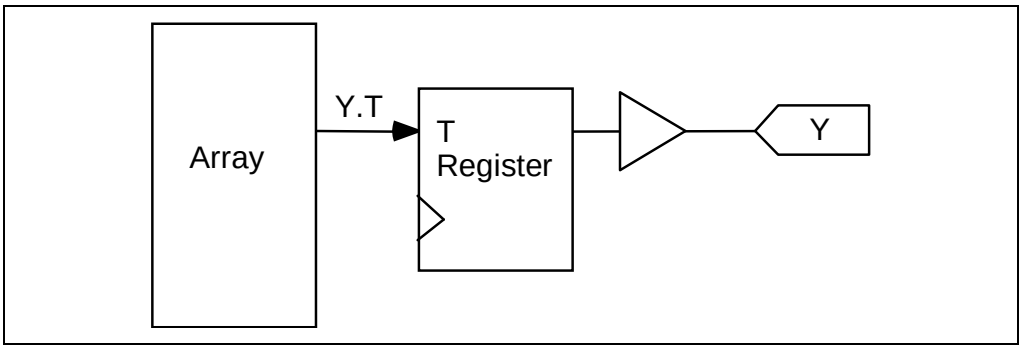


Figure 6-48. .T Extension

The .T extension specifies the T input for a T register. The use of the T extension itself causes the compiler to configure the macrocell as a T register. Special consideration should be given to devices with T registers and programmable polarity before the register. Since T registers toggle when the incoming signal is true, the behavior will be changed when the polarity is changed since the incoming signal is now inverted before reaching the register. It is best to declare pins that will use T registers as active high always.

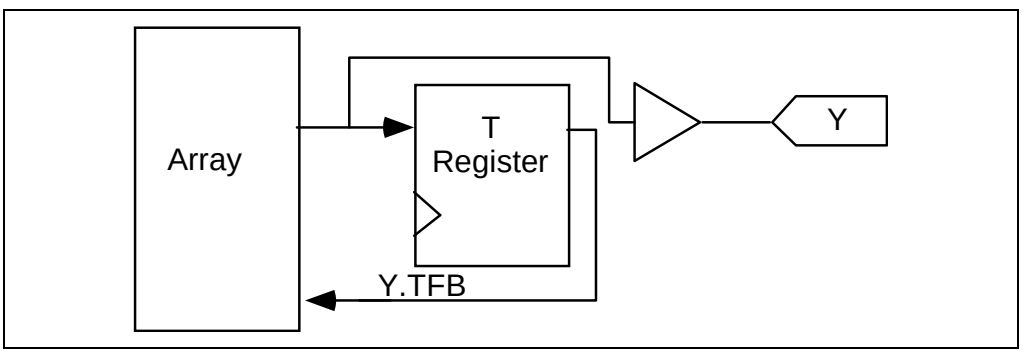


Figure 6-49. .TFB Extension

The .TFB extension is used in special cases where a programmable output macrocell is configured as combinatorial but the T register still remains connected to the output. The .TFB extension provides a means to use the feedback from the register. Under normal conditions, when an output is configured as registered, the feedback from the register is selected by using no extension.

Boolean Logic Review

Table 6-12 lists the rules that the CUPL compiler uses for evaluating logic expressions. These basic rules are listed for reference purposes only.

Table 6-12. Logic Evaluation Rules

Commutative Property:

A & B = B & A
A # B = B # A

Associative Property:

A & (B & C) = (A & B) & C
A # (B # C) = (A # B) # C

Distributive Property:

A & (B # C) = (A & B) # (A & C)
A # (B & C) = (A # B) & (A # C)

Absorptive Property:

A & (A # B) = A
A # (A & B) = A

DeMorgan's Theorem:

!(A & B & C) = !A # !B # !C
!(A # B # C) = !A & !B & !C

XOR Identity:

A $ B = (!A & B) # (A & !B)
!(A $ B) = A $!B = !A $ B = (!A & !B) # (A& B)

Theorems:

A & 0 = 0	A & 1 = A
A # 0 = A	A # 1 = 1
A & A = A	A & !A = 0
A # A = A	A # !A = 1

Expressions

Expressions are combinations of variables and operators that produce a single result when evaluated. An expression may be composed of any number of sub-expressions.

Expressions are evaluated according to the precedence of the particular operators involved. When operators with the same precedence appear in an expression, evaluation order is taken from left to right. Parentheses may be used to change the order of evaluation; the expression within the innermost set of parentheses is evaluated first.

In Table 6-13, note how the order of evaluation and use of parentheses affect the value of the expression.

Table 6-13. Sample Expressions

Expression	Result	Comments
A # B & C	A # B & C	
(A # B) & C	A & C # B & C	Parentheses change order
!A & B	!A&B	
!(A & B)	!A # !B	DeMorgan'sTheorem
A # B & C # D	A # D # B & C	
A # B & (C # D)	A # B&C # B & D	Parentheses change order

Logic Equations

Logic equations are the building blocks of the CUPL language. The form for logic equations is as follows:

[!] var [.ext] = exp ;

where

var is a single variable or a list of indexed or non-indexed variables defined according to the rules for the list notation (see the subtopic, List Notation in this chapter). When a variable list is used, the expression is assigned to each variable in the list.

.ext is an optional extension to assign a function to the major nodes inside a programmable device (see Table 1-11).

exp is an expression; that is, a combination of variables and operators (see "Expressions" in this chapter).

= is the assignment operator; it assigns the value of an expression to a variable or set of variables.

! is the complement operator.

The complement operator can be used to express the logic equation in negative true logic. The operator directly precedes the variable name (no spaces) and denotes that the expression on the right side is to be complemented before it is assigned to the variable name. Use of the complement operator on the left side is provided solely as a convenience. The equation may just as easily be written by complementing the entire expression on the right side.

Older logic design software that did not provide the automatic DeMorgan capability (output polarity assigned according to the pin variable declaration) required the use of the complement operator when using devices with inverting buffers.

Place logic equations in the "Logic Equation" section of the source file provided by the template file.

Logic equations are not limited solely to pin (or node) variables, but may be written for any arbitrary variable name. A variable defined in this manner is an intermediate variable. An intermediate variable name can be used in other expressions to generate logic equations or additional intermediate variables. Writing logic equations in this "top down" manner yields a logic description file that is generally easier to read and comprehend.

Place intermediate variables in the "Declarations and Intermediate Variable Definitions" section of the source file.

The following are some examples of logic equations:

Table 6-14. Sample Logic Equations

```
SEL_0=A15 & !A14;          /* A simple, decoded output pin */
Q0.D=Q1 & Q2 & Q3;         /* Output pin w/ D flip-flop    */
Q1.J = Q2 # Q3;            /* Output pin w/ JK flip-flop   */
Q1.K = Q2 & !Q3;
MREQ=READ # WRITE;         /* Intermediate Variable        */
SEL_1=MREQ & A15;          /* Output intermediate var      */
[D0..3] = 'h'FF;           /* Data bits assigned to constant*/
[D0..3].oe = read;         /* Data bits assigned to variable */
```

APPEND Statements

In standard logic equations, normally only one expression is assigned to a variable. The **APPEND** statement enables multiple expressions to be assigned to a single variable. The format is as follows.

APPEND [!]var[.ext] = expr ;

where

! is the complement operator to optionally define the polarity of the variable.

var is a single variable or a list of indexed or non-indexed variables in standard list format.

.ext is an optional extension that defines the function of the variable.

= is the assignment operator.

expr is a valid expression.

; is a semicolon to mark the end of the statement.

The expression that results from multiple **APPEND** statements is the logical OR of all the **APPEND** statements. If an expression has not already been assigned to the variable, the first **APPEND** statement is treated as the first assignment.

The following example shows several APPEND statements.

```
APPEND Y = A0 & A1 ;
APPEND Y = B0 & B1 ;
APPEND Y = C0 & C1 ;
```

The three statements above are equivalent to the following equation.

```
Y = (A0 & A1) # (B0 & B1) # (C0 & C1) ;
```

The **APPEND** statement is useful in adding additional terms (such as reset) to state-machine variables or constructing user-defined functions (see the subtopics, State Machine Syntax and User-Defined Functions in this chapter).

Set Operations

All operations that are performed on a single bit of information, for example, an input pin, a register, or an output pin, may be applied to multiple bits of information grouped into sets. Set operations can be performed between a set and a variable or expression, or between two sets.

The result of an operation between a set and a single variable (or expression) is a new set in which the operation is performed between each element of the set and the variable (or expression). For example

```
[D0, D1, D2, D3] & read
```

evaluates to:

```
[D0 & read, D1 & read, D2 & read, D3 & read]
```

When an operation is performed on two sets, the sets must be the same size (that is, contain the same number of elements). The result of an operation between two sets is a new set in which the operation is performed between elements of each set.

For example

```
[A0, A1, A2, A3] & [B0, B1, B2, B3]
```

evaluates to:

```
[A0 & B0, A1 & B1, A2 & B2, A3 & B3]
```

Bit field statements (see the subtopic, Bit Field Declaration Statements in this chapter) may be used to group variables into a set that can be referenced by a single variable name. For example, group the two sets of variables in the above operation as follows:

```
FIELD a_inputs = [A0, A1, A2 A3] ;
FIELD b_inputs = [B0, B1, B2, B3] ;
```

Then perform a set operation between the two sets, for example, an AND operation, as follows:

```
a_inputs & b_inputs
```

When numbers are used in set operations, they are treated as sets of binary digits. A single octal number represents a set of three binary digits, and a single decimal or hexadecimal number represents a set of four binary digits. Table 6-15 lists the representation of numbers as sets.

Table 6-15. Equivalent Binary Sets

Number	Binary Set	Number	Binary Set
'O'X	[X, X, X]	'H'X	[X,X,X,X]
'O'0	[0, 0, 0]	'H'0	[0,0,0,0]
'O'1	[0, 0, 1]	'H'1	[0,0,0,1]
'O'2	[0, 1, 0]	'H'2	[0,0,1,0]
'O'3	[0, 1, 1]	'H'3	[0,0,1,1]
'O'4	[1, 0, 0]	'H'4	[0,1,0,0]
'O'5	[1, 0, 1]	'H'5	[0,1,0,1]
		[1, 1, 0]	'H'6 [0,1,1,0]
'O'7	[1, 1, 1]	'H'7	[0,1,1,1]
'D'0	[0,0,0,0]	'H'8	[1,0,0,0]
'D'1	[0,0,0,1]	'H'9	[1,0,0,1]
'D'2	[0,0,1,0]	'H'A	[1,0,1,0]
'D'3	[0,0,1,1]	'H'B	[1,0,1,1]
'D'4	[0,1,0,0]	'H'C	[1,1,0,0]
'D'5	[0,1,0,1]	'H'D	[1,1,0,1]
'D'6	[0,1,1,0]	'H'E	[1,1,1,0]
'D'7	[0,1,1,1]	'H'F	[1,1,1,1]

Numbers may be effectively used as “bit masks” in logic equations using sets. An example of this application is the following 5-bit counter.

```
field count = [Q3, Q2, Q1, Q0];
count.d = 'b' 0001 & (!Q0)
    # 'b' 0010 & (Q1 $ Q0)
    # 'b' 0100 & (Q2 $ Q1 & Q0)
    # 'b' 1000 & (Q3 $ Q2 & Q1 & Q0);
```

The equivalent logic equations written without set notation are as follows:

```
Q0.d = !Q0;
Q1.d = Q1 $ Q0;
Q2.d = Q2 $ Q1 & Q0;
Q3.d = Q3 $ Q2 & Q1 & Q0;
```

Equality Operations

Unlike other set operations, the equality operation evaluates to a single Boolean expression. It checks for bit equality between a set of variables and a constant. The format for the equality operation is as follows:

[var, var, ... var]: constant ;

bit_field_var:constant ;

where

[var, var, ... var] is a list of variables in shorthand notation.

constant is a number (hexadecimal by default).

bit_field_var is a variable defined using a bit field statement.

: is the equality operator.

; is a semicolon used to mark the statement end.

Square brackets do not indicate optional items, but delimit variables in a list.

Format1 is used between a list of variables and a constant value. Format 2 is used between a bit field variable and a constant value.

The bit positions of the constant number are checked against the corresponding positions in the set. Where the bit position is a binary 1, the set element is unchanged. Where the bit position is a binary 0, the set element is negated. Where the bit position is a binary X, the set element is removed. The resulting elements are then ANDed together to create a single expression. In the following example, hexadecimal D (binary 1101) is checked against A3, A2, A1, and A0.

select = [A3..0]:'h'D ;

The set elements A3, A2, and A0 remain unchanged because the corresponding bit position is one or true. Set element A1 is negated because its corresponding bit position is zero or false. Therefore, the above expression is equivalent to the following expression:

select = A3 & A2 & !A1 & A0 ;

In the following example, binary 1X0X is checked against A3, A2, A1, A0.

```
select = [A3..0]:'b'1X0X ;
```

The set element A3 remains unchanged because the corresponding bit position is one or true. Set element A1 is negated because its corresponding bit position is zero or false. Set elements A2 and A0 are removed from the expression because the corresponding bit positions are “don't-cared.” Therefore, the above expression is equivalent to the following equation:

```
select = A3 & !A1 ;
```

In addition to address decoding, the equality operator can be used to specify a counter or state machine. For example, a 5-bit counter can be specified using the following notation:

```
FIELD count = [Q0..3];
Q0.J = count:0 # count:2 # count:4 # count:6
     # count:8 # count:A # count:C #  count:E ;
Q0.K = count:1 # count:3 # count:5 # count:7
     # count:9 # count:B # count:D #  count:F ;
Q1.J = count:1 # count:5 # count:9 # count:D ;
Q1.K = count:3 # count:7 # count:B # count:F ;
Q2.J = count:3 # count:B ;
Q2.K = count:7 # count:F ;
Q3.J = count:7 ;
Q3.K = count:F ;
```

The equality operator can also be used with a set of variables that are to be operated upon identically. The following syntax can be used as a time-saving convenience:

```
[var, var, ... , var]:op
```

which is equivalent to:

```
var op var op ... var
```

where

op is the &, # or $ operator (or its equivalent if an alternate set of operators has been defined).

var is any variable name.

For example, the following three expressions

```
[A3,A2,A1,A0]:&
[B3,B2,B1,B0]:#
[C3,C2,C1,C0]:$
```

are equivalent respectively to:

```
A3 & A2 & A1 & A0
B3 # B2 # B1 # B0
C3 $ C2 $ C1 $ C0
```

The equality operation can be used with an equivalent binary set to create a function table description of the output values. For example, in the following Binary-to-BCD code converter, output values are assigned by using the equality operation to define the inputs, and equivalent binary sets to group the output.

```
        FIELD  input = [in3..0] ;
        FIELD  output = [out4..0] ;
/*    in3..0  ->out4..0*/
$DEFINE  L  'b'0
$DEFINE  H  'b'1
output          = input:0 & [L, L, L, L,  L]
                # input:1 & [L, L, L, L, H]
# input:2 & [L L,  L, H, L]
    # input:3 & [L, L, L, H, H]
                # input:4 & [L, L, H, L, L]
    # input:5 & [L, L, H, L, H]
    # input:6 & [L, L, H, H, L]
                # input:7 & [L, L, H, H, H]
                # input:8 & [L, H, L, L, L]
                # input:9 & [L, H, L, L, H]
                # input:A & [H, L, L, L, L]
                # input:B & [H, L, L, L, H]
                # input:C & [H, L, L, H, L]
                # input:D & [H, L, L, H, H]
```

```
                    # input:E & [H, L, H, L, L]
                    # input:F & [H, L, H, L, H];
        $UNDEF L
        $UNDEF H
```

Indexed Variable Bit Fields and Equality

Indexed variables, field statements and the range function operate with each other in tight union. This section will attempt to illustrate this relationship.

As discussed earlier in this chapter, indexed variables can be used as an easy way to declare multiple variables with few actual lines of code.

For example

```
Pin [2..4] = [AD0..2];
```

expands to:

```
Pin 2 = AD0;
Pin 3 = AD1;
Pin 4 = AD2;
```

The FIELD statement is used to group a set of related signals into one element. It works by using a 32 bit field where each bit in the field represents one of the members of the field. If there are less than 32 members then the extra bits are ignored. For example:

```
Pin 2 = VAR_A;
Pin 3 = VAR_B;
Pin 4 = VAR_C;
Pin 15 = ROM_SEL;
FIELD ADDR = [VAR_A,VAR_B,VAR_C];
```

The following figure shows how the variables VAR_A, VAR_B and VAR_C map into the bit field.

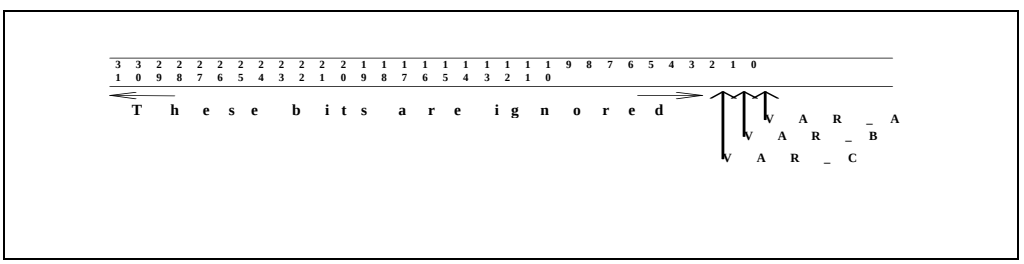


Figure 6-50. Bit field mapping of member variables

Now suppose that we had an output as follows:

ROM_SEL = ADDR:3;

The contents of the bit field for this equation would be as follows:

“XXXXXXXXXXXXXXXXXXXXXXXXXXXXX011”

This would result in the following equations:

ROM_SEL = !VAR_A & VAR_B & VAR_C;

When using indexed variables, the internal representation changes slightly. The index number of the variable determines its position in the bit field. Therefore, VAR0 always resides in bit position 0 regardless of the declaration of the field. The two following declarations both have the identical internal representation.

field ADDR = [VAR0, VAR1, VAR2];
field ADDR = [VAR2, VAR1, VAR0];

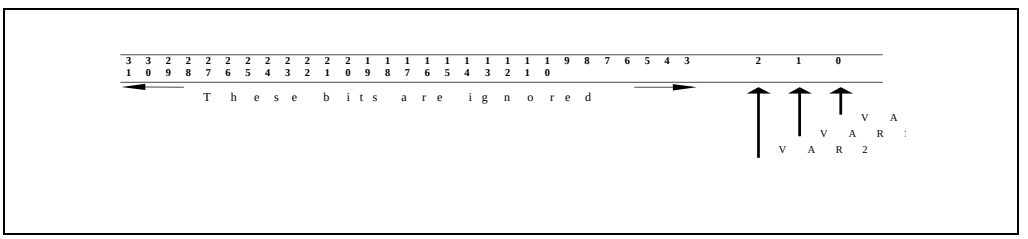


Figure 6-51. Bit field representation with indexed variables

Now suppose that we had an output as follows:

ROM_SEL = ADDR:3;

The contents of the bit field for this equation would be as follows:

“XXXXXXXXXXXXXXXXXXXXXXXXXXXXX011”

This would result in the following equations:

ROM_SEL = !VAR2 & VAR1 & VAR0;

If we take a set of variables that use a higher index we can see that the way indexed variables are handled may affect the output differently than we expect. If the variables used are VAR17, VAR18 and VAR19 then the bit map changes accordingly. The

equivalence with 3 now does not work because 3 only maps into bits 0, 1 and 2. What needs to be done is to add zeroes to move the desired equivalence up to the desired range.

Now suppose that we had an output as follows:

```
FIELD ADDR = [VAR18, VAR17, VAR16];
ROM_SEL = ADDR:3;
```

The variables would map into the bit field ADDR as follows:

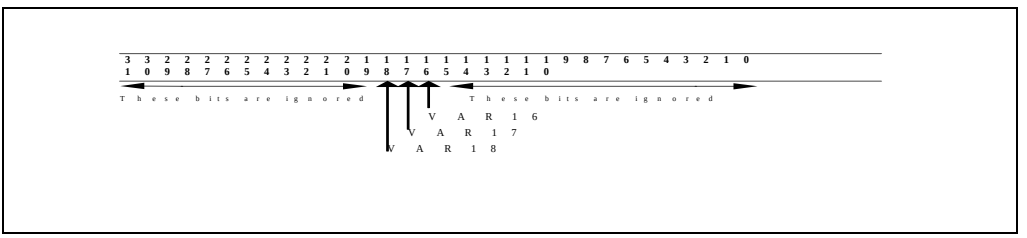


Figure 6-52. Bit field representation with indexed variables not starting at 0

If we attempt to apply an equivalence of three to this bit field, the bits will not match correctly.

The following line shows how the constant three maps onto the bit field.

“XXXXXXXXXXXXXXXXXXXXXXXXXXXXX011”

Notice that the significant bits in the above equivalence does not map over the bits representing the variables. What needs to be done to make this correct is to append enough zeroes to the end of the constant so that it represents what we truly want.

```
ROM_SEL = ADDR:30000;
```

This will now produce the correct results since the bit map from this constant is as follows:

“XXXXXXXXXXXXX011000000000000000”

```
ROM_SEL = !VAR18 & VAR17 & VAR16;
```

Range Operations

The range operation is similar to the equality operation except that the constant field is a range of values instead of a single value. The check for bit equality is made for each constant value in the range. The format for the range operation is as follows:

[var, var, ... var]:[constant_lo..constant_hi] ;

bit_field_var:[constant_lo..constant_hi] ;

where:

[var, var, ... var] is a list of variables in shorthand notation.

bit_field_var is a variable that has been defined using a bit field statement.

: is the equality operator.

; is a semicolon used to end the statement.

[constant_lo constant_hi] are numbers (hexadecimal by default) that define the range operation.

Square brackets do not indicate optional items, but delimit items in a list

The first format specifies the range operation between a list of variables and a range of constant values. The second format specifies a range operation between a bit field variable and a range of constant values.

All numbers greater than or equal to **constant_lo** and less than or equal to **constant_hi** are used to create ANDed expressions as in the equality operation. The sub-expressions are then ORed together to create the final evaluated expression. For example, the **RANGE** notation can be used to look for a decoded hex value between 1100 and 1111 on an address bus containing A3, A2, A1, and A0. First, define the address bus, as follows:

FIELD address = [A3..A0]

Then write the **RANGE** equation:

select = address:[C..F] ;

This is equivalent to the following equation:

select = address:C # address:D # address:E # address:F ;

This equation expands to:

```
Select      =       A3 & A2 & !A1 & !A0
            #       A3 & A2 & !A1 &  A0
      #     A3 & A2 &  A1 & !A0
```

```
                #       A3 & A2 &  A1&          A0 ;
```

The logic minimization capabilities within CUPL reduce the previous equation into a single product term equivalent. The range minimization works as follows. First, lines one and two are combined and lines three and four are combined to produce the following equation:

```
select      =       A3 & A2 & !A1 & (!A0 # A0)
    #       A3 & A2 &  A1 & (!A0 # A0) ;
```

Since the expression (!A0 # A0) is always true, it can be removed from the equation, and the equation reduces to:

```
select      =       A3 & A2 & !A1
            #       A3 & A2 &  A1 ;
```

By the same process, the equation reduces to the following:

```
select      =       A3 & A2 & (!A1 # A1) ;
```

Since the expression (!A1 # A1) is always true, removing it reduces the equation to the single product term:

```
select      =       A3 & A2 ;
```

When either the equality or range operations are used with indexed variables, the **CONSTANT** field must contain the same number of significant bit locations as the highest index number in the variable list. Index positions not in the pin list or field declaration are DON'T CAREd in the operation.

In the following example, pin assignments are made, an address bus is declared, and a decoded output is asserted over the hexadecimal memory address range 8000 through BFFF.

```
PIN    [1..4]  = [A15..12] ;
FIELD  address = [A15..12] ;
chip_select = address:[8000..BFFF] ;
```

Although the variables A15, A14, A13, and A12 are the only address inputs to the device, a full 16-bit address is used in the range expression. The most significant bit, A15, determines that the field is a 16-bit field. The lower order address bits (A0 through A11) are effectively DON'T CAREd in the equation, because the variable index numbers are used to determine bit position. Even though the lower order bits are not present in the

device, the constant value is written as though they did exist, generating a more meaningful expression in terms of documentation.

Consider, for example, the following application that decodes a microprocessor address for an I/O port:

```
PIN  [3..6] = [A7..10] ;
FIELD ioaddr = [A7..10];
```

The order of the field declaration is not important when using indexed variables

```
io_port = ioaddr:[400..6FF] ;
```

Since the most significant bit is A10, an 11-bit constant field is required (although three hex digits form a 12-bit address, the bit position for A11 is ignored).

Address bits A0 through A6 are DON'T CAREd in the expression. Without the bit position justification, the range equation would be written as

```
io_port = ioaddr:[8..D] ;
```

This expression doesn't clearly document the actual I/O address range that is desired.

The original equation without the range operation could be written as follows:

```
io_port =    A10 & !A9 & !A8 & !A7
     #       A10 & !A9 & !A8 &  A7
     #       A10 & !A9 &  A8 & !A7
     #       A10 & !A9 &  A8 &  A7
     #       A10 &  A9 & !A8 & !A7
     #       A10 &  A9 & !A8 &  A7 ;
```

CUPL reduces this equation to the following:

```
io_port = A10 & !A9  #  A10 &  A9 & !A8 ;
```

Careless use of the range feature may result in the generation of huge numbers of product terms, particularly when fields are composed of variables with large index numbers. The algorithm for the range does a bit-by-bit comparison of the two constant values given in the range operation, starting with index variable 0 (whether it exists in the field or not). If the value of the bit position for **constant_lo** is less than that for **constant_hi**, the variable for that bit position is not used in the generation of the ANDed expressions. When the value of the bit position for **constant_lo** is equal to or greater than that for **constant_hi**, an ANDed expression is created for all constant values between this new value and the original **constant_hi** value.

For example, consider the following logic equation that uses the range function on a 16-bit address field.

```
field  address = [A15..12] ;
board_select = address:[A000..DFFF] ;
```

Figure 6-53 shows how the CUPL algorithm treats this equation.

```
RANGE FUNCTION ALGORITHM

              A A A A A A A A A A A A A A A A

bit           1 1 1 1 1 1
position ->   5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
              -----------

constant_hi   1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
constant_lo   1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
                        ^
                        |
              End of DON'T CARE
```

Figure 6-53. Range Function Algorithm

The algorithm ignores all bit positions lower than position 13, because for these positions **constant_lo** is less than **constant_hi**. Figure 6-54 shows the result.

```
RANGE FUNCTION ALGORITHM

            A A A A A A A A A A A A A A

bit         1 1 1 1 1 1
position -> 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1
            -----------

constant_hi 1 1 0 x x x x x x x x x x x x
constant_lo 1 0 1 x x x x x x x x x x x x
```

Figure 6-54. Range Function Results

The following two product terms are generated as a result of the range function in Figure 1-54.

A15 & A14 & !A13

A15 & !A14 & A13

The following equation is another example using the range function.

board_select = address:[A000..D000] ;

Because the values of **constant_lo** and **constant_hi** match for the least significant bits, the algorithm generates product terms as follows:

1010 0000 0000 0000

1010 0000 0000 0001

1010 0000 0000 0010

1010 0000 0000 0011

.

1100 1111 1111 1111

1101 0000 0000 0000

The number of product terms generated is over twelve thousand (4096 x 3 + 1).

Truth Tables

Sometimes the clearest way to express logic descriptions is in tables of information. CUPL provides the **TABLE** keyword to create tables of information. The format for using **TABLE** is as follows:

TABLE var_list_1 => var_list_2 {

input_n => output_n ;
.
.
input_n => output_n ;
}

where

var_list_1 defines the input variables.

var_list_2 defines the output variables.

input_n is a decoded value (hex by default) or a list of decoded values of **var_list_1**.

output_n is a decoded value (hex by default) of **var_list_2**.

{ } are braces to begin and end the assignment block.

=> specifies a one-to-one assignment between variable lists, and between input and output numbers.

First, define relevant input and output variable lists, and then specify one-to-one assignments between decoded values of the input and output variable lists. Don't-care values are supported for the input decode value, but not for the output decode value.

A list of input values can be specified to make multiple assignments in a single statement. The following block describes a simple hex-to-BCD code converter:

```
FIELD  input = [in3..0] ;
FIELD  output = [out4..0] ;
TABLE  input => output {
0=>00;          1=>01;          2=>02;          3=>03;
4=>04;          5=>05;          6=>06;          7=>07;
8=>08;          9=>09;          A=>10;          B=>11;
C=>12;          D=>13;          E=>14;          F=>15;
```

```
}
```

The following example illustrates the use of a list of input numbers to do address decoding for various-sized RAM, ROM, and I/O devices. The address range is decoded according to the rules (in terms of indexed variable usage) for the range operation (see the subtopic, Range Operations in this chapter).

```
PIN [1..4]  = [a12..15] ;          /* Upper 4 address*/
PIN 12  = !RAM_sel ;/* 8K x 8 RAM    */
PIN 13  = !ROM_sel ;/* 32K x 8 ROM  */
PIN 14  = !timer_sel ; /* 8253 Timer      */
FIELD address = [a15..12] ;
FIELD decodes = [RAM_sel,ROM_sel,timer_sel] ;
TABLE address  => decodes {
[1000..2FFF] => 'b'100;          /* select RAM     */
[5000..CFFF] => 'b'010;          /* select ROM     */
F000 => 'b'001;                  /* select timer      */
}
```

State-Machines

This section describes the CUPL state machine syntax, providing a brief overview of its use, a definition of a state machine, and explaining in detail the CUPL state machine syntax.

The state-machine approach used with the CUPL compiler-based PLD language permits bypassing the gate and equation level stage in logic design and to move directly from a system-level description to a PLD implementation. Additionally, unlike assembler-based approaches, the state-machine approach allows clear documentation of design, for future users.

State-Machine Model

A synchronous state machine is a logic circuit with flip-flops. Because its output can be fed back to its own or some other flip-flop's input, a flip-flop's input value may depend on both its own output and that of other flip-flops; consequently, its final output value depends on its own previous values, as well as those of other flip-flops.

The CUPL state-machine model, as shown in Figure 1-52, uses six components: inputs, combinatorial logic, storage registers, state bits, registered outputs, and non-registered outputs.

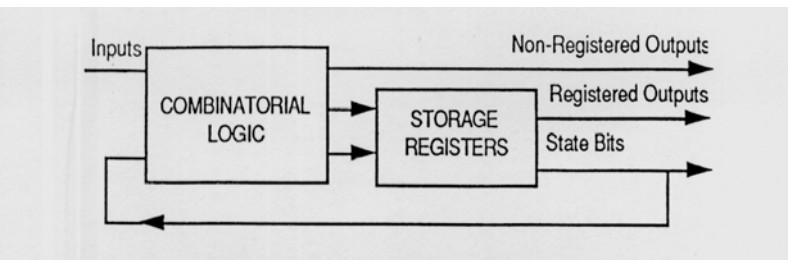


Figure 6-55. State Machine Model

The following definitions refer to Figure 6-55.

Inputs - are signals entering the device that originate in some other device.

Combinatorial Logic - is any combination of logic gates (usually AND-OR) that produces an output signal that is valid Tpd (propagation delay time) nsec after any of the signals

that drive these gates changes. Tpd is the delay between the initiation of an input or feedback event and the occurrence of a non-registered output.

State Bits - are storage register outputs that are fed back to drive the combinatorial logic. They contain the present-state information.

Storage Registers - are any flip-flop elements that receive their inputs from the state machine's combinatorial logic. Some registers are used for state bits: others are used for registered outputs. The registered output is valid Tco (clock to out time) nsec after the clock pulse occurs. Tco is the time delay between the initiation of a clock signal and the occurrence of a valid flip-flop output.

Figure 6-56 shows the timing relationships between the state machine components.

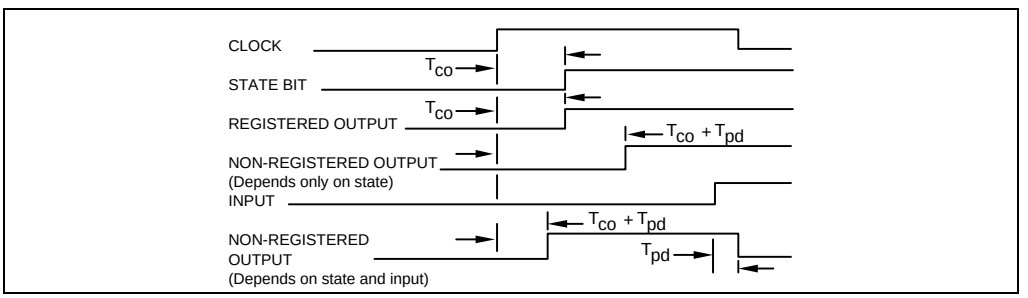


Figure 6-56. State MachineTiming Diagram

For the system to operate properly, the PLD’s requirements for setup and hold times must be met. For most PLDs, the setup time (Tsu) usually includes both the propagation delay of the combinatorial logic and the actual setup time of the flip-flops. Tsu is the time it takes for the result of either feedback or an input event to appear at the input to a flip-flop. A subsequent clock input cannot be applied until this result becomes valid at the flip-flop's input. The flip-flops can be either D, D-CE, J- K, S-R, or T types.

Non-registered Outputs - are outputs that come directly from the combinatorial logic gates. They may be functions of the state bits and the input signals (and have asynchronous timing), or they may be purely dependent on the current state-bit values, in which case they become valid Tco + Tpd nsec after an active clock edge occurs.

Registered Outputs - are outputs that come from the storage registers but are not included in the actual state-bit field (that is, a bit field composed of all the state bits). State- machine theory requires that the setting or resetting of these registered outputs

depends on the transition from a present state to a next state. This allows a registered output to be either set or reset in a given state depending upon how the machine came to be in that state. Thus, a registered output can assume a hold operation mode. In the hold mode, the registered output will remain at its last value as long as the current state transition does not specify an operation on that registered output.

This hold mode of operation is available only for devices which use D-CE, J-K, or S-R type flip-flops.

State Machine Syntax

To implement the state machine model, CUPL supplies a syntax that allows the describing of any function in the state machine.

The **SEQUENCE** keyword identifies the outputs of a state machine and is followed by statements that define the function of the state machine. The format for the **SEQUENCE** syntax is as follows:

```
SEQUENCE state_var_list {
PRESENT state_n0
        IF (condition1)   NEXT   state_n1;
        IF (condition2)   NEXT   state_n2 OUT  var;
        DEFAULT           NEXT   state_n0;
        PRESENT state_n1
                NEXT   state_n2;
        .
        .
PRESENT  state_nn statements ;
}
```

where

state_var_list is a list of the state bit variables used in the state machine block. The variable list can be represented by a field variable.

state_n is the state number and is a decoded value of the **state_variable_list** and must be unique for each **PRESENT** statement.

statements are any of the conditional, next, or output statements described in the following subsections of this section.

; is a semicolon used to mark the end of a statement.

{ } are braces to mark the beginning and end of the state machine description.

Symbolic names defined with the **$DEFINE** command may be used to represent **state_numbers**.

The **SEQUENCE** keyword causes the storage registers and registered output types generated to be the default type for the target device. For example, by using the **SEQUENCE** keyword in a design with a P16R8 target device, the state storage registers and registered outputs will be generated as D-type flip-flops.

The storage registers for certain devices can be programmed as more than one type. In the case of the F159 (Signetics PLS159), they can be either D or J-K type flip-flops. By default, using the **SEQUENCE** statement with a design for the F159 will cause the state storage registers and registered outputs to be generated as J-K type flip-flops. To override this default, the **SEQUENCED** keyword would be used in place of the **SEQUENCE** keyword. This would cause the state registers and registered outputs to be generated as D-type flip-flops.

Along with the **SEQUENCE** and **SEQUENCED** keywords are the **SEQUENCEJK**, **SEQUENCERS**, and **SEQUENCET** keywords. Respectively, they cause the state registers and registered outputs to be generated as J-K, S-R, and T-type flip-flops.

The subsections that follow describe the types of statements that can be written in the state-machine syntax. Statements use the **IF**, **NEXT**, **OUT** and **DEFAULT** keywords.

Unconditional NEXT Statement

This statement describes the transition from the present state to a specified next state. The format is:

```
PRESENT state_n
NEXT  state_n ;
```

where

state_n is a decoded value of the state bit variables that are the output of the state machine.

A symbolic name can be assigned with the **$DEFINE** command to represent **state_n**.

Because the statement is unconditional (that is, it describes the transition to a specific next state), there can be only one **NEXT** statement for each **PRESENT** statement.

The following example specifies the transition from binary state 01 to binary state 10.

```
PRESENT 'b'01
NEXT 'b'10 ;
```

Figure 6-57 shows the transition described in the example above.

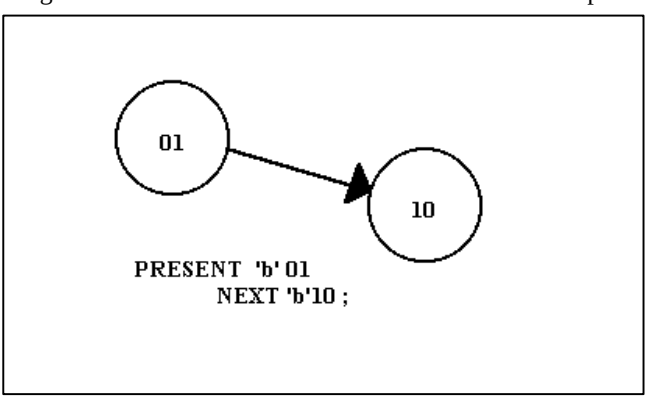


Figure 6-57. Unconditional Next Statement Diagram

For the transition described in the example and figure above, CUPL generates the following equations, depending on the type of flip-flop that is specified:

D-Type Flip-Flop

```
Q1.D = !Q1 & Q0;
Q0.D = 'b'0;      /* implicitly resets */
```

J-K-Type Flip-Flop

```
Q1.J = !Q1 & Q0;
Q1.K = 'b'0;
Q0.J = 'b'0;
Q0.K = !Q1 & Q0;
```

S-R-Type Flip-Flop

```
Q1.S = !Q1 & Q0;
Q1.R = 'b'0;
Q0.S = 'b'0;
Q0.R = !Q1 & Q0;
```

D-CE-Type Flip-Flop

```
Q1.D  = !Q1 & Q0;
Q1.CE = !Q1 & Q0;
Q0.D  = 'b'0;
Q0.CE = !Q1 & Q0;
```

T-Type Flip-Flop

```
Q1.T  = !Q1 & Q0;
Q0.T  = !Q1 & Q0;
```

Conditional NEXT Statement

This statement describes the transition from the present state to a next state if the conditions in a specified input expression are met. The format is as follows.

```
PRESENT state_n
IF expr NEXT state_n;
```

```
        .
        .
        .
    IF  expr  NEXT  state_n;
    [DEFAULT  NEXT state_n;]
```

where

state_n is a decoded value of the state bit variables that are the output of the state machine.

expr is any valid expression (see the subtopic, Expressions in this chapter).

; is a semicolon used to mark the end of a statement.

The value for each state number must be unique.

More than one conditional statement can be specified for each **PRESENT** statement.

The **DEFAULT** statement is optional. It describes the transition from the present state to a next state if none of the conditions in the specified conditional statements are met. In other words, it describes the condition that is the complement of the sum of all the conditional statements.

Be careful when using the **DEFAULT** statement. Because it is the complement of all the conditional statements, the **DEFAULT** statement can generate an expression complex enough to greatly slow CUPL operation. In most applications, one or two conditional statements can be specified instead of the **DEFAULT** statement.

The following is an example of two conditional **NEXT** statements without a **DEFAULT** statement.

```
PRESENT 'b'01
    IF INA NEXT 'b'10;
    IF !INA NEXT 'b'11;
```

Figure 6-58 shows the transitions described by the above example.

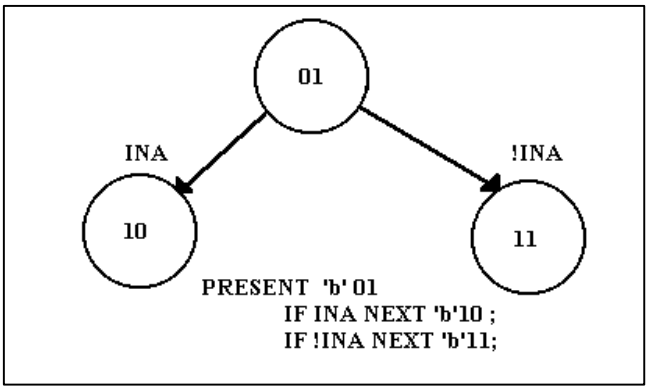


Figure 6-58. Conditional NEXT Statement Diagram

For the transitions described in the above example and figure, CUPL generates the following equations, depending on the type of flip-flop that is specified:

D-Type Flip-Flop

```
Q1.D = !Q1 & Q0;
Q0.D = !Q1 & Q0 & !INA;
```

D-CE-Type Flip-Flop

```
Q1.D  = !Q1 & Q0;
Q1.CE = !Q1 & Q0;
Q0.D  = !Q1 & Q0 & !INA;
Q0.CE = !Q1 & Q0 & INA;
```

J-K-Type Flip-Flop

```
Q1.J = !Q1 & Q0;
Q1.K = 'b'0;
Q0.J = 'b'0;
Q0.K = !Q1 & Q0 & INA;
```

S-R-Type Flip-Flop

```
Q1.S = !Q1 & Q0;
```

```
Q1.R = 'b'0;
Q0.S = 'b'0;
Q0.R = !Q1 & Q0 & INA;
```

T-Type Flip-Flop

```
Q1.T = !Q1 & Q0;
Q0.T = !Q1 & Q0 & INA;
```

The following is an example of two conditional statements with a **DEFAULT** statement.

```
PRESENT 'b'01
    IF INA & INB NEXT 'b'10';
    IF INA & !INB NEXT 'b'11;
            DEFAULT NEXT 'b'00;
```

Figure 6-59 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.

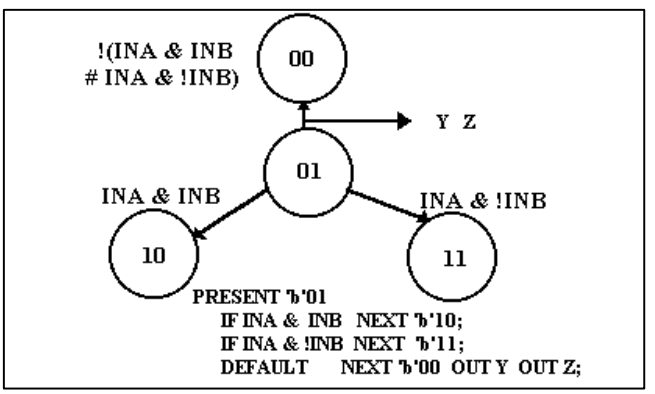


Figure 6-59. Conditional NEXT Statement with Default Diagram

Use the negation mode only for D-CE, J-K, T or S-R type flip-flops; D-type flip-flops implicitly reset when assertion is not specified.

For the transitions described in the above example and figure, CUPL generates the following equations, depending on the type of flip-flop that is specified.

D-Type Flip-Flop

```
Q1.D = !Q1 & Q0 & INA;
Q0.D = !Q1 & Q0 & INA & !INB;
```

D-CE-Type Flip-Flop

```
Q1.D  = !Q1 & Q0 & INA;
Q1.CE = !Q1 & Q0 & INA;
Q0.D  = 'b'0;
Q0.CE = !Q1 & Q0 & !INA
        # !Q1 & Q0 & INA & INB;
```

J-K-Type Flip-Flop

```
Q1.J = !Q1 & Q0 & INA;
Q1.K = 'b'0;
Q0.J = 'b'0;
Q0.K = !Q1 & Q0 & INA & INB
       # !Q1 & Q0 & !INA;
```

S-R-Type Flip-Flop

```
Q1.S = !Q1 & Q0 & INA;
Q1.R = 'b'0;
Q0.S = 'b'0;
Q0.R = !Q1 & Q0 & INA & INB
       # !Q1 & Q0 & !INA;
```

T-Type Flip-Flop

```
Q1.T  = !Q1 & Q0 & INA;
Q0.T  = !Q1 & Q0 & !INA
        # !Q1 & Q0 & INA & INB;
```

Unconditional Synchronous Output Statement

This statement describes a transition from the present state to a next state, specifies a variable for the registered (synchronous) outputs associated with the transition, and defines whether the variable is logically asserted. The format is as follows:

```
PRESENT state_n
    NEXT state_n  OUT [!]var... OUT [!]var;
```

where

state_n is a decoded value (default hex) of the state bit variables that are the output of the state machine.

var is a variable name declared in the pin declarations. It is not a variable from the **SEQUENCE state_var_list**.

! is the complement operator; use it to logically negate the variable, or omit it to logically assert the variable.

; is a semicolon used to mark the end of a statement.

The square brackets indicate optional items.

The **PIN** declaration statement (see the subtopic, Pin Declaration Statements in this chapter) determines whether the variable, when asserted, is active-HI or active-LO. For example, if the variable has the negation symbol (!var) in the pin declaration, when it is asserted in the **OUT** statement, its value is active-LO.

Use the negation mode only for D-CE, J-K, T or S-R type flip-flops; D-type flip-flops implicitly reset when assertion is not specified.

The following is an example of an unconditional synchronous output statement.

```
PRESENT 'b'01
    NEXT 'b'10 OUT Y OUT Z ;
```

Figure 6-60 shows the transition and output variable definition described in the example above.

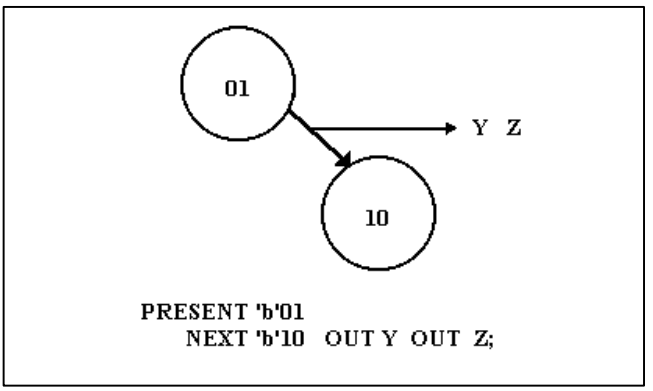


Figure 6-60. Unconditional Synchronous Output Diagram

For the synchronous output definitions in the example and figure above, CUPL generates the following equations, depending on the type of flip-flop that is specified.

D-Type Flip-Flop

```
Y.D  =  !Q1  &  Q0;
(not defined for Z output)
```

D-CE Type Flip-Flop

```
Y.D =       !Q1 & Q0;
Y.CE        =       !Q1 & Q0;
Z.D =       'b'0;
Z.CE        =       !Q1 & Q0;
```

J-K-Type Flip-Flop

```
Y.J  =      !Q1  &  Q0;
Y.K =       'b'0;
Z.J  =      'b'0;
Z.K =       !Q1  &  Q0;
```

S-R-Type Flip-Flop

```
Y.S =       !Q1  &  Q0;
Y.R =       'b'0;
Z.S =       'b'0;
Z.R =       !Q1  &  Q0;
```

T-Type Flip-Flop

```
Y.T =       !Q1 & Q0;
Z.T =       !Q1 & Q0;
```

Conditional Synchronous Output Statement

This statement describes a transition from the present state to a next state, specifies a variable for the registered (synchronous) outputs associated with the transition, and defines whether the variable is logically asserted if the conditions specified in an input expression are met. The format is as follows:

```
PRESENT state_n
    IF expr NEXT state_n OUT [!]var...OUT [!] var;
                .
                .
    IF expr NEXT state_n OUT [!]var...OUT [!]var;
            [ [DEFAULT] NEXT state_n OUT [!]var;]
```

where

state_n is a decoded value (default hex) of the state bit variables that are the output of the state machine.

var is a variable name declared in the pin declarations. It is not a variable from the **SEQUENCE state_variable_list**.

! is the complement operator; use it to logically negate the variable, or omit it to logically assert the variable.

; is a semicolon used to mark the end of a statement.

expr is any valid expression.

The square brackets indicate optional items.

The **PIN** declaration statement (see the subtopic, Pin Declaration Statements in this chapter) determines whether the variable, when asserted, is active-HI or active-LO. For example, if the variable has the negation symbol (!var) in the pin declaration, when it is asserted in the **OUT** statement, its value is active-LO.

Use the negation mode only for J-K or S-R-type flip-flops; D-type flip-flops implicitly reset when assertion is not specified.

The **DEFAULT** statement is optional. It describes the transition from the present state to a next state, and defines the output variable, if none of the conditions in the specified conditional statements are met. In other words, it describes the condition that is the complement of the sum of all the conditional statements.

Be careful when using the **DEFAULT** statement. Because it is the complement of all the conditional statements, the **DEFAULT** statement can generate an expression complex enough to greatly slow CUPL operation. In most applications, one or two conditional statements can be specified instead of the **DEFAULT** statement.

The following is an example of conditional synchronous output statements without a **DEFAULT** statement.

```
PRESENT 'b'01
    IF INA NEXT 'b'10 OUT Y;
    IF !INA NEXT 'b'11 OUT Z;
```

Figure 6-61 shows the transitions and outputs defined by the statements in the example above.

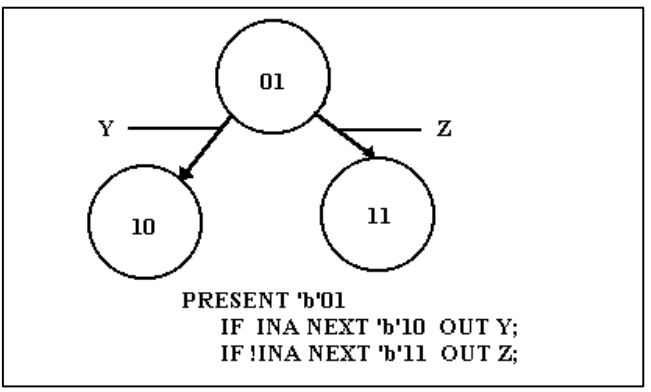


Figure 6-61. Synchronous Conditional Output Diagram

For the synchronous output definitions in the example and figure above, CUPL generates the following equations, depending on the type of flip-flop specified:

D-Type Flip-Flop

```
Y.D =    !Q1 & Q0 & INA;
Z.D =    !Q1 & Q0 & !INA;
```

D-CE Type Flip-Flop

```
Y.D =    !Q1 & Q0 & INA;
Y.CE     =    !Q1 & Q0 & INA;
Z.D =    !Q1 & Q0 & !INA;
Z.CE     =    !Q1 & Q0 & !INA;
```

J-K-Type Flip-Flop

```
Y.J  =   !Q1 & Q0 & INA;
Y.K =    'b'0;
Z.J  =   !Q1 & Q0 & !INA;
Z.K =    'b'0;
```

S-R-Type Flip Flop

```
Y.S =       !Q1 & Q0 & INA;
Y.R =       'b'0;
Z.S =       !Q1 & Q0 & !INA;
Z.R =       'b'0;
```

T-Type Flip-Flop

```
Y.T =       !Q1 & Q0 & INA;
Z.T =       !Q1 & Q0 & !INA;
```

The following is an example of conditional output statements with a **DEFAULT** statement.

```
PRESENT 'b'01
    IF INA & INB NEXT 'b'10;
    IF INA & !INB NEXT 'b'11;
      DEFAULT NEXT 'b'00 OUT Y
            OUT !Z;
```

Figure 6-62 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.

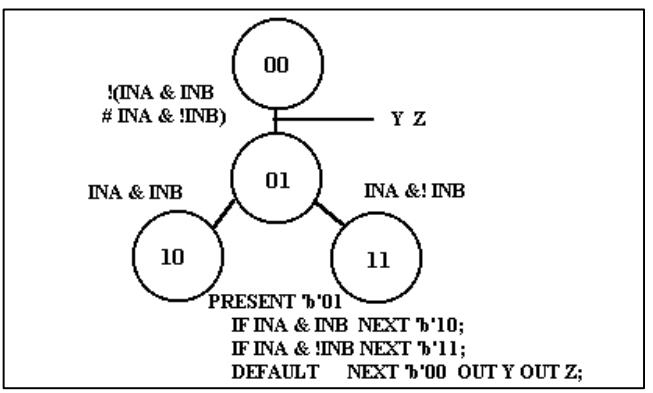


Figure 6-62. Synchronous Conditional Output with Default Diagram

For the transitions described in the above example and figure, CUPL generates the following equations, depending on the type of flip-flop that is specified.

D-Type Flip-Flop

```
Y.D =       !Q1  &  Q0  &  !INA;
(not defined for Z output)
```

D-CE-Type Flip-Flop

```
Y.D =       !Q1 & Q0 & !INA;
Y.CE        =       !Q1 & Q0 & !INA;
Z.D =       'b'0;
Z.CE        =       !Q1 & Q0 & INA;
```

J-K-Type Flip-Flop

```
Y.J =       !Q1  &  Q0  &  !INA;
Y.K =       'b'0;
Z.J =       'b'0;
Z.K =       !Q1  &  Q0  &  !INA;
```

S-R-Type Flip-Flop

```
Y.S =       !Q1  &  Q0  &  !INA;
Y.R =       'b'0;
Z.S =       'b'0;
Z.R =       !Q1  &  Q0  &  !INA;
```

T-Type Flip-Flop

```
Y.T =       !Q1 & Q0 & !INA
Z.T =       !Q1 & Q0 & INA;
```

Unconditional Asynchronous Output Statement

This statement specifies variables for the non-registered (asynchronous) outputs associated with a given present state, and defines when the variable is logically asserted. The format is as follows:

```
PRESENT state_n
    OUT  var  ...  OUT  var ;
```

where:

state_n is a decoded value (default hex) of the state bit variables that are the output of the state machine.

var is a variable name declared in the pin declarations. It is not a variable from the **SEQUENCE state_var_list**.

; is a semicolon used to mark the end of a statement.

The **PIN** declaration statement (see the subtopic, Pin Declaration Statements in this chapter) determines whether the variable, when asserted, is active-HI or active-LO. For example, if the variable has the negation symbol (!var) in the pin declaration, when it is asserted in the **OUT** statement, its value is active-LO.

Negating the variable (with the complement operator) is not a valid format for this statement.

Only one output statement can be written for each present state. However, multiple variables can be defined using more than one **OUT** keyword.

The following is an example of an unconditional asynchronous output statement.

```
PRESENT 'b'01
    OUT Y  OUT Z;
```

Figure 6-63 shows the outputs defined by the statements in the example above.

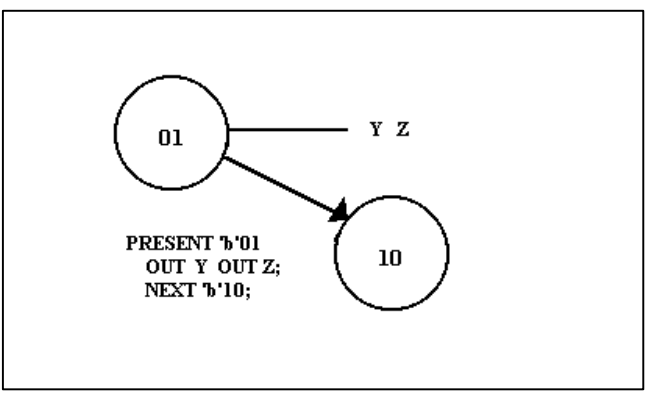


Figure 6-63. Unconditional Asynchronous Output Diagram

For the asynchronous output definitions in the example and figure above, CUPL generates the following equations:

```
Y = !Q1 & Q0;
Z = !Q1 & Q0;
```

Conditional Asynchronous Output Statement

This statement specifies variables for the non-registered (asynchronous) outputs associated with a given present state, and defines when the variables are logically asserted, if the conditions in an input expression are met. The format is as follows:

```
PRESENT state_n
    IF  expr  OUT  var ... OUT var;
            .
            .
    IF  expr  OUT  var ... OUT var;
            [DEFAULT   OUT  var ... OUT var;]
```

where

state_n is a decoded value (default hex) of the state bit variables that are the output of the state machine.

var is a variable name declared in the pin declarations. It is not a variable from the **SEQUENCE** statement.

expr is any valid expression.

; is a semicolon used to mark the end of a statement.

The square brackets indicate optional items.

The **PIN** declaration statement determines whether the variable, when asserted, is active-HI or active-LO. For example, if the variable has the negation symbol (!var) in the pin declaration, when it is asserted in the **OUT** statement, its value is active-LO.

Negating the variable (with the complement operator) is not a valid format for this statement. Multiple output statements can be written for each present state, and define multiple variables using the **OUT** keyword.

The **DEFAULT** statement is optional. It defines the output variable if none of the conditions in the specified conditional statements are met. In other words, it describes the condition that is the complement of the sum of all the conditional statements.

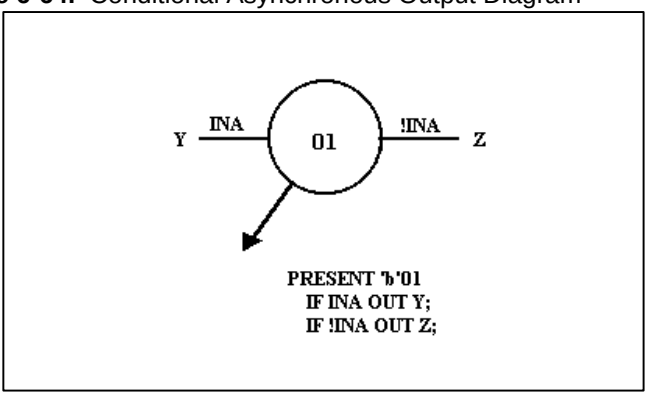
Be careful when using the **DEFAULT** statement. Because it is the complement of all the conditional statements, the **DEFAULT** statement can generate an expression complex enough to greatly slow CUPL operation. In most applications, one or two conditional statements can be specified instead of the **DEFAULT** statement.

The following is an example of conditional asynchronous output statements without a default statement.

```
PRESENT 'b'01
    IF INA OUT Y;
    IF !INA OUT Z;
```

Figure 6-64 shows the outputs defined by the statements in the above example.

Figure 6-64. Conditional Asynchronous Output Diagram



For the asynchronous output definitions in the example and figure above, CUPL generates the following equations:

```
Y = !Q1 & Q0 & INA;
Z = !Q1 & Q0 & !INA;
```

The following is an example of conditional asynchronous output statements with a **DEFAULT** statement.

```
PRESENT 'b'01
    IF INA & INB OUT X;
    IF INA & !INB OUT Y;
    DEFAULT OUT Z;
```

Figure 6-65 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.

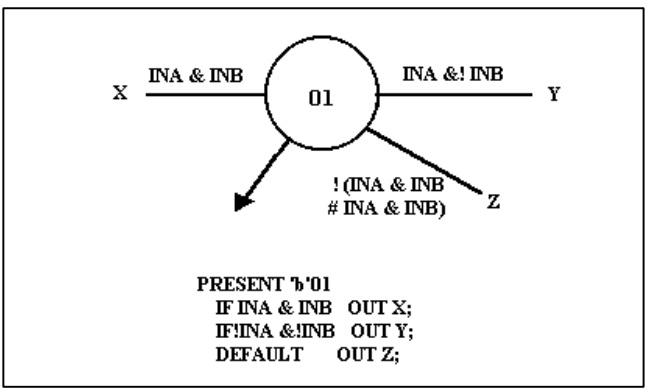


Figure 6-65. Conditional Asynchronous Output with Default Diagram

For the transitions described in the above example and figure, CUPL generates the following equations, depending on the type of flip-flop that is specified.

```
X = !Q1 & Q0 & INA & !INB;
Y = !Q1 & Q0 & INA & INB;
Z = !Q1 & Q0 & !INA;
```

One-Hot-Bit State Machines

Using this option will cause the compiler to generate state machine equations as 'one-hot-bit'. This has some distinct advantages in register rich architectures. The fanin is reduced making routing much easier and timing problems associated with variable length feedback paths from register to register are eliminated. To use this feature you define each of your states with a "one-hot-bit" pattern. Currently, CUPL treats all state machines in the design as "one-hot-bit" if the option is used. Future generations of this feature will allow mixing of "normal" and "one-hot-bit" state machines in the same design by using advanced syntax.

Sample State-Machine Syntax File

This section provides an example of a simple two-bit counter implemented with state-machine syntax.

Figure 6-66 shows a diagram of the counter operation.

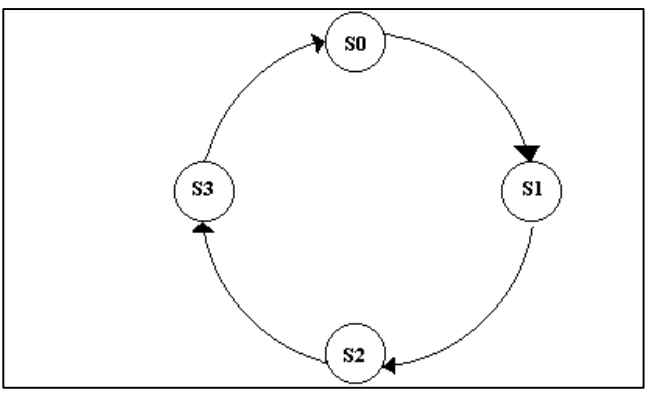


Figure 6-66. Simple 2-Bit Counter Diagram

The **$DEFINE** command assigns symbolic names to the states of the counter, and the **SEQUENCE** statement defines the transitions between states.

```
$DEFINE  S0  0       /* assign symbolic names */
$DEFINE  S1  1       /* to states             */
```

```
$DEFINE  S2  2
$DEFINE  S3  3
FIELD  count  =  [Q1, Q0];
/* assign field variable to statebits */
SEQUENCE  count  {
     PRESENT S0     NEXT  S1 ;
     PRESENT S1     NEXT  S2 ;
     PRESENT S2     NEXT  S3 ;
     PRESENT S3     NEXT  S0 ;
}
```

See the example, Decade Up/Down Counter for another illustration of a state machine implementation.

Defining Multiple State Machines

The CUPL syntax allows for more than one state machine to be defined within the same PLD design. When multiple state machines are defined, occasionally the designer would like to have the state machines communicate with each other. That is, when one state machine reaches a certain state another state machine may begin. There are two methods of accomplishing state machine communication: using set operations on the state bits or defining a "global" signal that can be accessed by both state machines. If the **One-Hot Bit** state machine option is used, all state machines in the design file are compiled to One-Hot.

In one state machine a conditional statement can contain another state machine's name followed by a state number or range of state numbers. The conditional statement will become TRUE when the other state machine reaches that particular state or states. The same case is true when using a register that is accessed by multiple state machines. However, this method requires the use one of the devices output or buried registers. Depending on the situation, the global register could also be combinatorial which may make a difference as to when the state machine receives the information from another state machine.

Condition Syntax

The **CONDITION** syntax provides a higher-level approach to specifying logic functions than does writing standard Boolean logic equations for combinatorial logic. The format is as follows:

```
CONDITION  {
    IF expr0 OUT  var ;
      .
      .
    IF exprn OUT  var ;
    DEFAULT  OUT  var ;
    }
```

where

expr is any valid expression.

var is a variable name declared in the pin declaration. It can also be a list of indexed or non-indexed variables in list notation.

; is a semicolon used to mark the end of a statement.

The **CONDITION** syntax is equivalent to the asynchronous conditional output statements of the state machine syntax, except that there is no reference to any particular state. The variable is logically asserted whenever the expression or **DEFAULT** condition is met.

The variable cannot be logically negated in this format.

Be careful when using the **DEFAULT** statement. Because it is the complement of all the conditional statements, the **DEFAULT** statement can generate an expression complex enough to greatly slow CUPL operation. In most applications, one or two conditional statements may be specified instead of the **DEFAULT** statement.

The following is an example of a 2 to 4 line decoder for the **CONDITION** syntax. The two data inputs, A and B, select one of four decoded outputs, Y0 through Y3, whenever the **ENABLE** signal is asserted. The **NO_MATCH** output is asserted if none of the other four outputs are true.

```
PIN [1,2] = [A,B] ;     /* Data Inputs */
PIN 3 = !enable ;       /* Enable Input */
PIN [12..15] = [Y0..3] ; /* Decoded Outputs */
PIN 14 = no_match ;  /* Match Output */
CONDITION {
    IF enable & !B & !A       out  Y0 ;
    IF enable & !B & A        out  Y1 ;
    IF enable & B & !A        out  Y2 ;
    IF enable & B & A         out  Y3 ;
    }
```

The **DEFAULT** expression of the above example is equivalent to the following logic equation

```
no_match = !( enable & !B & !A)
# enable & !B &  A
# enable &  B & !A
# enable &  B &  A ;
```

which reduces to the following:

```
no_match = !enable ;
```

User-Defined Functions

The **FUNCTION** keyword permits the creating of personal keywords by encapsulating some logic as a function and giving it a name. This name can then be used in a logic equation to represent the function. The format for user-defined functions is as follows:

FUNCTION name ([parameter$_0$,....,parameter$_n$])

{ body }

where

name is any group of valid symbols used to reference the function. Do not use any of the CUPL reserved keywords.

parameter is an optional variable used to reference variables when the function is used in a logic equation. It cannot be an expression.

body is any combination of logic equations, truth tables, state-machine syntax, condition syntax, or user function.

( ) are parentheses used to enclose the parameter list.

{ } are braces used to enclose the body of the function.

The square brackets indicate optional items.

The statements in the body may assign an expression to the function, or may be unrelated equations.

When using optional parameters, the number of parameters in the function definition and in the reference must be identical. The parameters defined in the body of the function are substituted for the parameters referenced in the logic equation.

For example, the following defines an exclusive OR function:

```
FUNCTION  xor(in1, in2) {
/* in1 and in2 are parameters */
xor  =  in1  &  in2  #  !in1  &  in2 ;
```

```
}
```

An xor can be used in an equation with the inputs A and B passed as parameters, as follows:

```
Y = xor(A,B) ;
```

The result is the following logic equation assignment for the output variable Y:

```
Y = A & !B # !A & B ;
```

When a function variable is referenced in an expression, the compiler takes the following action:

1. A special function invocation variable is assigned for the function name and its arguments. This variable name is not user accessible.

2. The rest of the expression is evaluated.

3. The function body, with the invocation parameters substituted, is evaluated.

4. The function invocation variable is assigned an expression according to the body of the function. If no assignment is made in the body statements, the function invocation variable is assigned the value of 'h'o.

Functions must be defined before they may be referenced. Functions are not recursive; that is, a function body may not include a reference of the function being defined.

The following example shows a user-defined function to construct state-machine-type transitions for non-registered devices without internal feedback (such as PROMs).

```
FUNCTION   TRANSITION(present_state,
                      next_state,
                      input_conditions ) {
APPEND  state_out = state_in:present_state &
                      input_condition &
```

```
                next_state;
}
```

The function defined in the example above is used in the following example to implement a simple up/down counter as a series of **TRANSITION** function references:

```
PIN [10,11] = [Qin0..1];                /* Registered PROM */
/*output feed back externally on input pins      */
PIN [12,13] = [count0..1] ;             /*Count Control      */
PIN [1,2] = [Q0..1] ;                   /* PROM Outputs     */
FIELD state_in  = [Qin0..1] ;
FIELD state_out  =[Q0..1] ;
count_up = !count1 & !count0 ;          / * count up       */
count_dn = !count1 & count0 ;           /* count down       */
hold_cnt  =  count1;                    /* hold count       */
$DEFINE  STATE0  'b'00
$DEFINE  STATE1  'b'01
$DEFINE  STATE2  'b'10
$DEFINE  STATE3  'b'11
/* (transition function definition made here)      */
TRANSITION(STATE0,      STATE1,         count_up) ;
TRANSITION(STATE1,      STATE2,         count_up) ;
TRANSITION(STATE2,      STATE3,         count_up) ;
TRANSITION(STATE3,      STATE0,         count_up ;
TRANSITION(STATE0,      STATE3,         count_dn) ;
TRANSITION(STATE1,      STATE0,         count_dn) ;
TRANSITION(STATE2,      STATE1,         count_dn) ;
TRANSITION(STATE3,      STATE2,         count_dn) ;
TRANSITION(STATE0,      STATE0,         hold_cnt) ;
TRANSITION(STATE1,      STATE1,         hold_cnt) ;
TRANSITION(STATE2,      STATE2,         hold_cnt) ;
TRANSITION(STATE3,      STATE3,         hold_cnt) ;
```

7. Simulator Reference

This chapter explains how to use the **CSIM** program to create test vectors for the programmable logic device under design. Test vectors specify the expected functional operation of a PLD by defining the outputs as a function of the inputs. Test vectors are used both for simulation of the device logic before programming and for functional testing of the device once it has been programmed. **CSIM** can generate JEDEC-compatible downloadable test vectors.

Input Files

A test specification source file (***filename*.SI**) is the input to **CSIM**. It contains a functional description of the requirements of the device in the circuit.

The source file may be created using a standard text editor like DOS EDIT or Windows Notepad in non-document mode.

The input pin stimuli and output pin test values entered in the source file are compared to the actual values calculated from the logic equations in the CUPL source file. These calculated values are contained in the absolute file (***filename*.ABS**), which is created during CUPL operation when the **-a** flag on the command line is specified. The absolute file must be created during CUPL operation before running **CSIM**.

CSIM must also be able to access the device library file, **CUPL.DL**, which contains a description of each of the target devices supported in the current version of **CSIM**.

The library describes the physical characteristics of each device, including internal architecture, number of pins, and type of registers available, and the logical characteristics, including registered and non-registered pins, feedback capabilities, register power-on state and register control features.

Reference the target device using device mnemonics. Each mnemonic is composed of a device family prefix and industry-standard part number suffix. Table 2-1 lists the device mnemonic prefixes.

Output Files

The simulator output is the following two files: a simulation listing file and an optional JEDEC downloadable fuse link file.

A simulation listing file (***filename*.SO**) contains the results of the simulation. It has the same filename as the input test specification file.

All header information is displayed in the listing file with any header errors marked appropriately. Each complete vector is assigned a number. Any output tests that failed are flagged with the actual (simulator-determined) output value displayed. Each variable in error is listed along with the expected (user-supplied) value. Any invalid or unexpected test values are listed along with an appropriate error message.

The simulator output listing can also be output to the screen (using the **-v** option on the command line).

An optional JEDEC downloadable fuse link file (***filename*.JED**) contains structured test vectors. **CSIM** appends the test vectors to an existing ***filename*.JED** created during CUPL operation.

CSIM does not support multi-device files as does CUPL. CSIM only simulates the first device of a multi-device file.

Virtual Simulation

Virtual simulation allows you to create a design without a target device and simulate it. It is possible, therefore, to get a working design before deciding what architecture it will be targeted to. This will be especially useful for designs that will be eventually partitioned or that require a fitter.

Usage of the virtual simulator is transparent. When you simulate any design, CSIM will examine what the device is and simulate the design accordingly. You do not need to learn any new commands or syntax. Just use the VIRTUAL device mnemonic when compiling and simulating to take advantage of the Virtual simulator.

Virtual simuation is also used to simulate FPGA designs. When a full architectural simulation is not possible due to the proprietary nature of the device internals or the level of complexity of the internal logic resources, Virtual simulation is the next best alternative for your design verification phase.

Running CSIM

The command line for CSIM is

csim [-flags] [library] [device] source

where

-flags is the following set of simulator options:

-l create listing file.

-j append test vectors to JEDEC file.

-n use source filename for JEDEC file.

-v display simulation results to terminal.

-u use specified library for simulation.

library is the library name and path name if the **-u** flag is being used to specify a library other than the default library.

device must be the same device mnemonic as was used in the CUPL compilation. Specifying the device is optional; if a device is not specified, **CSIM** uses the device CUPL compiled (contained in the **.ABS** file).

source is the user-created ASCII test specification file (***filename*.SI**). The extension **.SI** is assumed for the source file and may be omitted when giving the **CSIM** command.

The square brackets indicate optional items.

Simulator Option Flags

Multiple option flags can be specified when running **CSIM**. A hyphen must be used before the first flag entered, but can be omitted for subsequent flags. Spaces may also be placed between the flags. For example, the following two **CSIM** command lines are equivalent:

csim -l -v -j p16r4 waitgen Return

csim -lvj p16r4 waitgen Return

CSIM can be typed without any flags, to see the command line format and a list of the option flags.

Table 7-1 lists descriptions of the **CSIM** option flags.

Table 7-1. Simulator Option Flags

Option Flag	Description
-j	Appends the structured test vectors generated by the simulation onto the existing JEDEC download file.
-l	Generates a simulation listing file (*filename*.SO.) The input and output values for each variable are listed. Error messages are listed following each vector, with the signal name in error displayed.
-n	Allows the source filename to be used as the JEDEC filename instead of using the name in the NAME field of the source file.
-v	Displays the contents of the listing file to the screen. When the simulation data begins to appear on the screen, type Ctrl-S to stop the display (and any key to start it again) or Ctrl-C to cancel the simulation.
-u	Overrides the default device library specified in the environment. Specify the complete path and library name. This option is of particular use on systems that have special libraries created for unique or custom devices.

Header Information

Header information which is entered must be identical to the information in the corresponding **CUPL** logic description file. If any header information is different, a warning message appears, stating that the status of the logic equations could be inconsistent with the current test vectors in the test specification file. Table 7-2 lists the keywords used for header information.

Table 7-2. CSIM Header Keywords

PARTNO	NAME
REVISION	DATE
DESIGNER	COMPANY
ASSEMBLY	LOCATION
DEVICE	FORMAT

When creating a test specification file, begin by copying the contents of the corresponding **CUPL** source file to the test specification file, to assure proper header information. Then delete everything except the header information from the test specification file.

Comments

Comments can be placed anywhere within the test specification file. Comments can be used to explain the contents of the specification file or the function of certain test vectors. A comment begins with a slash-asterisk (**/***) and ends with an asterisk-slash (***/**). Comments can span multiple lines and are not terminated by the end of a line. However, comments cannot be nested.

Statements

CSIM provides the keywords, **ORDER**, **BASE**, and **VECTORS** to write statements in the source file that determine the simulation output and how it is displayed. The following sections describe how to write statements with the CUPL keywords.

ORDER Statement

Use the **ORDER** keyword to list the variables to be used in the simulation table, and to define how they are displayed. Typically, the variable names are the same as those in the corresponding **CUPL** logic description file.

Place a colon after **ORDER**, separate each variable in the list with a comma, and terminate the list with a semicolon. The following is an example of an **ORDER** statement:

```
ORDER: inputA, inputB, output ;
```

Only those variables that are actually used in the simulation must be listed.

The polarity of the variable name can be different than was declared in the **CUPL** logic description file, allowing simulation of active-LO outputs with an active-HI simulation vector. The variable names can be entered in any order; **CSIM** automatically creates the proper order and polarity of the resulting vector to match the requirements of the JEDEC download format for the device.

When indexed variables are used in the **ORDER** statement, they can be expressed in list notation format. However, since the **ORDER** statement is already in list form, square brackets are not needed to delimit the **ORDER** set. The following is an example of two equivalent **ORDER** statements; the first statement lists all the variables, and the second is written in list form.

```
ORDER: A0, A1, A2, A3, SELECT, !OUT0, !OUT1;
ORDER: A0..3, SELECT, !OUT0..1 ;
```

In list notation format, the polarity of the first indexed variable (**!OUT0** in the above example) determines the polarity for the entire list.

Bit fields that are declared in the **CUPL** logic description file can be referenced by their single variable name. Bit fields can also be declared in the test specification file for **CSIM**, using **FIELD** declaration statements (see Bit Field Declaration Statements). The **FIELD** statement must appear before the **ORDER** statement.

The **ORDER** statement can be used to specify the format of the vector results in the simulator listing file (or on the screen if screen output is specified.) By default, variable values are displayed without spaces between columns. For example, the following **ORDER** statement

```
ORDER: clock, input, output ;
```

generates the following display in the output file (using sample values):

0001: C0H
0002: C1L

Spaces can be inserted between columns by using the **%** symbol and a decimal value between **1** and **80**. For example, the following **ORDER** statement

ORDER: clock, %2, input, %2, output ;

generates the following display in the output file:

0001: C 0 H
0002: C 1 L

> The ORDER statement must be terminated by a semicolon.

Text can be inserted into the output file by putting a character string, enclosed by double quotes (**“ ”**,) into the **ORDER** statement. (Do not place text in the **ORDER** statement if waveform output will be used.) For example, the following **ORDER** statement

ORDER: “Clock is ”, clock,
“ and input is ”, input,
“ output goes ”, output ;

produces the following result in the output file:

0001: Clock is C and input is 0 output goes H
0002: Clock is C and input is 1 output goes L

BASE Statement

In most cases, each variable in the **ORDER** statement (except for **FIELD** variables) has a corresponding single character test value that appears in the test vector table of the output file. Multiple test vector values can be represented with quoted numbers. Use single quotes for input values and double quotes for output values. Enter a **BASE** statement to specify how each quoted number is expanded. The format for the **BASE** statement is:

BASE: name;

where

name is either octal, decimal or hex.

Follow **BASE** with a colon.

The base statement must be terminated by a semicolon.

The default base for quoted test values is hexadecimal. The **BASE** statement must appear in the file before the **ORDER** statement.

If the base is decimal or hexadecimal, quoted numbers expand to four digits; if the base is octal, they expand to three digits. For example, a test vector entered as '7' is interpreted as follows:

1 1 1 Base is octal

or

0 1 1 1 Base is decimal

or

0 1 1 1 Base is hex

More than one hexadecimal or octal digit may be entered between quotes. For example, '563' expands to the following:

1 0 1 1 1 0 0 1 1 Base is octal

or

0 1 0 1 0 1 1 0 0 0 1 1 Base is decimal

or

0 1 0 1 0 1 1 0 0 0 1 1 Base is hex

Quoted values may also be used with all other test values. For example, if the base is set to octal

“XX” expands to X X X X X X

“LL” expands to L L L L L L

“45” expands to H L L H L H

Quoted values cannot contain *.

Test values for **FIELD** variables can be expressed either individually (for example, 001, HHLL) or with quoted values (for example, '1', “C”). When quoted values are used, the value is automatically expanded to the number of variables in the field. For example, for the following address field

```
FIELD address = [A0..5] ;
```

A test value of

```
/*
    A     A     A     A     A     A
    5     4     3     2     1     0
    ---------------------------------*/
    1     1     1     0     0     1
```

could be written using single test values, or

```
'39'
```

using quoted test values.

VECTORS Statement

Use the **VECTORS** keyword to prefix the test vector table. Following the keyword, include test vectors made up of single test values or quoted test values (see the subtopic, Base Statement in this chapter). Each vector must be contained on a single line. No semicolons follow the vector. Table 7-3 lists allowable test vector values.

Table 7-3. Test Vector Values

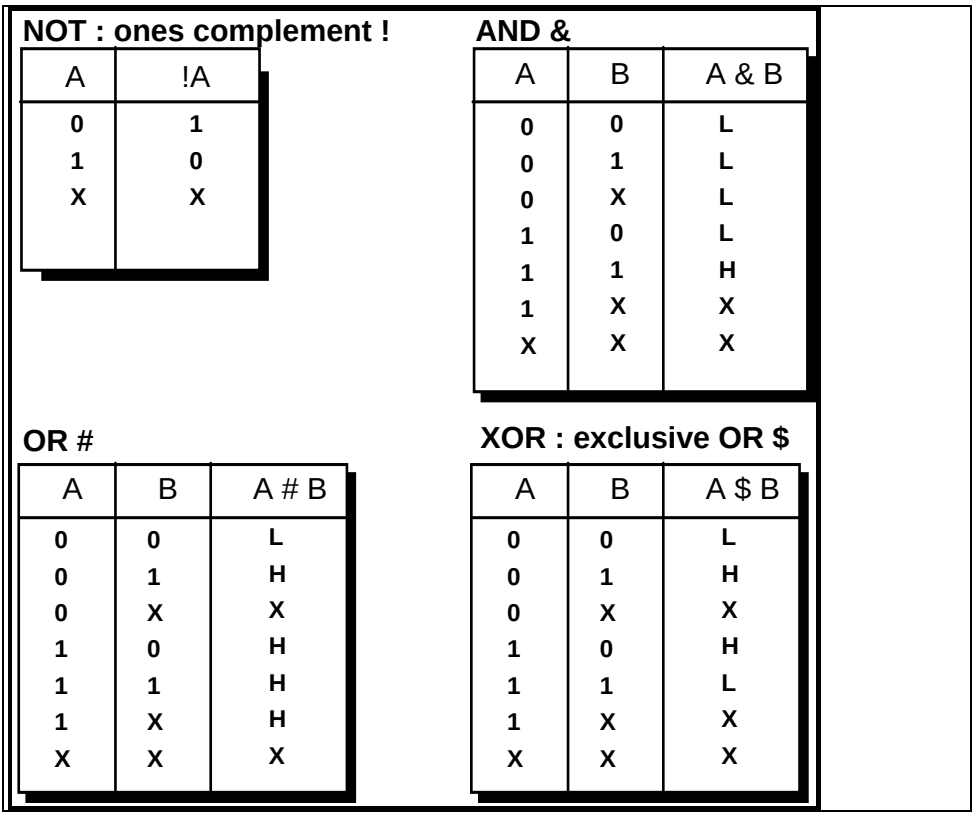
Test Value	Description
0	Drive input LO (0 volts) (negate active-HI input)
1	Drive input HI (+5 volts) (assert active-HI input)
C	Drive (clock) input LO, HI, LO
K	Drive (clock) input HI, LO, HI
L	Test output LO (0 volts) (active-HI output negated)
H	Test output HI (+5 volts) (active-HI output asserted)
Z	Test output for high impedance
X	Input HI or LO, output HI or LO.

	Note: Not all device programmers treat X on inputs the same; some put it to 0, some allow input to be pulled to 1, and some leave it at the previous value.
N	Output not tested
P	Preload internal registers (value is applied to !Q output)
*	Outputs only -simulator determines test value and substitutes in vector
' '	Enclose input values to be expanded to a specified BASE (octal, decimal, or hex). Valid values are 0-F and X.
“ ”	Enclose output values to be expanded to a specified BASE (octal, decimal, or hex.) Valid values are 0-F, H, L, Z, and X.

The following is an example of a test vector table:

```
VECTORS:
0 0 1 1 1 'F' Z “H”     /* test outputs HI */
0 1 1 0 0 '0' Z “L”     /* test outputs LO*/
```

Unlike many other simulators, **CSIM** treats the DON'T-CARE (state X) as any other value. State X is not assumed to be 0 on input and N on the output. The X state allows specific determination of which inputs affect the output value, according to the rules listed in the truth tables in Figure 7-1.



NOT : ones complement !

A	!A
0	1
1	0
X	X

AND &

A	B	A & B
0	0	L
0	1	L
0	X	L
1	0	L
1	1	H
1	X	X
X	X	X

OR #

A	B	A # B
0	0	L
0	1	H
0	X	X
1	0	H
1	1	H
1	X	H
X	X	X

XOR : exclusive OR $

A	B	A $ B
0	0	L
0	1	H
0	X	X
1	0	H
1	1	L
1	X	X
X	X	X

Figure 7-1. Vector Truth Tables

Preload

Use the **P** test value on the clock pin of a registered device to preload internal registers of a state machine or counter design to a known state, if the device does not have a dedicated TTL-level preload pin. The device programmer uses a supervoltage to actually load the registers. All input pins to the device are ignored and hence should be defined as X. The values that appear for registered variables are loaded into the **!Q** output of the register. These values (0 or 1) are absolute levels and are not affected by output polarity nor inverting buffers. The following is an example of a preload sequence for an active-

LO output variable in a device with an inverting buffer between the register Q output and device pin:

```
ORDER: clock, input1, input2 , !output ;
VECTORS:
P X X 1     /* reset flip-flop */
            /* !Q goes to 1 */
            /* Q goes to 0 */
0 X X H     /* output is HI due to */
            /* inverting buffer */
```

CSIM can simulate and generate preload test vectors even for devices that do not have preload capability. However, not all PLDs are capable of preload using a supervoltage. Some devices have dedicated preload pins to use for this purpose. **CSIM** does not verify whether the device under simulation is actually capable of preload because parts from different manufacturers exhibit different characteristics. Before using the preload capability, determine whether the device being tested is physically capable of supervoltage preloading.

Clocks

Most synchronous devices (devices containing registers with a common clock tied to an output pin) use an active-HI (positive edge triggered) clock. To assure proper **CSIM** operation for these devices, always use a **C** test value (not a 1 or 0) on the clock pin. For synchronous devices with an active-LO (negative edge triggered) clock, use the **K** test value on the clock pin.

Asynchronous Vectors

When writing test vectors for a circuit with asynchronous feedback, changing two test values at once can create a spike condition that produces anomalous results. It shows the diagram for a circuit with three inputs [A, B, and C] and an output at Y that feeds back.)

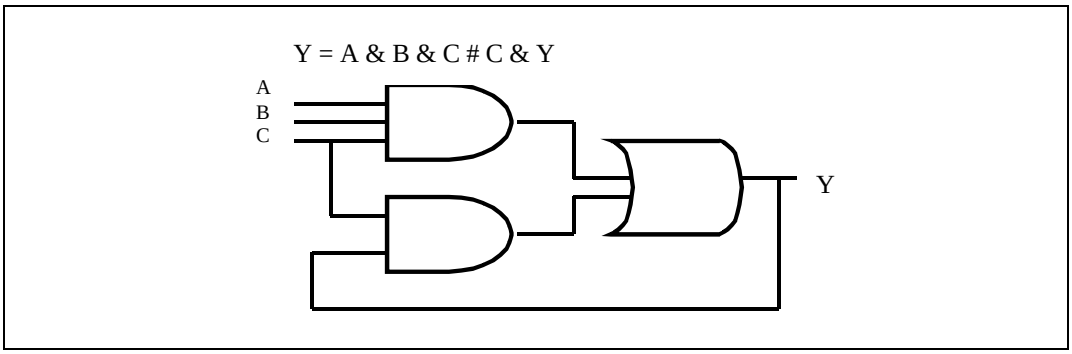


Figure 7-3. Circuit with Feedback

The equation for the output at Y is as follows:

Y = A & B & C # C & Y

The vectors table in Figure 7-4 shows an expected low output at Y based on the specified input values.

			A	B	C	Y
0001	0	0	0	L		
0002	0	1	1	L		
0003	1	0	1	L		

Figure 7-4. Vectors Table for Circuit with Feedback

Because one of the inputs is 0 in each of the vectors, the AND gate defined by A, B, and C produces a low output. The low value feeding back from the Y output keeps the other AND gate low also. Therefore, the OR gate (driven by the output of the two AND gates) and consequently the output at Y remain low for the specified test vectors.

However, when the programmer operates on the test vectors, it applies values serially, beginning with the first pin. Because two test values change between vectors, the programmer creates intermediate results (labeled “a” in Figure 7-5).

			A	B	C	Y
0001	0	0	0	L		
0001a	0	1	0	L		
0002	0	1	1	L		

0002a	1	1	1	H
0003	1	0	1	H

Figure 7-5. Vectors Table with Intermediate Results

The intermediate result, [0002a], produces a high value for the output at Y. This high value feeds back and combines with the "1" value specified for input C in vector [0003] to produce a high output for the AND gate and consequently for the OR gate and for the output at Y. This high value conflicts with the expected low value specified in the third test vector, and the result is a spike condition.

By taking care to always change only one value between test vectors, the spike condition described above can be avoided. Also, in the source specification file, it is possible to specify a **TRACE** value of 1, 2, or 3 (rather than the default value of 0) that instructs **CSIM** to display intermediate results in the output file (see "**TRACE**" in the following section, Simulator Directives).

I/O Pin simulation

When writing test vectors for a design that has input/output capability and a controllable output enable (OE), the test vector value placed at the I/O pin will depend on the value of the output enable. If the output enable is active, the I/O pin needs an output test value (L, H, *,...). If the output enable becomes inactive, a Hi-Z (Z) will appear on the I/O pin. At this time, input test values (0, 1, ...) can be placed on the I/O pin allowing that pin to behave as an input pin. When the output enable is activated again, the test values for that pin will reflect the output of the macrocell.

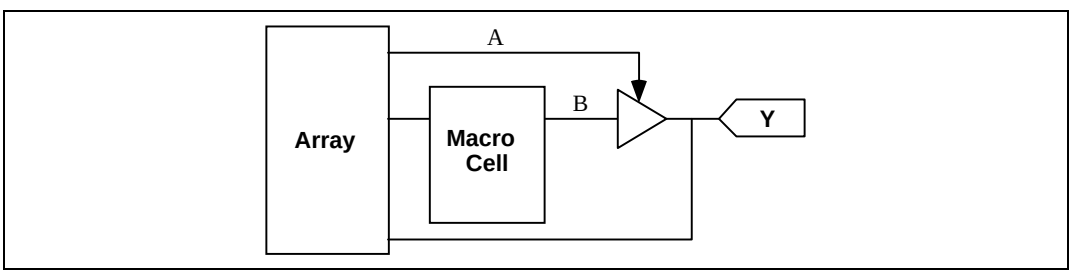


Figure 7-6. I/O Pin Simulation

The following equations express the boolean equation representation of Figure 7-6:

```
Y = B;
Y.OE = A;
```

When A is TRUE, the output of the macrocell (B) will appear at the pin (Y). When A is FALSE, the output enable will be deactivated and a Hi-Z will appear at the pin (Y). After the output enable is deactivated, input values can be placed on the pin. Here is an example of what the simulation file will look like:

```
Order:  A, %1, B, %3, Y;
Vectors:
1 0   L      /* OE is ON */
1 1   H
0 0   Z      /* OE is OFF */
0 0   1      /* a valid input value can be
                placed on pin Y */
1 0   L      /* OE is ON again */
```

Multiple ORDER statements

CSIM allows several ORDER statements to be defined in a single SI file. For example, if the file TEST.SI has the following contents:

```
Name        test;
Partno      XXXXX;
Date        XX/XX/XX;
Revision    XX;
Designer    XXXXX;
Company     XXXXX;
Assembly    XXXXX;
Location    XXXXX;
Device      g16v8;

Order: A, %1, B, %1, X, %1, Y;
Vectors:
 0 0 H L
 0 1 H H
 1 0 H H
 1 1 L L
 0 X H X
 X 0 H X
 1 X X X
 X 1 X X
Order: A, B, X;
Vectors:
 0 0 H
 0 1 H
 1 0 H
 1 1 L
 0 X H
 X 0 H
 1 X X
 X 1 X
```

Figure 7-7. TEST.SI

The file TEST.SO will look like this:

```
CSIM: CUPL Simulation Program
Version 4.2a Serial# ...
Copyright (c) 1983, 1991 Logical Devices, Inc.
CREATED Wed Dec 04 02:14:12 1991
LISTING FOR SIMULATION FILE: test.si
1: Name         test;
2: Partno       XXXXX;
3: Date         XX/XX/XX;
4: Revision     XX;
5: Designer     XXXXX;
6: Company      XXXXX;
7: Assembly     XXXXX;
8: Location     XXXXX;
9: Device       g16v8;
10:
11: Order: A, %1, B, %1, X, %1, Y;
12:
================
    A B X Y
================
0001: 0 0 H L
0002: 0 1 H H
0003: 1 0 H H
0004: 1 1 L L
0005: 0 X H X
0006: X 0 H X
0007: 1 X X X
0008: X 1 X X
25: Order: A, B, X; 26:
============
    ABX
============
0010: 00H
0011: 01H
0012: 10H
0013: 11L
0014: 0XH
0015: X0H
0016: 1XX
0017: X1X
```

Figure 7-8. TEST.SO

Random Input Generation

The value R can appear wherever a 0 or a 1 to allow CSIM to generate a random input value for the corresponding signal in that test vector.

R can only be used to generate random input values

For example if the following is used in the SI file:

```
$repeat 10;
C   0 RRR 1RRRRRRR  ********
```

CSIM generates these test vectors:

```
0035: C     0       000 10001011    HLLLHLH
0036: C     0       000 11100111    HHHLLHHH
0037: C     0       110 10111101    HHHHLHHL
0038: C     0       111 11000100    HLLLHLLH
0039: C     0       101 10001011    LHLHHHLL
0040: C     0       101 10000110    LLHHLHLL
0041: C     0       010 10000001    LHHLLLLL
0042: C     0       000 10010000    HLLHLLLL
0043: C     0       001 11110100    LHHHHLHL
0044: C     0       001 10011110    LHLLHHHH
```

Simulator Directives

CSIM provides six directives that can be placed on any row of the file after the **VECTOR** statement. All directive names begin with a dollar sign and each directive statement must end with a semicolon. Table 7-1 lists the **CSIM** directives.

Table 7-1. CSIM Directives

$MSG	$REPEAT	$TRACE
$SIMOFF	$SIMON	$EXIT

$MSG

Use the **$MSG** directive to place documentation messages or formatting information into the simulator output file. For example, a header for the simulator function table, listing the variable names, may be created. The format is as follows:

```
$MSG “any text string” ;
```

In the output table, the text string appears without the double quotes.

Blank lines can be inserted into the output, for example, between vectors, by using the following format:

```
$MSG “” ;
```

The $MSG directive can be also used to place markers in the simulator output file. The markers will be displayed on the screen at display waveform time (if the “w” flag was set). To mark a vector, place the following statement on the line preceding the vector to be marked:

```
$MSG”mark”
```

$REPEAT

The **$REPEAT** directive causes a vector to be repeated a specified number of times. Its format is:

```
$REPEAT n ;
```

where

n is a decimal value between 1 and 9999.

The vector following the **$REPEAT** directive is repeated the specified number of times.

The **$REPEAT** directive is particularly useful for testing counters and state transitions. Use the asterisk (*****) to represent output test values supplied by **CSIM**. The following example shows a a 2-bit counter from a CUPL source file, and a **VECTORS** statement using the **$REPEAT** directive to test it.

From CUPL:

```
Q0.d = !Q0 ;
Q1.d = !Q1 & Q0 # Q1 & !Q0 ;
```

In CSIM:

```
ORDER: clock, input, Q1, Q0 ;
VECTORS:
0 0 X X       /* power-on condition      */
P X 1 1       /* reset the flip-flops    */
0 0 H H
$REPEAT 4 ;          /* clock 4 times    */
C 0 * *
```

The above file generates the following test vectors:

```
0 0 X X
P X 1 1
0 0 H H
C 0 L L
C 0 L H
C 0 H L
C 0 H H
```

CSIM supplies four sets of vector values.

$TRACE

Use the **$TRACE** directive to set the amount of information that **CSIM** prints for the vectors during simulation. The format is

$TRACE n ;

where

n is a decimal value between 0 and 4.

Trace level 0 (the default) turns off any additional information and only the resulting test vectors are printed.

When non-registered feedback is used in a design, the value for the output feeding back is unknown for the first evaluation pass of the vector. If the new feedback value changes any output value, the vector is evaluated again. All outputs must be identical for two passes before the vector is determined to be stable.

Trace level 1 prints the intermediate results for any vector that requires more than one evaluation pass to become stable. Any vector that requires more than twenty evaluation passes is considered unstable.

Trace level 2 identifies three phases of simulation for designs using registers. The first phase is “Before the Clock,” where intermediate vectors using non-registered feedback are resolved. The second phase is “At the Clock,” where the values of the registers are given immediately after the clock. The third phase is “After the Clock,” where the outputs utilizing feedback are resolved as in trace level 1.

Trace level 3 provides the highest level of display information possible from **CSIM**. Each simulation phase of “Before Clock,” “At Clock,” and “After Clock” is printed and the individual product term for each variable is listed. The output value for the AND gate is listed along with the value of the inputs to the AND array.

Trace level 4 provides the ability to watch the logical value before the output buffer. Using **$TRACE 4**, **CSIM** only reports the true output pin values, and assigns a "**?**" to inputs and buried nodes. For combinatorial output, trace level 4 displays the results of the OR term. For registered outputs, trace level 4 shows the Q output of the register.

The following example uses a p22v10:

```
pin 1 = CLK;
pin 2 = IN2;
```

```
pin 3 = IN3;
....
pin 14 = OUT14;
pin 15 = OUT15;
....
OUT 14.D = IN2;
OUT 14.AR = IN3;
OUT 14.OE = IN4;
....
```

The simulation result file is:

```
ORDER CLK, IN2, IN3, IN4, . OUT14, OUT15 . ;
******before output buffer******
    ???? .. LL ...
0001:0011 .. HH ...
.....
******before output buffer******
    ????   HH...
0004 C100...ZZ
.....
```

$EXIT

Use the **$EXIT** directive to abort the simulation at any point. Test vectors appearing after the **$EXIT** directive are ignored. This directive is useful in debugging registered designs in which a false transition in one vector causes an error in every vector thereafter.

Placing a **$EXIT** command after the vector in error directs attention to the true problem, instead of to the many false errors caused by the incorrect transition.

$SIMOFF

Use the **$SIMOFF** simulator directive to turn off test vector evaluation. Test vectors appearing after the **$SIMOFF** directive are only evaluated for invalid test values and the correct number of test values. This directive is useful in testing asynchronously clocked designs in which **CSIM** is unable to correctly evaluate registered outputs.

$SIMON

Use the **$SIMON** simulator directive to cancel the effects of the **$SIMOFF** directive. Test vectors appearing after the **$SIMON** directive are evaluated fully.

Fault Simulation

An internal fault can be simulated for any product term, to determine fault coverage for the test vectors. The format for this option is as follows:

STUCKL n ;

or

STUCKH n ;

where

n is the JEDEC fuse number for the first fuse in the product term.

The documentation file (***filename*.DOC**) fuse map lists the fuse numbers for the first fuse in each product term in the device.

Format 1 forces the product term to be stuck-at-0.

Format 2 forces the product term to be stuck-at-1. The **STUCK** command must be placed between the **ORDER** and **VECTORS** statements.

Variable Declaration (VAR)

Syntax: VAR <var_name> = <var_list>;

<var_name> - string of up to 20 characters that can be letters,digits or _ (underscore), but cannot end with a digit.

<var_list> - a list of symbols from the order statement (single, grouped or fields), previously defined variables, separated by commas.

<var_list> = [!]<field> | [!]<group> | [!]<var> [..[!]<var> | ,<var_list>]

Action:
It groups all the entities contained in <var_list> under one generic name for further

references. It is similar to the FIELD statement, except this statement cannot appear before the ORDER statement. It is used between the ORDER statement and the VECTORS statement.

Example:

VAR Z = Q7..4;

All the following commands can be placed only in the test vectors section of the SI file, after the VECTORS keyword.

Assignment Statement ($SET)

Syntax: $SET <variable> = <constant>;

<variable> = <single_sym> | <field> | <defined_variable>

<constant> = <quoted_val> | <tv_string>

<quoted_val> = numbers enclosed in single/double quotes representing inputs/outputs. They will be expanded according to the base in effect and should not contain “don't care” values.

<tv_string> = string of test vector values. The number of values must be equivalent to the number of bits in the variable that they are assigned to.

Action:
It assigns a constant value to a symbol, field or variable. It takes effect immediately, but affects only the user values of the variable; the last simulation results are unchanged. Can appear anywhere in the test vector section.

Example:

```
$set input = '3F';  /*  single  quotes  for  inputs  */
$set output = "80"; /*  double  quotes  for  outputs  */
$set Z = HHHH;  /* test vector values for a 5-bit output  variable */
```

Arithmetic and Logic Operations ($COMP)

Syntax: $COMP <variable> = <expression>;

<variable> = <single_sym> | <field> | <defined_variable>

<expression> = any logic or arithmetic expression in which the operands can be variables (like above) or constants.

The allowed constants are decimal numbers (unquoted). Parentheses are permitted.

Operator	Function	Precedence
!	NOT	1
&	AND	2
#	OR	3
$	XOR	4

Table 7-2. Logic Operators

Operator	Function	Precedence
*	multiplication	1
/	division	1
+	addition	2
-	subtraction	2

Table 7-3. Arithmetic Operators

The logical and arithmetic operators can be mixed freely in an expression. Normally the logical operators have a higher precedence, however, this rule can be overridden by using parentheses.

Action:
It evaluates the expression and assigns the result to the variable. The current values of the operands (user values) are used in evaluating the expression. Takes effect immediately,

but affects only the user values of the variable; the last simulation results are unchanged. Can appear anywhere in the test vector section.

Examples:

$COMP A = (!B + C) * A + 1;
$COMP X = (Z / 2) # MASK;

Generate Test Vector ($OUT)

Syntax: $OUT;

Action:
Triggers the simulation for the current values of the symbols and generates a test vector. It is useful when used after the $set and $comp command because it allows the previously assigned values to take effect in vector evaluation.

Example:

The following set of commands in the SI file:

```
ORDER: _CLOCK, %3, _OE, %3, shift, %1, input, %2, output;
VECTORS:
0 0 'X' XXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$set shift = '0';
$set input = '80';
$set output = "80";
$out;
```

Figure 7-10. .SI File

This will produce this result in the SO file:

```
0001: 0   0   XXX XXXXXXXX  LLLLLLLL
0002: C   0   000 10000000  HLLLLLLL
```

Figure 7-11. .SO File

Conditional Simulation ($IF)

Syntax: $IF <condition> :
<block_1>

```
[ $ELSE :
<block_2> ]
$ENDIF;
```

<condition> = <var_list> <logic_operators> <constant>

logic operators :

=	equal
#	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

<constant> = <quoted_val> | <tv_string>

<block_1>,<block_2> = any sequence of statements, including test vectors

The $ELSE branch is optional.

Action:
The condition is evaluated using the current simulation value of the variable. If the result is true, <block_1> is executed; otherwise, if $ELSE is present, <block_2> is executed. $ENDIF marks the end of the IF statement.

Looping Constructs

FOR statement

Syntax:
```
$FOR <count> = <n1>..<n2> :
<block>
$ENDF;
```

<count> = the counter of the FOR loop; it takes values between <n1> and <n2>

<n1>,<n2> = limits for <count> values; should be positive decimal numbers.

<block> = any sequence of statements, including test vectors

Action:
Step 1. <count> is initialized with the first value, <n1>.
Step 2. execute <block>.
Step 3. if <count> = <n2> STOP;
otherwise <count> is incremented by 1 (if <n1> less than <n2>) or decremented by 1 (if <n1> greater than <n2>) then repeat steps 2 and 3.

WHILE Statement

Syntax: $WHILE <condition> :
<block>
$ENDW;

<condition> = same as IF condition

<block> = any sequence of statements

Action:
Step 1: Evaluate condition; if false then STOP
else continue with step 2.
Step 2: Execute <block>.
Step 3: Continue with step 1.

DO..UNTIL Statement

Syntax: $DO:
<block>
$UNTIL <condition> ;

<condition> = same as IF condition

<block> = any sequence of statements

Action:
Step 1: Execute <block>.
Step 2: Evaluate condition; if true then STOP,
else continue with step 1.

IF and repetitive statements can be nested; however, the maximum number of nested statements is 10.

MACRO and CALL Statements

Macro Definition

Syntax: $MACRO name(<arg_list>);
<macro_body>
$MEND;

name = the macro name

<arg_list> = symbolic names, separated by commas

<macro_body> = any sequence of statements, except $MACRO (including macro calls)

Argument names can appear in the macro body wherever a variable name or a constant is allowed. They cannot substitute operators, special characters or reserved words.

Macro Call

Syntax: $CALL name(<act_arg_list>);

name = the name of a previously defined macro

<act_arg_list> = actual arguments list

The actual arguments can be variable names, constants or even macro arguments, if the CALL statement is placed within a macro body.

Action:
It executes the statements that form the invoked macro body by replacing any occurrence of a macro argument with the corresponding actual argument.

In order to successfully complete a macro call, check if the actual arguments fit the syntax of the macro body, that is they won't cause a syntax error by replacing the corresponding formal argument.

Example:

```
$MACRO m1(a,b,c);              /* Macro definition */
$set shift = a;
$set shift = b;
$set output = c;
$MEND;

$CALL m1('0','80',********);   /* Macro call */
```

The following statements will be executed:

```
$set shift = '0';
$set shift = '80';
$set output = ********;
```

The following is full example of how these statements work and how they can help the user simulate his design without entering a lot of test vectors.

These two SI files produce the same output:

1. Old way:

```
Name         Barrel22;
Partno       CA0006;
Date         05/11/96;
Revision     02;
Designer     Engineer;
Company      Logical Devices, Inc.;
Assembly     None;
Location     None;
Device       g20v8a;

ORDER:  CLOCK, %3, OE, %3, shift, %1, input, %2, output;
VECTORS:
0    0 'X' XXXXXXXX  HHHHHHHH    /* power-on reset state */
C    0 '0' 10000000  HLLLLLLL    /* shift 0 */
C    0 '1' 10000000  LHLLLLLL    /* shift 1 */
C    0 '2' 10000000  LLHLLLLL    /* shift 2 */
C    0 '3' 10000000  LLLHLLLL    /* shift 3 */
C    0 '4' 10000000  LLLLHLLL    /* shift 4 */
C    0 '5' 10000000  LLLLLHLL    /* shift 5 */
C    0 '6' 10000000  LLLLLLHL    /* shift 6 */
C    0 '7' 10000000  LLLLLLLH    /* shift 7 */
C    0 '0' 01111111  LHHHHHHH    /* shift 0 */
C    0 '1' 01111111  HLHHHHHH    /* shift 1 */
C    0 '2' 01111111  HHLHHHHH    /* shift 2 */
C    0 '3' 01111111  HHHLHHHH    /* shift 3 */
C    0 '4' 01111111  HHHHLHHH    /* shift 4 */
C    0 '5' 01111111  HHHHHLHH    /* shift 5 */
C    0 '6' 01111111  HHHHHHLH    /* shift 6 */
C    0 '7' 01111111  HHHHHHHL    /* shift 7 */
```

Figure 7-12. .SI File

2. New way:

```
ORDER:  CLOCK, %3, OE, %3, shift, %1, input, %2, output;
VECTORS:
0 0 'X' XXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$set shift = '0';
$set input = '80';
$set output = "80";
$for i = 1..16 :
$out;
$if shift = '7':
$set shift = '0';
$set input = '7f';
$set output = "7f";
$else:
$comp shift = shift + 1;
$comp output = output / 2;
$if input = '7f':
$comp output = output # 128;
$endif;
$endif;
$endf;
```

Figure 7-13. .SI File.

or, using macros:

```
ORDER:  CLOCK, %3, OE, %3, shift, %1, input, %2, output;
VECTORS:
$macro m1(x,y,z);
$set shift = x;
$set input = y;
$set output = z;
$mend;

$macro m2(a,b,c,d);
$call m1(a,b,c);
$for i = 1..8 :
$out; $comp shift = shift + 1;
$comp output = output / 2 + d;
$endf;
$mend;
0 0 'X' XXXXXXXX LLLLLLLL /* power-on reset state */
$set _CLOCK = C;
$call m2('0','80',"80", 0);
$call m2('0','7f',"7f", 128);
```

Figure 7-14. .SI File

3. The Output:

```
CSIM: CUPL Simulation Program Version
4.8a Serial# ...
Copyright (c) 1983, 1997 Logical Devices, Inc.
CREATED Wed Dec 04 03:00:11 1997
LISTING FOR SIMULATION FILE: barrel22.si
1: Name   Barrel22;
2: Partno CA0006;
3: Date   05/11/96;
4: Revision      02;
5: Designer      Engineer;
6: Company       Logical Devices, Inc.;
7: Assembly      None;
8: Location      None;
9: Device g20v8a;
10:
11: FIELD input = [D7,D6,D5,D4,D3,D2,D1,D0];
12: FIELD output = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
13: FIELD shift = [S2,S1,S0];
14:
15: ORDER: CLOCK, %3, OE, %3, shift, %1, input, %2, output;
16:
17: var X = Q7;
18: var Y = Q7..4;
19:
=====================================
      C
      L
      O
      C   O   shi
      K   E   ft  input     output
=====================================
0001: 0   0   XXX XXXXXXXX  LLLLLLLL
0002: C   0   000 10000000  HLLLLLLL
0003: C   0   001 10000000  LHLLLLLL
0004: C   0   010 10000000  LLHLLLLL
0005: C   0   011 10000000  LLLHLLLL
0006: C   0   100 10000000  LLLLHLLL
0007: C   0   101 10000000  LLLLLHLL
0008: C   0   110 10000000  LLLLLLHL
0009: C   0   111 10000000  LLLLLLLH
```

Figure 7-15. .SO File

```
0010: C   0   000 01111111  LHHHHHHH
0011: C   0   001 01111111  HLHHHHHH
0012: C   0   010 01111111  HHLHHHHH
0013: C   0   011 01111111  HHHLHHHH
0014: C   0   100 01111111  HHHHLHHH
0015: C   0   101 01111111  HHHHHLHH
0016: C   0   110 01111111  HHHHHHLH
0017: C   0   111 01111111  HHHHHHHL
```

Figure 7-15. .SO File sheet 2

There is one thing the user must keep in mind when creating a simulation input file using the new syntax:

If one or more $SET or $COMP commands are placed right before some conditional statement (IF, WHILE, UNTIL) without any intermediate $OUT statement, the values set by those commands (user values) will not affect the condition value, as the condition is evaluated using the last simulation values of the variables involved.

For example, let's assume that we want to generate the following simulation output:

```
ORDER: CLOCK,clr,dir,!OE,%2,count,%1,carry;
var mode = clr,dir;
VECTORS:
C 100     LLLL L  /* synchronous clear to state 0 */
C 000     LLLH L  /* count up to state 1          */
C 000     LLHL L  /* count up to state 2          */
C 000     LLHH L  /* count up to state 3          */
C 000     LHLL L  /* count up to state 4          */
C 000     LHLH L  /* count up to state 5          */
C 000     LHHL L  /* count up to state 6          */
C 000     LHHH L  /* count up to state 7          */
C 000     HLLL L  /* count up to state 8          */
C 000     HLLH H  /* count up to state 9 - carry  */
```

Figure 7-16. Expected Output

The following sequence will generate a wrong output:

```
ORDER: CLOCK,clr,dir,!OE,%2,count,%1,carry;
var mode = clr,dir;
VECTORS:
C 100 LLLL L $set mode = '0';
$for i=1..9 :
$comp count = count + 1;
$if count="9":
$set carry = H;
$endif;
$out;
$endf;
```

Figure 7-17. Simulation Input (Incorrect)

that is:

```
0001: C 100  LLLL L
0002: C 000  LLLH L
0003: C 000  LLHL L
0004: C 000  LLHH L
0005: C 000  LHLL L
0006: C 000  LHLH L
0007: C 000  LHHL L
0008: C 000  LHHH L
0009: C 000  HLLL L
0010: C 000  HLLH H
                  ^
[0019sa] user expected (L) for carry
```

Figure 7-18. Simulation Output

This is because the value for count used in the evaluation of the IF condition for vector 10 was the current simulation value (that is the one displayed in vector 9) and not the one set by $comp command.

The correct sequence is:

```
C 100 LLLL L
$set mode = '0';
$for i=1..9 :
$if count="8":
$set carry = H;
$endif;
$comp count = count + 1;
$out;
$endf;
```

Figure 7-19. Simulation Input (Correct)

8. Design Example

In this design a TC74HC138 IC will be implemented in a pld. It is a 3-8 line decoder. If the device is enabled, 3 binary select inputs (A, B, C) determine which of the outputs go low. If the enable input G1 is low or either of the active-low inputs G2A or G2B are high, decoding is inhibited and the 8 outputs will go to a tristate level.

Step 1: Create the PLD file from template

The first step is to create the pld file file using design template. This dialog is launched from the File New menu.

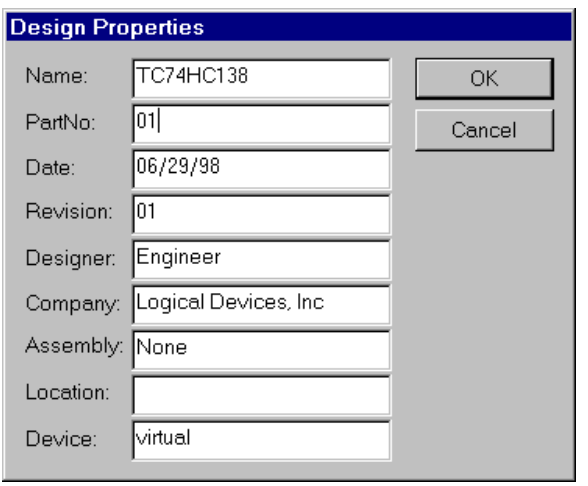


Figure 1A New Design File Template

After filling in the header information we specify that our design has six inputs and eight outputs with no pinnodes the following is loaded into the WinCUPL editor.

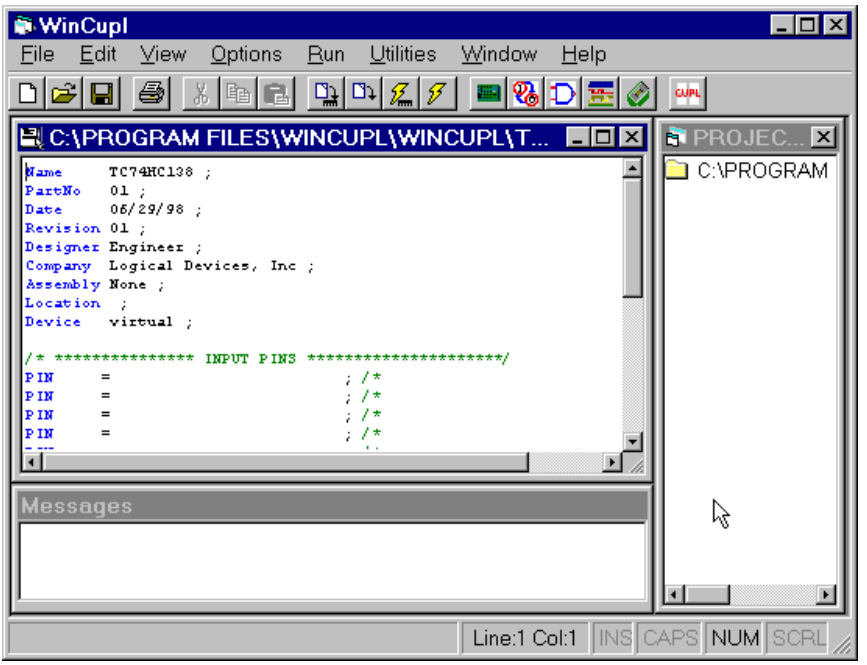


Figure 2A Design File

General editor settings such as font size and highlighting can be customized from the Options menu.

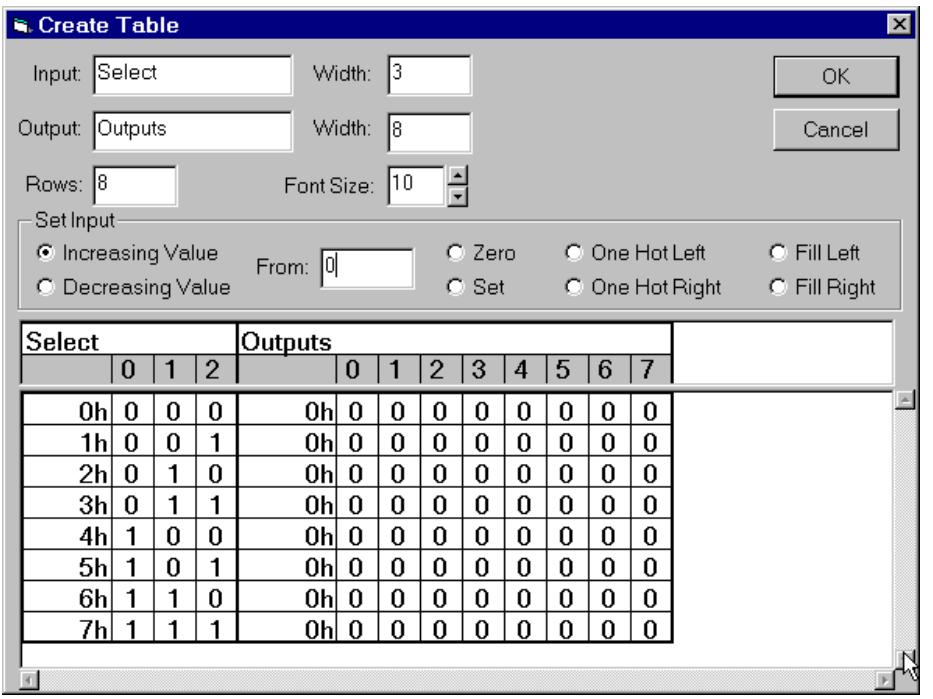
We now enter our pin declarations as shown below.

```
/* *************** INPUT PINS *********************/
PIN    =   G1              ; /*  Enable Input        */
PIN    =   !G2A            ; /*  Enable Input        */
PIN    =   !G2B            ; /*  Enable Input        */
PIN    =   A               ; /*   Binary Input       */
PIN    =   B               ; /*                      */
PIN    =   C               ; /*                      */

/* *************** OUTPUT PINS *********************/
PIN    =    ![Y0..Y7]              ; /* Selected Output          */
```

Step 2: Create the Binary Truth Table

For this design we will create a binary truth table. This is done by selecting Insert Table from the Edit menu. We will assign our select inputs the field name Select and our Y0..7 output the field name Outputs. The number of rows is eight and we want our input field to increase by one starting at zero. The complete dialog is shown in Figure 3A.



Select				Outputs								
	0	1	2		0	1	2	3	4	5	6	7
0h	0	0	0	0h	0	0	0	0	0	0	0	0
1h	0	0	1	0h	0	0	0	0	0	0	0	0
2h	0	1	0	0h	0	0	0	0	0	0	0	0
3h	0	1	1	0h	0	0	0	0	0	0	0	0
4h	1	0	0	0h	0	0	0	0	0	0	0	0
5h	1	0	1	0h	0	0	0	0	0	0	0	0
6h	1	1	0	0h	0	0	0	0	0	0	0	0
7h	1	1	1	0h	0	0	0	0	0	0	0	0

Figure 3A Truth Table Dialog

The following code is then placed in the design file at the last cursor location before the Truth Table Dialog is launched.

```
FIELD Select = [A,B,C];
```

```
FIELD Outputs = [Y0..7];

TABLE Select => Outputs {
   'b'000 => 'b'00000000;
   'b'001 => 'b'00000000;
   'b'010 => 'b'00000000;
   'b'011 => 'b'00000000;
   'b'100 => 'b'00000000;
   'b'101 => 'b'00000000;
   'b'110 => 'b'00000000;
   'b'111 => 'b'00000000;}
```

Step 3: Set Binary Truth Table Values

The next step is to fill the table with the decoder values. The completed table is shown below for this design.

```
TABLE Select => Outputs {
   'b'000 => 'b'00000001;
   'b'001 => 'b'00000010;
   'b'010 => 'b'00000100;
   'b'011 => 'b'00001000;
   'b'100 => 'b'00010000;
   'b'101 => 'b'00100000;
   'b'110 => 'b'01000000;
   'b'111 => 'b'10000000;}
```

Step 4: Assign Output Enables

We need to assign the output enable terms for the Y0..7 values so that if G1 goes low or G2A or G2B goes high (False) they will be tristated. Notice that since we declared G2A and G2B to be active-low we use the inversion symbol to assign the False state (H) to the equation.

```
[Y0..7].OE = !G1 # !G2A # !G2B;
```

Step 5: Compile The Design

We now use the Options menu to specify the compiler options. Since we are targetting a virtual device we will only specify the expanded product terms (.doc) file and a listing file (.lst) file to view any errors that are in the file.

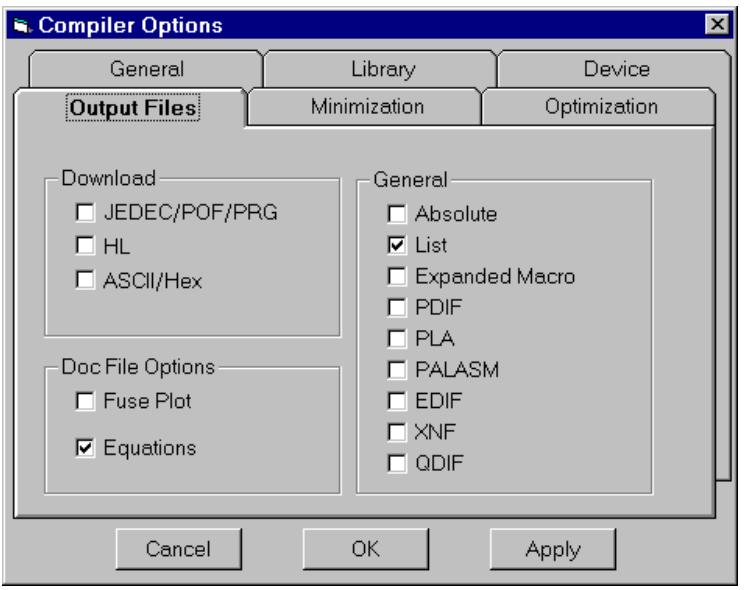


Figure 4A Compiler Options Dialog

We then successfully compile the file. The documentation file generated is listed below.

```
*******************************************************************************
                                    TC74HC138
*******************************************************************************

CUPL(WM)        4.9a Serial# 00000000
Device          virtual  Library DLIB-h-39-1
Created         Mon Jun 29 11:16:42 1998
Name            TC74HC138
Partno          01
```

```
Revision        01
Date            06/29/98
Designer        Engineer
Company         Logical Devices, Inc
Assembly        None
Location

================================================================================
                            Expanded Product Terms
================================================================================

Outputs =>
    Y0 , Y1 , Y2 , Y3 , Y4 , Y5 , Y6 , Y7

Select =>
    A , B , C

Y0 =>
    !A & !B & !C

Y0.oe  =>
    !G1
  # !G2A
  # !G2B

Y1 =>
    !A & !B & C

Y1.oe  =>
    !G1
  # !G2A
  # !G2B

Y2 =>
    !A & B & !C

Y2.oe  =>
    !G1
  # !G2A
  # !G2B

Y3 =>
    !A & B & C

Y3.oe  =>
    !G1
  # !G2A
```

```
  # !G2B

Y4 =>
    A & !B & !C

Y4.oe  =>
    !G1
  # !G2A
  # !G2B

Y5 =>
    A & !B & C

Y5.oe  =>
    !G1
  # !G2A
  # !G2B

Y6 =>
    A & B & !C

Y6.oe  =>
    !G1
  # !G2A
  # !G2B

Y7 =>
    A & B & C

Y7.oe  =>
    !G1
  # !G2A
  # !G2B


================================================================================
                                  Symbol Table
================================================================================

Pin Variable                                      Pterms   Max     Min
Pol   Name              Ext     Pin     Type      Used   Pterms  Level
--- --------            ---     ---     ----    ------  ------  -----

    A                           0        V        -       -       -
    B                           0        V        -       -       -
    C                           0        V        -       -       -
    G1                          0        V        -       -       -
```

```
 !  G2A                          0        V        -       -       -
 !  G2B                          0        V        -       -       -
    Outputs                      0        F        -       -       -
    Select                       0        F        -       -       -
 !  Y0                           0        V        1       0       1
    Y0                     oe    0        X        3       0       1
 !  Y1                           0        V        1       0       1
    Y1                     oe    0        X        3       0       1
 !  Y2                           0        V        1       0       1
    Y2                     oe    0        X        3       0       1
 !  Y3                           0        V        1       0       1
    Y3                     oe    0        X        3       0       1
 !  Y4                           0        V        1       0       1
    Y4                     oe    0        X        3       0       1
 !  Y5                           0        V        1       0       1
    Y5                     oe    0        X        3       0       1
 !  Y6                           0        V        1       0       1
    Y6                     oe    0        X        3       0       1
 !  Y7                           0        V        1       0       1
    Y7                     oe    0        X        3       0       1


LEGEND    D : default variable         F : field      G : group
          I : intermediate variable    N : node       M : extended node
          U : undefined                V : variable   X : extended
variable
          T : function
```

Step 6: Create Simulation File

The next step is to create a simulation file to verify the logic of the design. We do this by launching the WinSim editor. Upon start-up no simulation input file will be found by the simulator. Selecting File New and seelcting our design file will fill in the header information from the pld file. The signals are then added to the simulator database.

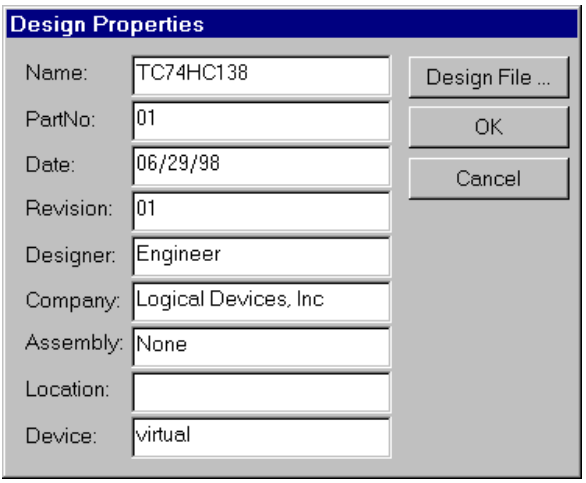


Figure 5A Simulator Header Information

A blank simulator window will then be displayed.

Step 7: Add Simulation Signals And Vectors

The next step is to add the signals we want to test to the simulation file. We do this by selecting Signal Add from the menu. This displays the Add Signal dialog. Signals that are not being currently displayed can be selected. Buses are identified by a '*" at the end of the field name. These fields must be declared in the pld file. Individual signals and the bus that contains them can be displayed.

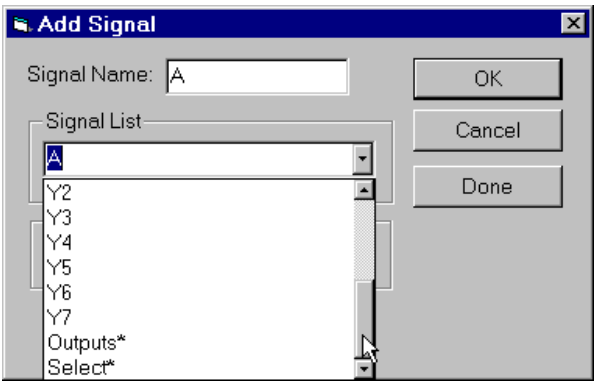


Figure 6A Add Signal Dialog

After selecting the signals to be tested the number of vectors to be displayed is then selected. For this example we will use twelve. This is done by selecting Signal Add Vector and entering the number of vectors.

Step 8: Specifying Simulation Values

The value of an individual signal can be specified by clicking on areas of the cell. To make this easier right clicking on the mouse when in the signal display window will display the available selections.

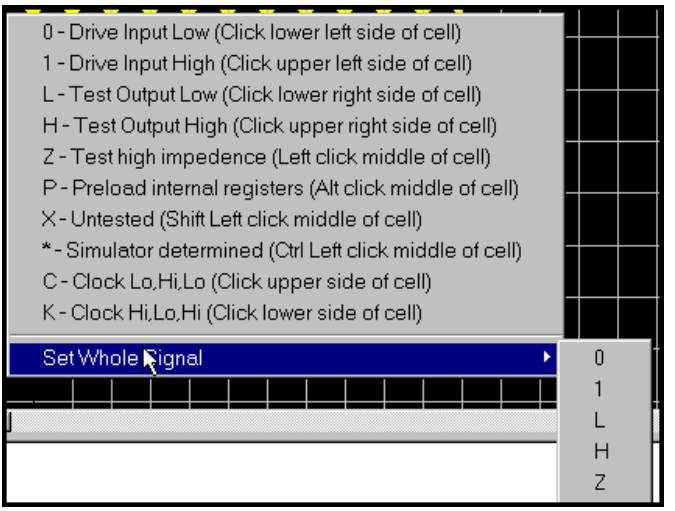


Figure 7A Setting Signal Values

Bus values can be set by expanding the bus signal (right click in name box) and setting the indivual signals or selecting that vector on the ruler and changing the value.

The finished vectors are displayed below.

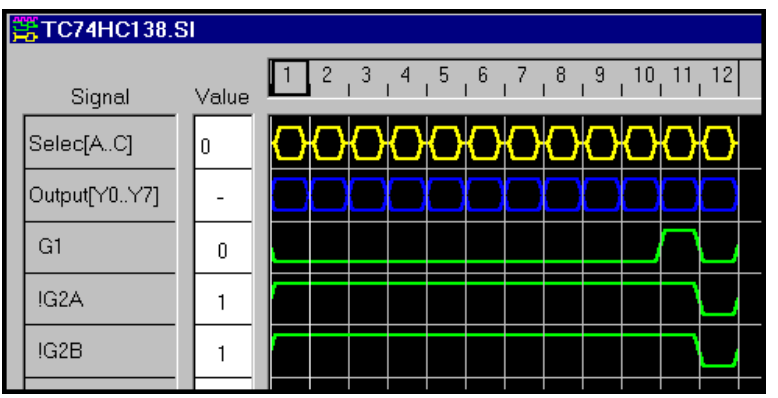


Figure 8A Vector Display

The output values can be left Simulator Determined (*) and those results assigned to the simulation file after the simulation is run.

Step 9: Examine Results

The simulator is then run from the Simulator menu. In this example we allowed the simulator to determine the Output field values. If there had been an error on a vector it would be displayed in red. Individual signals can be examined by selecting the View Signal Definitions and selecting the output signal. The Signal Definitions box will display the reduced equations for that signal to determine the cause of the error.

9. Sample Pld Files

This chapter lists the logic description files that are included in the CUPL package to illustrate how **CUPL** and **CSIM** implement various designs.

FILE: ADDER.PLD
DEVICES: PAL16L8, PAL16P8, 82S153
5-bit asynchronous adder implemented as a ripple-carry through four adder-slice circuits. Each adder-slice was implemented using a user-defined function.

FILE: ADDER_TT.PLD
DEVICES: RA9P8 (512x8 PROM)
5-bit asynchronous adder implemented using a truth table. Makes use of nested $REPEAT statements.

FILE: BARREL22.PLD
DEVICES: PAL22V10
8-bit registered barrel shifter with synchronous presetting capability.

FILE: BUSARB.PLD
DEVICES: 82S105
Multiprocessor bus arbiter having two machines in one design.

FILE: COUNT8.PLD
DEVICES: PAL20X8
8-bit counter with parallel load, clear, and hold using XOR capability.

FILE: COUNT8A.PLD
DEVICES: PAL20X8
8-bit counter with parallel load, clear, and hold using set notation.

FILE: COUNT10.PLD
DEVICES: PAL16RP4, GAL16V8
5-bit up/down decade counter with synchronous clear capability. An asynchronous ripple carry output is provided for cascading multiple devices.

FILE: COUNT13.PLD
DEVICES: PAL32R16
13-bit counter using set notation with parallel load hold and clear.

FILE: CYP_CNT.PLD
DEVICES: CY7C330
Up/Down counter with preloadable upper and lower limits.

FILE: DATASEP.PLD
DEVICES: EP600
Single density 8” floppy disk data separator.

FILE: DECADE.PLD
DEVICES: 82S157
5-bit synchronous free-running decade counter that uses the complement-array to force invalid states to reset the counter registers. State machine syntax is used.

FILE: FLOP.PLD
DEVICES: PAL16R8, PAL16RP8, 82S159
Using D-type flip-flop to create a 2-bit counter (four ways).

FILE: GATES.PLD
DEVICES: PAL16L8, PAL16P8 , 82S153
Simple use of NOT, AND, OR, and XOR gates.

FILE: HEXDISP.PLD
DEVICES: RA5p8 (32x8 PROM)
Hexidecimal to 7-segment decoder used for displaying numbers.

FILE: IODECODE.PLD
DEVICES: PAL12L6 , PAL12P6, 82S153
A chip select signal generator for I/O functions. It also enables the data bus transceiver for both memory and I/O write cycles.

FILE: IOPORT.PLD
DEVICES: PAL20RA10
7-bit register with handshake logic used to interface between a microprocessor and I/O port.

FILE: KEYBOARD.PLD
DEVICES: 82S100
Converts the rows and columns of a matrix keyboard and generates the corresponding ASCII code required for the key.

FILE: LOOKUP.PLD
DEVICES: RA8P8 (256 x 8 EPROM)
Arithmetic lookup table that calculates the perimeter of a circle given the radius. Truth table syntax is used.

FILE: MDECODE.PLD
DEVICES: PAL16L8, PAL16P8 , 82S153
A memory RAS generator and CAS signal initiator. It also enables the data bus transceiver for both memory and I/O read cycles.

FILE: MULTIBUS.PLD
DEVICES: PAL23S8
Simple MULTIBUS arbiter supports parallel and serial priority.

FILE: PRIORITY.PLD
DEVICES: PALR19L8
Priority Interrupt Encoder for the Motorola 68000 using both Boolean equations and Conditional syntax. The use of input registers is shown.

FILE: RIPPLE8.PLD
DEVICES: PAL20RA10
8-bit ripple counter with asynchronous load.

FILE: SHFTCNT.PLD
DEVICES: 82S105
5-bit counter/shifter using SR-type flip-flops.

FILE: SHFTCNT4.PLD
DEVICES: 82S159
5-bit counter/shifter using JK-type flip-flops.

FILE: SHFTCNT6.PLD
DEVICES: 82S167
5-bit counter/shifter using SR-type flip-flops.

FILE: STEPPER.PLD
DEVICES: PALT19R6
Memory mapped stepper motor controller interfaced to the 8048 single chip microprocessor.

FILE: TCOUNTER.PLD
DEVICES: EP600
16-bit up/down counter with built-in shift register using toggle flip-flops.

FILE: TTL.PLD
DEVICES: PAL16L8
Multiple TTL chip representation using $Macros from the $Include file.

Any of these logic description files can be viewed or printed out, or they can be input to **CUPL** to generate documentation or download files. A corresponding test specification file (***filename*.SI**) is also provided for each logic description file, so that **CSIM** can be run to verify the designs.

The following examples describe key points of the following designs (the logic description file for each design is shown in parentheses):

- Simple gates (GATES.PLD)
- TTL conversion (WGTTL.PLD)
- Two-bit counter (FLOPS.PLD)
- Decade up/down counter using state-machine syntax (COUNT10.PLD)
- Seven-segment display decoder (HEXDISP.PLD)

10. Trouble Shooting

Contacting Customer Support

Before contacting Customer Support, make sure to collect the following information:

Make sure that you have a semicolon at the end of each statement. You would be surprised at the number of files we get that contain only this problem. Examine the header section of the PLD file in particular since a missing semicolon in this area will often cause strange results.

Check to make sure that all comment blocks are closed. Many times designers start a comment with /* but forget to close it with */. What happens is that the compiler continues reading until it finds an end of comment marker */. Everything read is considered a comment and is therefore invisible to the compiler.

- The CUPL serial number
- The CUPL version number
- The device mnemonic

By E-Mail www.logicaldevices.com

By Telephone (303 279-6868

By FAX (303) 278-6868

11. Error Messages

List all error messages with common corrections.

CUPL error messages are intended to be self-explanatory. This appendix provides additional information describing them.

Some of the CUPL programs, such as **CUPL** and **CSIM**, are composed of individual modules. Error messages are numbered and listed according to the program and module in which they occur. The suffix to the error message number identifies the program and module.

Module	Suffix
CUPL processor	ck
CUPLX preprocessor	cx
CUPLA source file parser	ca
CUPLB equation fitter	cb
CUPLM minimizer	cm
CUPLC fusemap generator	cc
CSIM processor	sk
CSIMA logic simulator	sa

This appendix lists the error messages by modules in the same order as they appear in the table above. The error messages within each module are listed in numerical order.

CUPL provides three levels of error messages: warnings, errors, and fatals.

warnings — do not prevent CUPL from continuing, but indicate a problem that should be corrected.

errors — allow CUPL to continue but must be corrected before future compiles.

fatals — prevent CUPL from continuing and must be corrected.

Error messages with indexes greater than 1000 are program errors. This section does not individually list program errors. Possible causes for program errors are bad data in a source file caused by disk errors or word processors in document mode; or previous errors continuing to propagate unexpected circumstances. If the cause of a program error cannot be determined, gather as much information as possible on the conditions in effect when the error occurred, then call CUPL support.

Error messages report the line number on which the error was detected; however, the cause of the error may be on a previous line. If the message doesn't seem to apply to the reported line, look at preceding lines for the source of the error.

CUPL ERROR MESSAGES

This section describes the errors for the **CUPL**, **CUPLX**, **CUPLA**, **CUPLB**, **CUPLM**, and **CUPLC** modules.

CUPL Module Error Messages

0001ck could not open: “filename”

Fatal. CUPL cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002ck could not execute program: “program name”

Fatal. CUPL is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003ck could not find PATH in ENVIRONMENT

Fatal. The PATH assignment has not been made in the ENVIRONMENT.

0004ck could not find LIBCUPL in ENVIRONMENT

Fatal. The LIBCUPL assignment has not been made in the ENVIRONMENT.

0005ck could not find program: “program name”

Fatal. CUPL is unable to locate the CUPL programs using the PATH in the ENVIRONMENT.

0006ck insufficient memory to execute program: “filename”

Fatal. Not enough program storage available to load and execute the program. Refer to Chapter 1, “Introduction,” for the minimum memory requirements for the configuration being used.

0007ck invalid flag: “option flag”

Fatal. The option flag specified is not one of the allowable compilation flags. Verify proper command line flags and syntax as discussed in Chapter 2, “Using CUPL.”

0008ck out of memory: “condition”

Fatal. CUPL has used all available RAM memory which has been allocated by the operating system. Check for the existence of print spoolers, RAM disks, or other

memory-resident programs which may decrease the amount of memory available to the CUPL application.

0009ck file read error, unexpected end of file: "filename"

Fatal. CUPL encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media or the file has been corrupted.

0010ck Fitter could not fit design

Fatal. The external fitter has determined that it cannot fit the specified design.

0011ck Fatal fitter error during processing

Fatal. A fatal error occurred while executing the external fitter.

0012ck invalid library access key

Fatal. This version of CUPL is not compatible with the version of the device library file. This occurs when either CUPL or the device library, but not both, has been updated.

0013ck invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPL and the device library are not compatible.

0014ck bad library file: "filename"

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0015ck device not in library: "device"

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0016ck target device not specified

Fatal. The user did not specify a target device on the command line and the source file did not contain a DEVICE assignment in the header information.

10xxck program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact Logical Devices, Customer Support.

CUPLX Module Error Messages

0001cx could not open: “filename”

Fatal. CUPLX cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cx could not execute program: “program name”

Fatal. CUPLX is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cx no label given for command

Error. One of the preprocessor commands, $DEFINE, $UNDEF, $IFDEF, or $IFNDEF, was used without a succeeding label.

0004cx already defined: “label”

Error. The label was previously defined using $DEFINE. To redefine the label, first use $UNDEF to undefine the label, and then use $DEFINE to redefine it.

0005cx string error

Fatal. All preprocessor label string space has been used.

0006cx $else without $ifdef

Error. An $ELSE preprocessor command was used without being preceded by an $IFDEF or $IFNDEF command.

0007cx $endif without $ifdef

Error. An $ENDIF preprocessor command was used without being preceded by an $IFDEF or $IFNDEF command.

0008cx $ifdef nesting too deep

Error. The level of $IFDEF nesting exceeded twelve.

0009cx missing $endif

Error. An $IFDEF preprocessor command was used without being succeeded by an $ENDIF command.

0010cx invalid preprocessor command: “$command”

Error. The preprocessor command is unknown. Refer to Preprocessor Commands in Chapter 2 for a list of valid commands.

0011cx disk write error: “filename”

Fatal. CUPLX encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0012cx out of memory: “condition”

Fatal. CUPLX has used all the available RAM memory allocated by the operating system.

0013cx illegal character: “hex value”

Error. CUPLX has encountered an illegal ASCII value in the source file. Make sure the file was created in non-document mode on the word processor. This error can also be caused by files which were created over a serial modem upload/download link.

0014cx unexpected symbol:“symbol”

Fatal. CUPLX encountered a symbol that it was not expecting. This occurs when certain symbols are expected in a particular order and are either incorrect, misplaced or misspelled.

0015cx Repeat nesting too deep

Fatal. The level of Repeat nesting exceeded two.

0016cx duplicate Macro function name:“function”

Error. The Macro function name has already been previously defined. A duplicate Macro name will cause confusion when they are called.

0017cx missing Macro name

Fatal. A Macro was defined without a name. This macro will never be accessed.

0018cx incorrect number of parameters

Fatal. The number of parameters defined in the Macro function did not equal the number of parameters in the macro call. All parameters defined in the Macro function must be defined in the Macro call.

0019cx out of range

Fatal. The index number exceeded 1023. Valid index numbers are 0 - 1023.

0020cx internal stack overflow

Fatal. A mathematical expression was too complex for CUPLX to handle. The expression can be reduced by eliminating as many parenthetical expressions as possible. Expressions are evaluated from left to right using standard precedence. The user should take advantage of this.

0021cx expression contains undefined symbol: “symbol”

Fatal. A symbol appearing in the expression has not been defined in the source file or predefined by CUPL.

0022cx invalid library access key

Fatal. The version of CUPLX is not compatible with the version of the device library file. This occurs when either CUPLX or the device library, but not both, has been updated.

0023cx invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPLX and the device library are not compatible.

0024cx bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0025cx unexpected end-of-file

Fatal. CUPLX has unexpectedly reached the end-of-file.

0026cx reached end-of-file before ending comment

Fatal. CUPLX detected that a comment was not terminated before reaching the end-of-file. The beginning of the comment can be found by searching for the last occurrence of /* in the PLD file.

0027cx invalid syntax for preprocessor command: “$command”

Fatal. One of the preprocessor commands, $REPEAT or $MACRO, has been used improperly. The command syntax contains unexpected symbols.

10xxcx program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact Logical Devices customer support.

CUPLA Module Error Messages

0001ca could not open: “filename”

Fatal. CUPLA cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002ca invalid number: “number”

Error. Either the number is used improperly, or a previous syntax error caused the number to be used improperly.

0003ca invalid library access key

Fatal. The version of CUPLA is not compatible with the version of the device library file. This occurs when either CUPLA or the device library, but not both, has been updated.

0004ca invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPLA and the device library are not compatible.

0005ca bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0006ca device not in library: “device”

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0007ca invalid syntax: “symbol”

Error. Either the symbol is used improperly, or a previous syntax error caused the symbol to be used improperly.

0008ca too many errors

Fatal. CUPLA has encountered more than 30 errors.

0009ca missing symbol: “symbol”

Error. The missing symbol is required to make the specified statement valid.

0010ca vector too wide

Fatal. A variable list has more than 50 members.

0011ca expression already assigned to: “variable”

Error. The variable (either an intermediate or output variable) was previously assigned an expression. Use APPEND to make multiple expression assignments for the same variable.

0012ca vector size mismatch

Error. The number of members in the variable list on the left side of the equation does not match the number of variables on the right side.

0013ca undefined function: “function”

Error. The variable name used as a function reference has no corresponding function definition. Functions must be defined before they can be referenced.

0014ca variable already declared: “variable”

Error. The variable which was previously assigned an expression cannot be reassigned.

0015ca out of memory: “condition”

Fatal. CUPLA has used all available RAM memory which has been allocated by the operating system. Decrease the number of intermediate variables, fields, or numbers in order to reduce the size of the symbol table.

Note

This error is not a result of insufficient product terms in the device to implement a particular expression.

0016ca invalid number of function arguments: “number”

Error. The user has attempted to pass an incorrect number of arguments to the user-defined function. The number of arguments for the function reference does not match the number in the function definition.

0017ca disk write error: “filename”

Fatal. CUPLA encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0018ca intermediate var not assigned an expression: “variable”

Error. The intermediate variable was used as an input in an expression without having been assigned an expression. This error often occurs when a pin or intermediate variable in a logic expression is misspelled.

0019ca indexed and non-indexed vars in range or match expression

Warning. A list (or field variable) in a range or match expression contains both indexed (variable names ending in a number) and non-indexed variables. This type of operation cannot produce the expected results because of inability to hold relative bit positions in the field. It is recommended to use all non-indexed variables in a field for portability to future versions of CUPL.

0020ca index too large for range or match operation

Error. The index of a variable in a list or field exceeds the range or match values.

0021ca header item already declared

Error. One of the header statements was duplicated.

0022ca missing header item(s)

Warning. At least one of the header statements is missing.

0023ca invalid range arguments: always true (in range)

Error. A range has been specified which will always be true and is therefore not an actual range. CUPLA attempts to minimize range functions and does not allow a NULL range such as this. This happens with ranges such as [0000..FFFF] for a 16-bit address. This error can also be given if non-indexed list variables are used in a range expression.

0024ca range or match number larger than variable list

Warning. The range or match number exceeds the width of the bit field it is being applied to. Values exceeding the width of the bit field will be ignored.

0025ca range minimization error

Error. The range reduces to always false, that is, none of the bits in the range are active.

0026ca invalid table statement

Error. Input numbers cannot be mapped into more than one output number.

0027ca invalid present state number

Error. The present state number specified is not valid. This error can occur whenever the present state has not been properly defined as a number using the $DEFINE command.

0028ca invalid next state number

Error. The next state number specified is not valid. This error can occur whenever the next state has not been properly defined as a number using the $DEFINE command.

0029ca invalid flip-flop type for sequence statement: “type”

Error. The flip-flop type for this device cannot be used for building the requested sequential state machine.

0030ca intermediate dependent on itself: “variable”

Error. The intermediate variable was used in the expression defining the same intermediate variable. This error often occurs when an intermediate variable is misspelled or an output pin expression is being defined using feedback without declaring the output variable as a pin.

0031ca invalid minimization level: “level”

Error. The minimization level specified is invalid. Refer to “Running CUPL” in Chapter 2 for valid minimization levels.

0032ca invalid next state: “hex number”

Error. The next state value is invalid. This error can occur whenever the next state has not been properly defined as a number using the $DEFINE command or has not been identified as a present state using the present command.

0033ca multiple asynchronous defaults for state: “hex number”

Error. By definition, only one asynchronous default expression can be assigned for any one state. The resulting expression is the complement of all previous conditional (if) asynchronous expressions.

0034ca multiple synchronous defaults for state: “hex number”

Error. By definition, only one synchronous default expression can be assigned for any one state. The resulting expression is the complement of all previous conditional (if) synchronous expressions.

0035ca multiple unconditional statements for state: “hex number”

Error. By definition, only one unconditional synchronous statement can be given for any one state.

0036ca device does not support synchronous state machines

Fatal. The device specified for compilation cannot be used with the sequence statement since it does not support registered operations.

0037ca duplicate present state: “hex number”

Error. The present state number was identified in more than one PRESENT command. This can occur when symbolic state names are used to refer to states, but the $DEFINE command, used to define states, assigned the same number to more than one symbolic name.

0038ca target device not specified

Fatal. The user did not specify a target device on the command line and the source file did not contain a DEVICE assignment in the header information.

0039ca line exceeds maximum length

Error. The statement is greater than 256 characters long. Break the line up into shorter statements.

0040ca invalid or duplicate header name: “name”

Fatal. The NAME field in the header information must not be NULL. When more than one device is being defined in a logic description file, the NAME field in the header information must be unique.

0041ca don't care(s) not allowed for decimal number, treated as 0

Warning. “Don't-care” values, “X”, are valid only for binary, octal, and hexadecimal numbers.

0042ca range or match list completely don't cared, decoded as 0

Warning. The variable list in a range or match operation has been completely “don't-cared,” leaving an empty variable list. The empty variable list will be decoded into a 0.

0043ca invalid GROUP name: “variable name”

Fatal. The GROUP name must contain the keyword BLOCK_ followed by “variable name”. Ex. GROUP BLOCK_A=[X,Y]; where A is the variable name.

0044ca unexpected end-of-file

Fatal. CUPLA has unexpectedly reached the end-of-file.

0045ca reached end-of-file before ending comment

Fatal. CUPLA detected that a comment was not terminated before reaching the end-of-file. The beginning of the comment can be found by searching for the last occurrence of /* in the PLD file.

0046ca invalid DeMorgan level: "number"

Error. The DeMorgan level specified is not within the range of 0 to 2. The level defaults to 0.

0047ca vector size mismatch in comparison vector: "variable"

Fatal. The number of members in the variable list on the left side of the comparison does not match the number of variables on the right side.

0048ca fixed polarity device, reset DeMorgan level to 0: "variable"

Warning. The device specified does not have programmable polarity capability. Only devices with this capability can use different DeMorgan levels. The variable will be evaluated to fit the device's capability.

0049ca unknown DECLARE entity: "variable"

Fatal. Either the manufacturer's ID or the attribute in the DECLARE statement is unknown. DECLARE.DEF contains the information needed for using a DECLARE statement.

10xxca program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact Logical Devices, customer support.

CUPLB Module Error Messages

0001cb could not open: “filename”

Fatal. CUPLB cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cb could not execute program: “program name”

Fatal. CUPLB is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cb invalid file: “filename”

Warning. The file was not created by the current version of CUPL.

0004cb missing or mismatched parentheses:

Error. The number of open parentheses [(] and close parentheses [)] in the specified statement does not match.

0005cb invalid library access key

Fatal. The version of CUPLB is not compatible with the version of the device library file. This occurs when either CUPLB or the device library, but not both, has been updated.

0006cb invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPLB and the device library are not compatible.

0007cb bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0008cb device not in library: “device”

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0009cb pin/node “number” redeclared: “variable”

Error. The same pin number or variable name was used more than once in a pin declaration statement.

0010cb pin/node “number” invalid output: “variable”

Error. The variable being assigned an output expression was previously declared for an input-only pin.

0011cb unknown extension: “extension”

Error. The extension is unknown or invalid for the particular device. Refer to “Extensions” in Chapter 2 for a list of valid extensions. Check to make sure the device has the capability required.

0012cb pin/node “number” invalid usage: “variable”

Fatal. The pin number assigned to the variable is invalid for the target device specified.

0013cb pin/node “number” invalid output extension or usage: “variable”

Error. Either the extension is used improperly or it is not valid for the assigned pin/node.

0014cb invalid input: “variable” or pin/node “number” invalid input: “variable”

Error. The variable used as an input was previously assigned to an output that is neither bidirectional nor feeds back into the input array.

0015cb device not yet fully supported: “device”

Fatal. There is an entry for the device in the device library, but the device is not fully supported by the current version of CUPL.

0016cb no expression assigned to: “variable”

Warning. The variable requires an output expression assignment. This warning message is commonly given when all outputs in a bank have the same capability (reset, preset, and so on) and not all the variables have been assigned the same expression. It is given to remind the user that all outputs will be affected.

Note

This warning may be suppressed by assigning the variable to 'b'0 or 'b'1 as appropriate.

0017cb out of memory: “conditions”

Fatal. CUPLB has used all available RAM memory that has been allocated by the operating system, typically as a result of performing a DeMorgan or expansion operation on a large expression. If using fixed polarity devices, check to make sure that the pin variable declaration matches the polarity of the device. Also check whether an intermediate variable which has been expressed in sum-of-product form is being complemented.

This error does not result from insufficient product terms in the device to implement a particular expression.

0018cb missing flip-flop expression for: “variable”

Error. The matching flip-flop expression for a J-K or S-R type flip-flop is missing. Both inputs must have expressions assigned to them. An input may be assigned to 'b'0 or 'b'1 as appropriate.

0019cb DeMorgan's theorem invoked for: “variable”

Warning. DeMorgan's Theorem has been applied to the expression assigned to the variable. Unlike D- or T-type flip-flops, meaningful results are not guaranteed when a DeMorgan equivalent expression is applied to the logic input.

0020cb invalid mix of banked outputs: “variable”

Error. All outputs in a banked group must be used in the same manner. An attempt was made to mix registered and non-registered output types.

0021cb no expression allowed for: “variable”

Error. Logic expressions are not allowed for reset and preset nodes when the output has been specified as asynchronous. CUPL will generate the proper defaults.

0022cb pin/node “number” conflicting input architectures: “variable”

Error. A fuse-assigned input architecture must be used consistently in all expressions. An attempt was made to specify both fuse options in different expressions.

0023cb disk write error: “filename”

Fatal. CUPLB encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0024cb output defined for node which does not exist: "variable"

Error. Variable is defined for a pin or node number which does not exist.

0025cb output mutually excluded by previous output: “variable”

Error. Variable usage is mutually excluded by a previous usage or other output. A shared product term or terms has been defined more than once.

0026cb disk read error, unexpected end of file: “filename”

Fatal. CUPLB encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

10xxcb program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact LDI customer support.

CUPLM Module Error Messages

0001cm could not open: “filename”

Fatal. CUPLM cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cm could not execute program: “program name”

Fatal. CUPLM is unable to perform the next step in the compilation. Be sure that all of the CUPL program files exist on the same directory or disk.

0003cm invalid file: “filename”

Warning. The file was not created by the current version of CUPL.

0004cm out of memory: “conditions”

Fatal. CUPLM has used all available RAM memory which has been allocated by the operating system while performing logic reduction.

Note

This error does not result from insufficient product terms in the device to implement a particular expression.

0005cm disk write error: “filename”

Fatal. CUPLM encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0006cm invalid library access key

Fatal. The version of CUPLM is not compatible with the version of the device library. This occurs when either CUPLM or the device library, but not both, has been updated.

0007cm invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD or CUPLM and the device library are not compatible.

0008cm bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0009cm device is not in library: “device”

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0010cm design too complex for this minimization level

Fatal. CUPLM has exceeded the array size allowed on this machine while reducing a particular expression. Specify a more efficient minimization level.

0011cm disk read error, unexpected end of file: “filename”

Fatal. CUPLM encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

10xxcm program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact LDI customer support.

CUPLC Module Error Messages

0001cc could not open: “filename”

Fatal. CUPLC cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002cc invalid file: “filename”

Warning. The file was not created by the current version of CUPL.

0003cc invalid library access key

Fatal. The version of CUPLC is not compatible with the version of the device library. This occurs when either CUPLC or the device library, but not both, has been updated.

0004cc invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPLC and the device library are not compatible.

0005cc bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0006cc excessive number of product terms: “variable”

Error. The number of product terms needed to implement the logic expression for the given variable exceeds the capacity of the output pin for which it was declared.

0007cc invalid download format(s)

Warning. At least one of the download formats specified is not available for the target device. For example, the HL download format is not available for PALs or PROMs.

0008cc pin can not be used as input: “variable”

Error. The pin to which the variable is assigned provides no input or feedback capability.

0009cc header name undefined, using no_name

Error. The NAME field in the header information is missing. Since CUPLC uses this name to generate download files, the desired file will be created as “no_name” along with the appropriate extension.

0010cc disk write error: “filename”

Fatal. CUPLC encountered an I/O error trying to write the indicated file. This error usually occurs when there is insufficient disk space.

0011cc out of memory: “conditions”

Fatal. CUPLC has used all the available RAM memory allocated by the operating system.

Note

This error does not result from insufficient product terms in the device to implement a particular expression.

0012cc disk read error, unexpected end of file: “filename”

Fatal. CUPLC encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

0013cc conflicting usage of pinnode:"variable"

Error. Variable usage is mutually excluded by a previous usage of the pin or pinnode. A shared product term or terms has been defined more than once.

0014cc unknown extension encountered: “extension”

Warning. The translation of a CUPL extension into another file format could not be accomplished. The equation is still placed in the new file except the extension has been lost.

0015cc invalid local feedback from “variable name” to “variable name”

Fatal. The local feedback of a macrocell was used outside the quadrant. This means that the feedback of a local macrocell or the internal feedback of a global macrocell was used as input to another macrocell that is located in another quadrant.

0016cc exceeded number of expander product terms

Fatal. The number of expander product terms needed to implement the design exceeds the capacity of the target device for which it was declared.

0017cc global feedback in local product term: "variable"

Error. The feedback from a global variable is being used within a local product term. This is illegal to do when using the V5000 mnemonic. The variable shown is a local variable and it contains the global variable feedback.

0018cc couldn't find XILINX symbol: "symbol"

Error. The symbol was not found for the specified Xilinx device. This means that either the symbol cannot be used for the specified device or the MAC file for that device is corrupted.

0019cc couldn't map CUPL symbol to XILINX symbol: "symbol"

Error. An architecture specification in the design file cannot be mapped into the specified device. This means that the MAP file for the device does not contain the CUPL macro translation.

0020cc couldn't find CUPL macro symbol: "symbol"

Error. An internal CUPL macro was not found in the file CUPL2XIL.MAP. This file may be corrupted or incomplete.

0021cc Error found in XILINX data file

Fatal. An error has occurred while reading in one of the Xilinx information files. These files are designated by the MAC and MAP extensions.

0022cc unsupported extension: "extension"

Fatal. The translation of a CUPL extension into another file format could not be accomplished.

0023cc incorrect number of variables in DECLARE statement: "attribute"

Warning. The number of variables in the DECLARE statement does not match the number of expected variables defined in the DECLARE.DEF file.

0024cc too many XOR gates defined for output: "variable"

Fatal. The placement of XOR gates into the PLA file cannot be performed due to the device not having the resources to hold the output expression. If this error occurs, do not use the -kx flag when compiling.

10xxcc program error: “specifics”

Fatal. An operating system interface problem is suspected. Contact Logical Devices customer support.

CSIM ERROR MESSAGES

This section describes the error messages for the CSIM, CSIMA, and WCSIM modules:Error Messages.

CSIM Module Error Messages

0001sk could not open: “filename”

Fatal. CSIM cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002sk could not execute program: “program name”

Fatal. CSIM is unable to perform the next step in the simulation. Be sure that all of the CSIM program files exist on the same directory or disk.

0003sk could not find PATH in ENVIRONMENT

Fatal. The PATH assignment has not been made in the ENVIRONMENT.

0004sk could not find LIBCUPL in ENVIRONMENT

Fatal. The LIBCUPL assignment has not been made in the ENVIRONMENT.

0005sk could not find program: “program name”

Fatal. CSIM is unable to locate the CSIM program using the PATH in the ENVIRONMENT.

0006sk insufficient memory to execute program: “filename”

Fatal. Not enough program storage available to load and execute the program. Refer to the System Overview for the minimum memory requirements for the configuration being used.

0007sk invalid flag: “flag”

Fatal. The specified flag is not a valid option flag. Execute CSIM without arguments to get a listing of valid option flags.

0008sk out of memory: “condition”

Fatal. CSIM has used all the available RAM memory allocated by the operating system. Check for the existence of print spoolers, RAM disks, or other memory-

resident programs which may decrease the amount of memory available to the CUPL application program.

0009sk file read error, unexpected end of file: "filename"

Fatal. CUPL encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media or the file has been corrupted.

0010sk invalid library access key

Fatal. This version of CUPL is not compatible with the version of the device library file. This occurs when either CUPL or the device library, but not both, has been updated.

0011sk invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CUPL and the device libraryare not compatible.

0012sk bad library file: "filename"

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0013sk device not in library: "device"

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0014sk target device not specified

Fatal. The user did not specify a target device on the command line and the source file did not contain a DEVICE assignment in the header information.

10xxsk program error: “specifies”

Fatal. An operating system interface problem is suspected. Contact LDI customer support.

CSIMA Module Error Messages

0001sa could not open: “filename”

Fatal. CSIM cannot continue because of the failure to open the indicated file. Be sure the file exists if it is an input.

0002sa invalid number: “number”

Error. Either the number is used improperly, or a previous syntax error has caused the number to be used improperly.

0003sa invalid file format: “filename”

Warning. The file was not created by a compatible version of CUPL.

0004sa invalid library access key

Fatal. The version of CSIMA is not compatible with the version of the device library used in the simulation. This occurs when either CSIMA or the device library, but not both, has been updated.

0005sa invalid library interface

Fatal. Either the device library was not created using the CUPL library manager, CBLD, or CSIMA and the device library are not compatible.

0006sa bad library file: “library”

Fatal. Either the device library does not exist or the contents of the device library have been damaged.

0007sa device not in library: “device”

Fatal. Either the specified target device does not exist or an entry has not been made in the device library for the device.

0008sa invalid output format: “format”

Warning. The download format is not available for the target device; for example, the JEDEC download format is not available for PROMS.

0009sa invalid syntax: “symbol”

Error. Either the symbol is used improperly, or a previous syntax error has caused the symbol to be used improperly.

0010sa expecting device: “device”

Fatal. The target device is not the same as used when CUPL created the absolute file.

0011sa unknown symbol: “symbol”

Error. The symbol, used in the order statement, was not previously defined in the CSIM or CUPL source files.

0012sa too many symbols:

Fatal. The number of symbols in the order statement exceeds the number of symbols previously defined in the CSIM and CUPL source files.

0013sa excessive test value “value”

Error. The test vector value is greater than the maximum possible value defined in the order statement. This error will occur when there are too many test values.

0014sa insufficient test values

Fatal. The test vector value is less than the minimum possible value defined in the order statement. This error will occur when there are too few test values.

0015sa field already defined: “field”

Error. The field name was previously used in either the CSIM or CUPL source files.

0016sa too many errors

Fatal. CSIM has encountered too many errors to continue.

0017sa missing symbol “symbol”

Error. CSIM expected a keyword.

0018sa out of memory: “condition”

Fatal. CSIM has used all the available RAM memory allocated by the operating system.

0019sa user expected (value) for: “variable”

Error. The test value expected by the user in the .SI file did not match the actual value computed by CSIM.

0020sa unstable output: “variable”

Error. The output variable did not have the same test value for two continuous evaluation passes after the maximum twenty passes were attempted. Check the logic equation for an untestable design.

0021sa invalid test value: “value”

Error. Either the test value is an invalid test vector symbol or the test value is used improperly; that is, a test value of 0 is used for an output.

0022sa bad fault id: “JEDEC number”

Error. The JEDEC number, given as the fault ID, is not the address of the beginning of a product term.

0023sa could not read file: “filename”

Fatal. CSIM could not read from the specified file. This occurs when the contents of the file have been corrupted.

0024sa could not write file: “filename”

Fatal. CSIM could not write to the specified file. This occurs when the file is write protected or there is no room left on the disk.

0025sa inconsistent header information

Warning. The header information in the CSIM source file does not match the header information in the CUPL source file used to create the absolute file.

0026sa missing header item(s)

Warning. At least one of the header statements is missing.

0027sa old absolute file format for “filename”

Fatal. The absolute file was created by an incompatible version of CUPL.

0028sa statement too long

Fatal. The statement exceeds 256 characters.

0029sa invalid trace level: “number”

Error. The trace level must be a decimal number in the range of 0 through 4.

0030sa invalid character: “hex value”

Error. CSIMA has encountered an illegal ASCII value in the source file. Make sure the file was created in non-document mode on the word processor. This error can also be caused by files which were created over a serial modem upload/download link.

0031sa disk read error, unexpected end of file: “filename”

Fatal. CSIMA encountered an I/O error trying to read the indicated file. This error usually occurs when the file is being read from damaged media.

0032sa feedback usage of undefined output: “variable name”

Fatal. The variable name does not exist in the ORDER statement and it is being used as input/feedback for another variable. Simulation cannot occur until all relevant variables are defined.

0033sa pin number is undefined for: “variable name”

Fatal. When simulating a design in a specified device, CSIM needs to have all the pin numbers defined. The variable name in the PLD file was not assigned a pin number. The PLD file has to be recompiled with all the pin numbers in place.

10xxsa program error: “specifies”

Fatal. An operating system interface problem is suspected. Contact Logical Devices customer support.

Index

E

F