ATMEL – WinCUPL

USER'S MANUAL

Section 1

Introduction to Programmable Logic

1.1 What is Programmable Logic?

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

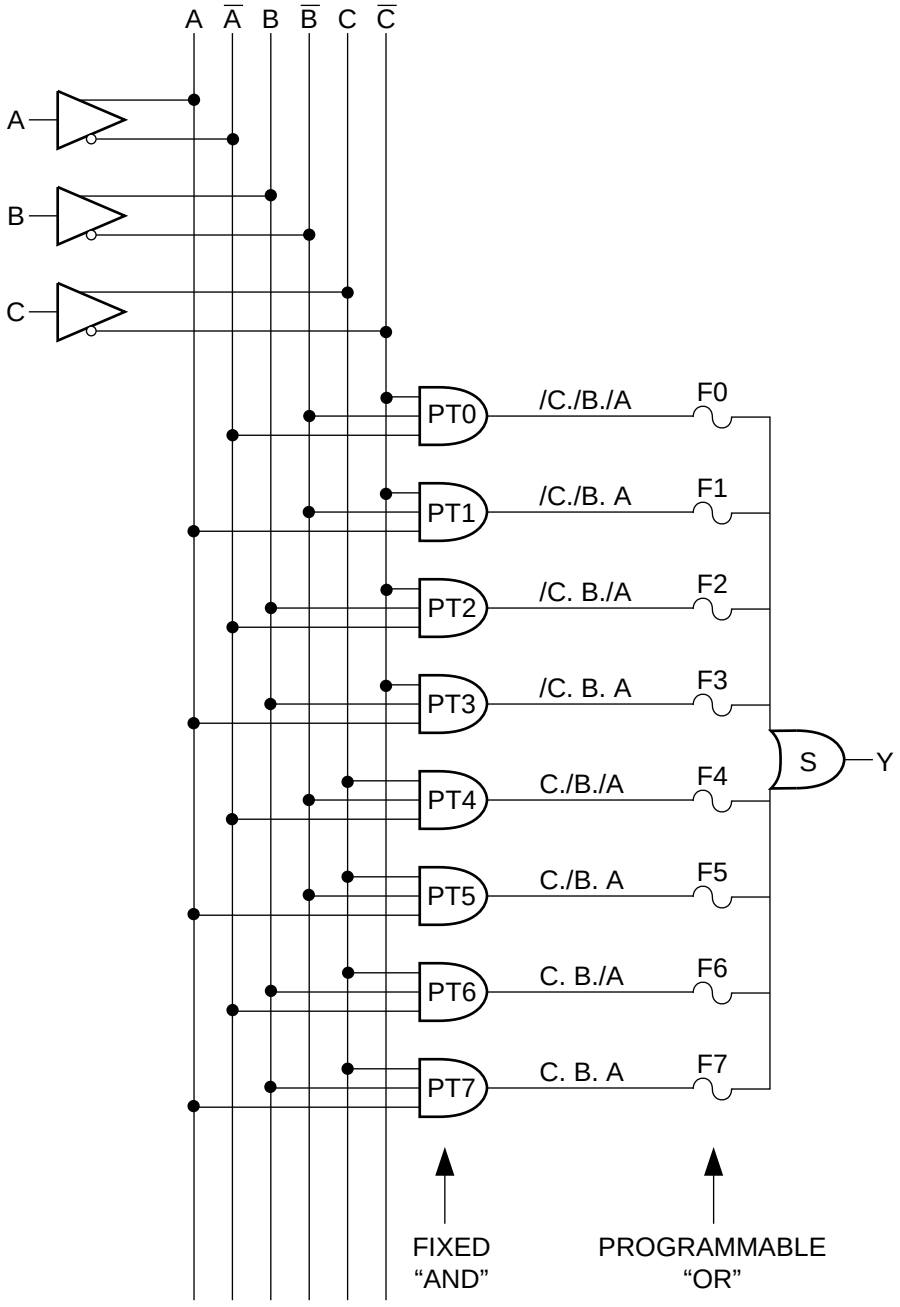
1.1.1 ASICs

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

1.1.2 Basic architecture of a user programmable device

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

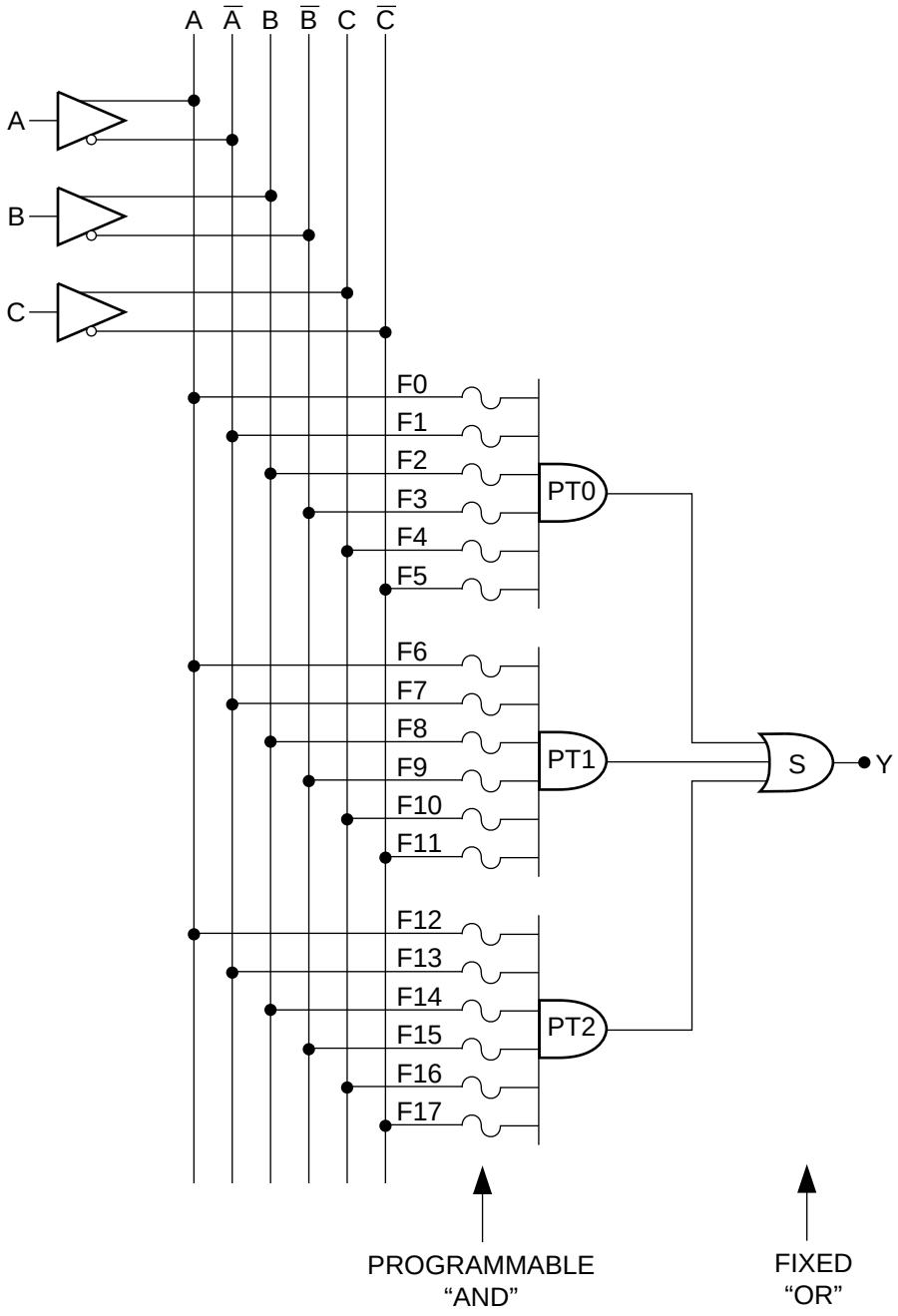
Figure 1-1. Elementary PROM architecture



1.1.3 PROMs

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

Figure 1-2. Elementary PAL architecture



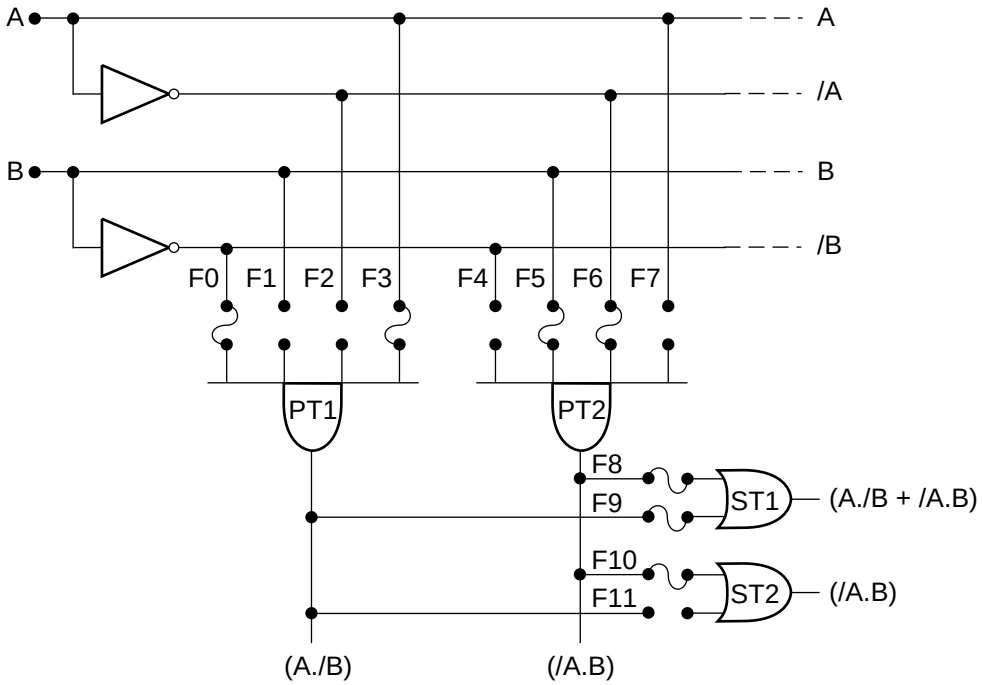
1.1.4 PALs

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

1.1.5 GALs

GALs are *Generic Array Logic* devices. They are designed to emulate many common PALs thought the use of macrocells. If a user has a design that is implemented using several common PALs, he may configure several of the same GALs to emulate each of the other devices. This will reduce the number of different devices in stock and increase the quantity purchased. Usually, a large quantity of the same device should lower the individual device cost. Also these devices are electrically erasable, which makes them very useful for design engineers.

Figure 1-3. Elementary PLA architecture



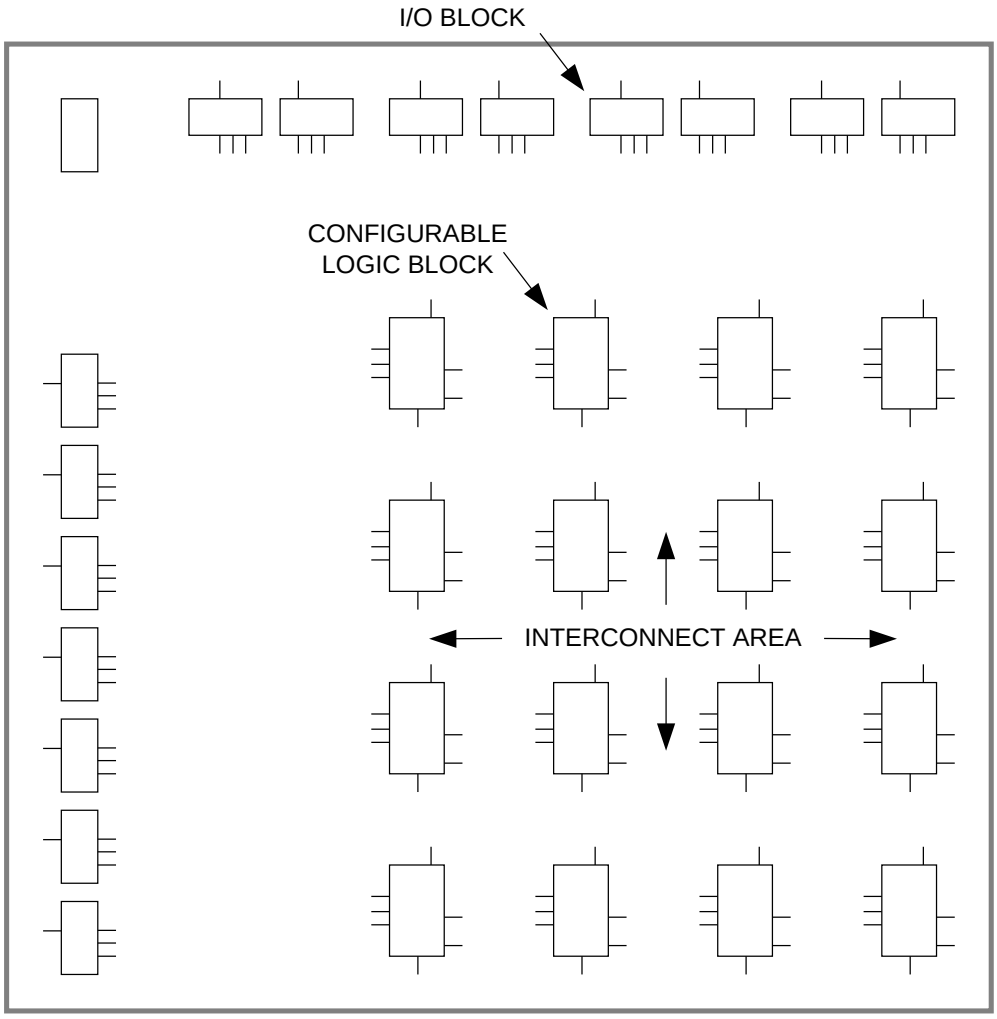
1.1.6 PLAs

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

1.1.7 Complex PLDs

Complex PLDs are what the name implies, *Complex Programmable Logic Devices*. They are considered very large PALs that have some characteristics of PLAs. The basic architecture is very much like a PAL with the capability to increase the amount of AND terms for any fixed OR term. This is accomplished by either stealing adjacent AND terms or using AND terms from an expander array. This allows for most any design to be implemented within these devices.

Figure 1-4. Elementary FPGA architecture



1.1.8 FPGAs

FPGAs are *Field Programmable Gate Arrays*. Simply put, they are electrically programmable gate array ICs that contain multiple levels of logic. FPGAs feature high gate densities, high performance, a large number of user-definable inputs and outputs, a flexible interconnect scheme, and a gate-array-like design environment. They are not constrained to the typical AND-OR array. Instead, they contain an interior matrix of configurable logic clocks (CLBs) and a surrounding ring of I/O blocks (IOBs). Each CLB contains programmable combinatorial logic and storage registers. The combinatorial logic section of the block is capable of implementing any Boolean function of its input variables. Each IOC can be programmed independently to be an input, and output with tri-state control or a bi-directional pin. It also contains flip-flops that can be used to buffer inputs and outputs. The interconnection resources are a network of lines that run horizontally and vertically in the rows and columns between the CLBs. Programmable switches connect the inputs and outputs of IOBs and CLBs to nearby lines. Long lines run the entire length or breadth of the device, bypassing interchanges to provide distribution of critical signals with minimum delay or skew. Designers using FPGAs can define logic functions of a circuit and revise these functions as necessary. Thus FPGAs can be designed and verified in a few days, as opposed to several weeks for custom gate arrays.

1.2 Device Technologies and Packaging

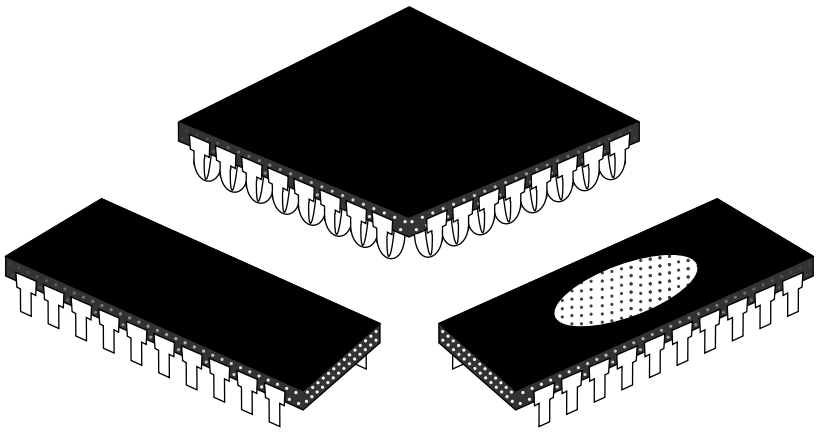
1.2.1 Device Technologies

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

1.2.2 Device Packaging

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

Figure 1-5. Picture of DIP and LCC devices



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

1.3 Programming Logic Devices

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

1.4 Functionally Testing Logic Devices

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

Section 2

Designing with the CUPL Language

When creating any design, it is generally considered good practice to implement the design using a “Top-Down” approach. A Top-Down design is characterized by starting with a global definition of the design, then repeating the global definition process for each element of the main definition, etc., until the entire project has been defined. CUPL offers many features that accommodate this type of design. This chapter describes the instructions that CUPL offers for implementing a design.

2.1 Declaration of Language Elements

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

2.1.1 Pin/Node Definition

Since the PIN definitions must be declared at the beginning of the source file, their definition is a natural starting point for a design. Nodes and pinnodes, used to define buried registers, should also be declared at the beginning of the source file. Pin assignment needs to be done if the designer already knows the device he wants to use. However, when creating a VIRTUAL design only the variable names that will later be assigned to pins need to be filled in. The area that normally contains the pin numbers will be left blank.

2.1.2 Defining Intermediate Variables

Intermediate variables are variables that are assigned an equation, but are not assigned to a PIN or NODE. These are used to define equations that are used by many variables or to provide an easier understanding of the design.

2.1.3 Using Indexed Variables

Variable names that end in a decimal number from 0 to 31 are referred to as indexed variables. They can be used to represent a group of address lines, data lines, or other sequentially numbered items. When indexed variables are used in bit field operations the variable with index number 0 is always the lowest order bit

Table 2-1. Using Number Bases

Number	Base	Decimal Value
‘b’0	Binary	0
‘B’1101	Binary	13
‘O’663	Octal	435
‘D’92	Decimal	92

Number	Base	Decimal Value
'h'BA	Hexadecimal	186
'O'[300..477]	Octal (range)	192..314
'H'7FXX	Hexadecimal (range)	32512..32767

2.1.4 Using Number Bases

All operations involving numbers in the CUPL compiler are done with 32-bit accuracy. Therefore, the numbers may have a value from 0 to 2^{32}-1. A number 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. Binary, octal, and hexadecimal numbers can have don't care ("X") values intermixed with numerical values.

2.1.5 Using List Notation

A list is a shorthand method of defining groups of variables. It is commonly used in pin and node declarations, bit field declarations, logic equations, and set operations. Square brackets are used to delimit items in the list.

Figure 2-1. Using The FIELD Statement

```
FIELD ADDRESS = [A7, A6, A5, A4, A3, A2, A1, A0];
FIELD DATA = [D7..D0];
FIELD Mode = [Up, Down, Hold];
```

2.1.6 Using Bit Fields

A bit field declaration assigns a single variable name to a group of bits. After making a bit field assignment using the *FIELD* keyword, the name can be used in an expression; the operation specified in the expression is applied to each bit in the group. 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 A0s will always occupy bit 0 in the bit field. This is mainly used for defining and manipulating address and data buses.

2.2 Usage of the Language Syntax

This section will discuss the logic and arithmetic operators and functions that are needed to create a Boolean equation design.

2.2.1 Using Logical Operators

Four standard logical operators are available for use: NOT, AND, OR, and XOR. The following table lists the operators and their order of precedence, from highest to lowest.

Table 2-2. Logical Operators

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

2.2.2 Using Arithmetic Operators And Functions

Six standard arithmetic operators are available for use in $repeat and $macro commands. The following table lists these operators and their order of precedence, from highest to lowest.

Table 2-3. Arithmetic Operators

Operator	Examples	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

One arithmetic function is available to use in arithmetic expressions being used in $repeat and $macro commands. The following table shows the arithmetic function and its bases.

Table 2-4. Arithmetic Function and Bases

Function	Base
LOG2	Binary
LOG8	Octal
LOG16	Hexadecimal
LOG	Decimal

The LOG function returns an integer value. For example:

```
LOG2(32) = 5 <==> 2**5 = 32

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

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

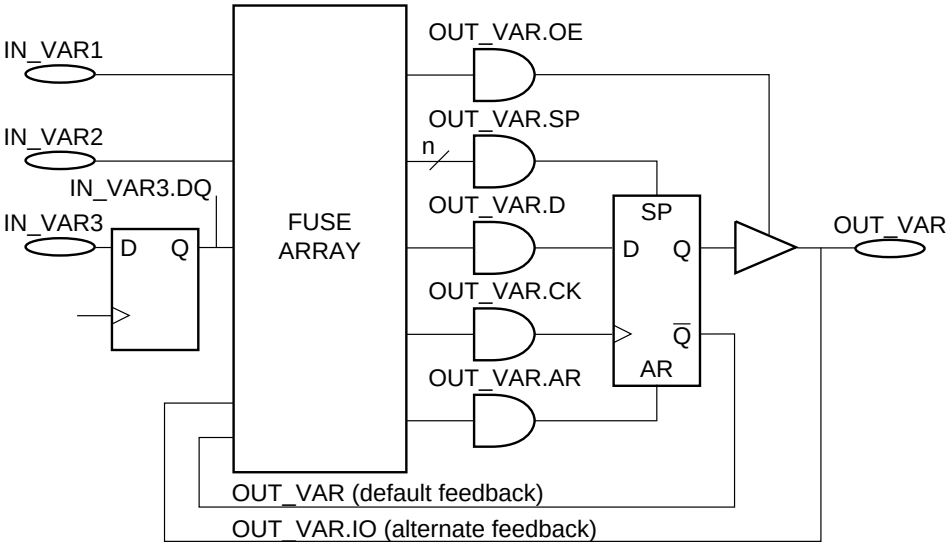
2.2.3 Using Variable Extensions

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 tri-state enables. 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. CUPL uses these extensions to configure the macrocells within a device. This way the designer does not have to know what fuses control what in the macrocells. To know what extensions are available for a particular device, use CBLD with the -e flag. A complete list of CUPL extensions can be found in the *CUPL PLD/FPGA Language Compiler* manual in the Extensions section of the CUPL Language chapter.

Table 2-5. Atmel PLD/CPLD's Variable Extensions

Extension	Side Used	Description
.AP	L	Asynchronous preset of flip-flop
.AR	L	Asynchronous reset of flip-flop
.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
.INT	R	Internal feedback path for registered macrocell
.IO	R	Pin feedback path selection
.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
.LQ	R	Q output of transparent input latch
.OE	L	Programmable output enable
.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
.T	L	T input of toggle output flip-flop
.TFB	R	T registered feedback path selection

Figure 2-2. Circuit Illustrating Extensions

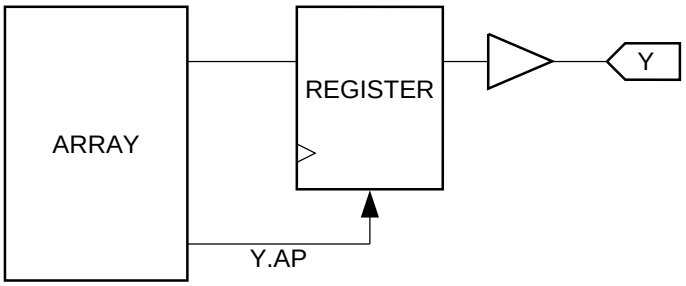



```
OUT_VAR.D = IN_VAR1 & OUT_VAR
          # !IN_VAR2 & IN_VAR3.DQ
          # !IN_VAR1 & OUT_VAR.IO
```

Figure 2-2 shows the use of extensions. Note that this figure does not represent an actual circuit, but shoes how to use extensions to write equations for different functions in a circuit.

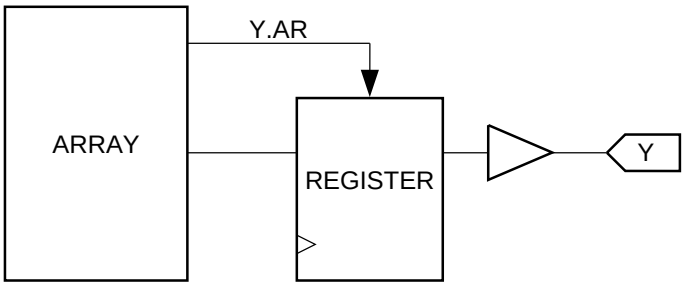
.AP Extension

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. This feature is supported on the Atmel ATF1500 family of devices.



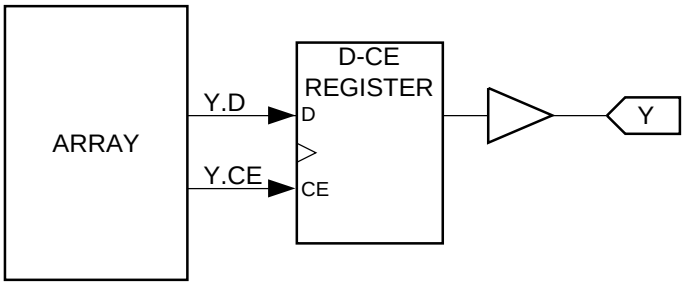
.AR Extension

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. Devices which have a pin-controlled reset inputs, such as the Atmel ATF1500 family also use this suffix.



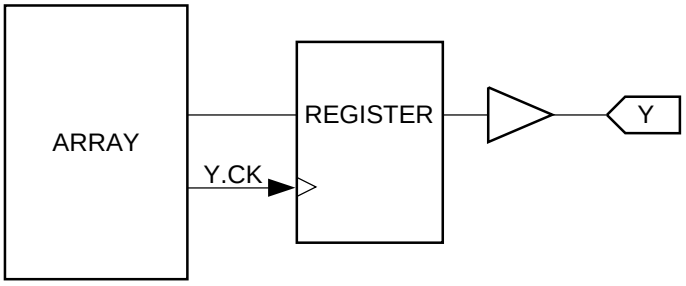
.CE Extension

The .CE extension is used for D-type registers which have a clock enable input (D-CE registers). It serves to specify the input to the Clock enable term of the register. In devices that have D-CE registers such as the ATV2500B and the ATF1500 family, the CE terms if not used must be set to binary 1, so that the registers behave as D registers. Normally, the CUPL compiler or Atmel ATF1500 family fitter will automatically do this for you.



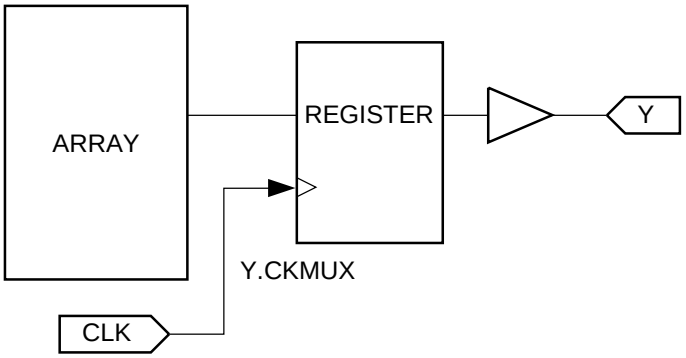
.CK Extension

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. Use this suffix to connect the clock for a register to the dedicated clock pin for any Atmel device that has this feature except the ATV750B (refer to .CKMUX Extension).



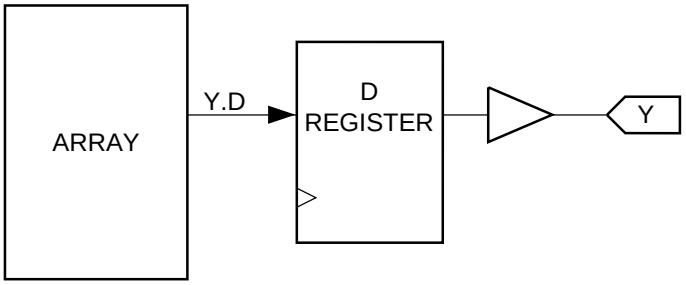
.CKMUX Extension

The .CKMUX extension is used to connect the pin clock to the register. This extension is used on the Atmel ATV750B device. Using this extension can reduce the number of product needed to implement the design into the Atmel device, and can increase the AC timing performance of the design. Other Atmel Devices such as the ATV2500/B and ATF1500 family also have dedicated clock pins, but they are not specified with the .CKMUX extension. They use the .CK extension instead.



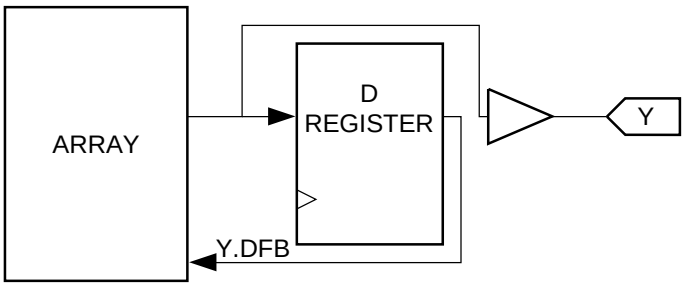
.D Extension

The .D extension is used to specify the D input to a D-type register. This options causes the compiler to configure the macrocells in the Atmel PLD device to D-type registers. For Atmel PLD's such as the ATF16V8B/20V8B/22V10B, ATV750/B and ATV2500/B the .D extension must be used for registered logic. Otherwise, CUPL will generate an error.



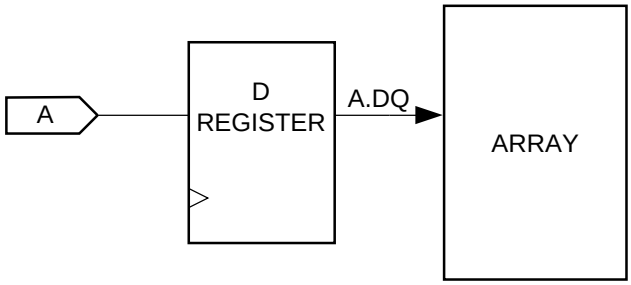
.DFB Extension

The .DFB extension is used when the macrocell on an Atmel device is configured for a combinatorial output but the D register still remains connected to the output. This configuration is supported on the ATV750/B and ATV2500/B devices. The .DFB allows the registered representation of the combinatorial output to be fedback internally into the Atmel Device. If you are interested in using this feature please contact Atmel PLD Applications.



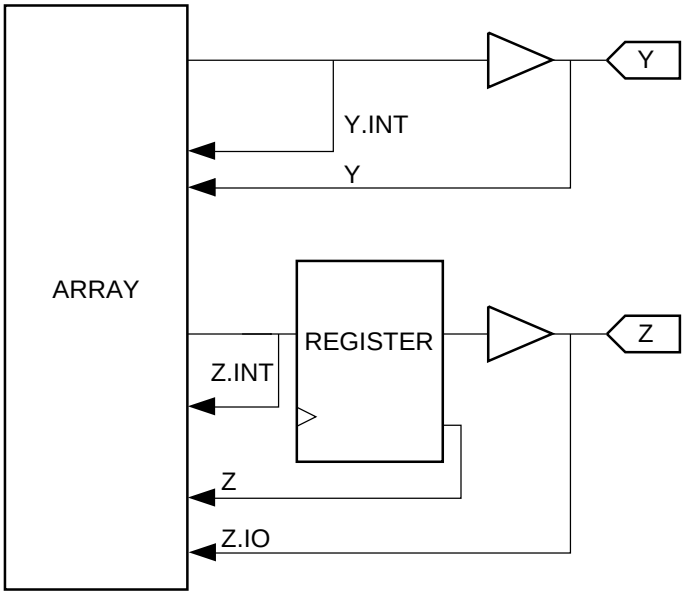
.DQ Extension

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. This feature is available on the ATF1500 family of device with 128 or greater macrocells. If you are interested in using this feature please contact Atmel PLD Applications.



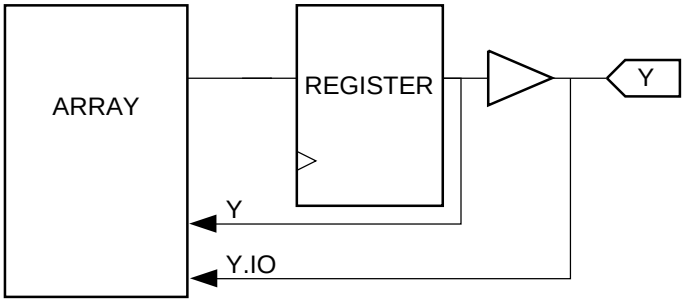
.INT Extension

The .INT extension is used for selecting an internal feedback path. This could be used to specify the buried combinatorial feedback path for either a registered or combinatorial output. This feature is available for the ATF1500 family of devices with 128 or greater macrocells. If you are interested in using this feature contact Atmel PLD applications.



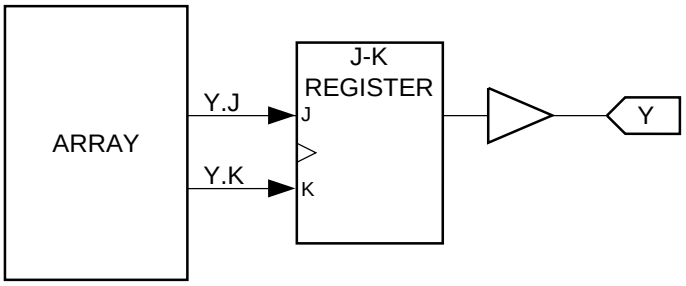
.IO Extension

The .IO extension is used to select pin feedback when the macrocell. This is useful, when a design requires using an I/O pin an input and also requires buried logic to be used within the same macrocell. It is also useful for implementing bi-directional outputs in CUPL. For examples on how implement bi-directional I/O in CUPL refer to the *Tips for Using Test Vectors* Application Note in the Atmel Configurable Logic databook, or contact Atmel PLD applications.



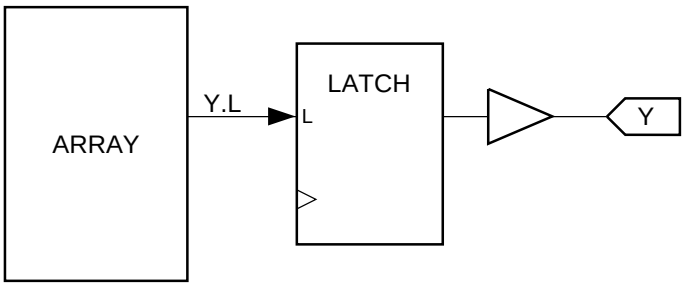
.J and .K Extensions

The .J and .K extensions are used to specify J and K inputs to a J-K register. Logic equations using these extensions can written and the REGISTER SELECT keyword when the D or T-type target register is specified. This keyword is supported in CUPL versions 4.6 and greater.



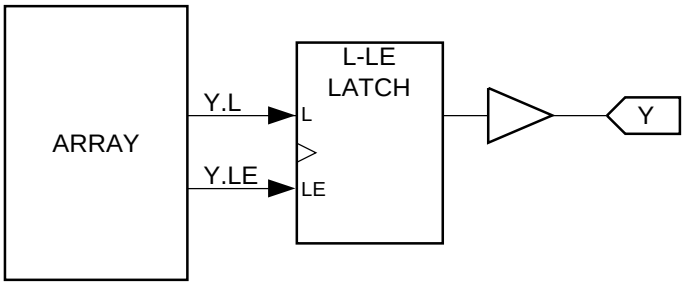
.L Extension

The .L extension is used to specify input into a latch. This extension is required to use the level-triggered latch feature available on the ATF1500 family of devices. The use of the .L extension causes the compiler to configure the macrocell as a latched output.



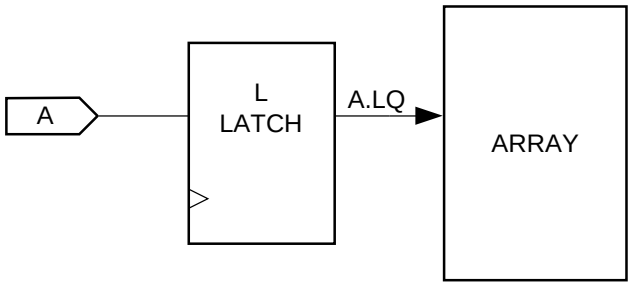
.LE

The .LE extension is used to specify the latch enable equation for a latch. It is required for designs using the level-triggered latch feature available on the ATF1500 family of devices. The .LE extension causes a product term to be connected to the latch enable.



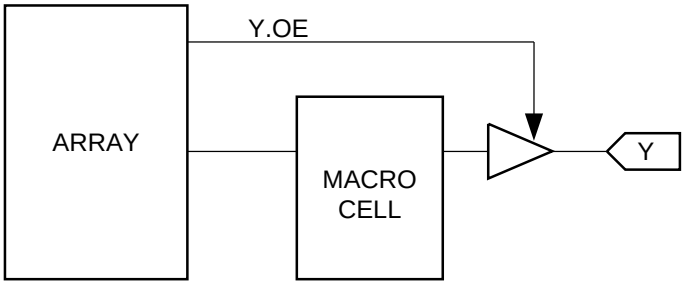
.LQ Extension

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 a output latch. This feature is available on the ATF1500 family of devices with 128 or more macrocells. If you are interested in using this feature contact Atmel PLD applications.



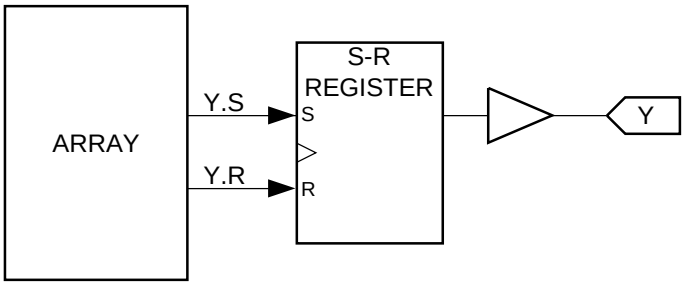
.OE Extension

The .OE extension is used to specify a product term driven output enable signal. It is required for using bi-directional I/O and the individually programmable output-enable product terms available on Atmel ATV750/B, ATV2500/B and ATF1500 family of devices. Atmel Devices which have a pin-controlled OE inputs, such as the ATF1500 family also use this suffix.



.S and .R Extensions

The .R and .S extensions are used to specify R (reset) and S (set) inputs to a SR register. Logic equations for the ATF1500 family can be written using these extensions if the REGISTER SELECT keyword is used and a D or T-type target register is specified. This keyword is supported in CUPL versions 4.6 and greater.

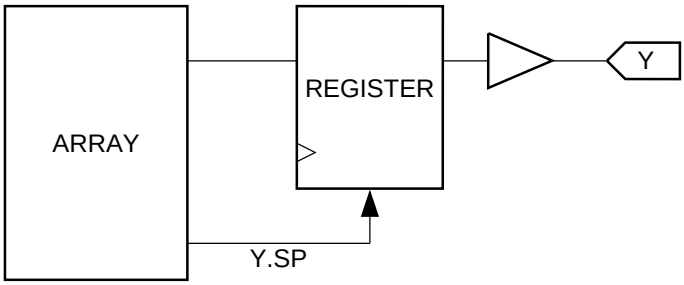


.SP Extension

The .SP extension is used to set the Synchronous Preset of a register to an expression. For example. the equation:

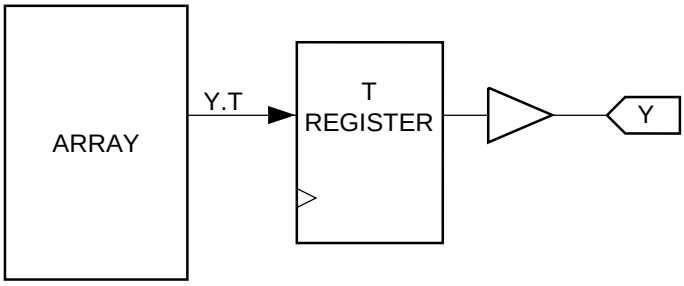
Y.SP = A & B ; /* **A** and **B** are inputs */

causes the output **Y** to be preset synchronous with the local clock used in the macrocell when inputs **A** and **B** are true. This feature is supported on Atmel ATF22V10B, ATV750/B and ATV2500/B devices which share Synchronous preset product terms.



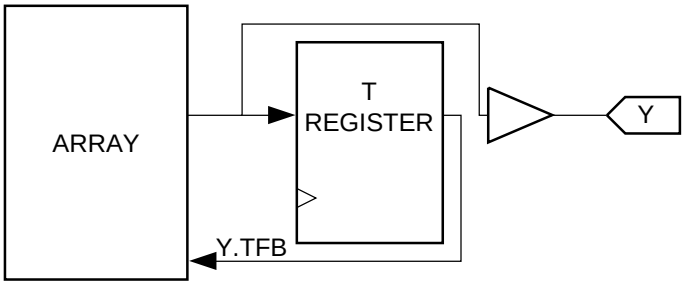
.T Extension

The .T Extension specifies the T input for a T register. The use of the T extension causes the compiler to configure the macrocell as a T-type register. This feature is supported on the Atmel ATV750B/2500B and ATF1500 family of devices.



.TFB Extension

The .TFB extension is used when the macrocell on an Atmel device is configured for a combinatorial output but the T register still remains connected to the output. This configuration is supported on the ATV750B and ATV2500B devices. The .TFB allows the registered representation of the combinatorial output to be fedback internally into the Atmel Device. If you are interested in using this feature, please contact Atmel PLD Applications.



2.2.4 Defining Logic Equations

Logic equations are the building clocks 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 list notation. When a variable list is used, the expression is assigned to each variable in the list

.ext is an option variable extension to assign a function to the major nodes inside programmable Devices.

exp is an expression; that is, a combination of variables and operators.

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

! is the complement operator.

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 APPENDed logic equation is logically ORed to the original equation for that variable. The format for using the APPEND statement is identical to defining a logic equation except the keyword APPEND appears before the logic equation begins.

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

2.2.5 Using 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).

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.

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.

2.2.6 Using 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 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.

The equality operator can also be used with a set of variables that are to be operated upon identically. For example, the following three expressions:

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

are equivalent respectively to:

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

2.2.7 Using 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.

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;
```

2.3 Advanced Language Syntax

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

2.3.1 Defining 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. 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 decoded 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:

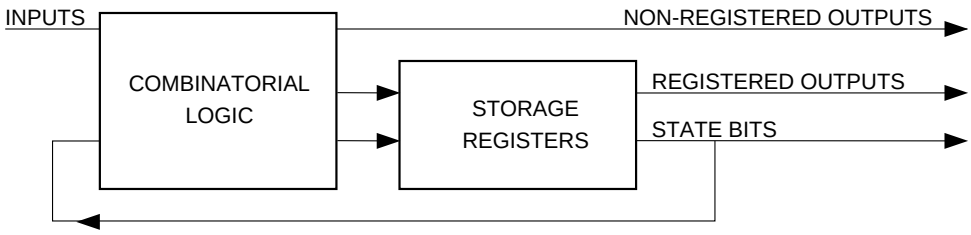
```
FIELD input = [in3..0];
FIELD output = [out3..0];
TABLE input => output {
   0=> 00; 1=>01;  2=>02;  3=>03;
   4=>04;  5=>05;  6=>06;  7=>07;
   8=>08;  9=>09;  A=>10;  B=>11;
   C=>12;  D=>13;  E=>14;  F=>15;
}
```

2.3.2 Defining State Machines

A state machine, according to AMD/MMI, is “a digital device which traverses through a predetermined sequence of states in an orderly fashion.” 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 2-3, uses six components: inputs, combinatorial logic, storage registers, state bits, registered outputs, and non-registered outputs.

Figure 2-3. State Machine Model



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

Combinatorial Logic - is any combination of logic gates (usually AND-OR) that produces an output signal that is valid Tpd (propagation delay time) nsec after any of the signals that drive these gates changes. Tpd is the delay between the initiation of an input or feedback event and the occurrence of a non-registered output.

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

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

To implement a state machine, 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 SEQUENCE keyword causes the storage registers and registered output types generated to be the default type for the target device. Along with the SEQUENCE keyword are the *SEQUENCED, SEQUENCEJK, SEQUENCERS, and SEQUENCET keywords*. Respectively, they force the state registers and registered outputs to be generated as D, J-K, S-R, and T-type flip-flops. The format for the SEQUENCE syntax is as follows:

```
SEQUENCE state_var_list {
   PRESENT state_n0
      IF (condition1)NEXT  state_n1;
      IF (condition2) NEXT state_n2    OUT out_n0;
      DEFAULT        NEXT  state_n0;
   PRESENT state_n1
      NEXT  state_n2;
   .
   .
   .
   PRESENT state_nn statements;
}
```

where

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

state_n is the state number and is a decode 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 subsection.

2.3.3 Defining Multiple State Machines

The CUPL syntax allows for more than one state machine to be defined within the same PLD design. When multiple state machines are defined, occasionally the designer would like to have the state machines communicate with each other. That is, when one state machine reaches a certain state another state machine may begin. There are two methods of accomplishing state machine communication: using set operations on the state bits or defining a “global” register that can be accessed by both state machines.

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

2.3.4 Using Condition Statement

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;
   }
```

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.

2.3.5 Defining A Function

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 a follows:

```
FUNCTION name ([Parameter0 ,...,Parametern])
{     body   }
```

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. 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'0.

2.3.6 MIN Declaration Statements

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

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 ;
```

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,

```
MIN [var, var, ... var] = level
```

.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 minimization levels available: None, Quick, Quine McClusky, Presto, Espresso.

The following are examples of valid MIN declarations.

```
MIN async_out    = 0;  /* no reduction */
MIN [outa, outb] = 1;  /* Quine McClusky reduction */
MIN count.d      = 4;  /* Espresso 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.

Section 3

Using the CUPL Compiler

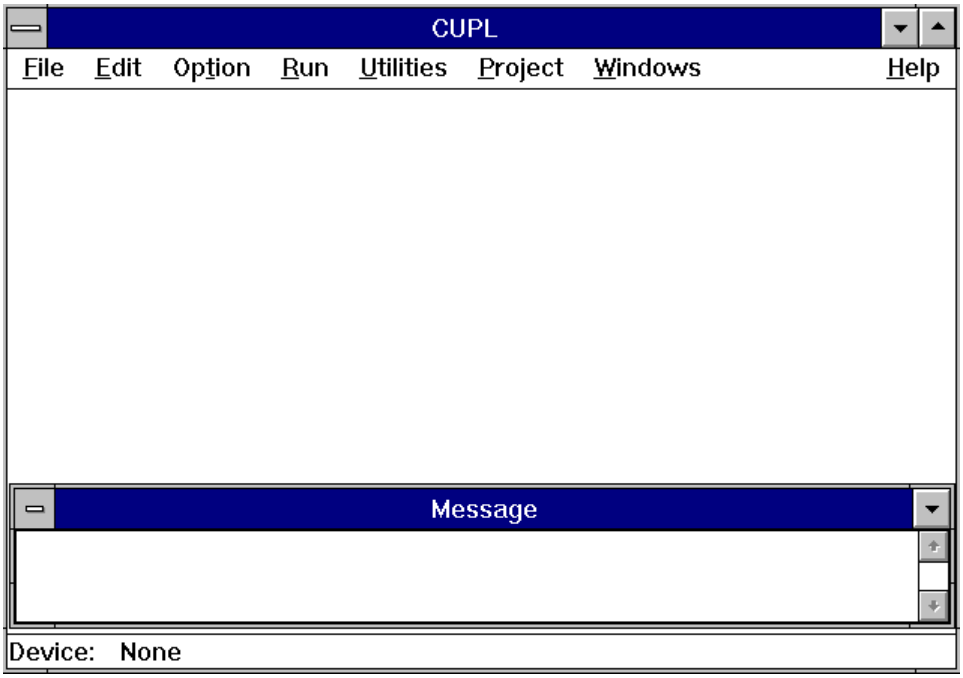
This chapter briefly describes CUPL source file operations and the types of output that CUPL creates under the CUPL for Windows environment.

3.1 About The Compiler

The CUPL compiler is a program that takes a text file consisting of high level directives and commands and creates files consisting of more primitive information. This information is either used by a device programmer to program a logic function into a programmable logic device or by a simulator to simulate the design.

To start CUPL, double click on the CUPL icon in the program manager for the Windows 3.X operating system, or click on Start, <Programs>, <CUPL>, CUPL in Windows 95.

Figure 3-1. CUPL's Main Screen



3.1.1 CUPL's Menu

File Menu - Controls and features relating to general program manipulation.

New - Opens a template PLD file for a new design.

Open - Opens an existing file for modification.

Save - Saves the current file being modified.

Save As - Save the current file as a new file with a different name.

Print - Print the currently selected document.

Exit - Exit the program.

Edit Menu - Controls and utilities for editing files.

Cut - Moves the selected text to the clipboard.

Copy - Copies the selected text to the clipboard.

Paste - Paste text from the clipboard to the current cursor location.

Delete - Delete the selected text.

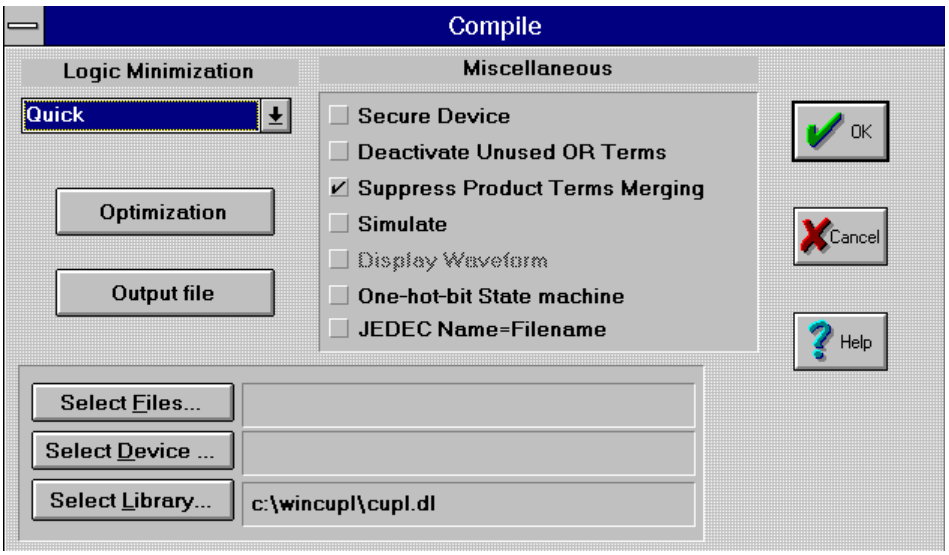
Copy Message - Copy the contents of the message window to the clipboard.

Search - Search for a text string in the body of text.

Line To - Advance to the line number selected.

Option Menu - Menu for selecting options related to CUPL's performance and compilation.

Figure 3-2. CUPL's Compiler Options

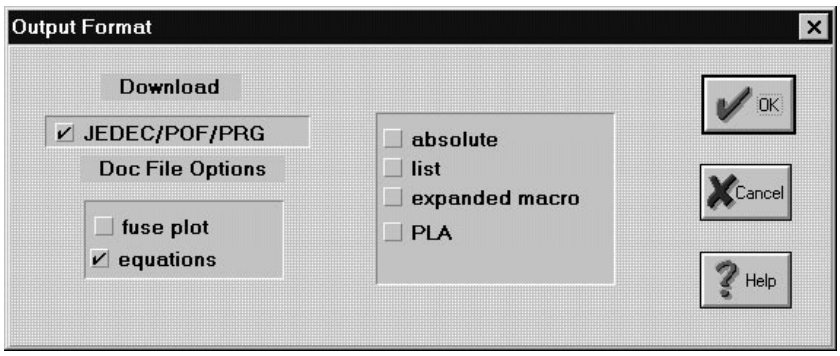


Compiler Options - Options directly affecting CUPL's compiler in minimization, optimization, and selecting output file formats.

Logic Minimization - Select the level of minimization desired on the entire design. Please note that pin by pin minimization is available. See Section 2.3.6–MIN Declaration Statement for more information on CUPL's minimization techniques.

Optimization - Select the optimizations desired on the entire design. Please note that pin by pin optimization is also available. See Section 2.3.6–MIN Declaration Statement for more information on CUPL's optimization techniques.

Figure 3-3. Output Format Files



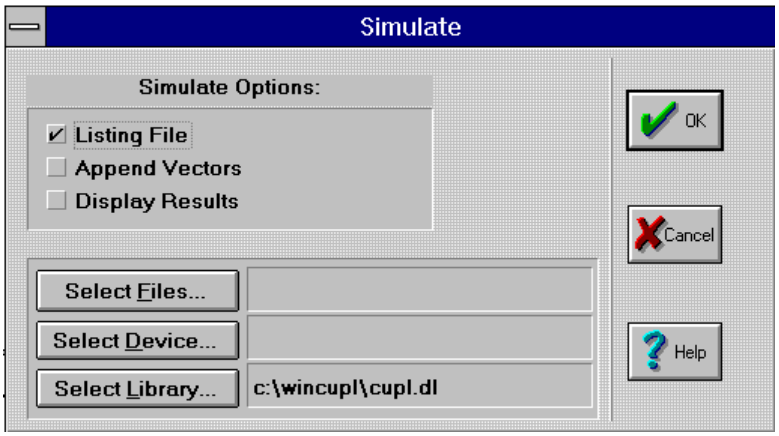
Output file - Select the output files needed for the design.

Download - Select the file type to download to the programmer.

DOC File Options - Select the options for the .DOC file.

Output - Several output formats are available from the compilation.

Figure 3-4. Simulator Options



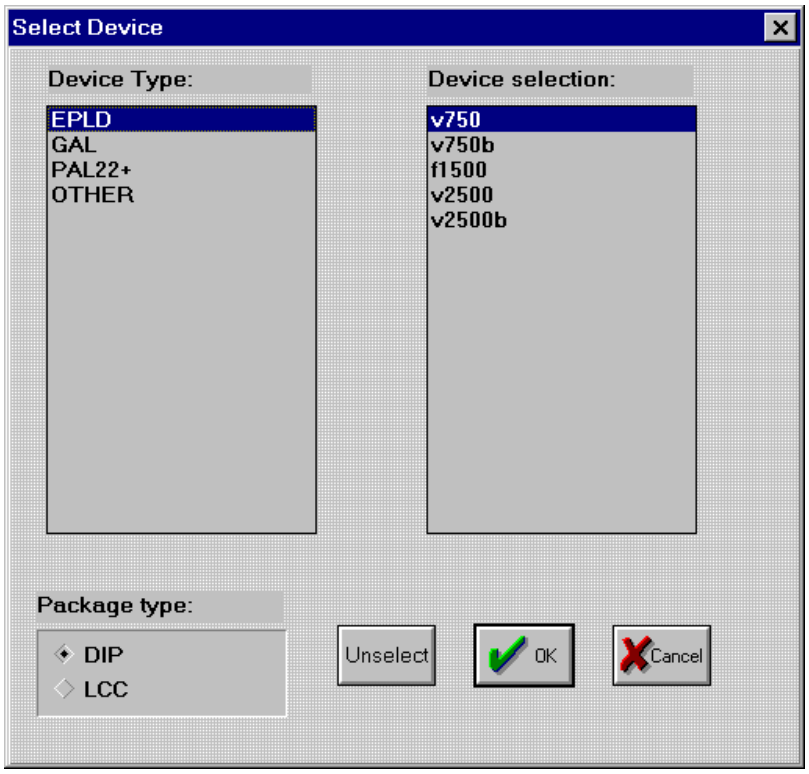
Simulator Options - Options related to simulation of the .PLD file.

Listing File - Create a simulation output file (.SO).

Append Vectors - Add test vectors to a JEDEC file.

Display Results - Display the waveform outputs graphically.

Figure 3-5. Device Selection Dialog Box



Select Device - Allows the user to select a device to target. *Using this option is not necessary and will override the device selection in the .PLD file.*

To select the device, click on the general type of PLD it is. Next select DIP or PLCC nmeumonic and specific type of device. Note that if the device type is only available in PLCC it only appears in the DIP section.

Select Library - Allows the user to choose a user supplied library.

Select Files - Allows the user to specify which file should be compiler.

Preferences - User defined preferences affecting environment

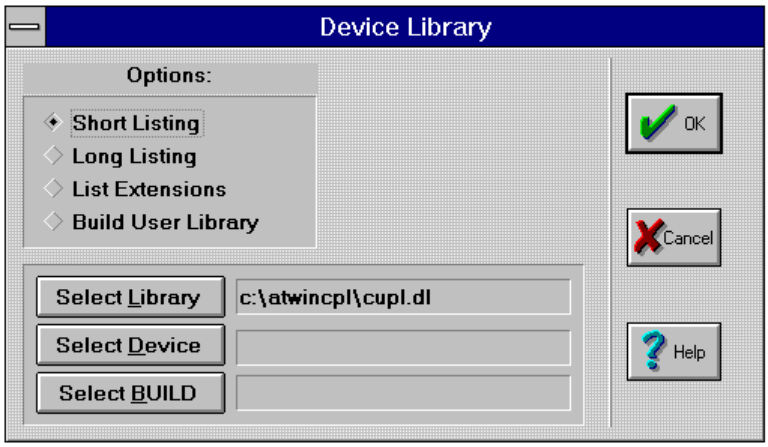
Run Menu - Compile, simulate, and analysis.

Device Specific Compile - Compile the currently selected design for the specific device selected.

Device Specific Simulate - Simulate the currently selected design for the specific device selected.

Utilities Menu - Additional useful utilities.

Figure 3-6. Device Library



Device Library - utility for manipulating CUPL's device library.

Options - Allows the user to specify what action is to be taken.

Short Listing - Outputs to the message box a wide list of all the devices nmeumonics contained in the selected device library.

Long Listing - Outputs to the message box a list of all the device mnemonics in the selected library along with revision information, number of pins, number of fuses, and total number of available product terms in the device.

List extensions - Lists the extensions of the selected device, or if no device is selected it lists extensions of all devices, from the library selected to the message window.

Build User Library - Build a user library from an existing library.

Select Library - Allows the user to select a device library.

Select Device - Allows the user to select a device (*see figure 3-6*).

Select Build - Select a user build file.

Calculator - Calls Windows calculator.

File Manager - Calls Windows File Manager.

DOS Prompt - Calls Windows DOS Prompt.

Project - CUPL's project option.

Load - Loads a project file for a .PLD file.

Save - Saves a project file which includes compiler and simulator settings.

Windows - Manipulation of multiple document interface windows.

Cascade - Cascade open windows.

Tile - Tile all open windows.

Arrange Icons - Arrange the icons of minimized windows in the CUPL window.

Help - On-line help files and general information about CUPL.

Index - Open the help file for CUPL for Windows.

Using Help - Information on how to use the help menu.

About - Opens the about CUPL dialog box. Contains version information.

3.2 Output File Format Descriptions

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

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.

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

Section 4

CUPL Tutorial

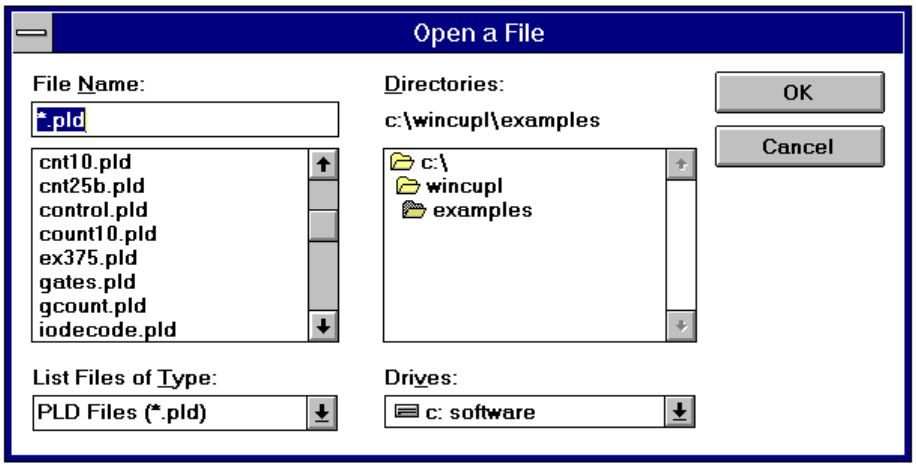
This section covers an example of how to use CUPL for Windows to compile a simple program and basically show the general flow of a design using CUPL.

4.1 Tutorial for Gates

Start CUPL by double clicking on the CUPL icon in Windows 3.X, or Windows 95 or WinNT click on Start, <programs>, <CUPL>, CUPL.

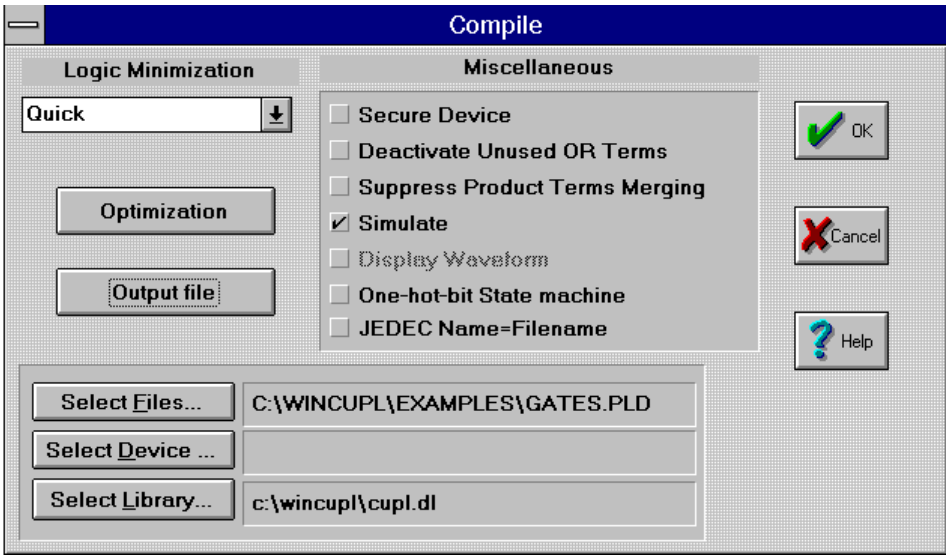
Under the file menu click on open.

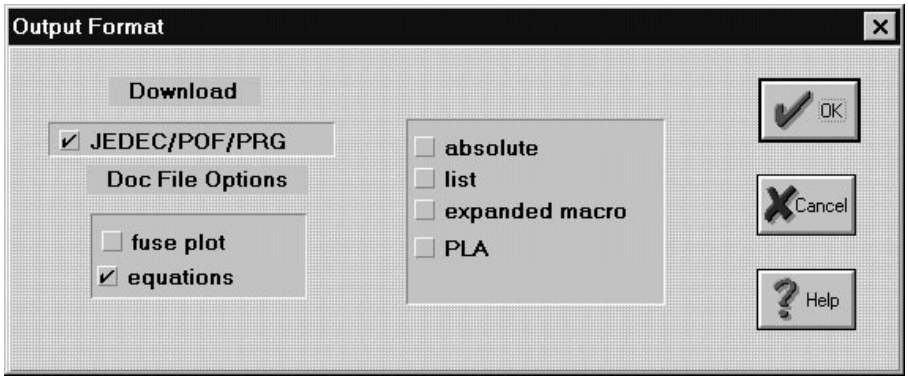
Figure 4-1. Open Dialog Box



Select gates.pld and click OK. The file gates.pld should appear in the CUPL windows. Take a second to look over the file. This file illustrates the use of CUPL's basic combinatorial logic.

Figure 4-2. GATES.PLD Compiler Options

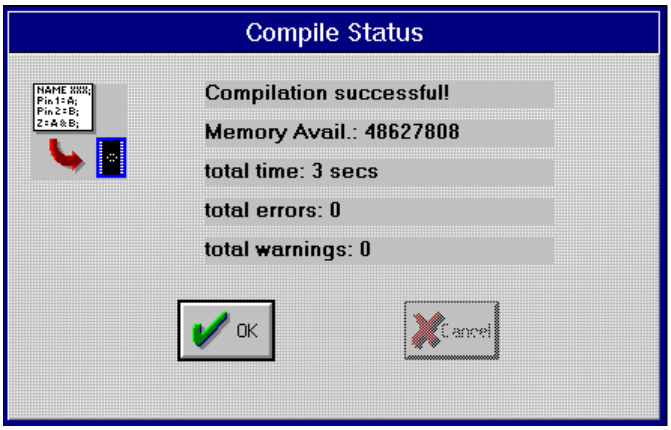




Click on <Options> and <Compiler Options> to bring up the compiler options dialog box. Click the simulate box and then click on <Output File>. Select JEDEC and absolute to produce a JEDEC file with test vectors. It is also useful to select Expanded Macro and listing files to get information on the compilation.

Once the compiler options are set, the file is ready to be compiled. Under the Compile menu, select Device Specific Compile, or press F9. If the file has been modified, it will need to be saved before a compile is performed. This function will compile the file, create the JEDEC file, simulate, and add the test vectors to the program.

Figure 4-3. Compile Status



After the compilation is completed, there will be several files created. The file gates.jed is used to download to a programmer in order to program the devices and do the functional testing. The file gates.so is the simulation output file, and displays the logic simulation CUPL created. The Expanded Macro file, gates.doc, is the logic, after minimization, the CUPL implemented into the device. The last file created is the Listing file, gates.lst, and is used to display errors in the source file.

4.2 Tutorial for COUNT10

This tutorial covers advance CUPL syntax of **State Machines** and **Conditional Statements**.

Open up the file COUNT10.PLD (in C:\WINCUPL\EXAMPLES directory) and look it over. The file uses the CUPL state machine syntax and conditional statements to demonstrate a four bit up/down decade counter with synchronous clear capability. The up, down, and clear statements control the direction and reset of the counter. An asynchronous ripple carry signal is generated when the counter reaches terminal count.

Click on <Options> and <Compiler Options> to bring up the compiler options dialog box. Click on <Output File> and select <JEDEC>, <absolute>, <simulation>, <Expanded Macro> and <list> options. Click on <Compile> and <Device Specific Compile> now to compile the file and generate the information files. Then open up the COUNT10.DOC file.

Figure 4-4. COUNT10.DOC

```
*************************************************************
                          Count10
*************************************************************

CUPL(WM)  4.7a Serial# MW-67999999
Device   g16v8ms Library DLIB-h-36-11
Created   Mon May 06 10:19:28 1996
Name   Count10
Partno   CA0018
Revision  02
Date   12/19/89
Designer  Kahl
Company   Logical Devices, Inc.
Assembly  None
Location  None
```

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

Q0.d =>
 !Q0 & !Q1 & !Q2 & Q3 & !clr
 # !Q0 & !Q3 & !clr

Q1.d =>
 !Q0 & !Q1 & !Q2 & Q3 & !clr & dir
 # Q0 & !Q1 & !Q3 & !clr & !dir
 # !Q0 & Q1 & !Q3 & !clr & !dir
 # Q0 & Q1 & !Q3 & !clr & dir
 # !Q0 & !Q1 & Q2 & !Q3 & !clr & dir

Q2.d =>
 !Q0 & !Q1 & !Q2 & Q3 & !clr & dir
 # Q0 & Q1 & !Q2 & !Q3 & !clr & !dir
 # !Q1 & Q2 & !Q3 & !clr & !dir
 # Q0 & Q2 & !Q3 & !clr & dir
 # !Q0 & Q1 & Q2 & !Q3 & !clr

Q3.d =>
 Q0 & !Q1 & !Q2 & Q3 & !clr & dir
 # !Q0 & !Q1 & !Q2 & !Q3 & !clr & dir
 # Q0 & Q1 & Q2 & !Q3 & !clr & !dir
 # !Q0 & !Q1 & !Q2 & Q3 & !clr & !dir

carry =>
 !Q0 & !Q1 & !Q2 & !Q3 & !clr & dir
 # Q0 & !Q1 & !Q2 & Q3 & !clr & !dir

clear =>
 clr

count =>
 Q3 , Q2 , Q1 , Q0

down =>
 !clr & dir

mode =>
 clr , dir

up =>
 !clr & !dir

carry.oe =>
 1
```

This is an example of how CUPL translates a state machine into simple Boolean logic. The CUPL state machine syntax is a very useful tool in designing counters, processes, or any sequence of events.

4.3 Tutorial for SQUARE.PLD

To start this example, click on <File> and <New>. This brings up a template file for modification. Start by filling out all of the header information. It is generally good practice to use have the Name field the same as the file name.

After the header information is supplied the pin declarations need to be made. For this design we will need 4 inputs and 8 outputs. A 16V8 in simple mode will accommodate this. Declare 4 input pins as the input bus and all of the I/O pins available as the output bus. Please note that you cannot name a signal **OUT** because it is a CUPL reserved word.

The next step is to define the **Field** statements for the signals. To do this look at the listing of the PLD file on the next page. Having the fields set up we can now define the **Table**. A general direct match is used to do this with the equating symbol (=>). CUPL also supports a repeat state that allows the user to quickly go through values without computing the value manually.

Figure 4-5. SQUARE.PLD file

```
Name  SQUARE;
Partno  XX;
Date  05/01/96;
Revision 01;
Designer Chip Willman;
Company Logical Devices Inc.;
Assembly None;
Location U1;
Device  G16V8;

/*************************************************************/
/* This Design Example is an example of a lookup table to */
/* produce the square of a number coming in.                 */
/*                                                           */
/*************************************************************/
/* Allowable Target Device Types:                            */
/*************************************************************/

/** Inputs **/
Pin [2..5] = [I0..2] ; /* Input bus line 4 bits */

/** Outputs **/
Pin [12..19] = [Ot0..7] ; /* Output bus line 8 bits */

/** Declarations and Intermediate Variable Definitions **/
Field input = [I3..0];
Field output = [Ot7..0];

/** Logic Equations **/
Table input=>output {
  'd'00 => 'd'000;
  'd'01 => 'd'001;
  'd'02 => 'd'004;
$REPEAT A = [3..15]
  'd'{A} => 'd'{A*A};
$REPEND
}
```

Included with this software are several other examples with useful demonstrations of CUPL syntax. The file EXAMPLES.TXT gives a description of most of the examples included in the package.

Section 5

CUPL Software Features

This chapter briefly describes several CUPL software packages supplied by Logical Devices. Please contact Logical Devices for more information. If you have questions about Atmel-CUPL or Atmel-WINCUPL please contact Atmel PLD applications.

5.1 CUPL - PALexpert

PALexpert contains the features mentioned in this package and supports 75 popular PAL, GAL and PROM architectures (approximately 1500 devices). This low cost CAE tool allows you to discover the benefits of designing with PLDs.

5.2 CUPL - PLDmaster

PLDmaster supports over 250 PAL, GAL, FPLA and PROM architectures which equates to over 3000 devices. With the more complex devices, it is possible for the designer to impliment larger designs into a single device

5.3 CUPL - Total Designer

Total Designer is the complete programmable logic design system including support most all industry devices including Complex PLDs and FPGAs. These include other competitor devices. Manufacturer specific place and route software and device fitters may not be included. In addition, partition software is provided for creating multiple PLD designs. Also SchemaQuik and ONCUPL are provided so that schematic entry designs can be created and translated into CUPL source files. All the programmable logic design software an engineer needs is in this package.

5.4 CUPL - Total Designer VHDL

5.4.1 OPTION for Total Designer

CUPL Total Designer VHDL transforms a VHDL design description into a Boolean design description and CUPL source design file. During processing the CUPL Total Designer VHDL program performs analysis, translation, and minimization. As an IEEE standard, VHDL descriptions are portable to other synthesis and simulation tools. VHDL allows the user to describe a design in any of three levels of abstraction: Structural (netlist like), data flow (like a PLD programming language), and behavioral (like a programming language).

5.5 ONCUPL

ONCUPL is a software tool that allows PLD designs to be done with schematic capture. A designer first draws the design with a schematic capture program. The schematic design is the converted into a netlist using a netlist extractor provided with the schematic capture program. ONCUPL then translate this netlist into a PLD file. This PLD file can be compiled with CUPL to produce any of the output files that CUPL is capable of producing.

ONCUPL is shipped with a library of symbols which can be used to implement the various macrocell architectures found in PLDs. Any design that will be processed by ONCUPL must be done using only ONCUPL symbols since these are specially structured for PLDs/ This usually means that existing TTL devices connected at the board level. Most often the design will have to be modified to some extent accommodate for the difference.

5.6 Liaison

Liaison - Logic Input Algorithm Interface for Symbolic Object Netlists - is a software system for converting schematic netlists to CUPL PLD files. As a more sophisticated translation tool than ONCUPL, LIAISON converts EDIF 2 0 0 netlists generated by schematic capture tools into its own internal format and the proceeds to translate this format into a PLD design file.

LIAISON automatically adjusts for differences in the target architecture by examining the design and the desired target to determine if there is a match. It then proceeds to implement the desired logic as closely as possible. LIAISON also adjusts for different register types, resorting to emulation where necessary.

LIAISON provides sophisticated and flexible symbols called COMPLEX symbols. These can be used to create customized symbols like a variety of TTL symbols.

5.7 PLPartition

PLPartition is a logic synthesis tool that works with CUPL for producing designs that span multiple PLDs. A design is created in CUPL using the device independent compilation feature. The .DOC output from this process is read into PLPartition where the designer then directs the software as to how to divide the logic and what devices to choose. The designer will set partitioning criteria such as how many solutions to produce, the maximum number of devices the solution can use, the percentage of product terms to use per output in each target device and several other optimization features. PLPartition then produces a list of solutions that match the users specified criteria. The designer can then choose one of these solutions and PLPartition will divide the logic in that manner.

PLPartiton has several interesting features. It can do automatic product term splitting. This means that if an equation cannot fit on a particular output Y then part of the equations placed on an unused output Y1 which is then fedback to the output Y where it is combined with the other part of the equation. This can allow more complex logic to fit into a device than may have been thought possible. The partitioning process can be optimized for minimum pin usage or minimum product term usage. This can be used to increase device usage efficiency by some simple analysis of the design.

PLPartition report file provides a mechanism for showing the designer how the logic was placed and the efficiency of the fit. This report file can be read back in by PLPartition in a future use of the same design so that pin placements can be retained if they were unchanged. This can potentially save board rework by retaining pinouts.

5.8 How to Contact Logical Devices

Logical Devices
1221 South Clarkston St.
Suite 200
Denver, CO 80202

e-mail: logdev@henge.com
Web: http://www.logical.com
Tel: (303)722-6868
Fax: (303)733-6868