

# CUPL Programmers Reference Guide

## Contents

CUPL Programmers Reference Guide .....	1
Programmers Reference Overview.....	4
1. CUPL Language Reference .....	5
1.1. Language Reference .....	5
1.1.1. LANGUAGE ELEMENTS .....	6
1.1.2. Variables .....	7
1.1.3. Indexed Variables .....	8
1.1.4. Reserved Words and Symbols .....	9
1.1.5. Numbers .....	10
1.1.6. Comments .....	11
1.1.7. List Notation .....	12
1.1.8. Template File.....	13
1.1.9. Header Information.....	15
1.1.10. Pin Declaration Statements.....	17
1.1.11. Node Declaration Statements .....	21
1.1.12. Bit Field Declaration Statements .....	23
1.1.13. MIN Declaration Statements .....	24
1.1.14. FUSE Statement.....	25
1.1.15. Preprocessor Commands .....	26
1.1.16. LANGUAGE SYNTAX.....	33
1.1.17. Logical Operators .....	34
1.1.18. Arithmetic Operators .....	35
1.1.19. Arithmetic Function .....	36
1.1.20. Extensions.....	37
1.1.21. Feedback Extensions Usage.....	40
1.1.22. Multiplexer Extension Usage .....	42
1.1.23. Extension Usage .....	44
1.1.24. Expressions .....	57
1.1.25. Logic Equations .....	58

1.1.26. Set Operations.....	60
1.1.27. Truth Tables.....	70
1.1.28. State-Machines .....	72
1.1.29. State-Machine Model.....	73
1.1.30. State Machine Syntax.....	75
1.1.31. Condition Syntax.....	90
1.1.32. User-Defined Functions .....	92
1.2 Simulator Reference.....	95
1.2.1. INPUT.....	96
1.2.2. OUTPUT.....	97
1.2.3. Virtual Simulation .....	98
1.2.4. Header Information.....	99
1.2.5. Comments .....	100
1.2.6. Statements.....	101
1.2.7. Simulator Directives .....	111
1.2.8. Fault Simulation .....	115
1.3. CUPL Modules.....	125
1.3.1. key features of the CUPL package include: .....	126
1.3.2. CUPL DATA FLOW.....	127
1.3.3 CUPL.DL .....	129
1.3.4. SYSTEM OVERVIEW .....	132
1.4 Minimization Techniques.....	135
1.5 Using the CUPL at the command line .....	136
1.7. Run CSIM using the following command line .....	140
1.7.1. Simulator Option Flags .....	141
2.1. SAMPLE DESIGN SESSION .....	143
2.1.1. STEP 1. EXAMINING THE DESIGN TASK.....	144
2.1.2. STEP 2. CREATING A CUPL SOURCE FILE .....	147
2.1.3. STEP 3. FORMULATING THE EQUATIONS.....	150
2.1.4. STEP 4. CHOOSING A TARGET DEVICE .....	152
2.1.5. STEP 5. MAKING THE PIN ASSIGNMENTS .....	154
2.1.6. STEP 6. RUNNING CUPL.....	156
2.1.7. STEP 7. CREATING A CSIM SOURCE FILE .....	163

2.1.8. STEP 8. RUNNING CSIM .....	166
2.2.1. EXAMPLE 1 SIMPLE GATES .....	172
2.2.2. EXAMPLE 2. CONVERTING A TTL DESIGN TO PLDs .....	174
2.2.3. EXAMPLE 3. TWO-BIT COUNTER.....	179
2.2.4. EXAMPLE 4. DECADE UP/DOWN COUNTER.....	182
2.2.5. EXAMPLE 5. SEVEN-SEGMENT DISPLAY DECODER.....	190
2.2.6. EXAMPLE 6. 4-BIT COUNTER WITH LOAD AND RESET.....	193
3 – Output Formats .....	195
3.1 DOWNLOADABLE FILE FORMATS .....	196
3.2 JEDEC Format .....	197
3.3 ASCII-Hex Format.....	200
3.4 HL Formats .....	201
3.5 DOCUMENTATION FILE FORMAT .....	207
3.6 PDIF FILE FORMAT .....	213
3.7 BERKELEY PLA FILE FORMAT .....	214
4 – Error Messages .....	215
4.1 CUPL Module Error Messages.....	216
4.2 CUPLX Module Error Messages.....	218
4.3 CUPLA Module Error Messages.....	221
4.4 CUPLB Module Error Messages.....	225
4.5 CUPLM Module Error Messages .....	228
4.6 CUPLC Module Error Messages .....	230
4.7 CSIM ERROR MESSAGES .....	233
4.8 CSIMA Module Error Messages .....	235
5 – Getting Help .....	238
5.1 Common Errors .....	238
5.2 How To Get Support .....	238
5.3 Contacting Technical Support .....	238

## **Programmers Reference Overview**

This Reference Guide gives an in-depth look into the CUPL Language. In this Reference Guide, you will find detailed information about the Compiler, a Language Reference look up, Manufacturer Specific Fitter Options, Library, ETC.

# **1. CUPL Language Reference**

## **1.1. *Language Reference***

This chapter explains the CUPL language elements and CUPL language syntax.

### **1.1.1. LANGUAGE ELEMENTS**

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

### 1.1.2. 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 1-2).

Variables cannot be the same as a CUPL reserved keyword (see Table 1-1).

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.

### 1.1.3. 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 LO-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.



**Note** 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.



**Note** 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:

<b>D0F</b>	<b>index number is not decimal</b>
<b>a36</b>	<b>index number out of range</b>

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



#### 1.1.4. Reserved Words and Symbols

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

**Table 1-1. CUPL Reserved Keywords**

APPEND	FORMAT	OUT
ASSEMBLY	FUNCTION	PARTNO
ASSY	FUSE	PIN
COMPANY	GROUP	PINNNOE
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 1-2 lists these reserved symbols.

**Table 1-2. CUPL Reserved Symbols**

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

### 1.1.5. 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 232 -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 1-3. Once a base change has occurred, that new base is the default base.

**Table 1-3. 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 1-4.

**Table 1-4. 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 1-5.

**Table 1-5. Sample Don't-Care Numbers**

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

### 1.1.6. 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.

Comments may span multiple lines and are not terminated by the end of a line. Comments cannot be nested. Some examples of valid comments are shown in Figure 1-1.

```
/******  
/* This is one way to create a title or */  
/* an information block */  
/******  
  
/*  
This is 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 1-1. Sample Comments**

### 1.1.7. 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]**

## 1.1.8. 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 1-2 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      =      ;      /*
Pin      =      ;      /*
Pin      =      ;      /*

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

/** Declarations and Intermediate Variable Definitions **/

/** Logic Equations **/
```

**Figure 1-2. 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).

### 1.1.9. 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 1-6 lists the CUPL header keywords and the information to provide with each keyword.

**Table 1-6. Header Information Keywords**

Keyword	Information
<b>Keyword</b>	<b>Information</b>
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.
FORMAT	Set a download output format override for the current logic description section. The valid values to use for the output format are:

h produce ASCII-hex output  
i produce Signetics HL output  
j produce JEDEC output

FORMAT overrides any option flag on the command line. It is useful in multi-device source files where different parts have incompatible output formats. More than one format value at a time may be specified to produce more than one type of output. The format value must be a lowercase letter.

The template file provides all the header keywords except **DEVICE** and **FORMAT**. An example of proper CUPL header information is as follows:

```
Name      WAITGEN ;
Partno    P9000183 ;
Revision  02 ;
Date      1/11/89 ;
Designer  Osann ;
Company   Logical Devices, Inc. ;
Assembly  PC Memory Board ;
Location  U106 ;
Device    F155;
Format    ij ;
```

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



### 1.1.10. 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.

### **Fig. 1-3. 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:

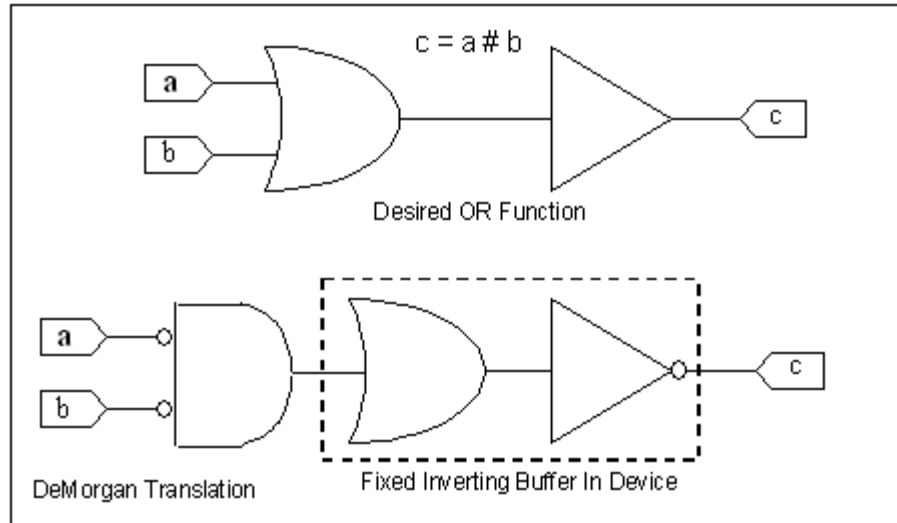
**c = a # b ;**

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 (see Documentation File Formats in Appendix C):

**$c \Rightarrow !a \& !b$**

Figure 1-4 shows the process described above.



**Figure 1-4. 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.

### 1.1.11. Node 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 82S105 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 **NODE** keyword to declare variable names for buried functions.

The format for node declarations is as follows:

**NODE** [!] **var** ;

where

**NODE** is a keyword to declare a variable name for a buried function.

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

**var** is a single variable name or list of variables grouped using the list notation.

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

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

Most internal nodes are active-level HI; therefore, the exclamation point should not be used to define the polarity of an internal signal as active-level LO. Using the exclamation point almost always causes the compiler to generate a significantly greater number of product terms. An exception is the complement array node, which, by definition, is an active-level LO signal.

Although no pin numbers are given in the declaration statement, CUPL assigns the variable name to an internal pseudo-pin number. These numbers begin with lowest possible number and are sequentially defined even if a node was assigned with the PINNODE statement. The assignment is automatic and determined by usage (flip-flop, complement array, and so on), so variable order is not a concern. However, once a node variable is declared, a logic equation must be created for the variable, or a compilation error results.

CUPL uses the node declaration to distinguish between a logic equation for a buried function and an intermediate expression.

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

```
NODE [State0..5]; /* Internal State Bit      */
NODE !Invert;    /* For Complement Array  */
```

An alternative for assigning buried functions instead of allowing CUPL to automatically assign them via the **NODE** keyword, is to use the **PINNODE** keyword. 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.

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

As with node declarations, most internal nodes are active-level HI; therefore, the exclamation point should not be used to define the polarity of an internal signal as active level LO. Using the exclamation point almost always causes the compiler to generate a significantly greater number of product terms. An exception is the complement array node, which by definition is an active-level LO signal.

A list of node numbers for all devices containing internal nodes is included in Appendix D. 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;    /* Complement Array */
PINNODE 25       = Buried;     /* Buried register part */
                                /* of an I/O macrocell */
                                /* with multiple */
                                /* feedback paths */

```

### 1.1.12. 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.



**Note** The square brackets do not indicate optional items. They are used to delimit items in the 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.



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

### 1.1.13. 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.



### 1.1.14. 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 MISER Bit or TURBO Bit 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 MISER or TURBO Bit.

### 1.1.15. 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 1-7 lists the available preprocessor commands. Each command is described in detail in this section.

**Table 1-7. 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 Start Comment
$DEFINE }      */  Alternate End Comment
$DEFINE /      !  Alternate Negation
$DEFINE *      &  Alternate AND
$DEFINE +      #  Alternate OR
$DEFINE :+     $  Alternate XOR
```



## Note

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.

<b>\$DEFINE</b>	<b>/</b>	<b>!</b>	<b>Alternate Negation</b>
<b>\$DEFINE</b>	<b>*</b>	<b>&amp;</b>	<b>Alternate AND</b>
<b>\$DEFINE</b>	<b>+</b>	<b>#</b>	<b>Alternate OR</b>
<b>\$DEFINE</b>	<b>:+:</b>	<b>\$</b>	<b>Alternate XOR</b>
<b>\$DEFINE</b>	<b>{</b>	<b>/*</b>	<b>Alternate Start Comment</b>
<b>\$DEFINE</b>	<b>}</b>	<b>*/</b>	<b>Alternate End Comment</b>

## \$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 4*/  
                                     /* 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 number1 to numbern. 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 files provided on diskette.

## **\$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.



### **1.1.16. 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.

## 1.1.17. Logical Operators

CUPL supports the four standard logical operators used for boolean expressions. Table 1-8 lists these operators and their order of precedence, from highest to lowest.

**Table 1-8. 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 1-5 list the Boolean Logic rules for each operator.

**NOT : ones complement !**

A	!A
0	1
1	0

**AND &**

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

**OR #**

A	B	A # B
0	0	0
0	1	1
1	0	1
1	1	1

**XOR : exclusive OR \$**

A	B	A \$ B
0	0	0
0	1	1
1	0	1
1	1	0

**Figure 1-5. Truth Tables**

### 1.1.18. 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 1-9 lists these operators and their order of precedence, from highest to lowest.

**Table 1-9 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
-	4-i	Subtraction	3

### 1.1.19. Arithmetic Function

CUPL supports one arithmetic function used for arithmetic expressions. The arithmetic expressions can only be used in the **\$REPEAT** and **\$MACRO** commands. Table 1-10 lists the function.

**Table 1-10 Arithmetic Function**

<b>Function</b>	<b>Base</b>
LOG2	Binary
LOG8	Octal
LOG16	Hexadecimal
LOG	Decimal

The LOG function returns an integer value. For example:

$\text{LOG2}(32) = 5 \iff 2^{**}5 = 32$

$\text{LOG2}(33) = \text{ceil}(5.0444) = 6 \iff 2^{**}6 = 64$

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

## 1.1.20. 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 1-11 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 1-11. Extensions**

<b><u>Extension</u></b>	<b><u>Side Used</u></b>	<b>Description</b>
.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 input 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 */
```

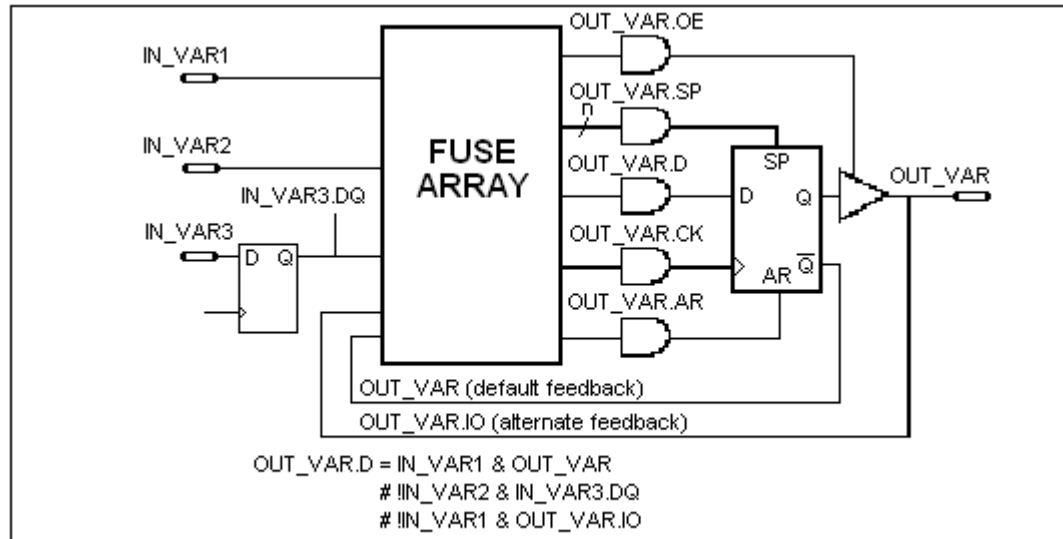
When the I/O pin is to be used as both an input and output, any new output enable expression that the user specifies overrides the default to enable the tri-state buffer at the desired time.

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 1-6 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 1-6. 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.



### Note

The DQ extension is used for input pins only

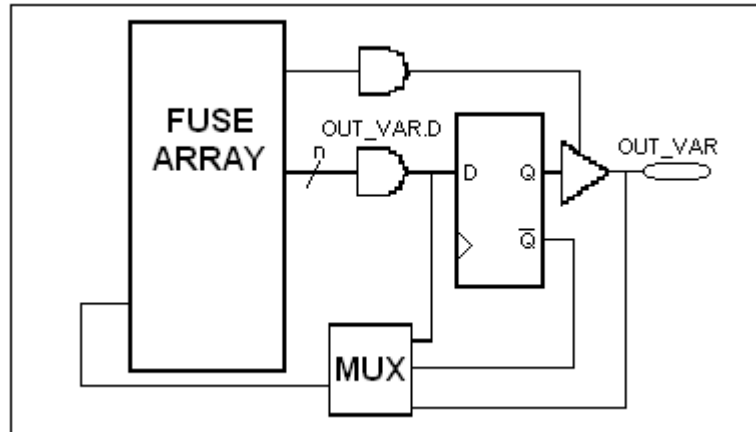
Additional equations can be written to specify other types of controls and control points. For example, an equation for the output enable can be written as follows:

$$\text{OUT\_VAR.OE} = \text{IN\_VAR1} \# \text{IN\_VAR2}$$

### 1.1.21. 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.

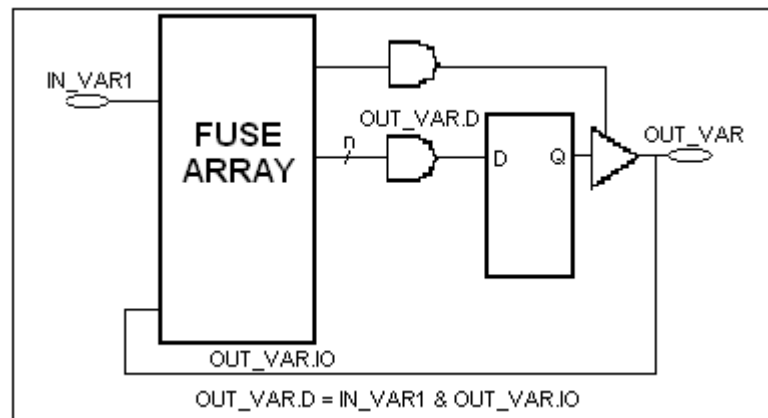
Figure 1-7 shows the EP300 programmable feedback capability.



**Figure 1-7. 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 1-6. 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.

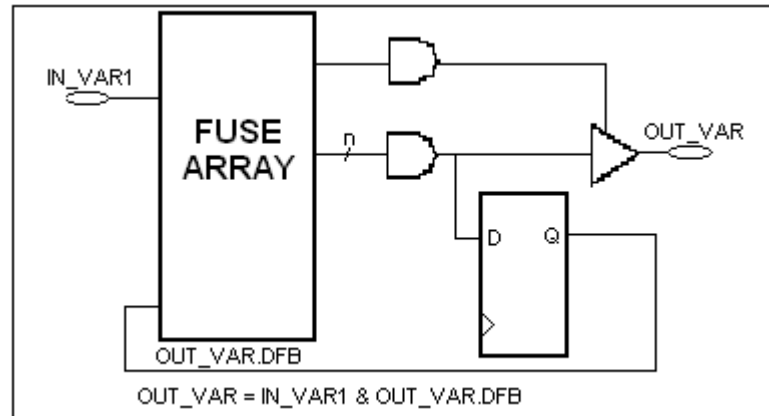
Figure 1-8 shows a registered output with pin feedback.



**Figure 1-8. Programmable Pin (I/O) Feedback**

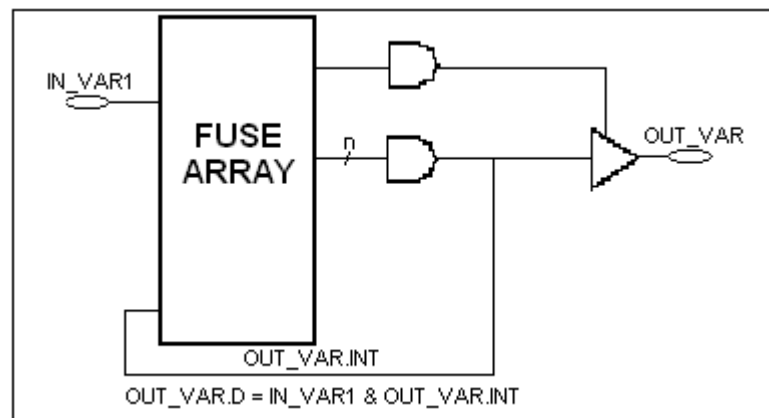
Figure 1-9 shows a combinatorial output with registered feedback.





**Figure 1-9. Programmable Registered Feedback**

Figure 1-10 shows a combinatorial output with internal feedback.

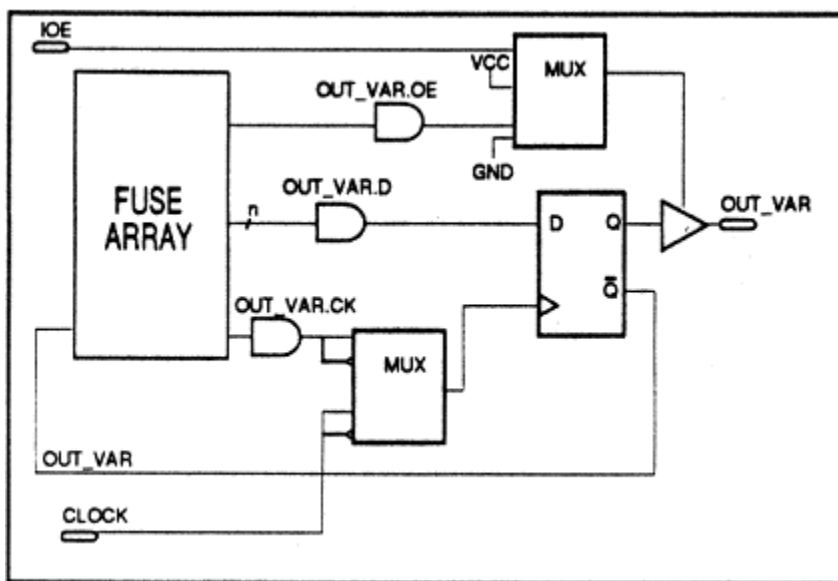


**Figure 1-10. Programmable Internal Feedback**

### 1.1.22. 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 1-11 shows the P29MA16 programmable clock and output enable capability.

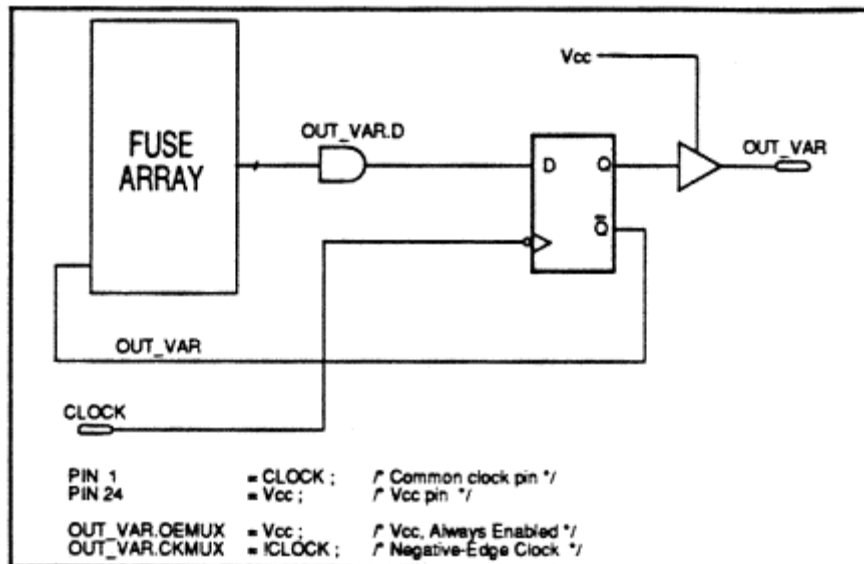


**Figure 1-11. Output with Output Enable and Clock Multiplexers**

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 1-12 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.

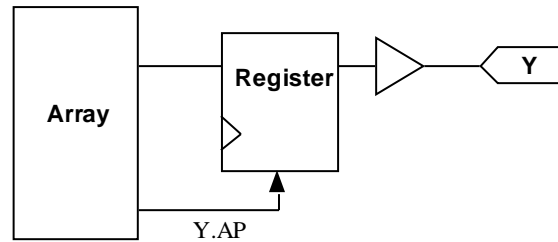


**Figure 1-12. Output with Output Enable and Clock Multiplexers Selected**

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.

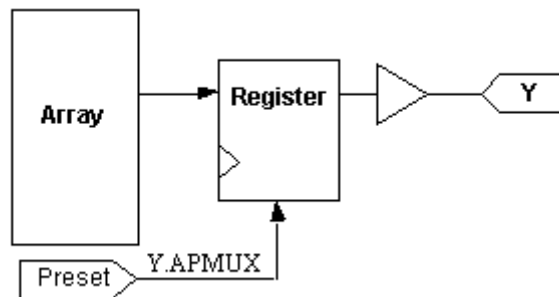
### 1.1.23. Extension Usage

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



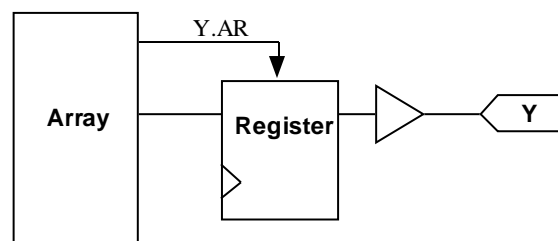
**Figure 1-13. .AP Extension**

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.



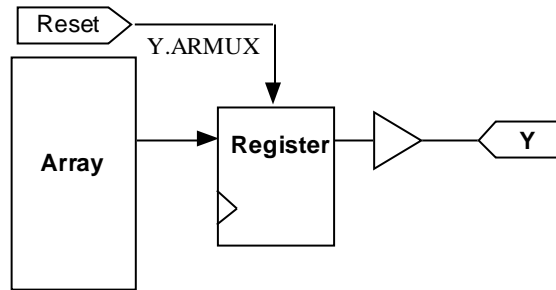
**Figure 1-14. .APMUX Extension**

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.



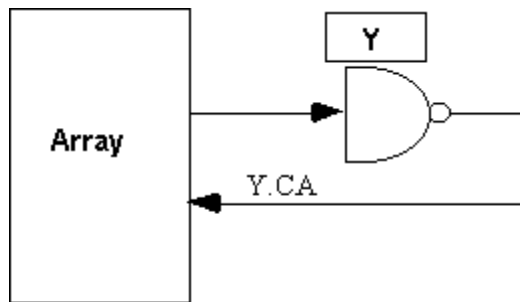
**Figure 1-15. .AR Extension**

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.



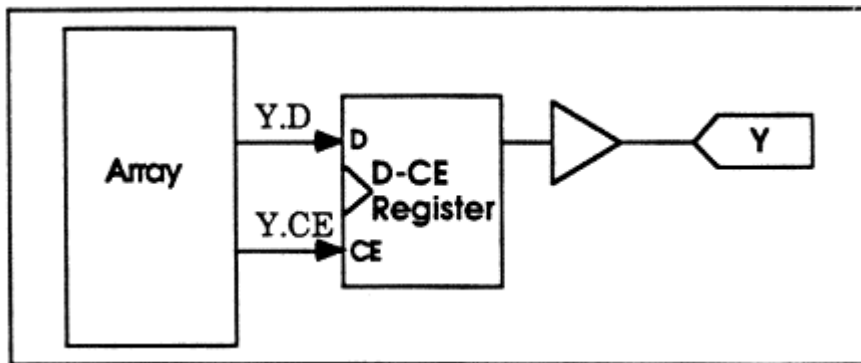
**Figure 1-16. .ARMUX Extension**

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.



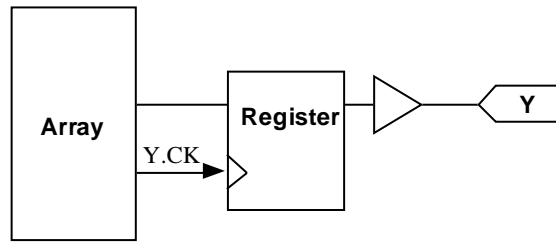
**Figure 1-17. .CA Extension**

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.(See Appendix B)



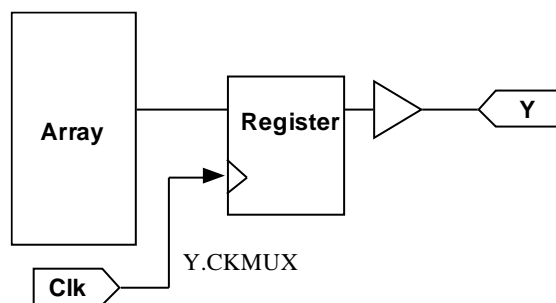
**Figure 1-18. .CE Extension**

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.



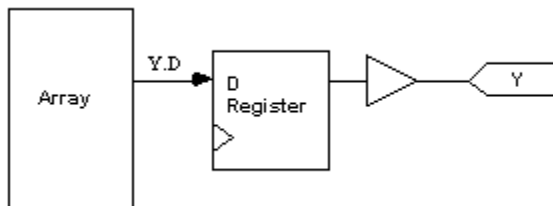
**Figure 1-19. .CK Extension**

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. To connect the clock to a pin directly, use the .CKMUX extension.



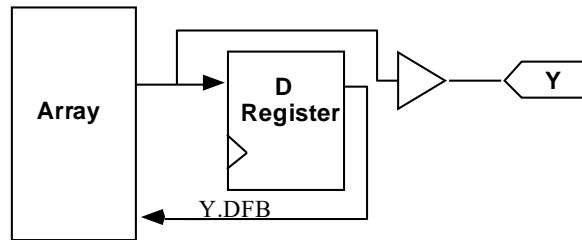
**Figure 1-20. .CKMUX Extension**

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.



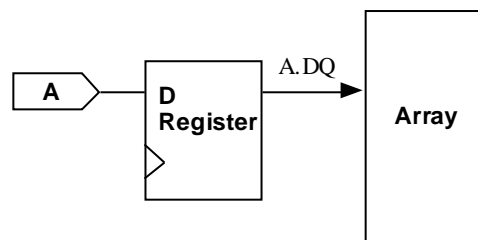
**Figure 1-21. .D Extension**

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.



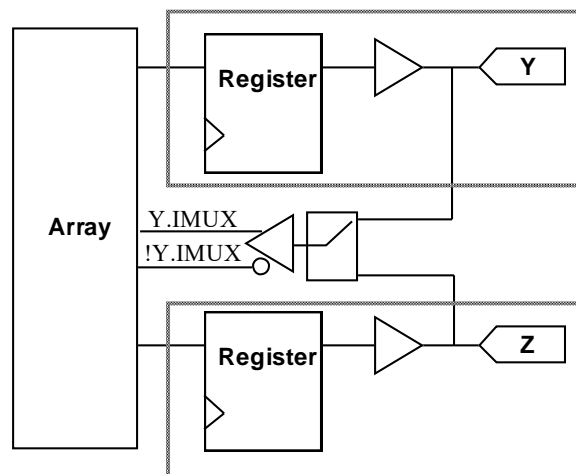
**Figure 1-22. .DFB Extension**

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.



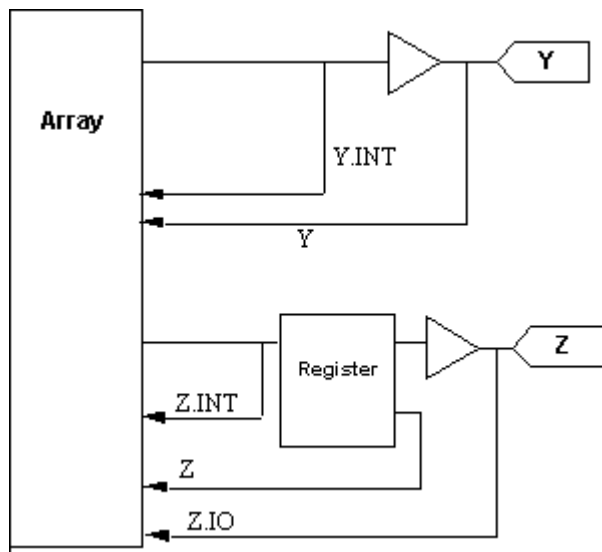
**Figure 1-23. .DQ Extension**

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.



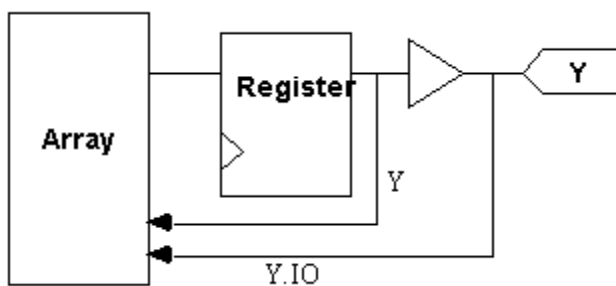
**Figure 1-24. .IMUX Extension**

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.



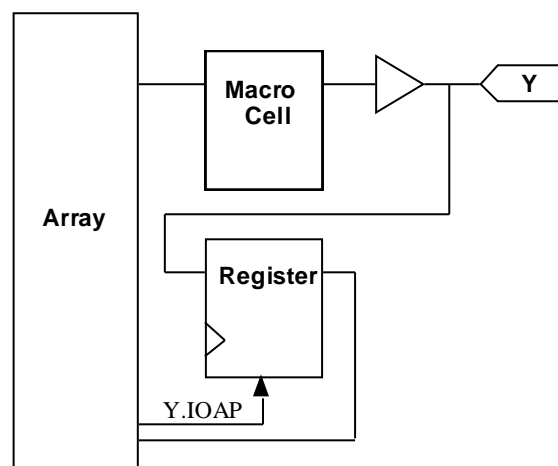
**Figure 1-25. .INT Extension**

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 1-26. .IO Extension**

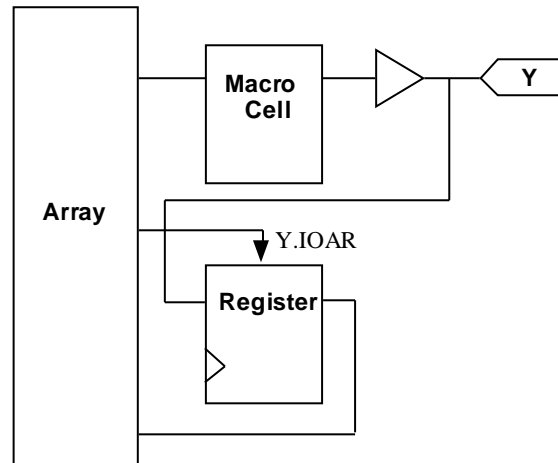
The .IO extension is used to select pin feedback when the macrocell is configured as registered.



**Figure 1-27. .IOAP Extension**

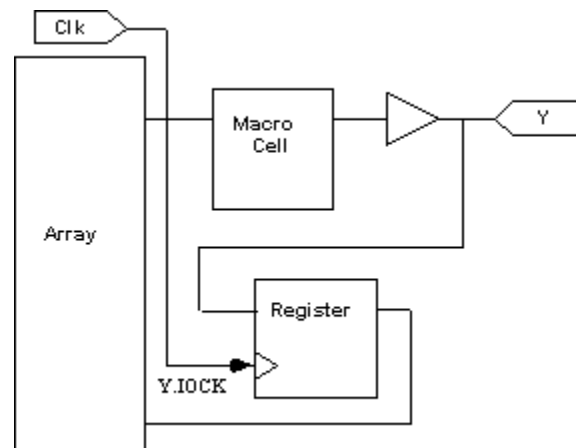


The .IOAP extension is used to specify the expression for Asynchronous Preset in cases where there is registered pin feedback from an output macrocell.



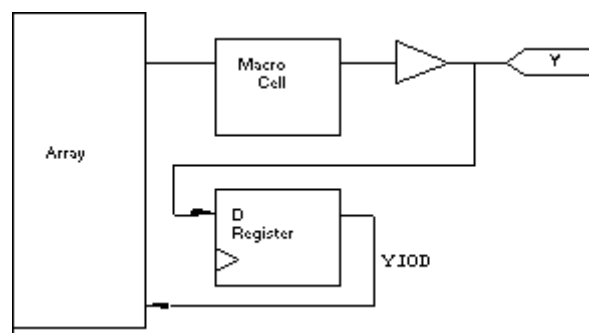
**Figure 1-28. .IOAR Extension**

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



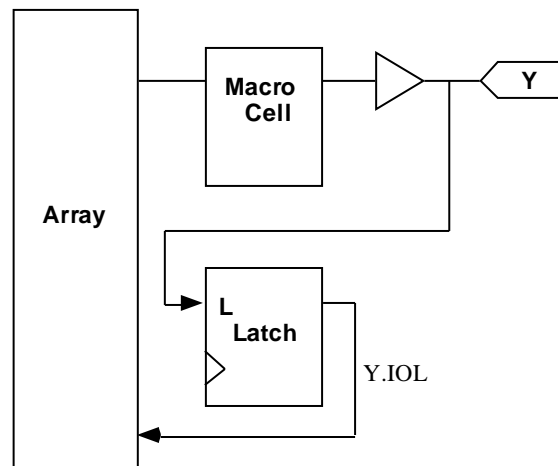
**Figure 1-29. .IOCK Extension**

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



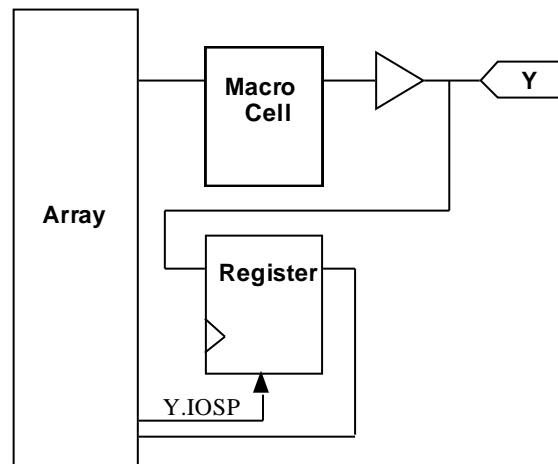
**Figure 1-30. .IOD Extension**

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



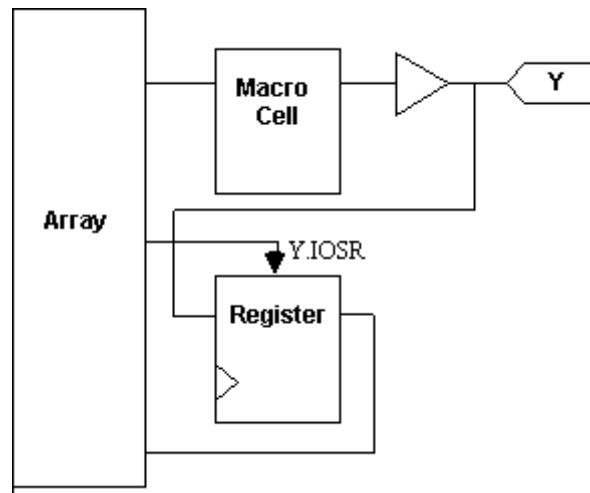
**Figure 1-31. .IOL Extension**

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



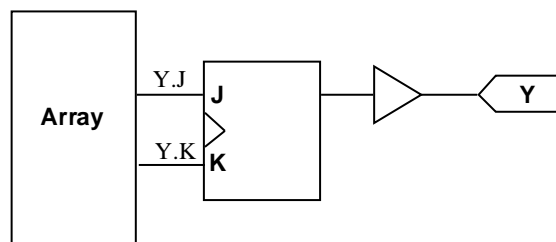
**Figure 1-32. .IOSP Extension**

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



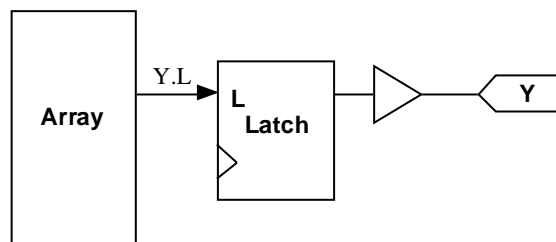
**Figure 1-33. .IOSR Extension**

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



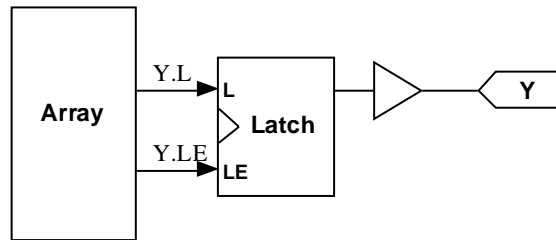
**Figure 1-34. .J and .K Extension**

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.



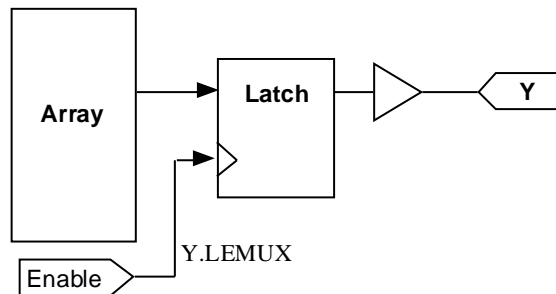
**Figure 1-35. .L Extension**

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.



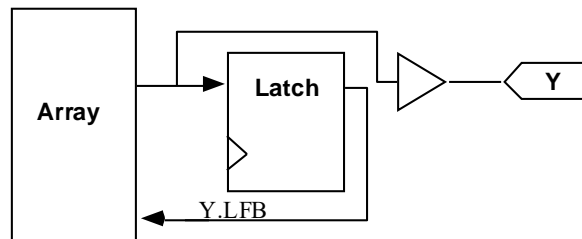
**Figure 1-36. .LE Extension**

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.



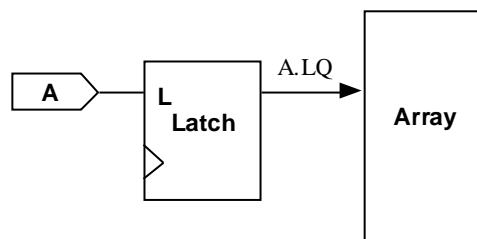
**Figure 1-37. .LEMUX Extension**

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



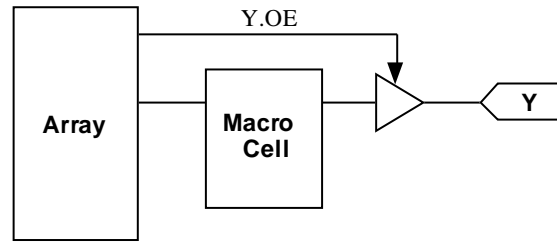
**Figure 1-38. .LFB Extension**

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.



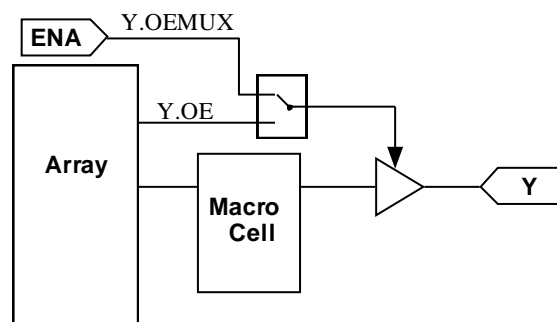
**Figure 1-39. .LQ Extension**

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.



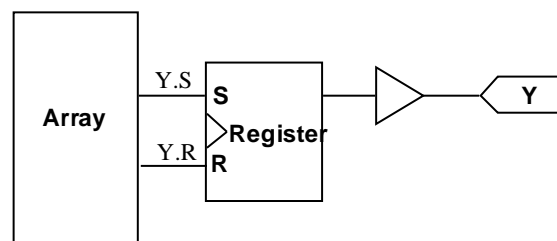
**Figure 1-40. .OE Extension**

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



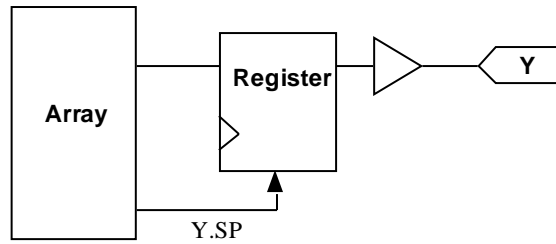
**Figure 1-41. .OEMUX Extension**

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.



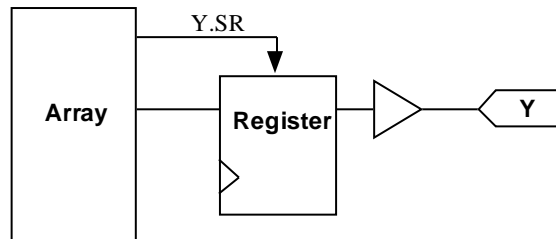
**Figure 1-42. .S and .R Extension**

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.



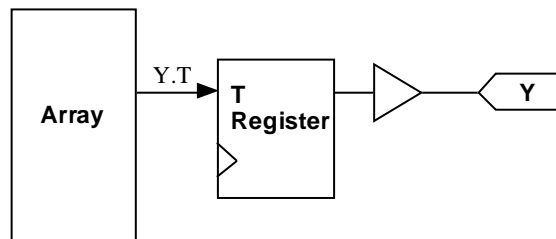
**Figure 1-43. .SP Extension**

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.



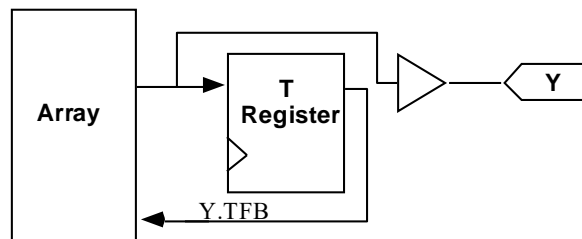
**Figure 1-44. .SR Extension**

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 1-45. .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 1-46. .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.

### 1.1.23.1 Boolean Logic Review

Table 1-12 lists the rules that the CUPL compiler uses for evaluating logic expressions. These basic rules are listed for reference purposes only.

**Table 1-12. 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:**

$$A \& 0 = 0 \quad A \& 1 = A$$

$$A \# 0 = A \quad A \# 1 = 1$$

$$A \& A = A \quad A \& \!A = 0$$

$$A \# A = A \quad A \# \!A = 1$$



## 1.1.24. 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 1-13, note how the order of evaluation and use of parentheses affect the value of the expression.

**Table 1-13. 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's Theorem
A # B & C # D	A # D # B & C	
A # B & (C # D)	A # B & C # B & D	Parentheses change order

### 1.1.25. 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:

```
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).

### 1.1.26. 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 1-14 lists the representation of numbers as sets.

**Table 1-14. Equivalent Binary Sets**

Number	Equivalent Binary Set	Number	Equivalent 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]
'O'6	[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]
'D'8	[1,0,0,0]		
'D'9	[1,0,0,1]		

Numbers may be effectively used as “bit masks” in logic equations using sets. An example of this application is the following 4-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:

1. **[var, var, ... var]: constant ;**
2. **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.



**Note** 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-care." 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 4-bit counter can be specified using the following notation:

**FIELD count = [Q0..3];**

<b>Q0.J</b>	<b>=</b>	<b>count:0 # count:2 # count:4 # count:6</b>	
		<b># count:8 # count:A # count:C #</b>	<b>count:E ;</b>
<b>Q0.K</b>	<b>=</b>	<b>count:1 # count:3 # count:5 # count:7</b>	
		<b># count:9 # count:B # count:D #</b>	<b>count:F ;</b>
<b>Q1.J</b>	<b>=</b>	<b>count:1 # count:5 # count:9 # count:D ;</b>	
<b>Q1.K</b>	<b>=</b>	<b>count:3 # count:7 # count:B # count:F ;</b>	
<b>Q2.J</b>	<b>=</b>	<b>count:3 # count:B ;</b>	
<b>Q2.K</b>	<b>=</b>	<b>count:7 # count:F ;</b>	
<b>Q3.J</b>	<b>=</b>	<b>count:7 ;</b>	
<b>Q3.K</b>	<b>=</b>	<b>count:F ;</b>	

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 Variables 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;
```

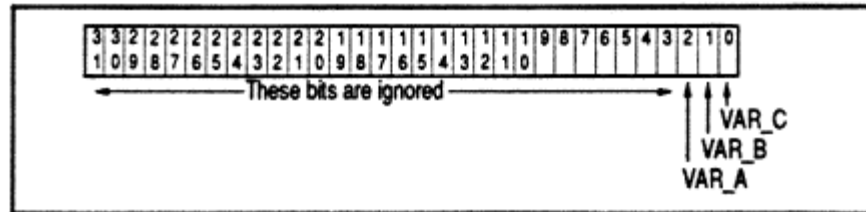
The FIELD statement is used to group a set of related signals into one element. It works by using a 32 bit field where each bit in the field represents one of the members of the field. If there are less than 32 members then the extra bits are ignored. For example:

```

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 1-47. 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:

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```

This would result in the following equations:

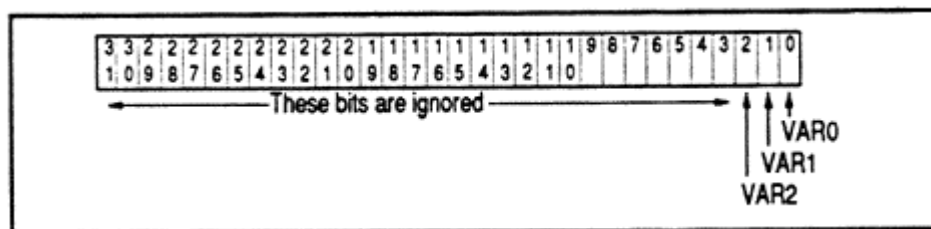
```
ROM_SEL = !VAR_A & VAR_B & VAR_C;
```

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 1-48. 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:

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```



This would result in the following equations:

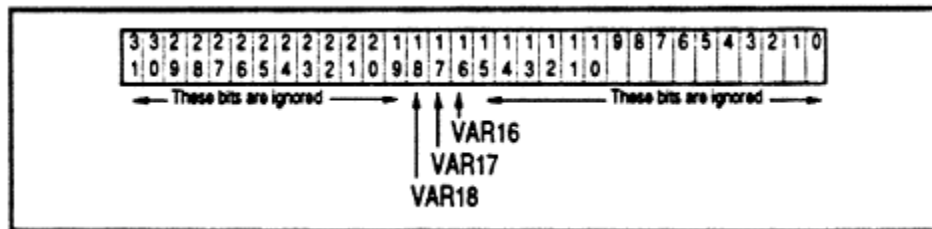
```
ROM_SEL = !VAR2 & VAR1 & VAR0;
```

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 1-49. 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.

```
"XXXXXXXXXXXXXXXXXXXXXXXXXXXX011"
```

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:

```
"XXXXXXXXXXXXX01100000000000000000"
```

```
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:

1. `[var, var, ... var]:[constant_lo..constant_hi];`
2. `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.



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

Format 1 specifies the range operation between a list of variables and a range of constant values. Format 2 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]; /*      order of 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 ;**



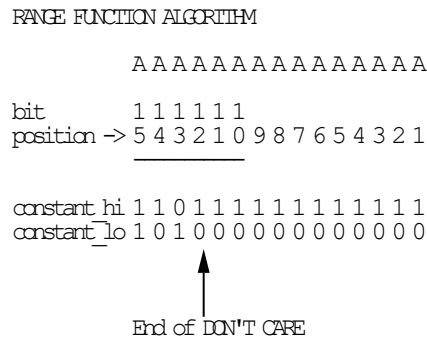
## Note

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.

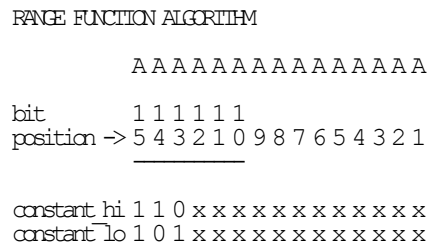
```
field address = [A15..12] ;
board_select = address:[A000..DFFF] ;
```

Figure 1-50 shows how the CUPL algorithm treats this equation.



**Figure 1-50. 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 1-51 shows the result.



**Figure 1-51. RangeFunction Results**

The following two product terms are generated as a result of the range function in Figure 1-51.

```
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 ( $4096 \times 3 + 1$ ). This number of product terms would probably produce an “out of memory” error message because CUPL cannot hold this many product terms in memory at one time.

### 1.1.27. 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 */
}
```

### **1.1.28. 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.

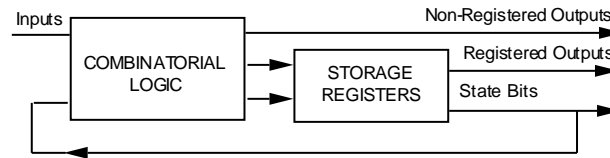
The state-machine approach used with the CUPL compiler-based PLD language permits bypassing the gate and equation level stage in logic design and to move directly from a system-level description to a PLD implementation. Additionally, unlike assembler-based approaches, the state-machine approach allows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



### 1.1.29. 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 1-52. State Machine Model**

The following definitions refer to the timing diagram in Figure 1-52.

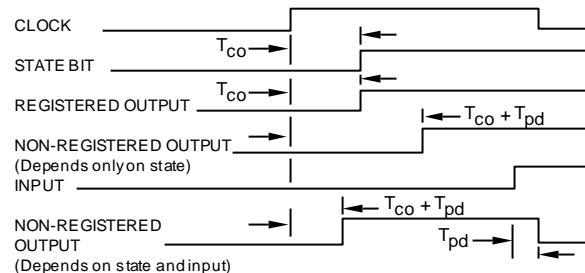
**Inputs** - are signals entering the device that originate in some other device.

**Combinatorial Logic** - is any combination of logic gates (usually AND-OR) that produces an output signal that is valid  $T_{pd}$  (propagation delay time) nsec after any of the signals that drive these gates changes.  $T_{pd}$  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  $T_{co}$  (clock to out time) nsec after the clock pulse occurs.  $T_{co}$  is the time delay between the initiation of a clock signal and the occurrence of a valid flip-flop output.

Figure 1-53 shows the timing relationships between the state machine components.



**Figure 1-53. State Machine Timing 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 ( $T_{su}$ ) usually includes both the propagation delay of the combinatorial logic and the actual setup time of the flip-flops.  $T_{su}$  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  $T_{co} + T_{pd}$  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.



### Note

or S-R type flip-flops.

This hold mode of operation is available only for devices which use D-CE, J-K

### 1.1.30. 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 statements ;  
  .  
  .  
  .  
  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.

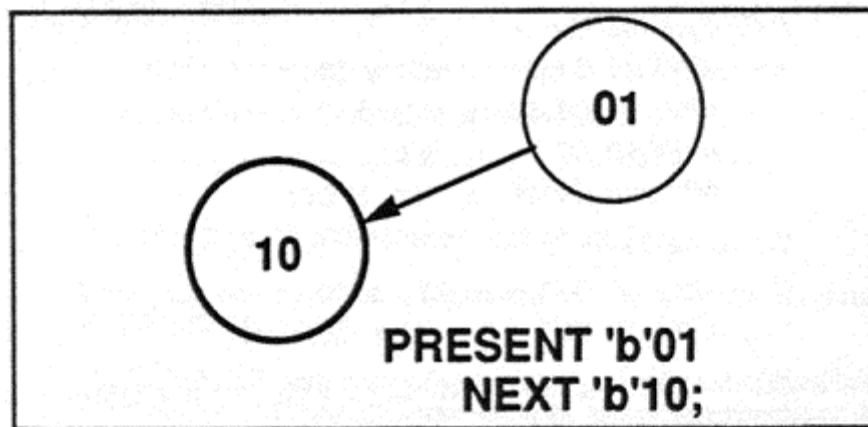
A symbolic name can be assigned with the **\$DEFINE** command to represent **state\_n**.

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 1-54 shows the transition described in the example above.



**Figure 1-54. 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*

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

*J-K-Type Flip-Flop*

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

*S-R-Type Flip-Flop*

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

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

*D-CE-Type Flip-Flop*

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

*T-Type Flip-Flop*

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

See the subtopic, **APPEND** Statements in this chapter for a description of the **APPEND** command.

## 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.



### Note

The square brackets indicate optional items.



### Note

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.



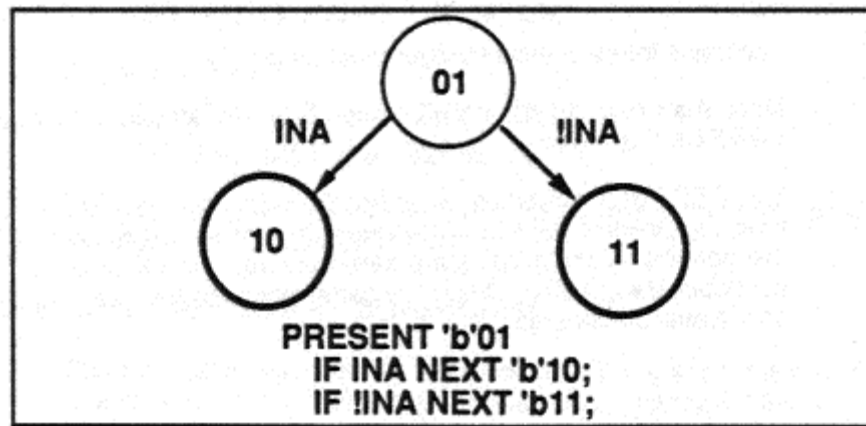
## Note

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 1-55 shows the transitions described by the above example.



**Figure 1-55. 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*

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

### *D-CE-Type Flip-Flop*

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

### *J-K-Type Flip-Flop*

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

### *S-R-Type Flip-Flop*

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

```

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

```

*T-Type Flip-Flop*

```

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

```

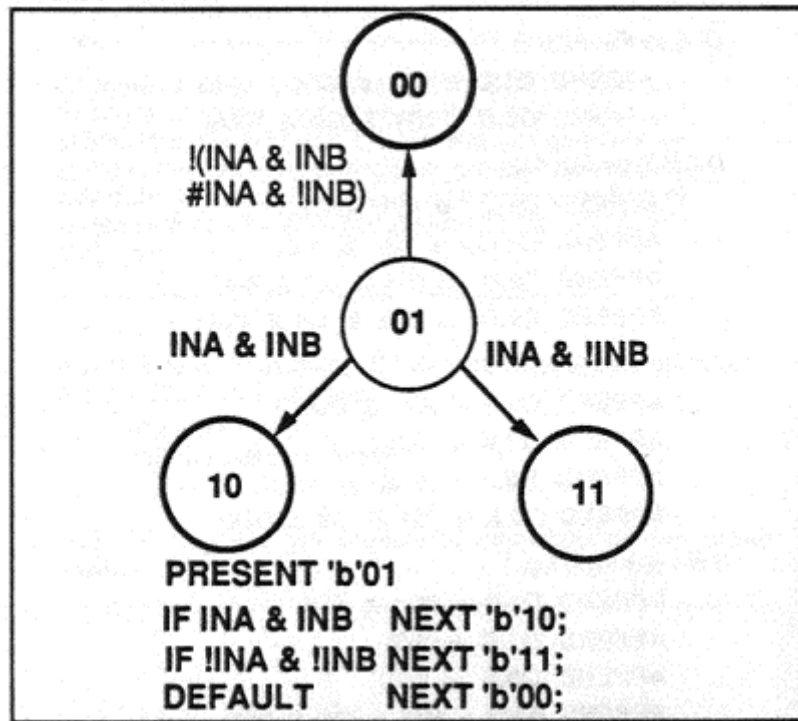
The following is an example of two conditional statements with a **DEFAULT** statement.

```

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

```

Figure 1-56 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.



**Figure 1-56. Conditional NEXT Statement 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*

```

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

```

*D-CE-Type Flip-Flop*

```

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

```

```

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

```

#### *J-K-Type Flip-Flop*

```

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

```

#### *S-R-Type Flip-Flop*

```

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

```

#### *T-Type Flip-Flop*

```

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



### **Note**

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.





## Note

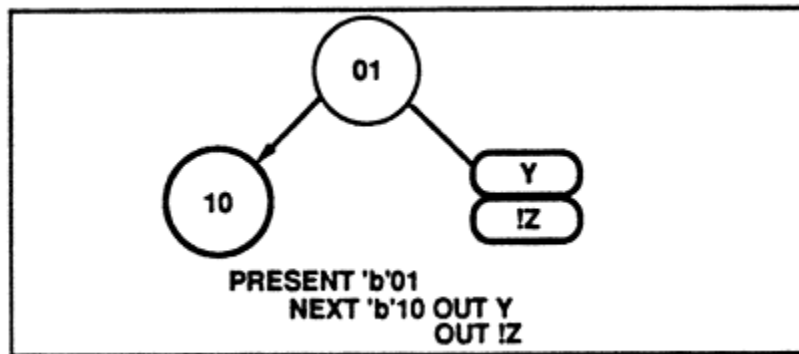
Use the negation mode only for D-CE, J-K, T or S-R type flip-flops; D-type flip-flops implicitly reset when assertion is not specified.

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

**PRESENT 'b'01**

**NEXT 'b'10 OUT Y OUT IZ ;**

Figure 1-57 shows the transition and output variable definition described in the example above.



**Figure 1-57. 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*

**APPEND Y.D = !Q1 & Q0;**

(not defined for Z output)

### *D-CE Type Flip-Flop*

**APPEND Y.D = !Q1 & Q0;**

**APPEND Y.CE = !Q1 & Q0;**

**APPEND Z.D = 'b'0;**

**APPEND Z.CE = !Q1 & Q0;**

### *J-K-Type Flip-Flop*

**APPEND Y.J = !Q1 & Q0;**

**APPEND Y.K = 'b'0;**

**APPEND Z.J = 'b'0;**

**APPEND Z.K = !Q1 & Q0;**

### *S-R-Type Flip-Flop*

**APPEND Y.S = !Q1 & Q0;**

**APPEND Y.R = 'b'0;**

**APPEND Z.S = 'b'0;**

**APPEND Z.R = !Q1 & Q0;**

### *T-Type Flip-Flop*

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



**Note** 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.



**Note** 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.



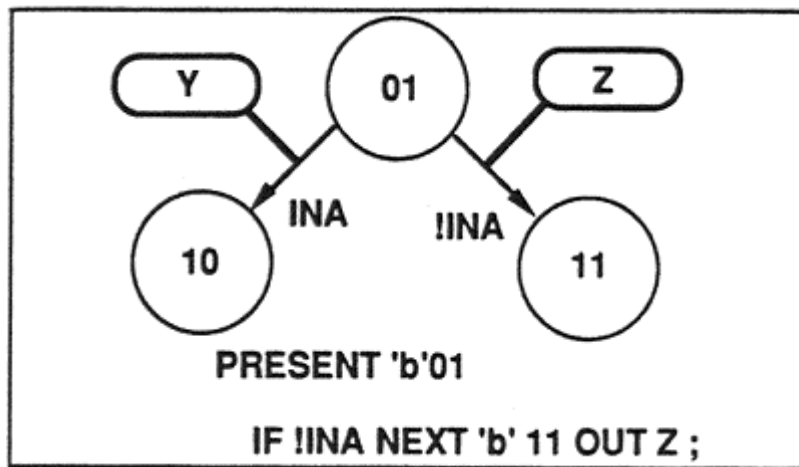
## Note

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 1-58 shows the transitions and outputs defined by the statements in the example above.



**Figure 1-58. 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*

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

### *D-CE-Type Flip-Flop*

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

### *J-K-Type Flip-Flop*

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

*S-R-Type Flip Flop*

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

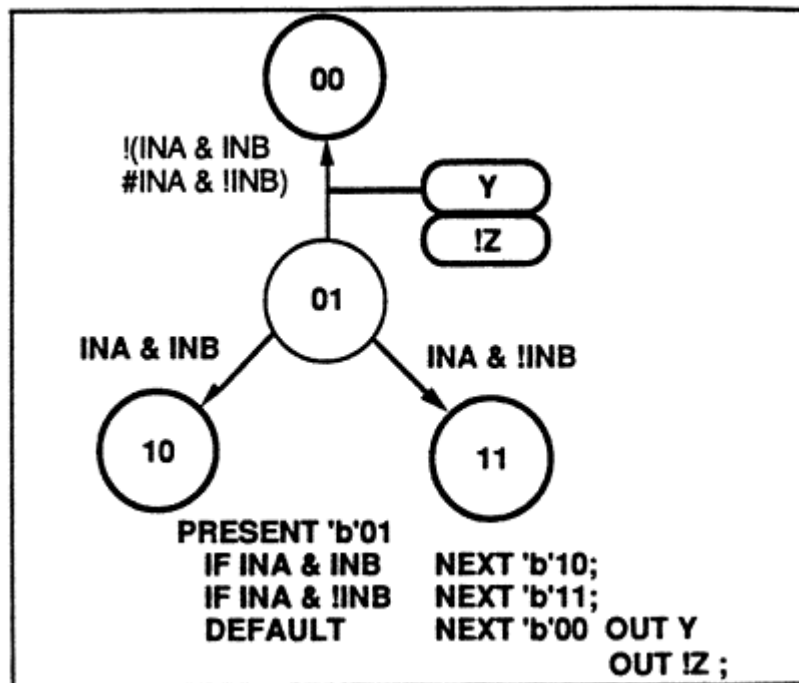
*T-Type Flip-Flop*

```
APPEND Y.T    =    !Q1 & Q0 & INA;  
APPEND 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 1-59 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.



**Figure 1-59. 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*

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

#### *D-CE-Type Flip-Flop*

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

#### *J-K-Type Flip-Flop*

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

#### *S-R-Type Flip-Flop*

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

#### *T-Type Flip-Flop*

```
APPEND Y.T      =    !Q1 & Q0 & !INA  
APPEND 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.

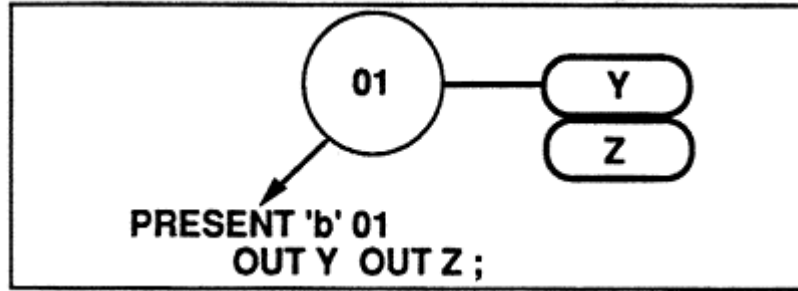
Negating the variable (with the complement operator) is not a valid format for this statement.

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 1-60 shows the outputs defined by the statements in the example above.



**Figure 1-60. Unconditional Asynchronous Output Diagram**

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

```
APPEND Y = !Q1 & Q0;
APPEND 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.



### Note

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.



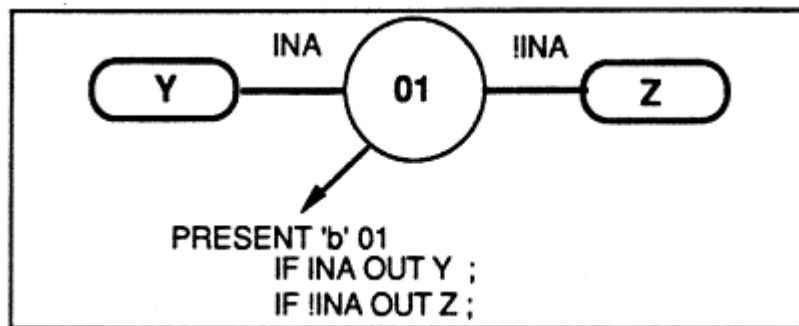
### Note

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 1-61 shows the outputs defined by the statements in the above example.



**Figure 1-61. Conditional Asynchronous Output Diagram**

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

```
APPEND Y = !Q1 & Q0 & INA;
APPEND 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 1-62 shows the transitions described by the above example. Note the equation generated by the **DEFAULT** statement.

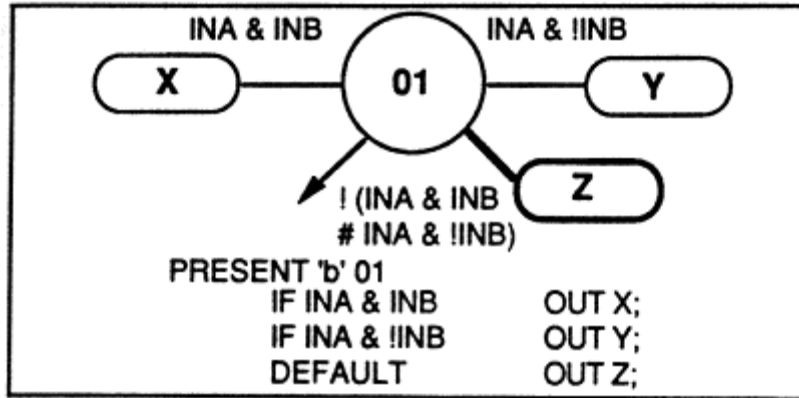


Figure 1-62. Conditional Asynchronous Output with Default Diagram

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.

```

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

```

## Sample State-Machine Syntax File

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

Figure 1-63 shows a diagram of the counter operation.

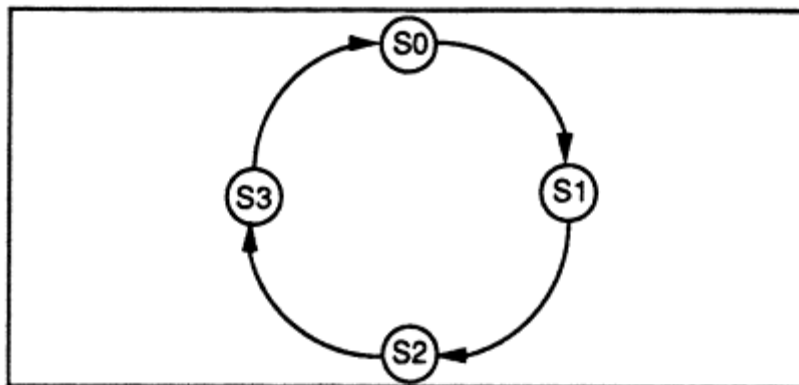


Figure 1-63. Simple 2-Bit Counter Diagram

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, in Chapter U5 for another illustration of a state machine implementation.

### 1.1.31. 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.



#### Note

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 ;
```

### 1.1.32. 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 ([parameter0,.....,parameterN])  
{      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.



#### Note

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.



### Note

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) ;
```

#### Input Files

A **logic description source file** (*filename.PLD*) is the input to **CUPL**. This file describes the logical functionality to assign to a specified target device.

The source file is created using a standard text editor. There are a wide variety of text editors available and the choice depends entirely upon personal preference. The only requirement is that it be able to produce a standard text file.

## **1.2 Simulator Reference**

This chapter explains how to use **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.

### 1.2.1. INPUT

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 EDLIN or WordStar 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.

**Table 2-1. CSIM Device Mnemonic Prefixes**

<b>Prefix</b>	<b>Device Family</b>
EP	Erasable Programmable Logic Device (EPLD)
G	Generic Array Logic (GAL)
F	Field Programmable Logic Array (FPLA)
F	Field Programmable Gate Array (FPGA)
F	Field Programmable Logic Sequencer (FPLS)
F	Field Programmable Sequence Generator (FPSG)
P	Programmable Logic Array (PAL)
P	Programmable Logic Device (PLD)
P	Programmable Electrically Erasable Logic (PEEL)
PLD	Pseudo Logical Device
RA	Bipolar Programmable Read-Only Memory (PROM)

For example, the device mnemonic for a PAL10L8 is P10L8; for an 82S100 the device mnemonic is F100. For bipolar PROMs, the suffix is the array size. For example, the device mnemonic for a 1024 x 8 bipolar PROM is RA10P8, since there are 10 address input pins and 8 data output pins.



## 1.2.2. OUTPUT

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.



### Note

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

### **1.2.3. 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.

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 simulation 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.

## 1.2.4. 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 2-3 lists the keywords used for header information (see the subtopic, Header Information in Chapter 1):

**Table 2-3. 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.

### **1.2.5. 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.

## 1.2.6. 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 in Chapter 2). 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**



### Note

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.



### Note

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



## Note

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 2-4 lists allowable 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 2-6.

**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 2-6. 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 */
```



### Note

**CSIM** can simulate and 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. (See Figure 2-7. It shows the diagram for a circuit with three inputs [A, B, and C] and an output at Y that feeds back.)

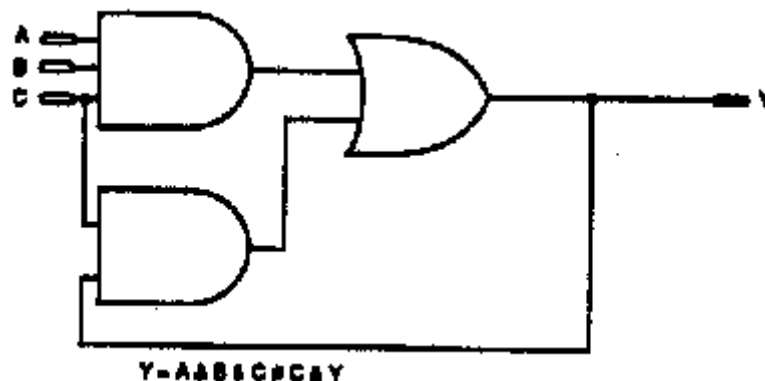


Figure 2-7. Circuit with Feedback

The equation for the output at Y is as follows:

$$Y = A \& B \& C \# C \& Y$$

The vectors table in Figure 2-8 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 2-8. 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 (labelled “a” in Figure 2-9).

	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

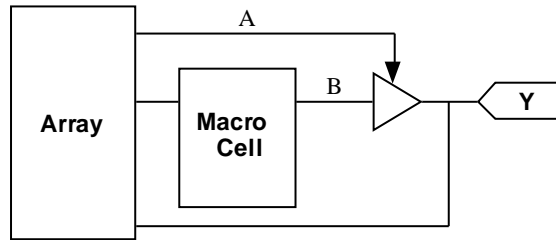
**Figure 2-9. Vectors Table with Intermediate Results**

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 2-10. I/O Pin Simulation**

The following equations express the Boolean equation representation of Figure 2-10:

```

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    gl6v8;

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 2-11. 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    gl6v8;
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 2-12. TEST.SO**

## Random Input Generation

A new test vector value is now supported:

R - it can appear wherever a 0 or a 1 can appear. When encountering such a value, CSIM generates a random value ( that is 0 or 1) for the corresponding signal in that test vector.



### Note

R can only be used to generate random input values

In the SI file:

```

$repeat 10;
C  0 RRR 1RRRRRRR  *****

```

**Figure 2-15. R in the .SI file**

In the SO file:

0035:	C	0	000	10001011	HLLLHLHH
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

**Figure 2-16. R in the .SO file**

## 1.2.7. 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 2-5 lists the **CSIM** directives.

Table 2-5. CSIM Directives

<b>\$MSG</b>	<b>\$REPEAT</b>	<b>\$TRACE</b>
<b>\$SIMOFF</b>	<b>\$SIMON</b>	<b>\$EXIT</b>

### **\$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 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;
....

```

**Figure 2-17. Using P22V10.**

Figure 2-17 shows the simulation result file:

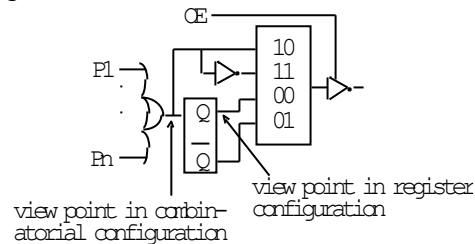
```

order CLK, IN2, IN3, IN4, . OUT14, OUT15 . ;
*****before output buffer*****
      ??? .. LL ...
0001:0011 .. HH ...
.....
*****before output buffer*****
      ???  HH...
0004 C100...ZZ
.....

```

**Figure 2-18. Simulation File.**

Figure 2-18 shows the virtual observation points when using trace level 4 with either a combinatorial configuration or a register configuration.



**Figure 2-19. Observation Points Using Trace Level 4.**

## **\$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**

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.

### 1.2.8. 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;



### Note

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 4-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 2-6. Logic Operators**

Operator	Function	Precedence
*	multiplication	1
/	division	1
+	addition	2
-	subtraction	2

**Table 2-7. 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 2-20. .SI file**

This will produce this result in the SO file:

```
0001: 0    0    XXX XXXXXXXX  LLLLLLLL  
0002: C    0    000 10000000  HLLLLLLL
```

**Figure 2-21. .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.



### Note

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.



### Note

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      05/11/89;
Revision  02;
Designer  Kahl;
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 2-22. .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 2-23. .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 2-24. .SI file**

### 3. The Output:

```

CSIM: CUPL Simulation Program Version
4.2a Serial# ...
Copyright (c) 1983, 1991 Logical Devices, Inc.
CREATED Wed Dec 04 03:00:11 1991
LISTING FOR SIMULATION FILE: barrel22.si
1: Name      Barrel22;
2: Partno    CA0006;
3: Date      05/11/89;
4: Revision  02;
5: Designer  Kahl;
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:
=====

```

	$\overline{C}$	$\overline{O}$	shi		
	C	E	ft	input	output
0001:	0	0	XXX	XXXXXXXXX	LLLLLLLLL
0002:	C	0	000	10000000	HLLLLLLLL
0003:	C	0	001	10000000	LHLLLLLLL
0004:	C	0	010	10000000	LLHLLLLLL
0005:	C	0	011	10000000	LLLHLLLLL
0006:	C	0	100	10000000	LLLLHLLLL
0007:	C	0	101	10000000	LLLLLHLLL
0008:	C	0	110	10000000	LLLLLLHLL
0009:	C	0	111	10000000	LLLLLLLHL

**Figure 2-25. .SO file sheet 1**

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	HHHHHLLH
0017:	C	0	111	01111111	HHHHHHHL

**Figure 2-26. .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 100LLLL L /* synchronous clear to state 0 */
C 000LLHL L /* count up to state 1 */
C 000LLHL L /* count up to state 2 */
C 000LLHL L /* count up to state 3 */
C 000LHLL L /* count up to state 4 */
C 000LHLH L /* count up to state 5 */
C 000LHHL L /* count up to state 6 */
C 000LHHH L /* count up to state 7 */
C 000HLLL L /* count up to state 8 */
C 000HLLH H /* count up to state 9 - carry */

```

**Figure 2-27. 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 2-29. 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 2-30. 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 2-31. Simulation Input (Correct)**

### 1.3. **CUPL Modules**

The CUPL package consists of the following programs: **CUPL** (Universal Compiler for Programmable Logic), **CSIM** (CUPL Simulator), **CBLD** (CUPL Build), and **PTOC** (PALASM to CUPL Translator).

Using CUPL, logic descriptions may be written and compiled. The logic descriptions can be assigned to specific programmable logic devices (PLDs). CUPL searches its library and creates a file which can be downloaded to a device programmer and, from there, the PLD can be programmed.

**CSIM** permits a design to be simulated before it is put into production. A file is created, describing the expected functionality of the PLD in terms of input and output values. **CSIM** compares the expected values to actual values calculated during CUPL operation. The DOS version of **CSIM** will also allow the simulation to be viewed in waveform.

**CBLD** allows the manipulation of the device libraries containing descriptions of the PLDs supported by CUPL.

**PTOC** converts PALASM designs to CUPL format.

### 1.3.1. 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. The CUPL language translator (**PTOC**) converts PALASM designs to CUPL, enabling the use of any PALASM designs.

**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.

### 1.3.2. CUPL DATA FLOW

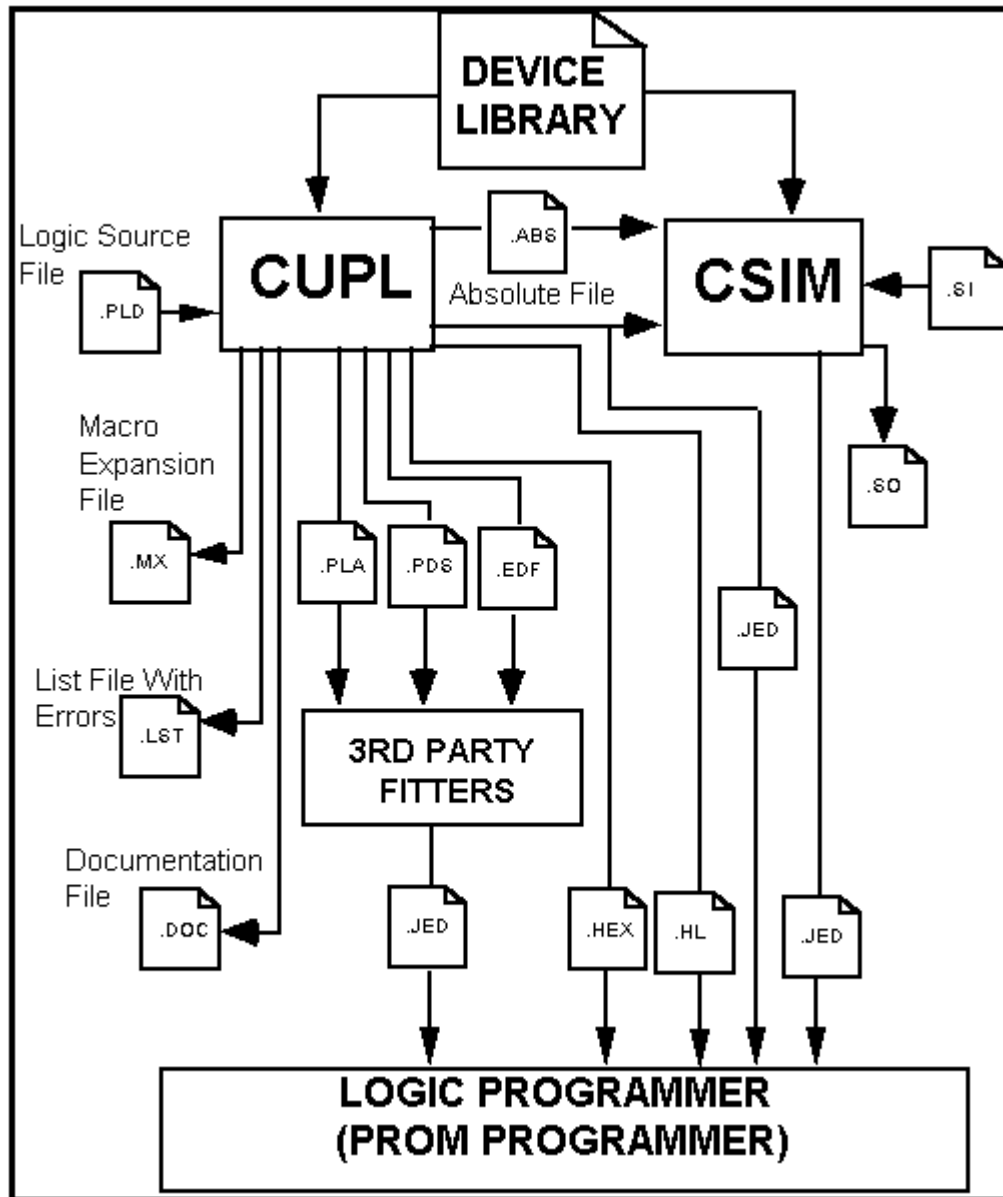
This section describes the CUPL data flow.

First, a logic description source file is created, using the CUPL language to describe the logic that will be assigned to a programmable logic device.

Then, run **CUPL** to compile the source file to create a fusemap file for downloading to a device programmer. On execution of **CUPL**, the '-a' flag may be used to cause **CUPL** to generate an '.ABS' file for later use by **CSIM**.

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**.

At this point, the verified fusemap file can be transferred to a device programmer. Figure U1-1 shows the CUPL data flow.



**Figure U1-1. CUPL Data Flow**

Output file types:

.SO	Simulator Output with errors
K.HL	HL Download file
.HEX	Hex Download file
K.JED	JEDEC file without test vectors
	JEDEC file with test vectors



### 1.3.3 CUPL.DL

The **CUPL** compiler must 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 CUPL. The library describes the physical characteristics of each device, including internal architecture, number of pins, and valid input and output pins, and also describes the logical characteristics, including registered and non-registered pins, number of product terms, fuse map information, and download format information.

The target device is referenced using device mnemonics. The mnemonic is composed of a device family prefix and industry standard part number suffix. Table U4-1 lists the device mnemonic prefixes.

#### U4-1. CUPL Device Mnemonic Prefixes

Symbol	Meaning
EP	Erasable Programmable Logic Device (EPLD)
G	Generic Array Logic (GAL)
F	Field Programmable Logic Array (FPLA)
F	Field Programmable Gate Array (FPGA)
F	Field Programmable Logic Sequencer (FPLS)
F	Field Programmable Sequence Generator (FPSG)
P	Programmable Logic Array (PAL)
P	Programmable Logic Device (PLD)
P	Programmable Electrically Erasable Logic (PEEL)
PLD	Pseudo Logical Device
RA	Bipolar Programmable Read Only Memory (PROM)



#### Note

A CUPL device mnemonic can reference multiple manufacturers. Please refer to mnemonic vs. manufacturer cross reference found in the file, Devices.TXT.

For example, the device mnemonic for a PAL10L8 is P10L8; for an 82S100 the device mnemonic is F100. For bipolar PROMs, the suffix is the array size; for example, the device mnemonic for a 1024 x 8 bipolar PROM is RA10P8, since there are 10 address input pins and 8 data output pins.

#### Device Selection

This section will describe how to tell the CUPL compiler what target device to use.

The development cycle, when designing with PLDs, is one of the first creating a source file and then compiling it to produce a JEDEC file which is then used to program an actual device. The CUPL compiler is like any compiler in that it processes a source file and produces output based on a target architecture. This compiler, however, has hundreds of different target architectures. This affords great flexibility because it allows a design to be implemented in many different architectures without making significant changes to the source file.

When we instruct the compiler to begin processing it must be supplied with several pieces of information. We must tell it what file contains the design equations, what type of output we want, and what device architecture is desired. It is the last part that we will discuss here.

## Device Mnemonics

CUPL uses a naming system for device architectures called device mnemonics. It is important to note that this naming system is for architectures, not for the devices themselves. For example, Altera EP312 and Intel 5AC312 both have the same architecture. Therefore, if you were doing a design with either of these devices from these manufacturers, you would tell CUPL that your target architecture is EP312. There is an addendum containing a cross reference of manufacturers' names for devices with their corresponding CUPL mnemonic.

## LCC and PLCC Devices

The default package type used by CUPL is the DIP package. Some devices have LCC or PLCC versions. These are sometimes referred to as Surface Mount Technology or SMT for short. Although the architecture of the SMT devices are the same as the DIP versions, the pinouts are different. For this reason, CUPL uses a different mnemonic for the SMT versions of device architectures. This involves adding the suffix "lcc" to the DIP mnemonic. The SMT version of the P16L8 is P16L8LCC.

## Specifying A Device

There are three ways to specify a device to the compiler on the DOS platform.

- In the source PLD file

- On the command line (not available on Macintosh version)

- From the menu (currently available on DOS only).

On UNIX or VMS there are currently two ways to specify a device, via the command line or in the PLD file. On the Macintosh the device is selected in the PLD file or from the device selection menu.

### Specifying a Device in the Source PLD File

If desired, the target architecture may be specified in the PLD source file. This is done by using the "DEVICE" field in the header section of the PLD file. The PLD file contains a header section which is comprised of several fields. Each field is composed of a keyword followed by some information.

The syntax of the "DEVICE" field is:

```
DEVICE mnemonic;
```

Example: DEVICE P16L8;

When the device architecture is specified in the PLD file, it is not necessary to specify it again on the command line or from the menu. If a device mnemonic is specified on the command line or from the menu, the device mnemonic specified in the PLD file is ignored. Thus, the command line or menu device selection has a higher priority when it comes to specifying a device.

### Specifying A Device On the Command Line

CUPL may be run from the command line or from a batch file or UNIX script. This command line contains option flags for directing the compiler in several ways. It also contains the name of the source file, an optional device library, and optionally, the device mnemonic. The command line looks as follows:

CUPL <flags> <source>

CUPL <flags> <library> <source>

CUPL <flags> <mnemonic> <source>

CUPL <flags> <library> <mnemonic> <source>.

When a device is specified on the command line, this becomes the target architecture, regardless of what has been specified in the PLD file.

Welcome to CUPL Universal Compiler for Programmable Logic. CUPL is a very versatile and powerful logic compiler, and can be used to create very sophisticated logic designs for PLDs. This section will introduce the user to CUPL.

### 1.3.4. SYSTEM OVERVIEW

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.

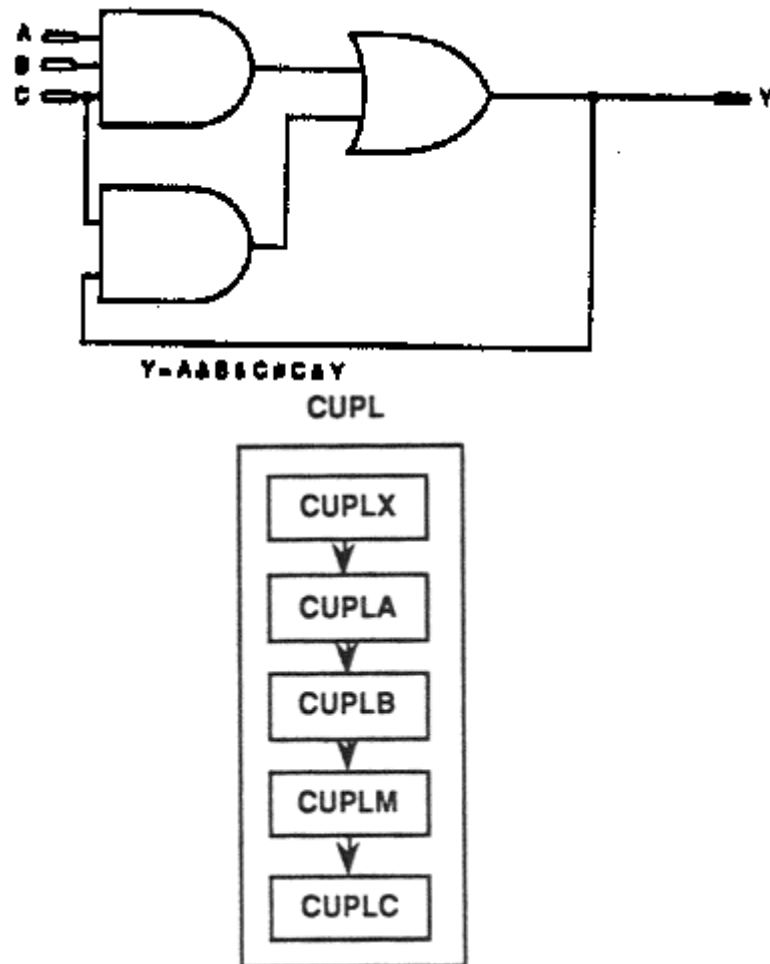


Figure U1-2. CUPL Execution Flow

#### **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,

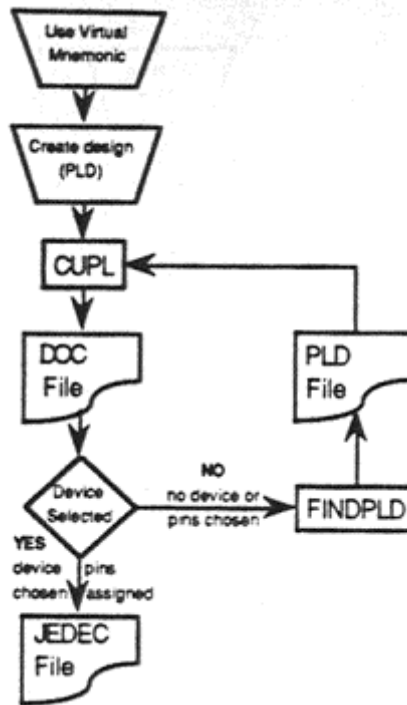
containing device model links, and bit mapped representation of the logic design.

#### **CUPLM: CUPL Logic Minimizer**

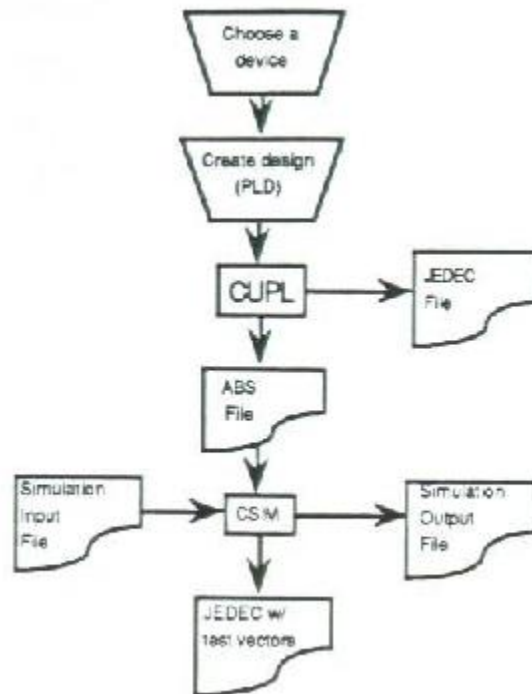
Executes logic minimization algorithms on the bit mapped logic generated in CUPLB. Processes only the equations for which reduction has been requested. If no reduction is requested then the intermediate file generated by CUPLB is renamed for use by CUPLC. Performs multiple output minimization for FPLA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

#### **CUPLC: CUPL Design Fitter**

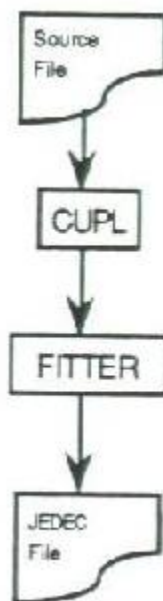
Determines if the design fits the target device architecture and builds a fuse map. The fuse map and symbol table are used to generate the documentation and JEDEC files.



**Figure U1-3. Device Independent Design Flow**



**Figure U1-4. Device Specific Design Flow**



**Figure U1-5. CUPL Device Fitting**

Some Fitter programs currently available include:

CUPLC	generic device fitter
FITR	MACH device fitter
FITMAPL	MAPL device fitter.

## 1.4 Minimization Techniques

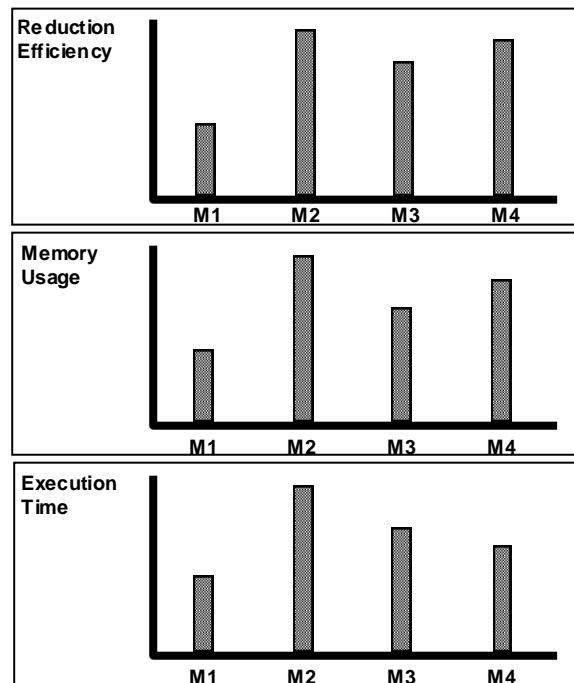


Figure U4-1. Logic Minimization Levels

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

## 1.5 Using the CUPL at the command line

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

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

where

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

<b>-j</b>	JEDEC download format
<b>-h</b>	ASCII-HEX download format
<b>-i</b>	HL download format
<b>-n</b>	use input filename for output file
<b>-a</b>	create absolute file
<b>-l</b>	create listing file
<b>-e</b>	create expanded macro definition file
<b>-x</b>	create expanded product-terms in documentation file
<b>-f</b>	create fuse plot/chip diagram in documentation file
<b>-p</b>	create PDIF database interchange format file
<b>-b</b>	create Berkeley PLA format file
<b>-c</b>	create PALASM format file
<b>-d</b>	deactivate unused OR terms
<b>-r</b>	disable product term merging
<b>-g</b>	program security fuse
<b>-u</b>	use specified library for compilation
<b>-s</b>	perform logic simulation after compilation
<b>-w</b>	perform simulation with waveform output (MS-DOS only)
<b>-m0</b>	no minimization
<b>-m1</b>	quick minimization (default)
<b>-m2</b>	Quine McCluskey
<b>-m3</b>	Presto
<b>-m4</b>	Expresso
<b>-q</b>	Microsoft format for error messages

**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.



### Note

brackets indicate optional items.

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 U4-2 lists descriptions of the **CUPL** option flags and output files. An introductory example will be presented in the next chapter.

**Table U4-2. Compiler Option Flags**

Option Flag	Description
<b>j</b>	Generates a JEDEC-compatible ASCII download file ( <b>filename.JED</b> ). The filename is not necessarily the same as the logic description filename input to <b>CUPL</b> . The <b>NAME</b> statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
<b>c</b>	This can be used to create an input file for other logic design tools and gate array fitters such as PDS2XNF from XILINX.
<b>h</b>	Generates an ASCII-hex download file ( <b>filename.HEX</b> ). This format is available only for PROMs. The filename is not necessarily the same as the logic description filename input to <b>CUPL</b> . The <b>NAME</b> statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
<b>i</b>	Generates an HL download file ( <b>filename.HL</b> ). This format is available only for the Signetics IFL devices. The filename is not necessarily the same as the logic description filename input to <b>CUPL</b> . The <b>NAME</b> statement in the header information section of the logic description file determines the download filename (see the subtopic, Header Information in this chapter).
<b>a</b>	Generates an absolute file ( <b>filename.ABS</b> ) for use by the <b>CSIM</b> logic simulation program.
<b>n</b>	Allows the source filename to be used as the JEDEC filename instead of using the name in the <b>NAME</b> field of the source file.
<b>l</b>	Generates an error listing file ( <b>filename.LST</b> ). 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.
<b>x</b>	Generates a documentation file ( <b>filename.DOC</b> ) 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.
<b>f</b>	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>P</b>	Generates a PDIF (P-CAD Database Interchange Format) file ( <b>filename.PDF</b> ) which can be translated by the <b>PDIFIN</b> program into a symbol for the PC-CAPS (P-CAD Schematic Capture) program. The generated symbol will contain packaging information for the PLD.
<b>B</b>	Generates a Berkeley PLA file ( <b>filename.PLA</b> ) for use by the Berkeley PLA tools, such as PLEASURE, or other PLA layout tools which use the Berkeley PLA format. The compiler
<b>d</b>	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** ds 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.
- 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 U4-3 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.

**Table U4-3. Boolean Logic Rules**

Expression	Result
!0	= 1
!1	= 0
A & 0	= 0
A & 1	= A
A & A	= A
A & !A	= 0

$$A \# 0 = A$$

$$A \# 1 = 1$$

$$A \# A = A$$

$$A \# !A = 1$$

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

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

## 1.7. Run CSIM using the following command line

**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.
- w** (MS-DOS only) simulate and display output file in                      waveform.
- d** (MS-DOS only) display an existing simulation  
output file in waveform.

**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.



### Note

The square brackets indicate optional items.

### 1.7.1. 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 2-2 lists descriptions of the **CSIM** option flags.

**Table 2-2. 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 ( <i>filename.SO.</i> ) 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 <span>Ctrl</span> - <span>S</span> to stop the display (and any key to start it again) or <span>Ctrl</span> - <span>C</span> 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.
-d	(MS-DOS only) Displays an existing simulation output file in waveform. Type <span>F1</span> , <span>F2</span> , <span>F5</span> , <span>F6</span> , <span>↑</span> , <span>↓</span> , <span>←</span> , <span>→</span> keys to view the waveform output. Type <span>F4</span> to exit.
-w	(MS-DOS only). Generates a simulation listing file and displays the output in waveform. Use the <span>F1</span> , <span>F2</span> , <span>F5</span> , <span>F6</span> , { <span>↑</span> , <span>↓</span> , <span>←</span> , <span>→</span> } keys to view the waveform output. Type <span>F4</span> to exit.

## #2. Design Example

Locate all of the useful design examples found in the CUPL manuals.

This chapter provides examples of using **CUPL** and **CSIM**. It is divided into two parts.

Part A provides step-by-step instructions through a sample design session. Part B describes some of the designs that can be implemented with the logic description files provided with the CUPL package.

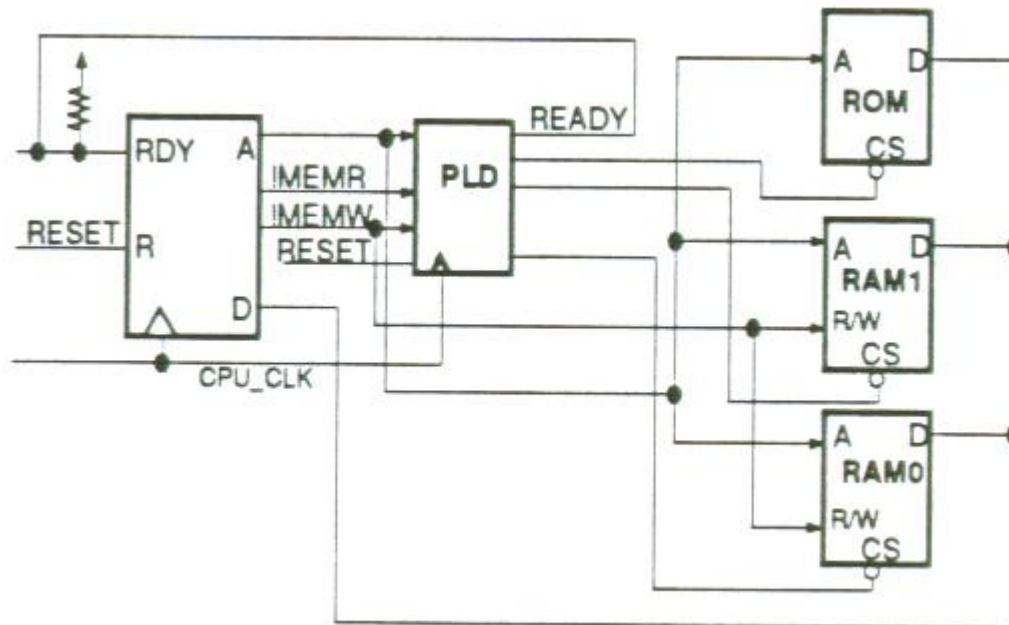
## **2.1. SAMPLE DESIGN SESSION**

This part provides step-by-step instructions through a sample design session using **CUPL** and **CSIM**. The steps in the process are:

- Step 1.     Examining the Design Task
- Step 2.     Creating the **CUPL** Source File
- Step 3.     Formulating the Equations
- Step 4.     Choosing a Target Device
- Step 5.     Making the Pin Assignments
- Step 6.     Running **CUPL**
- Step 7.     Creating the **CSIM** Source File
- Step 8.     Running **CSIM**

### 2.1.1. STEP 1. EXAMINING THE DESIGN TASK

The system in this programmable logic device (PLD) design example is microprocessor-based, with the CPU interfacing with ROM and RAM. Figure U5-1 shows a diagram of the system.



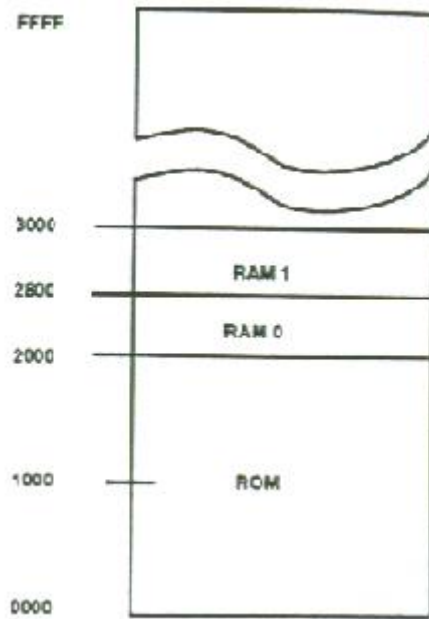
**Figure U5-1. Microprocessor-Based System**

A PLD provides a flexible interface between the CPU and peripherals by performing address decoding and timing control functions. As the diagram shows, a ROM (or PROM) is used for system control and two static RAMs are used for scratch pad and auxiliary memory functions.

In this sample session, a PLD will be designed that decodes the CPU's address using a memory map, and creates chip select signals for the ROM and RAM chips based upon CPU address and CPU data strobes.

The memory map in Figure U5-2 shows where the ROM and two RAM chips reside in the CPU's addressing space.



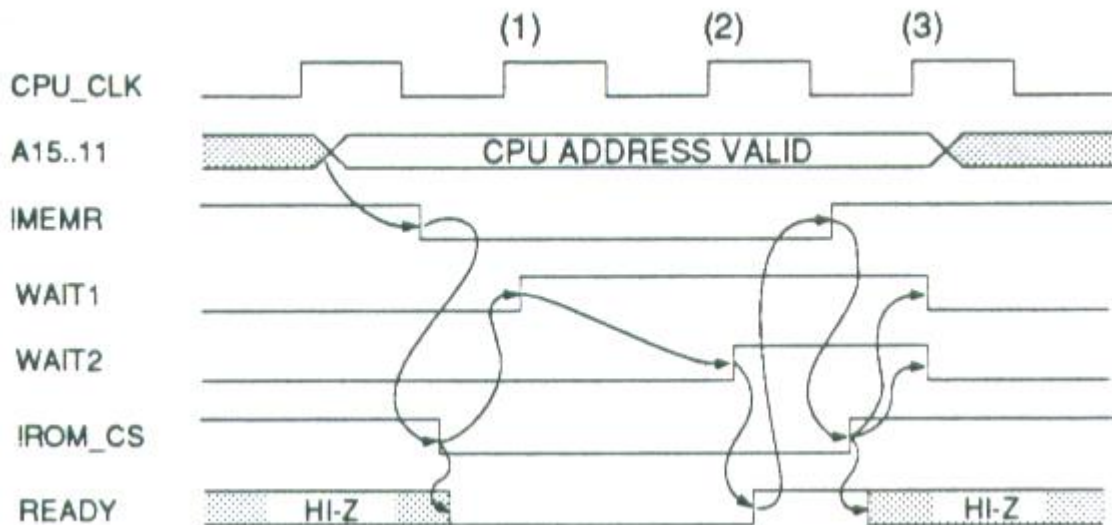


**Figure U5-2. Memory Map**

Addresses are marked and shown in hexadecimal in the memory map. Use this memory map when designing the logic for the PLD.

Because the ROM chip is slow, the PLD must be designed to perform a wait state generation that adds at least one CPU clock period to the ROM access.

The worm arrows on the timing diagram in Figure U5-3 show signals affected or created by other signals.



**Figure U5-3. Wait State Generator Timing Diagram**

A description of the operation of the timing diagram follows. The numbers in parentheses indicate the rising edge of the **CLOCK** signal.

A wait state sequence starts with the CPU address becoming valid prior to the memory read strobe. Only

the **!MEMR** signal needs to be considered, because the wait state is generated only for the ROM.

When the **!MEMR** strobe is active for an address corresponding to the ROM, the **!ROM\_CS** signal is asserted and turns on the three-state buffer, driving the **CPU READY** signal LO, (indicating not ready, or wait). The next rising edge of the CPU clock (1) after **!ROM\_CS** becomes active and sets the **WAIT1** signal. After one CPU clock period passes, the **WAIT2** signal is asserted (2); the wait state period (one CPU clock) is completed, causing the **CPU READY** signal to be driven HI, which causes the CPU to continue its read cycle and remove the **!MEMR** strobe at the appropriate time. The **!ROM\_CS** signal is negated, disabling the three-state buffer driving the ready signal and, at the next rising edge of the CPU clock (3), causing **WAIT1** and **WAIT2** to be reset. The wait state generator is now prepared for the next CPU access time.

## 2.1.2. STEP 2. CREATING A CUPL SOURCE FILE

In this step, a logic description file will be created to describe the design for the PLD. The logic description file serves as input to **CUPL**, which compiles the design for downloading to a device programmer.

To make it easy to set up the required format for the logic description file, **CUPL** provides a template file, **TMPL.PLD**, that can be copied into the file being used. First, choose a name for the file that reflects the use being designed for the PLD. Since this is a sample session, use the name **SAMPLE.PLD**. Copy **TMPL.PLD** to **SAMPLE.PLD**, by typing:

**copy tmp.pld sample.pld**



### Note

To move more quickly through this design example, it is not necessary to actually create and edit the **SAMPLE.PLD** file. The CUPL package provides a sample file, **WAITGEN.PLD**, that can be used instead.

A completed **SAMPLE.PLD** file is shown here to explain the different sections of the file. This is followed by step-by-step instructions for creating **SAMPLE.PLD**.

```
Name      Adder;
Partno    CA0016;
Date      10/08/85;
Rev       01;
Designer  Smith;
Company   Technology Inc;
Assembly  None;
Location  None;
Device    pl618;

/*****
/*
/* Four bit adder using the CUPL function statement.
/*
/* 4-bit asynchronous adder implemented as a ripple-carry
/* through four adder-slice circuits. Each adder-slice
/* takes a pair of 1-bit numbers (Xi, Yi) and the carry from
/* a previous slice (Cin) and produces their 1-bit sum (Zi)
/* and carry (Cout). Each adder-slice circuit is defined
/* using the CUPL function adder_slice(), which returns
/* the product directly and the carry as Cout.
*****/
/* Allowable Target Device Types : PAL16L8
*****/

/** Inputs **/

Pin [1..4] = [X1..4];          /* First 4-bit number */
Pin [5..8] = [Y1..4];          /* Second 4-bit number */

/** Outputs **/

Pin [12..15] = [Z1..4];         /* 4-bit sum */
Pin [16..18] = [C1..3];         /* Intermediate carry vaules */
Pin 19 = Carry;                 /* Carry for 4-bit sum */

/* Adder-slice circuit - add 2, 1-bit, numbers with carry */

function adder_slice(X, Y, Cin, Cout) {
    Cout = Cin & X               /* Compute carry */
    # Cin & Y
    # X & Y;
    adder_slice = Cin $ (X $ Y); /* Compute sum */
}

/* Perform 4, 1-bit, additions and keep the final carry */

Z1 = adder_slice(X1, Y1, 'h'0, C1); /* Initial carry = 'h'0 */
Z2 = adder_slice(X2, Y2, C1, C2);
```

```
Z3 = adder_slice(X3, Y3, C2, C3);
Z4 = adder_slice(X1, Y4, C3, Carry); /* Final carry value */
```

**Figure U5-4. Source File Example**

Use a text editor in non-document mode to open **SAMPLE.PLD**.

Figure U5-5 shows the template information that has been copied into the file.

```

                                TEMPLATE FILE

Name                           XXXXX;
Partno                         XXXXX;
Date                           XX/XX/XX;
Revision                       XX;
Designer                       XXXXX;
Company                        XXXXX;
Assembly                       XXXXX;
Location                       XXXXX;

/*****
/*
/*
/*
/*
/*
/* Allowable Target Device
/*
*****/

/** Inputs **/

Pin      =      ;      /*
Pin      =      ;      /*
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 **/
```

**Figure U5-5. SAMPLE.PLD Template Information**

The file can be edited, in order to enter specific header and title information, specify the input and output pins, and write the intermediate and logic equations.

In the header section, replace the XXXs with specific information referring to the company and the PLD being designed. Since this is a sample design, use the information provided (as shown in Figure U5-6) or any other desired information.

Below the header section is a title block with comment symbols (**/\*** and **\*/**). In the title block, type in information describing the design, as shown in Figure U5-6.

```

                                SAMPLE.PLD

Name                           Sample;
Partno                         P9000183;
Date                           07/16/87;
Revision                        02;
Designer                       Osann;
Company                        ATI;
Assembly                       PC Memory;
Location                       U106;


/*****
/* This device generates chip select signals for one */
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
/* the system READY line to insert a wait-state of at */
/* least one cpu clock for ROM accesses */
*****/
```

**Figure U5-6. SAMPLE.PLD Header and Title Block**

### 2.1.3. STEP 3. FORMULATING THE EQUATIONS

To make it easier to enter the specific equations for address decoding and wait state generation, first enter equations for intermediate variables. Intermediate variables are arbitrary names; that is, they do not represent specific pins. Enter the intermediate equations in the space provided in the **SAMPLE.PLD** file for “Declarations and Intermediate Variable Definitions.”

The first intermediate equation to enter is a bit field declaration to define the address bus. Use the name **MEMADR** (memory address) to represent the address, and type the equation as follows:

**FIELD MEMADR = [A15..11] ;**

In the system diagram in Figure U5-1, notice that the chip select signals for the static RAMs are not dependent solely upon address but must be asserted for either the **MEMW** or **MEMR** data strobes.

To simplify the equations for the static RAM chip select signals, create a signal called **MEMREQ** (memory request). Type the following:

**MEMREQ = MEMW # MEMR ;**

Whenever **MEMREQ** is used in other equations, CUPL substitutes **MEMW # MEMR** when it compiles.

Notice in the timing diagram in Figure U5-3 that the decoding of the address corresponding to the ROM combines with the **!MEMR** strobe to produce the ROM chip select (**ROM\_CS**), and to initiate the wait state sequence.

Create an intermediate variable, called **SELECT\_ROM**, representing the combination of the **!MEMR** strobe and the specific address decoding for the ROM's address space, by typing the following:

**SELECT\_ROM = MEMR & MEMADR : [0000..1FFF] ;**

After entering the above intermediate equations, the specific equations for address decoding and wait state generation may be entered.

If the signal **ROM\_CS**, which feeds back into the array, is being used to initiate the wait state timing, an additional pass delay is incurred through the PLD. Because the clock rate is relatively slow (4-8 MHz), in this example the additional delay is not a problem. However, at higher clock rates it is better to recreate the same logic (using the **SELECT\_ROM** intermediate) in the registered logic equations.

Create the ROM chip select (**ROM\_CS**) using the intermediate variable **SELECT\_ROM**, by typing:

**ROM\_CS = SELECT\_ROM ;**

The chip-selects for the two RAMs, **RAM\_CS0** and **RAM\_CS1**, are dependent on **MEMREQ** and the address bus being within the hexadecimal boundaries taken from the memory map. Use the CUPL range operation with the lower and upper boundaries of the desired address range to decode these signals. Type the following:

**RAM\_CS0 = MEMREQ & MEMADR : [2000..27FF];**

**RAM\_CS1 = MEMREQ & MEMADR : [2800..2FFF];**

Next, create the equations that relate to the wait state timing and generation. First, as shown in the timing diagram (Figure U5-3), a signal called **WAIT1** is required that responds to the selection of the ROM chip by being set at the next rising edge of the CPU clock. According to the rules for a D-type flip-flop, the

logic level at the D input is transferred to the Q output after the clock. Enter the equation for this signal, where **WAIT.D** represents the signal at the D input of the flip-flop within the PLD, by typing the following:

**WAIT.D = SELECT\_ROM & !RESET ;**

Notice that in the equation for **WAIT1.D**, the **!RESET** signal has been ANDed with the rest of the equation to perform a synchronous reset when the **RESET** signal is asserted.

Next, create the signal **WAIT2** at the next clock edge following the one that causes **WAIT1** to set, by making the equation for **WAIT2.D** dependent on the signal **WAIT1**. Since **WAIT2.D** must reset at the next clock edge following the removal of the CPU's access of the ROM, AND the variable, and then **SELECT\_ROM** into this equation by typing the following:

**WAIT2.D = SELECT\_ROM & WAIT1 ;**

This creates the signal **SELECT\_ROM** in accordance with the timing diagram (Figure U5-3) to indicate that the three-state buffer should be turned on while the ROM is being decoded and the **MEMR** data strobe is active. Therefore, enter the equation for the three-state output by typing the following:

**READY.OE = SELECT\_ROM ;**

While this equation determines when the three-state buffer actually drives its output and leaves the high impedance state, it does not determine which logic level the signal is driven to. The equation for **READY** determines the logic level to which the signal is driven; the signal should remain inactive at **READY** until the completion of a wait state period equal to one full CPU clock cycle. As this condition does not occur until **WAIT2** becomes set, type the equation for **READY** as follows:

**READY = WAIT2 ;**

## 2.1.4. STEP 4. CHOOSING A TARGET DEVICE

After the equations are completed, the next step is to identify a compatible, commercially available PLD. Points to consider when choosing a target device are:

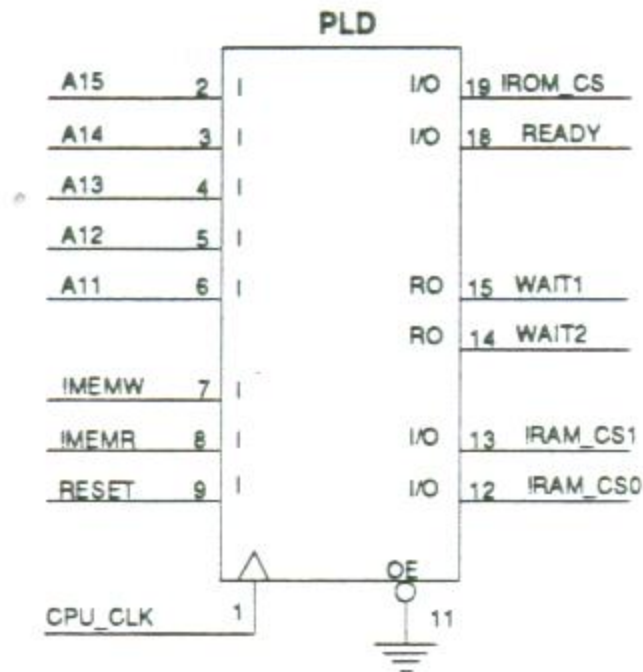
- The number of input pins required.

- The relative number of registered and non-registered output pins.

- Three-state output control (if required).

- The number of product terms required to implement the logic function for each equation.

The PLD package diagram in Figure U5-7 shows pin assignments configured to match up with a device similar to a PAL16R4 or an 82S155 IFL.



**Figure U5-7. Sample Pin Configuration**

In the pin configuration in Figure U5-7, the three chip select signals are assigned to I/O type pins that should always be in the output drive mode. The **READY** pin, attached to the **READY** signal on the CPU bus, is used in a controllable three-state mode. The two flip-flops that are needed to implement the wait state generator have been assigned to output pins that are internally connected to registers.

One of the registered outputs could be used to drive the **READY** signal directly, since the logical function of **READY** is the same as that of the signal **WAIT2**. However, use of the dedicated three-state output enable signal connected to pin 11 of the target device would be required. Since pin 11 controls the three state outputs of all four pins connected to internal registers, this defeats the ability to use the other two registered output pins for any purpose other than wait state generation.

It is better to keep options open by not using the dedicated three-state control, since it is difficult to predict all the changes that might be made during the evolution of a design. Therefore, pin 11 is tied to ground, which always enables the three-state outputs coming from the registers.



The PAL16R4 has at least seven product terms available on all outputs, which is an adequate number for this application. The IFL 82S155, which is a second source for this socket position, has a total of thirty-two product terms available for all outputs combined, which is also an adequate number for this application.

The PAL16R4 devices have only D-type flip-flops, whereas the 82S155 devices may be configured for either D or JK types. **CUPL** automatically chooses a D-type flip flop configuration because the equations entered for **WAIT1** and **WAIT2** in step 3 specified the **.D** extension.

## 2.1.5. STEP 5. MAKING THE PIN ASSIGNMENTS

Match the pin assignments to the pins in Figure U5-7 for a PAL16R4 or 82S155 device. First, in **SAMPLE.PLD** in the comment space labeled “Allowable Target Device Types,” type:

**pal16r4, 82s155** [Return](#)

To ensure consistent documentation when making the pin assignments, be certain that the signal polarities (signal levels) assigned are the same as those in the logic schematic.

Make the pin assignments as shown in Figure U5-8.

```

                                SAMPLE PIN ASSIGNMENTS
/**  Inputs  **/
Pin 1      = cpu_clk   ;      /* CPU clock                */
Pin [2..6] = [a15..11] ;      /* CPU Address Bus     */
Pin [7,8]  = ![memw,memr] ;    /* Memory Data Strokes */
Pin 9      = reset     ;      /* System Reset        */
Pin 11     = !oe        ;      /* Output Enable       */

/**  Outputs **/
Pin 19     = !rom_cs    ;      /* ROM Chip select     */
Pin 18     = ready      ;      /* CPU Ready signal    */
Pin 15     = wait1      ;      /* Start wait state    */
Pin 14     = wait2      ;      /* End wait state      */
Pin [13,12] = ![ram_cs1..0] ; /* RAM Chip selects    */
```

**Figure U5-8. SAMPLE.PLD Pin Assignments**

After making all the pin assignments, delete the extra “**pin = ;**” lines provided by the template file.

Figure U5-9 shows the completed logic description file, **SAMPLE.PLD**.

```

SAMPLE.PLD

Name          Sample;
Partno        P9000183;
Date          07/16/87;
Revision      02;
Designer      Osann;
Company       ATI;
Assembly      PC Memory;
Location      U106;

/*****
/* This device generates chip select signals for one
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives
/* the system READY line to insert a wait-state of at
/* least one cpu clock for ROM accesses
*****/
/** Allowable Target Device Types : PAL16R4, 82S155 **/
*****/
** Inputs **/
Pin 1    = cpu_clk    ;      /* CPU clock                      */
Pin [2..6] = [a15..11] ;      /* CPU Address Bus                */
Pin [7,8]  = ![memw,memr] ;    /* Memory Data Strobes (active low)*/
Pin 9      = reset    ;      /* System Reset                   */
Pin 11     = !oe       ;      /* Output Enable (active low)     */

** Outputs **/
Pin 19     = !rom_cs   ;      /* ROM chip select (active low) */
Pin 18     = ready     ;      /* CPU ready                      */
Pin 15     = wait1     ;      /* Wait state 1                   */
Pin 14     = wait2     ;      /* Wait state 2                   */
Pin [13,12] = ![ram_cs1..0] ; /* RAM chip select (active low) */

/** Declarations and Intermediate Variable Definitions */
Field memadr = [a15..11] ;      /* Give the address bus          */
                                   /* the Name "memadr"             */

memreq = memw # memr ;          /* Create the intermediate       */
                                   /* variable "memreq"             */
select_rom = memr & memadr:[0000..1FFF] ; /* = rom_cs                      */

** Logic Equations **/
rom_cs = select_rom;
ram_cs0 = memreq & memadr:[2000..27FF] ;
ram_cs1 = memreq & memadr:[2800..2FFF] ;

/* read as: when select_rom is true and reset is false */
wait1.d = select_rom & !reset ;

/* read as: when when select_rom is true and wait1 is true */
                                   /* Synchronous Reset            */
wait2.d = select_rom & wait1 ; /* wait1 delayed                */

ready.oe = select_rom ; /* Turn Buffer off */
ready = wait2 ; /* end wait */

```

**Figure U5-9. SAMPLE.PLD File**

## 2.1.6. STEP 6. RUNNING CUPL

When running **CUPL**, specify the target PLD, the source logic description file, and option flags to enable specific output files. In this step, compile the logic description file **SAMPLE.PLD** for the target device PAL16R4, and create the following output files:

**SAMPLE.ABS (-a flag)** - This is the absolute file for later use by **CSIM**, the **CUPL** simulator (This file is needed for step 7). It contains a condensed representation of the logical function to be programmed into a device. **CSIM** compares this representation to test vectors in a user-created input file to determine whether the response vectors in the input file are a correct response to the stimulus vectors.

**SAMPLE.DOC (-x, and -f flags)** - This is the documentation file. It provides fully expanded product terms for both intermediate and output pin variables, and a fuse plot and chip diagram.

**SAMPLE.LST (-l flag)** - This is the list file. It is a recreation of the description file, except line numbers have been added and any error messages generated during compilation are appended at the end of the file.

**SAMPLE.JED (-j flag)** - This is a JEDEC file for downloading to a device programmer. It contains a fuse pattern but no test vectors.



### Note

The **SAMPLE.JED** filename is determined by the **NAME** field in the header information section of the logic description file. When only one device is described in the file, be certain to use the same name (in this case, **SAMPLE**) as the filename.

To begin to compile and create the files described above, type the following command line: **cupl -jaxfl p16r4 sample**



### Note

If **SAMPLE.PLD** has not been created, type **WAITGEN** instead of **SAMPLE** to specify the sample file, **WAITGEN.PLD**, provided by CUPL. The filename for all output files created by CUPL is **WAITGEN** instead of **SAMPLE**.

The following messages appear on the screen, indicating how much time each CUPL module takes for completion. The actual time will vary depending on the system being used.

CUPL: Universal Compiler for Programmable Logic  
Version 4.XX Serial # XX-XXX-XXXX  
Copyright (C) 1983, 1990 Logical Devices, Inc.

```
cuplx
time:    2 secs
cupla
time:    2 secs
cuplb
time:    2 secs
cuplm
time:    1 secs
cuplc
time:    5 secs
total time: 12 secs
```

When the prompt appears, compilation is complete. **SAMPLE.LST** and **SAMPLE.DOC** are ASCII files, so it is possible to display them on the screen, open them with a text editing program, or print a hard copy of their contents.

The list file, **SAMPLE.LST**, is essentially a recreation of the source file with line numbers inserted and any error messages attached to the end. The line numbers facilitate the quick locating of error sources, if any are detected by **CUPL**.

Figure U5-10 shows the contents of **SAMPLE.LST**.

```

                                SAMPLE.LST
CUPL Version 4.XX Serial # XX-XXX-XXXX
Copyright (C) 1983,1990 Logical Devices, Inc.
CREATED Thur Jan 14 08:42:12 1990

LISTING FOR LOGIC DESCRIPTION FILE: sample.pld;

1:Name                          Sample;
2:Partno                        P9000183;
3:Date                          07/16/87;
4:Revision                      02;
5:Designer                      Osann;
6:Company                      ATI;
7:Assembly                      PC Memory;
8:Location                      U106;
9:
10:/******
11:/* This device generates chip select signals for one */
12:/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
13:/* the system READY line to insert a wait-state of at */
14:/* least one cpu clock for ROM accesses */
15:/******
16:/******
17:/* Allowable Target Device Types : PAL16R4, 82S155 */
18:/******
19:/* Inputs */
20:
21:Pin 1      = cpu_clk ; /* CPU clock */
22:Pin [2..6] = [a15..11] ; /* CPU Address Bus */
23:Pin [7,8]  = ![memw,memr] ; /* Memory Data Strobes */
24:Pin 9      = reset ; /* System Reset */
25:Pin 11     = !oe ; /* Output Enable */
26:
27:/* Outputs */
28:
29:Pin 19      = !rom_cs ; /*
30:Pin 18      = ready ; /*
31:Pin 15      = wait1 ; /*
32:Pin 14      = wait2 ; /*
33:Pin [13,12] = ![ram_cs1..0] ; /*
34:
35:/* Declarations and Intermediate Variable Definitions */
36:
37:Field memadr = [a15..11] ; /* Give the address bus */
38: /* the Name "memadr" */
39:
40:memreq = memw # memr ; /* Create the intermediate */
41: /* variable "memreq" */
42:
43:select_rom = memr & memadr:[0000..1FFF] ; /* = rom_cs */
44:
45:/* Logic Equations */
46:
47:rom_cs = select_rom;
48:ram_cs0 = memreq & memadr:[2000..27FF] ;
49:ram_cs1 = memreq & memadr:[2800..2FFF] ;
50:wait1.d = select_rom & !reset ;
51: /* Synchronous Reset */
52:wait2.d = select_rom & wait1 ; /* wait1 delayed */
53:ready.oe = select_rom ; /* Turn Buffer off */
54:ready = wait2 ; /* end wait */
Jedec Fuse Checksum (4D50)
Jedec Transmit Checksum (E88F)

```

**Figure U5-10. SAMPLE.LST**

Figure U5-11 shows the documentation file, **SAMPLE.DOC**, created by **CUPL**.

```

                                SAMPLE.DOC
*****
                                Sample
*****

CUPL                               4.XX Serial# XX-XXX-XXXX
Device                           p16r4 Library DLIB-d-26-11
Created                           Mon Aug 20 10:48:32 1990
Name                             Sample;
Partno                           P9000183;
Date                             04/1/90;
Revision                         02;
Designer                         Osann;
Company                         ATI;
Assembly                        PC Memory;
Location                        U106;

=====
                                Expanded Product Terms
=====

wait1.d =>
    !memr
    # a15
    # a14
    # a13
    # reset

select_rom =>
    !a13 & !a14 & !a15 & memr

wait2.d =>
    !memr
    # a15
    # a14
    # a13
    !wait

memadr =>
    a15,a14,a13,a12,a11

ready =>
    !wait2

ready.oe =>
    !a13 & !a14 & !a15 & memr
rom_cs =>
    !a13 & !a14 & !a15 & memr

```

**Figure U5-11. SAMPLE.DOC Sheet 1 of 4**

```

memreq =>
    memw
    # memr

ram_cs0 =>
    !a11 & !a12 & !a13 & !a14 & !a15 & memw
    # !a11 & !a12 & !a13 & !a14 & !a15 & memr

ram_cs1 =>
    a11 & !a12 & a13 & !a14 & !a15 & memw
    # a11 & !a12 & a13 & !a14 & !a15 & memr

rom_cs.oe =>
    1

ram_cs0.oe =>
    1

ram_cs1.oe =>
    1

```

Symbol Table							
Pin	Variable	Ext	Pin	Type	Pterms Used	Max Pterms	Min Level
---	-----	---	---	---	-----	-----	-----
	wait1		15	V	-	-	-
	wait1	d	15	X	5	8	1
	a11		6	V	-	-	-
	select_rom		0	I	1	-	-
	wait2		14	V	-	-	-
	wait2	d	14	X	5	8	1
	a12		5	V	-	-	-
	a13		4	V	-	-	-
	a14		3	V	-	-	-
	a15		2	V	-	-	-
!	oe		11	V	-	-	-
!	memr		8	V	-	-	-
	memadr		0	F	-	-	-
	ready		18	V	1	7	1
	ready	oe	18	X	1	1	1
!	memw		7	V	-	-	-
	cpu_clk		1	V	-	-	-
!	rom_cs		19	V	1	7	1
	reset		9	V	-	-	-
	memreq		0	I	2	-	-
!	ram_cs0		12	V	2	7	1
!	ram_cs1		13	V	2	7	1
	rom_cs	oe	19	D	1	1	0
	ram_cs0	oe	12	D	1	1	0
	ram_cs1	oe	13	D	1	1	0
LEGEND							
	F : field		D : default			M : extended node	
	N : node		I : Intermediate variable			T : function	
	V : variable		X : extended variable			U : undefined	

Figure U5-11. Sheet 2 of 4



```

=====
                        Fuse Plot
=====

Pin #19
0000 -----x-----
0032 -x---x---x-----x-----
0064 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0096 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0128 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0160 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0192 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0224 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #18
0256 -x---x---x-----x-----
0288 -----x-----
0320 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0352 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0384 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0416 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0448 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0480 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #17
0512 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0544 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0578 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0608 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0640 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0672 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0704 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0738 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #16
0768 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0800 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0832 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0864 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0896 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0928 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0960 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
0992 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #15
1024 -----x-----
1056 x-----x-----
1088 ---x-----
1120 -----x-----
1152 -----x-----
1184 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1216 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1248 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

Pin #14
1280 -----x-----
1312 x-----x-----
1344 ---x-----
1378 -----x-----
1408 -----x-----
1440 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1472 xxxxxxxxxxxxxxxxxxxxxxxxxxxx
1504 xxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

**Figure U5-11. Sheet 3 of 4**

```

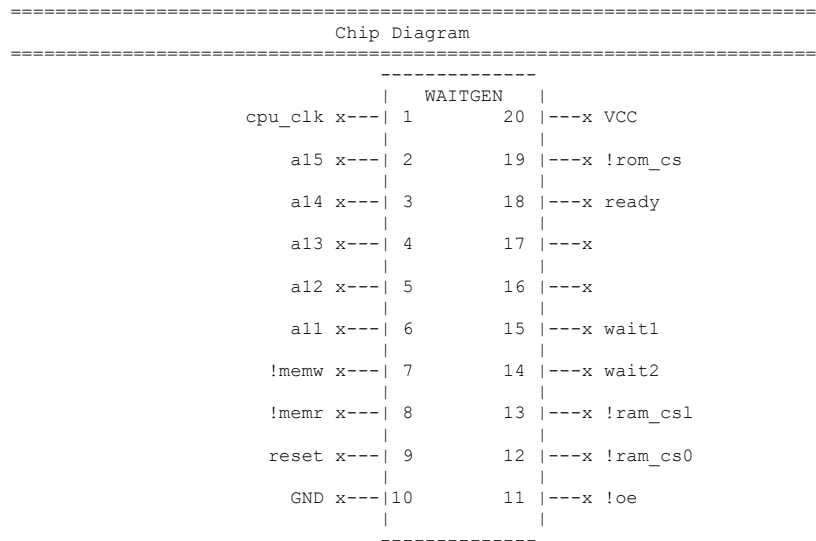
Pin #13
1536 -----
1568 -x---x--x---x--x---x-----
1600 -x---x--x---x--x---x-----
1632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

```

Pin #12
1792 -----
1824 -x---x--x---x--x---x-----
1856 -x---x--x---x--x---x-----
1888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```



**Figure U5-11. Sheet 4 of 4**

The expanded product terms for **WAIT1.D** and **WAIT2.D** show five product terms, because the fixed inverting output buffer (active-LO architecture) in the PAL16R4 causes CUPL to perform DeMorgan's Theorem on equations when an output variable has been declared as active-HI in the pin list for this particular device.

## 2.1.7. STEP 7. CREATING A CSIM SOURCE FILE

In this step, a simulation will be performed to verify the compiled design for the PAL16R4 device. Performing this step before downloading to a logic programmer decreases the probability of programming a device with incorrect logic.

Create a source specification file, **SAMPLE.SI**, containing test vectors for input to **CSIM**. **CSIM** compares the test vector inputs and expected outputs to the actual values contained in the **SAMPLE.ABS** file that was created during **CUPL** operation, and flags any discrepancies.

Figure U5-12 shows the contents of a sample source specification file.

```
Name                Sample;
Partno              P9000183;
Date                07/16/87;
Revision            02;
Designer            Osann;
Company             ATI;
Assembly            PC Memory;
Location            U106;

/*****
/* This device generates chip select signals for one */
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
/* the system READY line to insert a wait-state of at */
/* least one cpu clock for ROM accesses */
*****/

ORDER:
    cpu_clk, %2, a15, %2, a14, %2,
    a13, %2, a12, %2, a11, %2,
    !memw, %2, !memr, %2, reset, %2, !oe,
    %4, !ram_cs1, %2, !ram_cs0, %2, !rom_cs, %2,
    wait1, %2, wait2, %2, ready;

VECTORS:
/* 123456-leave six blanks to allow for numbers in .SO file */
$msg "          ! ! " ;
$msg "      c          r r ! " ;
$msg "      p          a a r " ;
$msg "      u          ! ! r   m m o w w r " ;
$msg "          _ m m e _ _ m a a e " ;
$msg "      c a a a a a e e s ! c c _ i i a " ;
$msg "      l 1 1 1 1 1 1 m m e o s s _ t t d " ;
$msg "      k 5 4 3 2 1 w r t e 1 0 s 1 2 y " ;
$msg "          " ;
$msg "      Power On Reset          " ;
$msg "      O X X X X X 1 1 1 0 H H H * * Z " ;
$msg "      Reset Flip Flops          " ;
$msg "      C X X X X X 1 1 0 0 H H H L L Z " ;
$msg "      Write RAM0          " ;
$msg "      0 0 0 1 0 0 0 1 0 0 H L H L L Z " ;
$msg "      Read RAM0          " ;
$msg "      0 0 0 1 0 0 1 0 0 0 H L H L L Z " ;
$msg "      Write RAM1          " ;
$msg "      0 0 0 1 0 1 0 1 0 0 L H H L L Z " ;
$msg "      Read RAM1          " ;
$msg "      0 0 0 1 0 1 1 0 0 0 L H H L L Z " ;
$msg "      Begin ROM read          " ;
$msg "      0 0 0 0 0 0 1 0 0 0 H H L L L L " ;
$msg "      Two clocks for wait state, Then drive READY High " ;
$repeat2;
$msg "      C 0 0 0 0 0 0 1 0 0 0 H H L * * * " ;
$msg "      End ROM Read          " ;
$msg "      0 0 0 0 0 0 1 1 0 0 H H H H H Z " ;
$msg "      End ROM Read          " ;
$msg "      C 0 0 0 0 0 0 1 1 0 0 H H H L L Z "
```

Figure U5-12. **SAMPLE.SI**

The source specification file contains three major parts: header information and title block, an **ORDER** statement, and a **VECTORS** statement.

**SAMPLE.SI** must have the same header information as **SAMPLE.PLD** to ensure that the proper files, including current revision level, are being compared against each other. Therefore, first copy **SAMPLE.PLD** to **SAMPLE.SI** and then use a text editor to delete everything in **SAMPLE.SI**, except the header and title block. Figure U5-13 shows the result.

```

Name                Sample;
Partno              P9000183;
Date                07/16/87;
Revision            02;
Designer            Osann;
Company             ATI;
Assembly            PC Memory;
Location            U106;

/*****
/* This device generates chip select signals for one */
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
/* the system READY line to insert a wait-state of at */
/* least one cpu clock for ROM accesses */
*****/

```

**Figure U5-13. SAMPLE.SI Header Information**

In the **ORDER** statement, list the input and output variables from **SAMPLE.PLD** to be included in test vectors. List the variables in the order in which they will be used in test variables; that is, put the clock variable, **CPU\_CLK**, first, followed by the other input variables. Put the output variables to the right. Separate variables with commas. Use the % symbol to insert spaces between the variables; put two spaces between each variable, and four spaces between the last input variable in the list, **!OE**, and the first output variable, **!RAM\_CS1**. Type the **ORDER** statement as follows:

**ORDER:**

```

CPU_CLK, %2, A15, %2, A14, %2,
A13, %2, A12, %2, ALL, %2,
!MEMW, %2, !MEMR, %2, RESET, %2, !OE,
%4, !RAM_CS1, %2, !RAM_CSO, %2, !ROM_CS, %2
WAIT1, %2, READY;

```

Following the **ORDER** statement, enter a **VECTORS** statement that creates a function table containing eleven test vectors (see Figure U5-12). First, to make the vectors easier to enter and understand, use the **\$MSG** command to create a heading for the function table. List the variable names in vertical columns in the same order and with the same spacing as specified in the **ORDER** statement, by typing the information in Figure U5-14.

```

/* 123456-leave six blanks to allow for numbers in .SO file */
$msg "                ! !                ";
$msg "      c                r r !                ";
$msg "      p                a a r                ";
$msg "      u                ! ! r      m m o w w r";
$msg "      _                m m e      _ m a a e";
$msg "      c a a a a a e e s !      c c _ i i a";
$msg "      l 1 1 1 1 1 1 m m e o      s s c t t d";
$msg "      k 5 4 3 2 1 w r t e      l 0 s 1 2 y";
$msg "      _____";

```

**Figure U5-14. Vectors Table Header**

Now enter the test vectors. Create the vectors by assigning a value to each of the input variables and an expected value to each of the output variables. Refer to Table 4-4 in Chapter 4, "Using CSIM", for allowable values to use for test vectors. Use the **\$MSG** directive to describe the device function tested by the function. The **ORDER** statement above specifies the spacing when creating the test vectors. For example, create the first vector, Power On Reset, by typing:

```

$msg " Power On Reset ";
0 X X X X 1 1 1 0 H H * * Z

```

Note that the output value (\*) has been used for **WAIT1** and **WAIT2** to instruct **CSIM** to calculate the

power-on state of the registers, since some devices power-on to X and some to H or L. Using the asterisk gives a more universal simulation file.

Type in the rest of the test vectors, as shown in Figure U5-15.

```
$msg "      Power On Reset                                ";
      O X X X X X 1 1 1 0      H H H * * Z
$msg "      Reset Flip Flops                            ";
      C X X X X X 1 1 0 0      H H H L L Z
$msg "      Write RAM0                                     ";
      0 0 0 1 0 0 0 1 0 0      H L H L L Z
$msg "      Read RAM0                                        ";
      0 0 0 1 0 0 1 0 0 0      H L H L L Z
$msg "      Write RAM1                                       ";
      0 0 0 1 0 1 0 1 0 0      L H H L L Z
$msg "      Read RAM1                                        ";
      0 0 0 1 0 1 1 0 0 0      L H H L L Z
$msg "      Begin ROM read                                   ";
      0 0 0 0 0 0 1 0 0 0      H H L L L L
$msg " Two clocks for wait state, Then drive READY High ";
$repeat2;
      C 0 0 0 0 0 0 1 0 0 0      H H L * * *
$msg "      End ROM Read                                     ";
      0 0 0 0 0 0 1 1 0 0      H H H H H Z
$msg "      End ROM Read                                     ";
      C 0 0 0 0 0 0 1 1 0 0      H H H L L Z
```

**Figure U5-15. Test Vectors**

The **\$REPEAT** directive in the test vectors causes the eighth vector to be repeated twice. The asterisks in the eighth vector for **WAIT1**, **WAIT2**, and **READY** tell **CSIM** to compute the output based on the inputs and place the results in the output file.

The value of the clock variable, **CPU\_CLK**, is 0 in some vectors and C in others. A value of 0 causes no clocking to occur. A value of C causes **CSIM** to examine the input values in the vector and also look back to the previous vector for any registered outputs that would be fed back internally prior to the clock. Then, after a clock is applied, **CSIM** computes the appropriate expected outputs for registered and non-registered variables.

After putting in the **VECTORS** statement, save the file. The next step is to run **CSIM** to perform the simulation.

## 2.1.8. STEP 8. RUNNING CSIM

When **CSIM** is run, it creates **SAMPLE.SO**, which contains the result of the simulation. Specify the **-I** flag to list any errors that might be generated.

To run **CSIM**, type: `csim -1 p16r4 sample`



### Note

If **WAITGEN.PLD** was used to run **CUPL** in step 6, specify **WAITGEN** instead of **SAMPLE** when running **CSIM**.

**CSIM** displays the amount of time to perform the simulation, as follows:

```
CSIM: CUPL Simulation Program
Version 4.XX Serial # XX-XXX-XXXX
Copyright (C) 1983, 1990 Logical Devices, Inc.
```

```
csima
time: 4 secs
total time: 4 secs
```

When the prompt reappears, the simulation is complete. **SAMPLE.SO** is an ASCII file, so it is possible to display it on the screen, print a hardcopy of it, or open it with a text editor.

Figure U5-16 shows the contents of **SAMPLE.SO**.

```

                                SAMPLE.SO

CSIM:    CUPL Simulation Program
Version 4.XX Serial # XX-XXX-XXXX
copyright (c) 1983,1990 Logical Devices, Inc.
CREATED Thur Aug 20 09:34:16 1990

1: Name                Sample;
2: Partno              P9000183;
3: Date               07/16/87;
4: Revision           02;
5: Designer           Osann;
6: Company            ATI;
7: Assembly           PC Memory;
8: Location           U106;
9:
10: /*****
11: /* This device generates chip select signals for one */
12: /* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives */
13: /* the system READY line to insert a wait-state of at */
14: /* least one cpu clock for ROM accesses */
15: /*****
16:
17: ORDER:
18:     cpu_clk, %2, a15, %2, a14, %2,
19:     a13, %2, a12, %2, a11, %2,
20:     !memw, %2, !memr, %2, reset, %2, !oe,
21:     %4, !ram_cs1, %2, !ram_cs0, %2, !rom_cs, %2,
22:     wait1, %2, wait2, %2, ready;
23:
=====
                                Simulation Results
=====

                                ! !
                                r r !
                                a a r
                                m m o w w r
                                _ _ m a a e
                                c a a a a a e e s !
                                l 1 1 1 1 1 m m e o
                                k 5 4 3 2 1 w r t e

                                1 0 s 1 2 y

Power On Reset
0001: O X X X X X X 1 1 1 0 H H H X X Z
Reset Flip Flops
0002: C X X X X X X 1 1 0 0 H H H L L Z
Write RAM0
0003: 0 0 0 1 0 0 0 1 0 0 H L H L L Z
Read RAM0
0004: 0 0 0 1 0 0 1 0 0 0 H L H L L Z
Write RAM1
0005: 0 0 0 1 0 1 0 1 0 0 L H H L L Z
Read RAM1
0006: 0 0 0 1 0 1 1 0 0 0 L H H L L Z
Begin ROM read
0007: 0 0 0 0 0 0 0 1 0 0 H H L L L L
Two clocks for wait state, Then drive READY High
0008: C 0 0 0 0 0 0 1 0 0 0 H H L H L L
0009: C 0 0 0 0 0 0 1 0 0 0 H H L H H H
End ROM Read
0010: 0 0 0 0 0 0 1 1 0 0 H H H H H Z
End ROM Read
0011: C 0 0 0 0 0 0 1 1 0 0 H H H L L Z

```

**Figure U5-16. SAMPLE.SO**

Compare **SAMPLE.SO** to the **SAMPLE.SI** file in Figure U5-12. Note that vectors 8 and 9 were created as a result of the **\$REPEAT** directive, and that **CSIM** has replaced the asterisks from **SAMPLE.SI** with the appropriate logic levels (H and L) for the **WAIT1**, **WAIT2** and **READY** signals.

Now that a successful simulation has been completed, test vectors can be added to the JEDEC file created while running **CUPL** (in step 6). Run the simulation again with the **-j** option flag, by typing:

**csim -j p16r4 sample** Return

If **WAITGEN.SI** was used to perform the simulation, specify it here instead of **SAMPLE.SI**.

Figure U5-17 shows the contents of **SAMPLE.JED**, which now contains both programming and testing information.

```
SAMPLE.JED

CUPL          4.XX   Serial# XX-XXX-XXXX
Device        p16r4  Library DLIB-d-26-11
Created       Thur Aug 20 09:52:02 1990
Name          Sample
Partno        P9000183
Revision      02
Date          12/16/89
Designer      Osann
Company       ATI
Assembly      PC Memory;
Location      U106;

*QP20
*QF2048
*G0
*F0
*L00000 11111111111111111111111111111111
*L00032 10111011101111111111111111011111
*L00256 10111011101111111111111111011111
*L00288 11111111111111111111111011111111
*L01024 11111111111111111111111011111111
*L01056 01111111111111111111111111111111
*L01088 11110111111111111111111111111111
*L01120 11111111011111111111111111111111
*L01152 11111111111111111111111111101111
*L01280 11111111111111111111111011111111
*L01312 01111111111111111111111111111111
*L01344 11110111111111111111111111111111
*L01376 11111111011111111111111111111111
*L01408 11111111111111111111011111111111
*L01536 11111111111111111111111111111111
*L01568 10111011011110110111101111111111
*L01600 10111011011110110111111110111111
*L01792 11111111111111111111111111111111
*L01824 10111011011110111011101111111111
*L01856 10111011011110111011111110111111
*C4D50
*V0001 0XXXXX111N0HHXXXXZHN
*V0002 CXXXXX110N0HLLXXZHN
*V0003 000100010N0LHLLXXZHN
*V0004 000100100N0LHLLXXZHN
*V0005 000101010N0HLLXXZHN
*V0006 000101100N0HLLXXZHN
*V0007 000000100N0HLLXXZLN
*V0008 C00000100N0HLLHXXZLN
*V0009 C00000100N0HHHXXZLN
*V0010 000000110N0HHHXXZHN
*V0011 C00000110N1HLLXXZHN
*3152
```

**Figure U5-17. SAMPLe.JED with Test Vectors**



## #2.2. SAMPLE PLD FILES

This section 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

4-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)

4-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

4-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

4-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)  
Hexadecimal to 7-segment decoder used for displaying numbers.

FILE: IOCODE.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  
4-bit counter/shifter using SR -type flip-flops.

FILE: SHFTCNT4.PLD  
DEVICES: 82S159  
4-bit counter/shifter using JK-type flip-flops.

FILE: SHFTCNT6.PLD  
DEVICES: 82S167  
4-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 (WGTTTL.PLD)

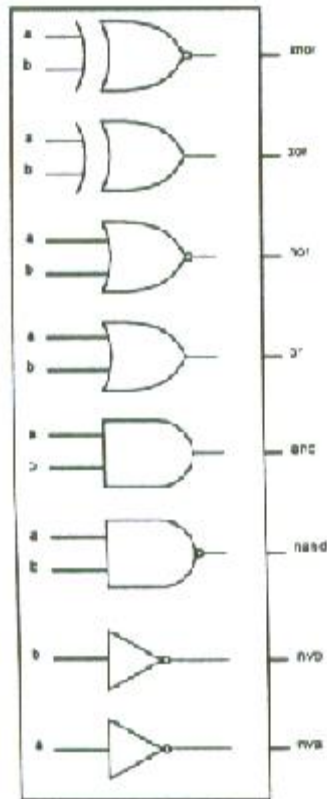
Two-bit counter (FLOPS.PLD)

Decade up/down counter using state-machine syntax (COUNT10.PLD)

Seven-segment display decoder (HEXDISP.PLD)

### 2.2.1. EXAMPLE 1 SIMPLE GATES

This example describes a design containing simple gates. Figure U5-18 shows the design.



**Figure U5-18. Design with Simple Gates**

The outputs are labelled to reflect the function of their gate; for example, the AND gate has an output labelled AND.

Figure U5-19 shows the **CUPL** source file (**GATES.PLD** provided in the CUPL package) that describes the design.

```

                                GATES.PLD

Name          Gates;
Partno        CA0001;
Date          07/16/87;
Designer      G Woolheiser;
Company       ATI;
Location      San Jose, CA.;
Assembly      Example;

/*****
/*
/* This is an example to demonstrate how CUPL
/* compiles simple gates
/*
/*
/* Target Devices: P16L8, P16P8, EP300, and 82S153
*****/

/* Inputs:  define inputs to build simple gates

Pin 1 = a;
Pin 2 = b;

/* Outputs:  define outputs as active HI levels

    For PAL16L8 and PAL16LD8, De Morgan's Theorem is
    applied to invert all outputs due to fixed
    inverting buffer in the device.

Pin 12 = inva;
Pin 13 = invb;
Pin 14 = and;
Pin 15 = nand;
Pin 16 = or;
Pin 17 = nor;
Pin 18 = xor;
Pin 19 = xnor;

/* Logic:  examples of simple gates expressed in CUPL */

inva = !a;          /* inverters
invb = !b;
and  = a & b;       /* and gate
nand = !(a & b);    /* nand gate
or   = a # b;       /* or gate
xor  = a $ b;       /* xor gate
nor  = !(a # b);    /* nor gate
xnor = !(a $ b);    /* exclusive nor gate

```

**Figure U5-19. Simple Gates Source File (GATES.PLD)**

The first part of the file provides archival information and a description of the intended function of the design, including compatible PLDs.

Pin declarations are made corresponding to the inputs and outputs in the design diagram.

In the “**Logic**” section of the file, equations describe each of the gates in the design.

For the PAL16L8 and PAL16LD8 devices, which contain fixed inverting buffers, **CUPL** applies DeMorgan’s Theorem to invert all outputs because they were all declared active-HI in the pin list. For example, during compilation, **CUPL** converts the following equation for an OR gate, on an output pin that has been declared as active high:

$$\text{or} = a \# b ;$$

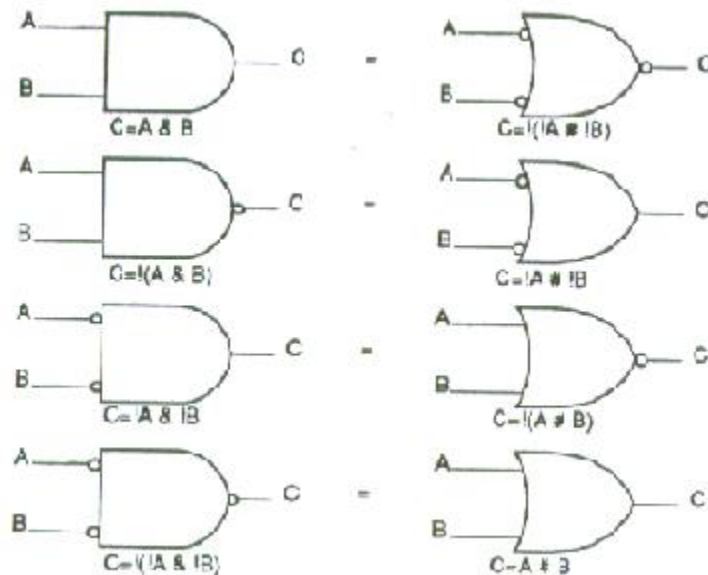
to the following single expanded product term (as shown in the documentation file):

$$\text{or} \Rightarrow !a \& !b$$

## 2.2.2. EXAMPLE 2. CONVERTING A TTL DESIGN TO PLDs

This example shows how to use a PLD to replace existing TTL circuitry. The conversion requires translating the gates of a TTL logic design into equivalent Boolean logic equations, which can then be compiled by **CUPL** and assigned to a PLD.

Figure U5-20 shows the TTL gate representations used in designing logic systems and the corresponding Boolean equation for each gate.

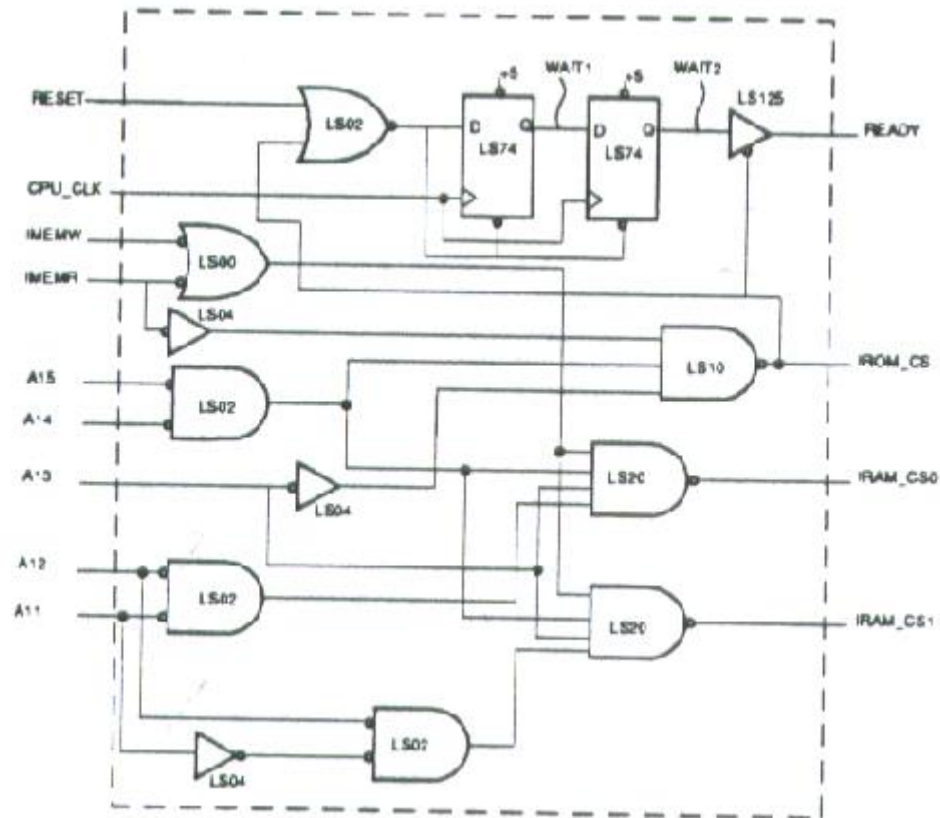


**Figure U5-20. TTL Gate Representations and Boolean Equations**

The basic conversion rules shown in Figure U5-20 are sufficient to write equations for each gate within a system of TTL gates when converting the logic to a PLD equivalent. CUPL uses an expression substitution process to build larger equations from the smaller expressions representing each gate in the TTL schematic. Expression substitution permits approaching a schematic one gate at a time.

Figure U5-21 shows the schematic for the TTL logic that is converted in Example 1.





**Figure U5-22. PLD Equivalent Diagram**

Note that the outputs of the internal gates (those that do not connect to the PLD output pins) are arbitrarily labelled with the variable names, A-H, to aid in entering equations in the logic description file. The logic description file used to convert this design is named **WGTTL.PLD** because it performs wait state generation and is based on a TTL design.

Figure U5-23 shows the contents of **WGTTL.PLD**.



```

WGTTL.PLD

Name          Sample;
Partno        P9000183;
Date          07/16/87;
Revision      02;
Designer      Osann;
Company       ATI;
Assembly      PC Memory;
Location      U106;

/*****
/* This device generates chip select signals for one
/* 8Kx8 ROM and two 2Kx8 static RAMs. It also drives
/* the system READY line to insert a wait-state of at
/* least one cpu clock for ROM accesses
*****/

/** Inputs **/

PIN 1         = cpu_clk      ; /* CPU clock          */
PIN [2..6]    = [a15..11]   ; /* CPU Address Bus   */
PIN [7 8]     = ![memw,memr] ; /* Memory Data Strobe*/
PIN 9         = reset       ; /* System Reset      */
PIN 11        = !oe         ; /* Output enable     */

/** Outputs **/

PIN 19        = !rom_cs     ; /* ROM Chip Select   */
PIN 18        = ready       ; /* CPU ready signal  */
PIN 15        = wait1       ; /* Start wait state  */
PIN 14        = wait2       ; /* End wait state    */
PIN [13,12]   = ![ram_cs1..0] ; /* RAM chip selects  */

/** Declarations and Intermediate Variable Definitions **/

a = !(!memw) # !(!memr) ;
b = !a15 & !a14 ;
c = !a13 ;
d = !a12 & !a11 ;
e = !a11 ;
f = !a12 & !oe ;
g = !(!rom_cs # reset) ;
h = !(!memr) ;

/** Logic Equations **/

!rom_cs = !(h & b & c);
!ram_cs0 = !(a & b & a13 & d) ;
!ram_cs1 = !(a & b & a13 & f) ;
wait1.d = g ;
wait2.d = wait1 & g ;
ready.oe = !(!(h & b & c)) ;
ready = wait2 ;

```

**Figure U5-23. WGTTL.PLD**

The header information is identical to that in **WAITGEN.PLD** (and **SAMPLE.PLD**; see Part A), because the functionality is the same.

The pin assignments match the schematic in Figure U5-22.

The logic equations for the internal gates are placed in the section of the file for “Declarations and Intermediate Variable Definitions.” The equations in this section use the output variable names, A-H, assigned in the schematic in Figure U5-22. For example, the AND gate LS02 is described by the following equation:

$$d = !a12 \ \& \ !a11 \ ;$$

The equations in this section can be simplified. For example, the double negations can be eliminated in the following equation:

$$a = !(!memw) \ \# \ !(!memr) \ ;$$

by entering:

```
a = memw # memr ;
```

The section of the file "Logic Equations" contains equations that describe the output signals of the PLD. These equations are written in terms of the intermediate equations that describe the outputs of the internal gates. For example, the AND gate, LS10, has **!ROM\_CS** as its output signal, and the signals H, B and C as inputs. Therefore, enter the following equation to describe LS10:

```
!rom_cs = !(h & b & c) ;
```

**WGTTL.PLD** to **WAITGEN.PLD** are not exactly alike, because the internal gates have been defined differently. When compiled, however, they perform the same function. This can be verified by simulating each logic description in turn with an identical simulation source file (**WGTTL.SI** and **WAITGEN.SI** provided in the CUPL package).

When converting a TTL design to a PLD, some slight changes in functionality are required. The asynchronous reset capability found on TTL flip-flops like the LS74 is not found in many of the commonly available PLDs. However, the same reset capability can be achieved by including a **RESET** variable in all product terms to ensure a synchronous reset at the clock.

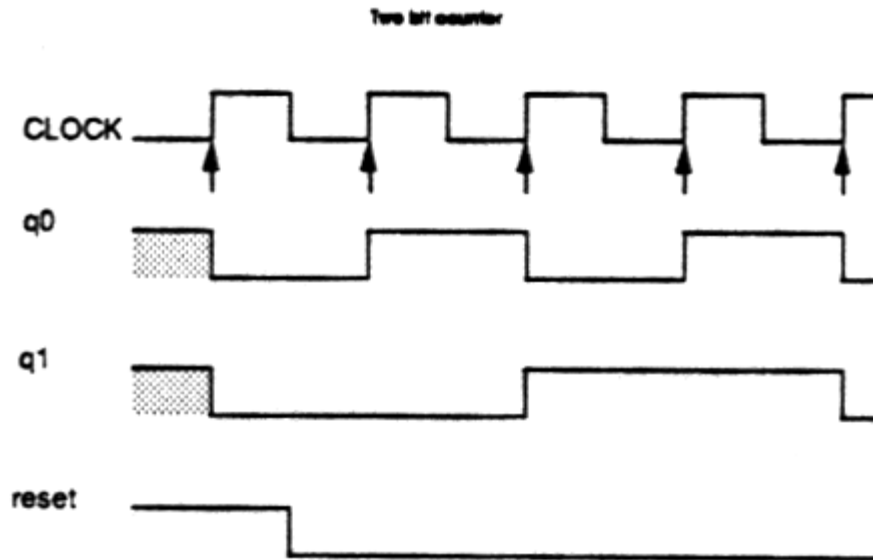
Therefore, **WGTTL.PLD** incorporates **!RESET** into the equation for G, which is used in the equations for both **WAIT1** (wait1.d = g;) and **WAIT2** (wait2.d = wait1.d & g). Although the functional nature of the synchronous reset does differ in timing from that of the asynchronous reset, the synchronous reset is sufficient for proper function of the device.

The simple methodology described in Example 2 allows the conversion of many TTL designs, especially those consisting of simple gates, to a PLD equivalent, regardless of the number of gate delays of TTL (logic) in the original design. In most cases, the only difference between the TTL design and the PLD is the total propagation delay through the circuit.

### 2.2.3. EXAMPLE 3. TWO-BIT COUNTER

This example demonstrates the implementation of a two-bit counter for a D-type flip-flop.

Figure U5-24 shows the timing diagram for the counter.



### Figure U5-24. Two-Bit Counter Timing Diagram

As indicated by the arrows, the registers are clocked on the rising edge of the clock signal.

Figure U5-25 shows the **CUPL** source file (**FLOPS.PLD**, provided in the CUPL package) to describe the two-bit counter design.



as:

**q1.d = q0 & !reset ;**

not as:

**q1.d = q0.d & !reset ;**

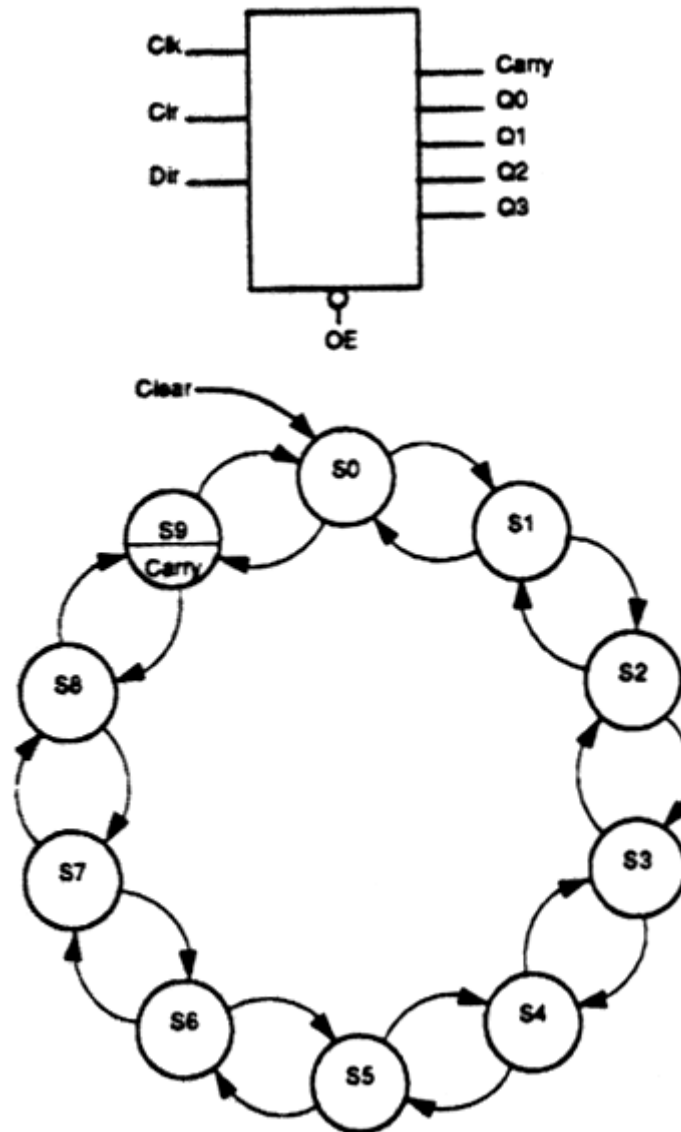
or:

**q1.d = q0.dq & !reset ;**

## 2.2.4. EXAMPLE 4. DECADE UP/DOWN COUNTER

This example describes a four-bit up/down decade counter with a synchronous clear capacity. The counter also provides an asynchronous ripple carry output for cascading multiple devices. The source file to implement the counter uses CUPL state machine syntax.

Figure U5-26 shows the counter design and its state.



**Figure U5-26. Up/Down Counter Diagram**

The input signal **dir** determines the direction of the count. When **dir** is high, the count goes down one on each clock; when **dir** is low, the count goes up one on each clock. The **clr** signal performs a synchronous reset.

Figure U5-27 shows the **CUPL** source file (**COUNT10.PLD**, provided in the CUPL package) that implements

the design.

```
COUNT10.PLD

Name          Count10;
Partno        CA0018;
Revision      02;
Date          07/16/87;
Designer      Kahl;
Company       ATI;
Location      None;
Assembly      None;
Device        pl6rp4;

/*****
/*
/*          Decade Counter
/* This is a 4-bit up/down decade counter with
/* synchronous clear capability. An asynchronous
/* ripple carry output is provided for cascading
/* multiple devices. CUPL state machine syntax
/* is used
*****/
/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
*****/
/* Inputs */
Pin 1 = clk;          /* counter clock
Pin 2 = clr;          /* counter clear input
Pin 3 = dir;          /* counter direction input
Pin 11 = !oe;         /* Register output enable

/* Outputs

Pin [14..17] = [Q3..0]; /* counter outputs
Pin 18 = carry;        /* ripple carry out

/* Declarations and Intermediate Variable Definitions */
field count = [Q3..0]; /* declare counter bit field */
$define S0 'b'0000
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001
field node = [clr,dir]; /* declare filed node control */
up = mode:0;           /* define count up mode
down = mode:1;         /* define count down mode
clear = mode:2..3];    /* define count clear mode
```

**Figure U5-27. Up/Down Counter Source File (COUNT10.PLD) Sheet 1 of 2**

```

sequence count { /* Logic Equations */
/* free running counter */

    present S0      if up      next S1;
                    if down    next S9;
                    if clear   next S0;
    present S1      if up      next S2;
                    if down    next S0;
                    if clear   next S0;
    present S2      if up      next S3;
                    if down    next S1;
                    if clear   next S0;
    present S3      if up      next S4;
                    if down    next S2;
                    if clear   next S0;
    present S4      if up      next S5;
                    if down    next S3;
                    if clear   next S0;
    present S5      if up      next S6;
                    if down    next S4;
                    if clear   next S0;
    present S6      if up      next S7;
                    if down    next S5;
                    if clear   next S0;
    present S7      if up      next S8;
                    if down    next S6;
                    if clear   next S0;
    present S8      if up      next S9;
                    if down    next S7;
                    if clear   next S0;
    present S9      if up      next S0;
                    if down    next S8;
                    if clear   next S0;
    out              carry; /* assert carry output */
}

```

**Figure U5-27. Sheet 2 of 2**

The first part of the file provides archival information and a description of the intended function of the design, including compatible PLDs.

Pin declarations are made corresponding to the inputs and outputs in the design diagram.

The “Declarations and Intermediate Variable Definitions” section contains declarations that simplify the notation.

The name “**count**” is assigned to the output variables Q3, Q2, Q1, and Q0.

The **\$DEFINE** command is used to assign names to ten binary states representing the state machine output. The state name can then be used in the logic equations to represent the corresponding binary number.

The **FIELD** keyword is used to combine the **clr** and **dir** inputs into a set called **mode**. **Mode** is defined by the following equations:

```

up = mode:0;
down = mode:1;
clear = mode [2..3];

```

**Mode** represents the inputs **clr** and **dir**, so the three equations above are equivalent to the following equations:

```

up = !clr & !dir ;
down = !clr & dir ;
clear = (clr & !dir) # (clr & dir) ;

```

The three modes are defined as follows:



**up** - Both the **dir** and **clr** inputs are not asserted.

**down** - The **dir** input is asserted and **clr** is not asserted.

**clear** - The **clr** input is asserted and **dir** is either asserted or not asserted.

The “**Logic Equations**” section contains the state machine syntax that specifies the states in the counter. In the first line, the **SEQUENCE** keyword identifies count (that is, Q3, Q2, Q1, and Q0) as the outputs to which the state values apply.

Conditional statements have been written to specify the transition from each possible present state to a next state, for each of the three modes. For example, when the present state is S4, if the mode is up, the counter goes to S5; if the mode is down the counter goes to S3; or if the mode is clear, the counter goes to S0. As example 4 shows, one advantage of the state machine syntax is that it clearly documents the operation of the design.

In Example 4, state 0 (binary value 0000) is defined, because it is the result of the **clr** signal. It is recommended that all designs have a valid 0000 defined to avoid being stuck at state 0. For example, in this design, if a state that hasn't been defined occurs at power-on, such as hexadecimal A-F, none of the conditions described in the equations is met, so the state goes to state 0 (hex value 0000). If 0000 has not been defined as a valid state, the counter stays at state 0.

Figure U5-28 shows how this example could have been written as a virtual design. It is the same file, but it has been modified where necessary to show the difference between a virtual design and a device specific design.

```

COUNT10.PLD

Name          Count10;
Partno        CA0018;
Revision      02;
Date          07/16/87;
Designer      Kahl;
Company       ATI;
Location      None;
Assembly      None;
Device        VIRTUAL;

/*****
/*
/*          Decade Counter
/* This is a 4-bit up/down decade counter with
/* synchronous clear capability. An asynchronous
/* ripple carry output is provided for cascading
/* multiple devices. CUPL state machine syntax
/* is used
*****/
/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
*****/
/** Inputs **/
Pin = clk;          /* counter clock
Pin = clr;          /* counter clear input
Pin = dir;          /* counter direction input
Pin = !oe;          /* Register output enable

/* Outputs

Pin = [Q3..0];      /* counter outputs
Pin = carry;        /* ripple carry out

/* Declarations and Intermediate Variable Definitions
field count = [Q3..0]; /* declare counter bit field
#define S0 'b'0000
#define S1 'b'0001
#define S2 'b'0010
#define S3 'b'0011
#define S4 'b'0100
#define S5 'b'0101
#define S6 'b'0110
#define S7 'b'0111
#define S8 'b'1000
#define S9 'b'1001
field node = [clr,dir]; /* declare filed node control
up = mode:0;          /* define count up mode
down = mode:1;        /* define count down mode
clear = mode:2..3];   /* define count clear mode

```

**Figure U5-28. Up/Down Counter Source File (virtual) Sheet 1 of 2**

```

/* Logic Equations */
sequence count { /* free running counter */

    present S0      if up      next S1;
                   if down     next S9;
                   if clear    next S0;
    present S1      if up      next S2;
                   if down     next S0;
                   if clear    next S0;
    present S2      if up      next S3;
                   if down     next S1;
                   if clear    next S0;
    present S3      if up      next S4;
                   if down     next S2;
                   if clear    next S0;
    present S4      if up      next S5;
                   if down     next S3;
                   if clear    next S0;
    present S5      if up      next S6;
                   if down     next S4;
                   if clear    next S0;
    present S6      if up      next S7;
                   if down     next S5;
                   if clear    next S0;
    present S7      if up      next S8;
                   if down     next S6;
                   if clear    next S0;
    present S8      if up      next S9;
                   if down     next S7;
                   if clear    next S0;
    present S9      if up      next S0;
                   if down     next S8;
                   if clear    next S0;

    out      carry; /* assert carry output */
}

```

**Figure U5-28. Sheet 2 of 2**

It is possible to use some of the features of the CUPL preprocessor to considerably shorten this PLD file. Figure U5-29 will show how this same file could be written with a \$REPEAT structure which reduces the file size considerably.

```

Name          Count10;
Partno        CA0018;
Revision      02;
Date          07/16/87;
Designer      Kahl;
Company       ATI;
Location      None;
Assembly      None;
Device        VIRTUAL;

/*****
/*
/*          Decade Counter
/* This is a 4-bit up/down decade counter with
/* synchronous clear capability. An asynchronous
/* ripple carry output is provided for cascading
/* multiple devices. CUPL state machine syntax
/* is used
*****/
/* Allowable Target Device Types: PAL16RP4, GAL16V8, EP300 */
*****/

/** Inputs **/

Pin = clk;          /* counter clock          */
Pin = clr;          /* counter clear input      */
Pin = dir;          /* counter direction input  */
Pin = !oe;          /* Register output enable   */

/* Outputs */

Pin = [Q3..0];      /* counter outputs          */
Pin = carry;        /* ripple carry out         */

/* Declarations and Intermediate Variable Definitions */

field count = [Q3..0]; /* declare counter bit field */
field node = [clr,dir]; /* declare filed node control */
up = mode:0;          /* define count up mode      */
down = mode:1;        /* define count down mode    */
clear = mode:2..3];   /* define count clear mode   */

/* state machine description */
sequence count {
present 0
    if up & !clear    next 1;
    if down & !clear  next 9;
    if clear          next 0;

$REPEAT i=[1..9]
present i
    if up & !clear    next {(i+1)%10};
    if down & !clear  next {(i-1)%10};
    if clear          next 0;
$REPEND

```

**Figure U5-29. Up/Down Counter Source File (virtual) Sheet 1 of 2**

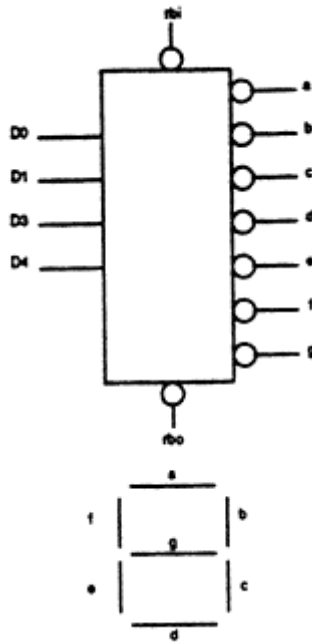
In this variation, we removed the \$DEFINE statements because we use the raw numbers instead. The most significant change is that we used a \$REPEAT loop to define most of the states instead of defining each state individually. It is possible to do this because all the states are identical except for the next state that each goes to. Notice that we define state 0 by itself and then all the other states are defined in one \$REPEAT loop. The \$REPEAT loop expands upon compilation to give a definition for each state. Notice that the statement indicating the next state is given as a calculation from the repeat variable 'i'. In the loop, 'i' represents the number of the current state. The next state is therefore 'i+1'. This will work for all states except the last state. In the last state, the state machine must go back to state 0. To accomplish this, the formula to calculate the next state is given as '(i+1)%10'. This means 'i+1' modulo 10. The number 10 represents the number of states. Therefore, when in state 9 the next state is calculated as 9+1 = 10 then modulo 10 which gives 0. A similar condition occurs in calculating the previous state except that we subtract 1 instead of adding it. You might have noticed that we defined state 0 separately. This was done because the \$REPEAT variables can only handle positive numbers. If we had defined state 0 in the

\$REPEAT loop this would result in evaluating to next state -1 and the compiler would produce an unexpected result.

### 2.2.5. EXAMPLE 5. SEVEN-SEGMENT DISPLAY DECODER

This example shows a hexadecimal-to-seven-segment decoder for driving common-anode LEDs. The design incorporates both a ripple-blanking input to inhibit the display of leading zeroes, and a ripple-blanking output for easy cascading of digits

Figure U5-30 shows the segment display decoder.



**Figure U5-30. Seven-Segment Display Decoder**

The segments in the display, labelled a-g, correspond to the outputs in the diagram.

Figure U5-31 shows the source file (**HEXDISP.PLD**, provided with the CUPL package).

```

                                HEXDISP.PLD

Name                           Hexdisp;
Partno                         CA0007;
Revision                       02;
Date                           07/16/87;
Designer                       T. Kahl;
Company                        ATL;
Location                       None;
Assembly                      None;

/*****
/*
/*
/*                                a                                */
/* This is a hexadecimal-to-seven-segment      ----- */
/* decoder capable of driving common-anode      |      |      */
/* LEDs. It incorporates both a ripple-        f|      |b      */
/* blanking input (to inhibit displaying       |  g  |      */
/* leading zeroes) and a ripple blanking output ----- */
/* to allow for easy cascading of digits       |      |      */
/*                                e|      |c      */
/*                                |      |      */
/*                                ----- */
/*                                d                                */
/*
*****/
/* Allowable Target Device Types: 32 x 8 PROM (82S123 or */
/* equivalent */
*****/
/** Input group (Note this is only a comment) */

pin [10..13] = [D0..3];      /* data input lines to display */
pin 14 = !rbi;               /* ripple blanking input */

/** Output Group ( Note this is only a comment ) */

pin [7..1] = ![a,b,c,d,e,f,g]; /* Segment output lines */
pin 9 = !rbo;                 /* Ripple Blanking output */

/** Declarations and Intermediate Variable Definitions */
field data = [D3..0];        /* hexadecimal input field */
field segment=[abcdefg];     /* Display segment field */
$define ON 'b'1              /* segment lit when logically "ON" */
$define OFF 'b'0             /* segment dark when logically "OFF" */

```

**Figure U5-31. Display Decoder Source File (HEXDISP.PLD) Sheet 1 of 2**

The first part of the file provides archival information and a description of the intended function of the design, including compatible PLDs.

Pin declarations are made corresponding to the inputs and outputs in the design diagram.

In the "Declarations and Intermediate Variables" section, field assignments are made to group the input pins into a set named data and the output pins into a set named segment. ON and OFF are defined respectively as binary 1 and binary 0.

```

/** Logic Equations **/
/*      a      b      c      d      e      f      g */
segment =
/* 0 */      [ ON,   ON,   ON,   ON,   ON,   ON,   OFF] & data:0 & !rbi
/* 1 */      # [OFF,  ON,   ON,   OFF,  OFF,  OFF,  OFF] & data:1
/* 2 */      # [ ON,  ON,   OFF,  ON,   ON,   OFF,  ON]  & data:2
/* 3 */      # [ ON,  ON,   ON,   ON,   OFF,  OFF,  ON]  & data:3
/* 4 */      # [OFF,  ON,   ON,   OFF,  OFF,  ON,   ON]  & data:4
/* 5 */      # [ ON,  OFF,  ON,   ON,   OFF,  ON,   ON]  & data:5
/* 6 */      # [ ON,  OFF,  ON,   ON,   ON,   ON,   ON]  & data:6
/* 7 */      # [ ON,  ON,   ON,   OFF,  OFF,  OFF,  ON]  & data:7
/* 8 */      # [ ON,  ON,   ON,   ON,   ON,   ON,   OFF] & data:8
/* 9 */      # [ ON,  ON,   ON,   ON,   OFF,  ON,   ON]  & data:9
/* A */      # [ ON,  ON,   ON,   OFF,  ON,   ON,   ON]  & data:A
/* B */      # [OFF,  OFF,  ON,   ON,   ON,   ON,   ON]  & data:B
/* C */      # [ ON,  OFF,  OFF,  ON,   ON,   ON,   OFF] & data:C
/* D */      # [OFF,  ON,   ON,   ON,   ON,   OFF,  ON]  & data:D
/* E */      # [ ON,  OFF,  OFF,  ON,   ON,   ON,   ON]  & data:E
/* F */      # [ ON,  OFF,  OFF,  OFF,  ON,   ON,   ON]  & data:F;

rbo = rbi & data:0;

```

**Figure U5-31. Sheet 2 of 2**

The logic equations are set up as a function table to describe the segments that are lit up by each input pattern. Comments create a header for the function table, listing the output segments across the top and the input numbers vertically down the side.

Each line of the table describes a decoded hex value and the segments of the display that the hex value turns on or off. For example, the line for an input value of 4 is written as follows:

**[OFF, ON, ON, OFF, OFF, ON, ON] & data:4**

The function table format expresses the intent of the design more clearly than equations; that is, the example above shows that an input value of 4 turns segment a off, segment b on, segment c on, and so on.



## 2.2.6. EXAMPLE 6. 4-BIT COUNTER WITH LOAD AND RESET

This example will present a counter that can be loaded and reset. The design is done using the VIRTUAL device but it could easily be implemented in almost any PLD that has four or more registers.

```
Name          Counter;
Partno        FL1201;
Revision      01;
Date          08/26/91;
Designer      RGT;
Company       LDI;
Location      None;
Assembly      None;
Device        VIRTUAL;
/*****
/* 4-bit counter
*****/

/** inputs */
PIN = clk; /* clock signal for registers */
PIN = load; /* load signal */
PIN = !ClrFlag;
PIN = [LoadPin0..3]; /* pins from which to load data */

/** outputs */
PIN = [CountPin0..3];

/* intermediate variables and fields */
field STATE_BITS = [Count0..3];
field LOAD_BUS = [LoadPin0..3];

/** state machine definition */
SequenceD STATE_BITS {
/* build a repeated loop for the states */
$REPEAT i = [0..15]
  Present 'h'{i}
    /* go to state 0 if clear signal is true */
    /* if the load signal is false go to the */
    /* next state. Note that the next state */
    /* is (current state + 1) modulo 16. */
    /* This causes the counter to wrap back */
    /* to 0 when in the last state */
    If !load Next 'h' {(i+1)%16};
    If !load & ClrFlag Next'b'0;
$REPEND
/* Add the load capability by using the
* APPEND statement. This has the effect of
* adding more inputs to the OR gate for
* this output. This equation states that if
* the load signal is true then the counter
* registers are loaded with data from the
* load pins. This is why 'load' was used in
* the equations for the 'IF' statements in
* the state definitions.
APPEND STATE_BITS.d = load & LOAD_BUS;
```

**Figure U5-32. Counter With Reset and Load.**

Since this is a virtual design, pin numbering can be ignored. The signals to be used are declared without any pin numbers.

When the load signal is asserted, the values at the load pins are fed into the state bit registers. When the 'Clear' signal is asserted, the state machine is forced to state 0.

The \$REPEAT loop is defined as [0 ..15]. Inside the loop, the present state is defined as 'h' {i}. This causes the numbers to be evaluated as hex numbers. The expansion that results from this is a state machine with 16 states ranging from 0 to F in hexadecimal. If the 'h' was left off, the states would have been expanded as 0 to 15 in decimal. The compiler would have interpreted these as hex anyway and states A-F would not have been defined. Additionally, State 10 in hex cannot exist with this number of statebits since that would require 5 statebits but all that we have is 4.

Notice that all the IF statements are ended with "!!load". This was done because we want to give loading the highest possible priority. Also this removes any possible conflict that could occur if load and some other signal were asserted at the same time.

The next state is calculated as  $(\text{present\_state} + 1) \bmod 16$ . This causes the counter to wrap back to the zero state when it is in the last state. We use modulo 16 because this is the number of states in the state machine.

The last part of this design involves adding the load capability to our design. For this we use the APPEND statement. The append statement causes the expression given to be ORed to the variable specified. This entire state machine eventually becomes a set of equations for each of the state bits. Our intent is to add another condition where the registers are loaded with the value from a set of input pins. Remember that we used 'load' in all the IF statements. That was to make sure that the equations generated by those IF statements, would not conflict with the APPEND statement.

```
CountPin0.d = !load & ???.....  
             # !load & ???.....;
```

**Figure U5-33. Equations Before APPEND**

```
CountPin0.d = !load & ???.....  
             # !load & ???.....  
             # load & LoadPin0;
```

**Figure U5-34. Equations After APPEND**

As an exercise, try to add up/down capability to this example. Also, try implementing it into an actual device.

### **3 – Output Formats**

### 3.1 DOWNLOADABLE FILE FORMATS

CUPL can output the files described below.

**A JEDEC-compatible ASCII download file (*filename.JED*)** for input to a device programmer.

**An ASCII Hex download file (*filename.HEX*)** available for PROMs.

**An HL download file (*filename.HL*)** available for Signetics IFL devices.

**An absolute file (*filename.ABS*)** for use by **CSIM**, the CUPL logic simulation program.

**An error listing file (*filename.LST*)** that lists errors in the original source file.

**A documentation file (*filename.DOC*)** that contains expanded logic equations, a variable symbol table, product term utilization, and fusemap information.

**P-CAD PDIF file (*filename.PDF*)** that can be translated by PDIFIN into a PC-CAPS symbol representing the pinouts of the programmable logic device..

**A Berkeley PLA file (*filename.PLA*)** for use by the Berkeley PLA tools.

**A Open PLA file (*filename.PLA*)** for use by various back end fitters

This appendix describes downloadable file formats and the **.DOC** file.

## 3.2 JEDEC Format

The JEDEC JC-42.1 standard consists of a transmission that begins with an ASCII Start-of-Text (STX) character, followed by various fields of information: an ASCII End-of-Text (ETX) character, and a transmission checksum. The allowed legal characters consist of printable ASCII characters (hex 20 through 7E) and the four control characters listed in table C-1.

**Table C-1. Control Characters**

STX	Start-of-Text	hex 02
ETX	End-of-Text	hex 03
LF	Line Feed	hex 0A
CR	Carriage Return	hex 0D

Figure C-1 shows a sample JEDEC file created by using **CUPL** and **CSIM**.

```
<STX>
Cupl          3.0  Serial # 0-00000-000
Device        p16r4  Library DLIB-h-24-11
Created       Tue Jul 07 15:22:33 1987
Name          WAITGEN
Partno        P9000183
Revision      02
Date          03/14/85
Designer      Osann
Company       P-CAD
Assembly      PC Memory
Location      U106
*QP20
*QF2048
*G0
*F0
*L00000 10110101110111111100111000110111
*C0307
*QV
*P 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
*V0001 CXXXXX110N0HLLZXXHN
*<ETX>6AA1
```

**Figure C-1. Sample JEDEC File**

The rest of this section describes the fields in the sample file in Figure C-1.

The design specification is the first field in the format. It includes all information between the STX and the first asterisk (\*). This information is for documentation purposes only and consists of the header information from the **CUPL** source file along with the version number of the compiler and device library.

Each of the fields after the design specification field begins with one of the single character identifiers shown in Table C-2.

**Table C-2. Field Identifiers**

A - *	N - *
B - *	O - *

C - Fuse check sum	P - Pin Order
D - Device type	Q - Value
E - *	R - *
F - Default fuse state	S - *
G - Security fuse	T - *
H - *	U - *
I - *	V - Test vector
J - *	W - *
K - *	X - *
L - Fuse link data	Y - *
M - *	Z - *

\* - indicates reserved for future use

Characters that have not been defined are reserved for future use.

Fields can be identified by multiple characters; for example, **QF** to indicate a value default fuse state.

The device field (**D**) is no longer supported. In CUPL 1.0 the device field contained a four-digit code that identified the device to be programmed. However, lack of consistent support by the device programmer manufacturers led to this field being dropped from the JEDEC file format for CUPL 2.0 and higher.

A value field, **QP**, describes the number of pins for the device; another value field, **QF**, describes the total number of programmable fuses in the device. Both values are decimal numbers.

The security fuse field (**G**) instructs the programmer to disable (G0) or enable (G1) the programming of the security fuse on the devices that contain this option. A single space follows the number for compatibility with certain manufacturers' equipment.

The default fuse state field (**F**) defines the state of the fuses that are not explicitly defined in the L field. Since CUPL 3.0 does not transmit all fuse states (to speed data transmission on large designs), this field must be recognized by the device programmer.

The fuse link field (**L**) contains the actual data. Each device fuse link is assigned a decimal number, starting with 0000. Each numbered fuse has two possible states: binary 0 specifies a low resistance link (**FUSE INTACT**) and binary 1 specifies a high resistance link (**FUSE BLOWN**).



## Note

Some manufacturers specify "test fuses" for purposes of running AC parameter tests on the device before programming. These fuses are not part of the fuse link data.

The **L** identifier begins the field and is followed by the number of the first fuse being defined in the field. When more than one binary value is specified, the additional values are assigned to fuses numbered consecutively from the first fuse number.

The next field is a fuse checksum (**C**) field. The checksum is a 16-bit hexadecimal value which is computed by adding 8-bit words formed from the specified state of each fuse link in the device. Link number 0 is the least-significant bit (lsb) and link number 7 is the most significant bit (msb) of word 0. Unspecified bits in the final 8-bit word are set to zero before computing the checksum. In the Figure C-1 the first thirty-two fuses generate four 8-bit words as follows:

			msb				lsb												
word	00	1	0	1	0	1	1	0	1	-->	AD								
word	01	1	1	1	1	1	0	1	1	-->	FB								

```

word 02 0 1 1 1 0 0 1 1 --> 73
word 03 1 1 1 0 1 1 0 0 --> EC
-----
0307

```

The optional test vector field (**V**) is created by running **CSIM** with the **-j** option flag. It contains functional test information for each device pin. The **QV** value field defines the number of test vectors which the file contains. Test vectors are numbered starting with 0001 and applied in numerical order to the device being tested. Table C-3 lists the valid conditions for any pin.

**Table C-3. Test Conditions**

0	-	Drive input LO (0 volts)
1	-	Drive input HI (+5 volts)
C	-	Drive input LO, HI, LO
K	-	Drive input HI, LO, HI
L	-	Test output LO (0 volts)
H	-	Test output HI (+5 volts)
Z	-	Test output for high impedance
X	-	Input undefined, Output untested
N	-	Power pins and Outputs not tested
P	-	Preload registers
Value given applied to $\bar{Q}$ of register		

The test conditions, as they appear in the vector, are applied to the device pins according to the sequence given in the pin order (**P**) field. In this example (figure C-1), the first condition is applied to pin 1 and the last to pin 20 of a 20 pin device. The **C** and **K** driving signals are presented after all other inputs are stable. The **L**, **H**, and **Z** conditions are tested after all inputs have been stabilized, including **C** and **K**.

The **P** driving signal on the clock pin is valid only for those devices capable of preloading registers with a supervoltage. Devices which use dedicated TTL-level preload pins must use the **C** or **K** driving signals on these pins to preload the registers.

The end of transmission is signified with a non-printing ASCII ETX character followed immediately by a transmission checksum (sum-check) of four ASCII hex characters. The checksum is the 16-bit sum of the ASCII values of all the transmitted characters between, and including, the starting STX and ending ETX characters. In the sample file (figure C-1), the transmission checksum calculates to 46C9, when taking into account a non-printing carriage return and line feed at the end of every line.

### 3.3 ASCII-Hex Format

The ASCII-hex format is generated for PROMs only. Data in this format is organized in sequential bytes separated by the execute character (space). Characters immediately preceding the execute character are interpreted as data bytes. The format may express the data bytes as either a single hex digit (x4 PROMs) or two hex digits (x8 PROM's)

An ASCII STX [Ctrl]-[B] character starts the transmission. A four-digit hexadecimal address, preceded by a \$ , A , and comma (**\$A,**) starts each line of 16 data bytes. An ASCII ETX [Ctrl]-[C] ends the data portion of the transmission. It is followed by forty spaces.

Figure C-2 shows a sample hex file.

```
^B
$A0000,00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
$A0010,10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
$A0020,20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
$A0030,30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
^C
$S07E0,
```

**Figure C-2. Sample Hex File**



### 3.4 HL Formats

The HL Download Format is generated for Signetics IFL devices only. Each device has its own unique format. All formats begin with STX [Ctrl]-[B] and end with ETX [Ctrl]-[C]. The following sections describe the format for each type of IFL device.

#### 82S100/101 FPLA

The active-level identifier **\*A** is followed by the states of the active levels for F7 to F0, where H denotes active-HI and L denotes active-LO. The product terms are described by the **\*P** identifier followed by a space and the P-term number. The input variable identifier **\*I** is followed by the input variables I15 to I0 and then the output function identifier **\*F** followed by F7 to F0.

Figure C-3 shows a sample file in this format.

```
^B
*A LHLHLHLH
*P 00 *I HHHHLLLLHHHHLLLL *F A.A.A.A.
.
.
*P 47 *I LLLLLLLLLLLLLLLLLL *F ....AAAA
^C
```

**Figure C-3. Sample 82S100/101 FPLA File**

#### 82S103 FPGA

The product terms start with the **\*G** identifier followed by a space and the term number. This is followed by the active-level identifier **\*A** and the active-level data. The input variable identifier **\*I** is followed by the input variables I15 to I0.

Figure C-4 shows a sample file in this format.

```
^B
*G 00 *A L *I HHHHLLLLHHHHLLLL
.
.
*G 47 *A H *I LLLLLLLLLLLLLLLLLL
^C
```

**Figure C-4. Sample 82S103 FPGA File**

#### 82S105 FPLS

The preset / output enable option is entered using the **\*A** identifier followed by H (preset) or L (output enable). The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the input variable identifier **\*I** followed by I15 to I0. The present state of the flip-flops is given with **\*P** followed by P5 to P0. The next state values follow with **\*N** and N5 to N0. The output function is described as **\*F** followed by F7 to F0.

Figure C-5 shows a sample file in this format.

```
^B
*AL
*T 00 *C.*ILLLLHHHHLLLLHHH*PHHLLL
      *NHHLLL *F HHHHHHH
.
.
*T 47 *C A *I LL----- *P LLLLLL
      *N HHHHHH *F HHHLLL-
^C
```

**Figure C-5. Sample 82S105 FPLS File**

## 82S151 FPGA

The direction of the I/O pins follows the **\*DIR** identifier, and the output polarity follows the **\*POL** identifier. The product terms are described by the **\*P** identifier followed by a space and the P-term number.

Control term numbers start with a **\*D** and then the term number. The input variable identifier **\*I** is followed by the input variables I5 to I0 and then the I/O feedback identifier **\*B** followed by B11 to B0.

Figure C-6 shows a sample file in this format.

```
^B
*DIR HHLHLLHLLHLLH *POL HHHHLLLLHLL
*P 00 *I HLLLLL      *B HL--HL--LLHH
.
.
*P 11 *I LLLLLL *B HHHHLLLLHLL
*D 02 *I ----HH    *B ----LLLLLL-
.
.
*D 00 *I LLLL-- *B HHHHHHLLLL-
^C
```

**Figure C-6. Sample 82S151 FPGA File**

## 82S153 FPLA

The output polarity identifier **\*POL** is followed by the states of the active levels for outputs B9 to B0, where H denotes active-HI and L denotes active-LO. The product terms are described by the **\*P** identifier followed by a space and the P-term number. Control term numbers start with a D and then the term number. The input variable identifier **\*I** is followed by the input variables I7 to I0. The feedback variables B9 to B0 follow the **\*BI** identifier and the output functions B9 to B0 follow the **\*BO** identifier.

Figure C-7 shows a sample file in this format.

```
^B
*POL HHLHLLHLLHH
*P 00 *I --HH--LL *BI --HL----- *BO A..A..A..A
.
.
*P 31 *I -----HH *BI HLHLHLHLHL *BO ....AA....
*P D9 *I --HHHHHH *BI ----HHHHLL
.
.
*P D0 *I LLLLLLLL *BI -----
^C
```

**Figure C-7. Sample 82S153 FPLA File**

## 82S155 FPLS

The output enable modes for groups A and B follow the **\*E** identifier. The flip-flop mode for each register follows an **\*F/F** identifier. The polarity for the output pins follows an **\*POL** identifier. The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the input variable identifier **\*I** followed by I3 to I0. The I/O feedback data follows the **\*B** identifier. The present state of the flip-flops is given with **\*QP** followed by Q3 to Q0. The next state values follow with **\*ON** and O3 to O0. The preset terms for groups PB and PA follow the preset identifier **\*P**. The reset terms for groups RB and RA follow the reset identifier **\*R**. The output function is described as **\*BO** followed by B7 to B0. The terms for flip-flop control, reset, preset, load, and output enable follow.

Figure C-8 shows a sample file in this format.

```

^B
*E AA *F/F A.A. *POL HLHLLHLH
*T 00 *C . *I HLLL *BI HL--HLHL *QP LH-
      *QN LLHH *P .. *R .. *BO .A.A.A.A
.
.
*T 31 *C A *I LLHH *BI ----HLHL *QP HLLL
      *QNHHHH *P .A *R .A *BO ..A.AA.A
*T FC *C . *I LLLL *BI LLLLHHHH *QP LLHH
*T LB *C . *I HLLL *BI --LL--LL *QP HHHH
*T LA *C . *I LL-- *BI LLLLHHHH *QP --LL
*T D3 *C . *I LLLL *BI LLLLLLL- *QP LLHH
.
.
*T D0 *C . *I LLLL *BI LLHHHLL *QP HLLH
^C

```

**Figure C-8. Sample 82S155 FPLS File**

## 82S157 FPLS

The output enable modes for groups A and B follow the **\*E** identifier. The flip-flop mode for each register follows a **\*F/F** identifier. The polarity for the output pins follows a **\*POL** identifier. The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the the input variable identifier **\*I** followed by I3 to I0. The I/O feedback data follows the **\*BI** identifier. The present state of the flip-flops is given with **\*QP** followed by Q5 to Q0. The next state values follow with **\*QN** and Q5 to Q0. The preset terms for group PA follows the preset identifier **\*P**. The reset terms for group PA follows the reset identifier **\*R**. The output function is described as **\*BO** followed by B5 to B0. The terms for Flip-Flop control, reset, preset, load and output enable follow.

Figure C-9 shows a sample file in this format.

```

^B
*E AA *F/F A.A. *POL HLHLLHLH
*T 00 *C . *I HLLL *BI HL--HL *QP LH--HL
      *QN LLHHHL *P . *R . *BO .A.A.A
.
.
*T 31 *C A *I LLHH *BI --HLHL *QP HLLHH
      *QNHHHLL *P A *R A *BO ..AA.A
*T FC *C . *I LLLL *BI LLLLHH *QP LLHHHH
*T PB *C . *I ---- *BI ----HH *QP ----LL
*T RB *C . *I HHHL *BI HLLLLL *QP HLLHH
*T LB *C . *I HLLL *BI --L-LL *QP HHHHLL
*T LA *C . *I LL-- *BI LLLHHH *QP --LLHH
*T D3 *C . *I LLLL *BI LLLLL- *QP LLHH-
.
.
*T D0 *C . *I LLLL *BI LLHHLL *QP HLLHLL
^C

```

**Figure C-9. Sample 82S157 FPLS File**

## 82S159 FPLS

The output enable modes for groups A and B follow the **\*E** identifier. The flip-flop mode for each register follows a **\*F/F** identifier. The polarity for the output pins follows a **\*POL** identifier. The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the the input variable identifier **\*I** followed by I3 to I0. The I/O feedback data

follows the **\*BI** identifier. The present state of the flip-flops is given with **\*QP** followed by Q7 to Q0. The next state values follow with **\*QN** and Q7 to Q0. The output function is described as **\*BO** followed by B7 to B0. The terms for Flip-Flop control, reset, preset, load and output enable follow.

Figure C-10 shows a sample file in this format.

```

^B
*E AA *F/F A.A.A.A. *POL LHLH
*T 00 *C . *I HLLL *BI HL-- *QP HLLLHH-
      *QN LLHLLHH *BO .A.A
.
.
*T 31 *C A *I LLHH *BI ---- *QP --HHHLL
      *QN LLLLHHHH *BO ...A
*T FC *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T PB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T RB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T LB *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T PA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T RA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T LA *C . *I LLLL *BI LLLL *QP LLLLHHHH
*T D3 *C . *I LLLL *BI LLLL *QP LLLLHHHH
.
.
*T D0 *C . *I LLLL *BI LLLL *QP LLLLHHHH
^C

```

**Figure C-10. Sample 82S159 FPLS File**

## 82S161 FPLA

The active level identifier **\*A** is followed by the states of the active levels for F7 to F0, where H denotes active-HI and L denotes active-LO. The product terms are described by the **\*P** identifier followed by a space and the P-term number. The input variable identifier **\*I** is then followed by the input variables I11 to I0 and the output function identifier **\*F** followed by F7 to F0.

Figure C-11 shows a sample file in this format.

```

^B
A LHLHLHLH
*P 00 *I LLLLHHHHLLLL *F A.A.A.A.
.
.
*P 47 *I LLLLLLLLLLLL *F ....AAAA
^C

```

**Figure C-11. Sample 82S161 FPLA File**

## 82S162 FPGA

The output polarity identifier **\*POL** is followed by the states of the active levels for outputs F4 to F0. The product terms start with the **\*G** identifier followed by a space and the term number. The input variable identifier **\*I** is followed by the input variables I15 to I0.

Figure C-12 shows a sample file in this format.

```

^B
*POL HLLL
*G 00 *I HHHLLLLLHHHHLLLL
.
.
*G 04 *I LLLLLLLLLLLLLLLLLL
^C

```

**Figure C-12. Sample 82S162 File**

### 82S163 FPGA

The output polarity identifier **\*POL** is followed by the states of the active levels for outputs F8 to F0. The product terms start with the **\*G** identifier followed by a space and the term number. The input variable identifier **\*I** is followed by the input variables I11 to I0.

Figure C-13 shows a sample file in this format.

```
^B
*G 00 *I HLLLHHHHLLLL
.
.
*G 08 *I LLLLLLLLLLLL
^C
```

**Figure C-13. Sample 82S163 FPGA File**

### 82S167 FPLS

The preset / output enable option is entered using the **\*A** identifier followed by H (preset) or L (output enable). The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the input variable identifier **\*I** followed by I13 to I0. The present state of the flip-flops is given with **\*P** followed by P7 to P0. The next state values follow with **\*N** and N7 to N0. The output function is then described as **\*F** followed by F3 to F0.

Figure C-14 shows a sample file in this format.

```
^B
*AL
*T 00 *C .*ILLHHHLLLLHHHH*PHHLLLLHH
          *NHHLLLLLL *F HHHH
.
.
*T 47 * A *I LL----- *P --LLLLLL
*N HLLLLHHH *F HL-
^C
```

**Figure C-14. Sample 82S167 FPLS File**

### 82S168 FPLS

The preset/output enable option is entered using the **\*A** identifier followed by H (preset) or L (output enable). The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the input identifier **\*I** followed by I11 to I0. The present state of the flip-flops is given with **\*P** followed by P9 to P0. The next state values follow with **\*N** and N9 to N0. The output function is described as **\*F** followed by F3 to F0.

Figure C-15 shows a sample file in this format.

```
^B
*AL
*T 00 *C.*ILLHHLLLLHHHH*PHHLLLLHHLH
          *NHHLHHLLLLL *F HHHH
.
.
*T 47 *C A *I LL----- *P --LLLLLL-
*N HLLLLHHH-L *F HL-
^C
```

**Figure C-15. Sample 82S168 FPLS File**

## 82S173 FPLA

The output polarity identifier **\*POL** is followed by the states of the active levels for outputs B9 to B0, where H denotes active-HI and L denotes active-LO. The product terms are described by the **\*P** identifier followed by a space and the P-term number. Control term numbers start with D and then the term number. The input variable identifier **\*I** is followed by the input variables I11 to I0. The feedback variables B9 to B0 follow the **\*B0** identifier.

Figure C-16 shows a sample file in this format.

```
^B
*POL HHLHHLHLLHH
*P 00 *I --HH--LLHH *BI --HL----- *BO A..A..A..A
.
.
*P 31 *I LL-----HH *BI HLHLHLHLHL *BO ....AA....
*P D9 *I LL--HHHHHH *BI ----HHHHLL
.
.
*P D0 *I --LLLLLLLLL *BI -----
^C
```

**Figure C-16. Sample 82S173 FPLA File**

## 82S179 FPLS

The output enable modes for groups A and B follow the **\*E** identifier. The flip-flop mode for each register follows a **\*F/F** identifier. The polarity for the output pins follows a **\*POL** identifier. The transition terms are described by the term identifier **\*T** followed by the term number. The complement array identifier **\*C** is followed by the value of this term. The input variables for the term are given with the input variable identifier **\*I** followed by I7 to I0. The I/O feedback data follows the **\*BI** identifier. The present state of the flip-flops is given with **\*QP** followed by Q7 to Q0. The next state values follow with **\*QN** and Q7 to Q0. The output function is described as **\*BO** followed by B7 to B0. The terms for flip-flop control, reset, preset, load, and output enable follow.

Figure C-17 shows a sample file in this format.

```
^B
*E AA *F/F A.A.A.A. *POL LHLH
*T 00 *C . *I HHLHHLH *BI HL-- *QP HHLHHLH-
      *QN LLHHLHHLH *BO .A.A
.
.
*T 31 *C *1 LLH--LHH *BI LLLL *QP --HHHHLL
      *QN LLLLHHHH *BO ...A
*T FC *C . *1 LLLHHHHL *BI LLLL *QP LLLLHHHH
*T PB *C . *I LLLHLHLL *BI LLLL *QP LLLLHHHH
*T RB *C . *I LLLLLLHL *BI LLLL *QP LLLLHHHH
*T LB *C . *I LLLHHHHL *BI LLLL *QP LLLLHHHH
*T PA *C . *I LLLHLHHL *BI LLLL *QP LLLLHHHH
*T RA *C . *I LLLL---- *BI LLLL *QP
*T LA *C . *I LLLL---H *BI LLLL *QP LLLLHHHH
*T D3 *C . *I LLLLLLLL *BI LLLL *QP LLLLHHHH
.
.
*T D0 *C . *I HHHHHHHH *BI LLLL *QP LLLLHHHH
^C
```

**Figure C-17. Sample 82S179 FPLS File**

### 3.5 DOCUMENTATION FILE FORMAT

This section describes the format for the documentation file (**filename.DOC**), including fuse plot information. A documentation file can be generated by specifying the **-x** option flag when running **CUPL**. Specifying the **-f** option generates a fuse plot in the documentation file.

Figure C-18 shows a sample documentation file.

```
*****
                        WAITGEN.DOC
*****
                        Waitgen
*****
CUPL                      3.0 Serial# 9-99999-999
Device
Created
Name
Partno
Revision
Date
Designer
Company
Assembly
Location
=====
                        Expanded Product Terms
=====
wait1.d =>
!memr
# a15
# a14
# a13
# reset
select_rom =>
!a13 & !a14 & !a15 & memr
wait2.d =>
!memr
# a15
# a14
# a13
# !wait1
memadr =>
a15 , a14 , a13 , a12 , a11
ready =>
!wait2
ready.oe =>
!a13 & !a14 & !a15 & memr
rom_os =>
!a13 & !a14 & !a15 & memr
memreq =>
memw
# memr
```

**Figure C-18. Sample Documentation File Sheet 1 of 5**

<pre> ram_cs0 =&gt; !a11 &amp; !a12 &amp; a13 &amp; !a14 &amp; !a15 &amp; memw #      !a11 &amp; !a12 &amp; a13 &amp; !a14 &amp; !a15 &amp; memr  ram_cs1 =&gt; a11 &amp; !a12 &amp; a13 &amp; !a14 &amp; !a15 &amp; memw a11 &amp; !a12 &amp; a13 &amp; !a14 &amp; !a15 &amp; memr  rom_cs.oe =&gt; 1  ram_cs0.oe =&gt; 1  ram_cs1.oe =&gt; 1 </pre>							
=====							
Symbol Table							
=====							
Pin	Variable		Pterms	Max	Min		
Pol	Name	Ext	Type	Used	Pterms	Level	
---	-----	---	---	---	---	---	---
	wait1		V	-	-	-	
	wait1	d	X	5	8	1	
	a11		V	-	-	-	
	select_rom		I	1	-	-	
	wait2		V	-	-	-	
	wait2	d	X	5	8	1	
	a12		V	-	-	-	
	a13		V	-	-	-	
	a14		V	-	-	-	
	a15		V	-	-	-	
!	oe		V	-	-	-	
!	memr		V	-	-	-	
	memadr		F	-	-	-	
	ready		V	1	7	1	
	ready	oe	X	1	1	1	
!	memw		V	-	-	-	
	cpu_clk		V	-	-	-	
!	rom_cs		V	1	7	1	
	reset		V	-	-	-	
	memreq		I	2	-	-	
!	ram_cs0		V	2	7	1	
!	ram_cs1		V	2	7	1	
	rom_cs	oe	D	1	1	0	
	rom_cs0	oe	D	1	1	0	
	rom_cs1	oe	D	1	1	0	
LEGEND    F:field       D:default variable       W:extended node							
N:node       I:intermediate variable    T:function							
V:variable   X:extended variable       U:undefined							

**Figure C-18. Sheet 2 of 5**



```

=====
                          Fuse Plot
=====
Pin #19
0000 -----
0032 -x---x-----x-----
0064 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0096 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0128 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0160 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0192 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0224 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #18
0256 -x---x---x-----x-----
0258 -----x-----
0320 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0352 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0384 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0416 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0448 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0480 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #17
0512 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0544 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0576 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0608 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0640 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0672 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0704 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0738 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin#16
0788 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0800 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0832 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0864 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0898 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0928 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0980 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0992 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #15
1024 -----x-----
1056 x-----
1088 ----x-----
1120 -----x-----
1152 -----x---
1184 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1216 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1248 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```

**Figure C-18. Sheet 3 of 5**

```

Pin #14
1280 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1312 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1344 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1378 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1408 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1440 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1472 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1504 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin#13
1536 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1568 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1600 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1632 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1664 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1696 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1728 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1760 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Pin #12
1792 -----
1824 -x---x--x---x---x---x-----
1856 -x---x--x---x---x-----x-----
1888 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1920 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1952 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
1984 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2016 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

LEGEND      X : fuse not blown
            - : fuse blown

```

**Figure C-18. Sheet 4 of 5**

```

=====
                          Chip Diagram
=====
*****
*
*
*
cpu_clk * 1
*****
*
Waitgen
*****
a15 * 2
*****
*
*****
a14 * 3
*****
*
*****
a13 * 4
*****
*
*****
a12 * 5
*****
*
*****
a11 * 6
*****
*
*****
!memw * 7
*****
*
*****
!memr * 8
*****
*
*****
reset * 9
*****
*
*****
GND * 10
*****
*
*****

*****
*****
*****
20 * VCC
*****
*
*****
19 * !rom_cs
*****
*
*****
18 * ready
*****
*
*****
17 *
*****
*
*****
16 *
*****
*
*****
15 * wait1
*****
*
*****
14 * wait2
*****
*
*****
13 * ram_cs1
*****
*
*****
12 * !ram_cs0
*****
*
*****
11 * !oe
*****
*
*****
*****

```

**Figure C-18. Sheet 5 of 5**

The first part of the file contains archival and revision information that is identical to the header information in the corresponding **CUPL** source file. The first part also contains version information about the device library and the CUPL program, and the date and time the file was created.

The next section of the file, Expanded Product Terms, contains the product terms generated by CUPL from the equations contained in the logic description file. **WAITGEN.PLD**, contained in the CUPL package, is the source logic description file for the sample documentation file in Figure C-18. Its contents can be viewed in order to compare the original logic equations with the product terms generated by **CUPL**.

CUPL generates product terms for the devices specified on the command line when running **CUPL** or with the **DEVICE** keyword in the logic description file. For example, some devices, such as PAL16L8s, contain fixed inverting buffers. In certain cases, to fit the logic to the device, CUPL performs DeMorgan's Theorem. For example, the logic description file is written for a PAL16L8 device; in the pin list, all outputs have been declared as active-HI. The following equation is written to specify an OR function.

**c = a # b ;**

However, the PAL16L8 contains a fixed inverting buffer. Because the inverting buffer cannot be changed, CUPL fits the logic to the device by performing a DeMorgan on the OR equation, which generates the following product term:

**c => !a & !b**

See Pin Declaration Statements in Chapter 2 for further information on how CUPL generates product terms for devices with fixed inverting buffers, when the output pins are asserted in the pin declaration statement.

The next section of the file, Symbol Table, provides information about each variable in the logic description file, including the pin number, extension, type of variable, number of product terms available, number of product terms used, and the minimization level used by CUPL.

If the maximum available product terms for a device is exceeded, CUPL displays an error message during compilation, naming the pin. However, the message doesn't indicate how much the limit was exceeded. The product term availability and use information in the Symbol Table (see Figure C-18) indicates if the number of available product terms was grossly exceeded, or was just slightly over the limit.

The next section, Fuse Plot, is generated by specifying the **-f** option flag when running **CUPL**. This section provides more detailed fuse information than that contained in the JEDEC file. The four-digit beginning JEDEC number for each product term is the number to use to reference **STUCK H** and **STUCK L** (see Fault Simulation in Chapter 4)

The last section, Chip Diagram, provides a diagram of the device showing the location of each variable name.

### 3.6 PDIF FILE FORMAT

This section describes the use of the PDIF (P-CAD Database Interchange Format) file (<F59>filename.PDF) generated by CUPL. For a detailed description of the PDIF-format file, see the PDIF User's Manual. A PDIF-format file can be generated by specifying the **-p** option flag when running **CUPL**.

The PDIF format is used as an interface to the P-CAD schematic capture program PC-CAPS. This is accomplished by translating the CUPL-generated PDIF-format file into a PC-CAPS symbol using the **PDIF-IN** program. The resulting symbol represents the logical representation of the PLD design. This includes pin packaging information, printed circuit board reference designator, PLD type, and design name.

Figure C-19 shows an example of a PC-CAPS symbol generated by **PDIF-IN**.

#### **Figure C-19. PC-CAPS Symbol Generated by PDIF-IN**

Refer to the *PDIF User's Manual* and the *PC-CAPS User's Manual* for instructions on running the **PDIF-IN** and PC-CAPS programs, respectively.

### 3.7 BERKELEY PLA FILE FORMAT

This section describes the format for the Berkeley PLA file (**filename.PLA**). The Berkeley PLA format is used as an interface format for PLA logic synthesis tools, such as the Berkeley PLA tools. A Berkeley PLA-format file can be generated by specifying the **-b** option flag when running **CUPL**.

Figure C-20 shows a sample Berkeley PLA-format file.

```
# Berkeley PLA format generated using
#
# CUPL                      3.0 Serial# 9-99999-999
# Device                   pl6rp4 Library DLIB-g-24-15
# Created                  Thu Feb 26 13:45:23 1987
# Name                     Count10
# Partno                   CA0018
# Revision                 01
# Date                     07/16/87
# Designer                 Kahl
# Company                  Assisted Technology
# Assembly                 None
# Location                 None
#
# Inputs 1 Q0 Q1 Q2
#          Q3 clr dir
# Outputs Q0.d Q1.d Q2.d Q3.d
#          carry carry.oe
.i 7
.o 6
.p 18
-00010- 100000
-0--00- 100000
-000101 010000
-10-000 010000
-01-000 010000
-11-001 010000
-001001 010000
-000101 001000
-110000 001000
--01000 001000
-1-1001 001000
-01100- 001000
-100101 000100
-000001 000100
-111000 000100
-000100 000100
-1001-- 000010
1----- 000001
.end
```

**Figure C-20. Sample Berkeley PLA-Format File**

The first part of the file contains archival and revision information. The **#** character indicates these are comments. This information is identical to the header information in the corresponding **CUPL** source file. There is also version information about the device library and CUPL program, the date and time the file was created, and a list of the input and outputs to the PLA.

The next section consists of a PLA description generated by CUPL from the equations contained in the logic description file, **COUNT10.PLD** (contained in the CUPL package). Its contents can be viewed to compare the original logic equations with the PLA description generated by CUPL.

The PLA description consists of fields to define the number of inputs '**i**', outputs '**o**', product terms '**p**', and a description of the AND and OR planes of the PLA with one line per product term. Connections in the AND plane are represented with a '**1**' for connection to the non-inverted input line and a '**0**' for connection to the inverted input line. No connection to an input line is indicated with a '**-**'. Connections in the OR plane are indicated by a '**1**'. No connections are indicated with a '**0**'. The end of the PLA description is indicated with an '**.end**' is actually caused by something else.

## 4 – Error Messages

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.

**Table 5-1. Error Message Module Suffixes**

Module	Suffix
CUPL processor	<u>ck</u>
CUPLX preprocessor	<u>cx</u>
CUPLA source file parser	<u>ca</u>
CUPLB equation fitter	<u>cb</u>
CUPLM minimizer	<u>cm</u>
CUPLC fusemap generator	<u>cc</u>
CSIM processor	<u>sk</u>
CSIMA logic simulator	<u>sa</u>
CBLD device library manager	ba
PTOC PALASM translator	pt

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.



### Note

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.

## 4.1 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.

## 4.2 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.

### 4.3 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.



## 4.4 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.



**Note**

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.

## 4.5 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.



## 4.6 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.



## 4.7 CSIM 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 library are 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.

## 4.8 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.

## **5 – Getting Help**

### **5.1 Common Errors**

### **5.2 How To Get Support**

The user of the CUPL product may need to have some questions answered or get help with the package in some way. If problems are encountered, LDI customer support may be contacted. There are several ways to do this.

By Telephone (305) 428-6868

By FAX (305) 428-1811

By Customer Support Electronic Bulletin Board Service (BBS) (305) 428-8014

The BBS requires that a modem be attached to a terminal or a computer. The BBS can be dialled, and messages may be left for the customer support staff. Additionally, files may be uploaded for review at LDI.

Baud:	300, 1200, 2400
Parity:	None
Data Bits:	8
Stop Bits:	1
Password:	user-defined

Special Interest Groups (SIGS) are also available; these can be a great source of information.

If assistance is needed with using CUPL, it is a good idea to contact your local dealer or distributor first, for their suggestions as to how to proceed.

### **5.3 Contacting Technical Support**

Before contacting Technical 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: This enables us to know if there is a known problem with your version.

The CUPL version number.

The device mnemonic that you were using when you encountered the problem.

Atmel EPLD Applications Engineering contacts:

Email: [pld@atmel.com](mailto:pld@atmel.com)

Tel: +1-408-436-4333