

Chapter 4

ABEL-HDL Design

ABEL-HDL is a hardware description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables.

The DesignDirect ABEL-HDL compiler and supporting software functionally verify ABEL-HDL designs through simulation. The compilers then implement the designs in a programmable IC. ABEL-HDL designs can also be transferred to other design environments through standard format design transfer files.

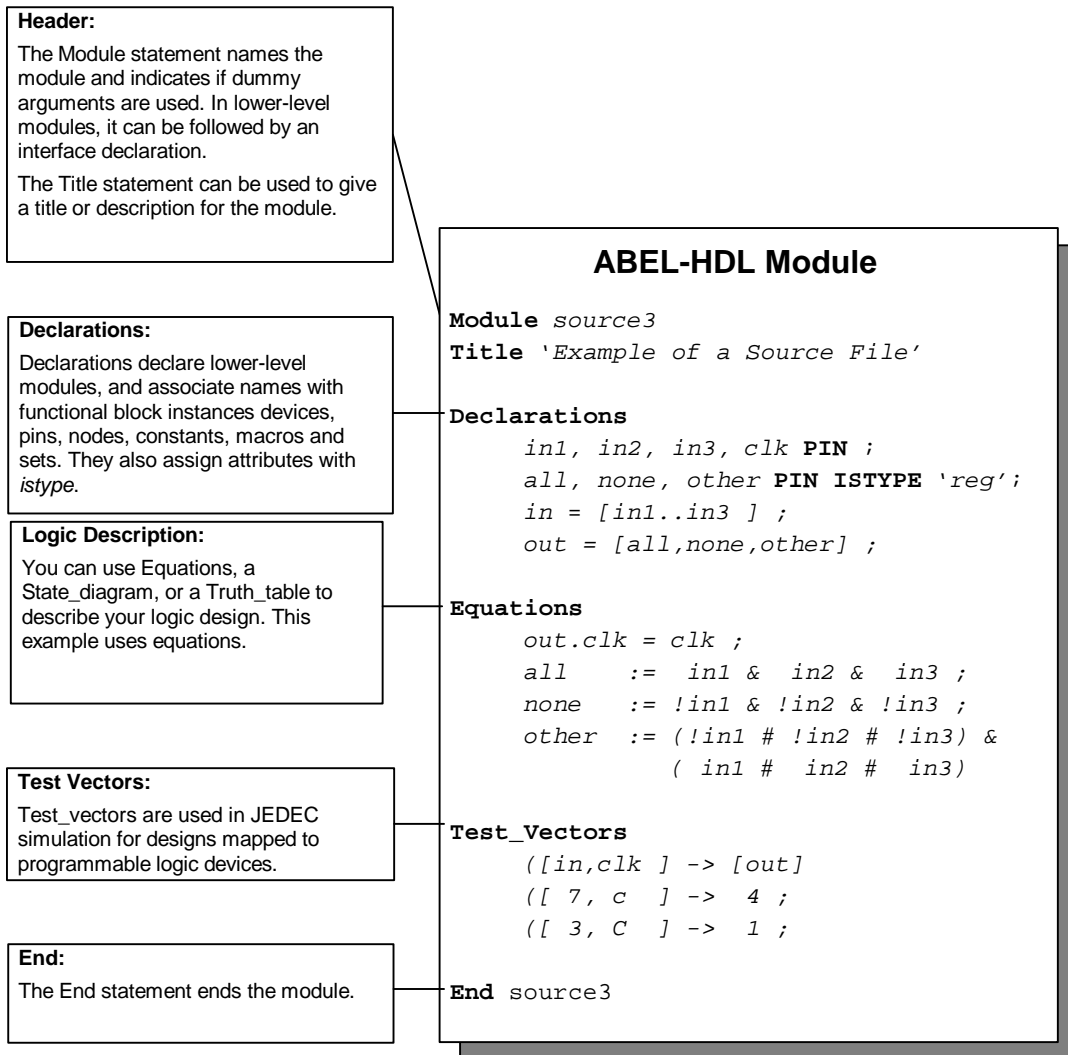
This chapter covers information on the following topics:

- A description of ABEL-HDL and an overview of its basic structure and syntax
- Design issues to consider
- The need to specify pin-to-pin attributes for declared signals
- Pin-to-pin vs. detailed description methods
- Using automatic polarity control
- Writing detailed equations that stimulate the inputs of alternate flip-flops
- Specifying feedback paths with "dot" extensions
- Using Don't Care optimization to reduce logic
- Using XOR attributes for architecture-independence
- Guidelines to help make state diagrams easy to read and maintain
- Describing a design using truth tables
- Using a template to create an ABEL-HDL source

What is ABEL-HDL?

ABEL-HDL is a hierarchical logic description language. ABEL-HDL design descriptions are contained in an ASCII text file in the ABEL Hardware Description Language (ABEL-HDL). For example, the ABEL-HDL code in Figure 1 describes a one-bit counter block.

Figure 1: Example ABEL-HDL code with descriptions



Language Structure

This section provides an overview of the basic structure and syntax of a design description in ABEL-HDL.

Note: A detailed description of the ABEL language is in the online Help system.

Basic Structure

ABEL-HDL source files can contain independent modules. Each module contains a complete logic description of a circuit or subcircuit. Any number of modules can be combined into one source file and processed at the same time

This section covers the basic elements that make up an ABEL-HDL source file module. A module can be divided into five sections, which are:

- Header
- Declarations
- Logic Description
- Test Vectors
- End

The elements of the source file are shown in the template in the figure following below. There are also directives that can be included in any of the middle three sections. The sections are presented briefly below, and then each element is introduced.

Structure Rules

The following rules apply to module structure:

- A module must contain only one header (composed of the MODULE statement and optional TITLE and OPTIONS statements).
- All other sections of a source file can be repeated in any order. However, declarations must immediately follow either the header or the DECLARATIONS keyword.
- No symbol (identifier) can be referenced before it is declared.

Header

The Header Section can consist of the following elements:

Module - The MODULE statement is required. It defines the beginning of the module and must be paired with an END statement. The MODULE statement also indicates whether any module arguments are used.

Title - The optional TITLE statement can be used to give a title or description for the module.

Declarations

The Declarations section specifies the names and attributes of signals used in the design; defines constants, macros, and states; declares lower-level modules and schematics; and optionally declares a device. Each module must have at least one Declarations section, and declarations affect only the module in which they are defined.

A Declarations section can consist of the following elements:

- Declarations Keyword
- Interface and Functional Block Declarations
- Signal Declarations (pin and node numbers optional)
- Constant Declarations
- Macro Declarations
- Library Declarations
- Device Declaration (one per module)

Logic Description

You can use one or more of the following elements to describe your design.

- Equations
- Truth Tables
- State Diagrams
- Fuses
- XOR Factors

In addition, dot extensions (like `ISTYPE` attributes in the Declarations section) enable you to more precisely describe the behavior of a circuit in a logic description that may be targeted to a variety of different devices.

Test Vectors Section

Test vectors are only used for Equation or JEDEC Simulation. A `Test_Vectors` section can consist of the following elements:

- Test Vectors
- Trace Statement
- Test Script

End Statement

The **END** statement ends (closes) the module, and is required.

Other Elements

Directives can be placed anywhere you need them.

Basic Syntax

The basic syntax of an ABEL-HDL source file includes the following:

- Supported ASCII characters
- Identifiers and keywords
- Constants
- Blocks
- Comments
- Numbers
- Strings
- Operators, expressions, and equations
 - Logical operators
 - Arithmetic operators
 - Relational operators
 - Assignment operators
 - Expressions
 - Equations
- Sets and set operation
- Arguments and argument substitution

Syntax Rules

Each line in an ABEL-HDL source file must conform to the following syntax rules and restrictions:

- A line can be up to 150 characters long.
- Lines are ended by a line feed (hex 0A), by a vertical tab (hex 0B), or by a form feed (hex 0C). Carriage returns in a line are ignored, so common end-of-line sequences, such as carriage return/line feed, are interpreted as line feeds. In most cases, you can end a line by pressing RETURN.
- Keywords, identifiers, and numbers must be separated by at least one space. Exceptions to this rule are lists of identifiers separated by commas, expressions where identifiers or numbers are separated by operators, or where parentheses provide the separation.
- Neither spaces nor periods can be imbedded in the middle of keywords, numbers, operators, or identifiers. Spaces can appear in strings, comments, blocks, and actual arguments. For example, if the keyword MODULE is entered as MOD ULE, it is interpreted as two identifiers, MOD and ULE. Similarly, if you enter 102 05 (instead of 10205), it is interpreted as two numbers, 102 and 5.
- Keywords can be uppercase, lowercase or mixed-case.
- Identifiers (user-supplied names and labels) can be uppercase, lowercase or mixed-case, but are case sensitive: the identifier, **output**, typed in all lowercase letters, is not the same as the identifier, **Output**.

Supported ASCII Characters

All uppercase and lowercase alphabetic characters and most other characters on common keyboards are supported. Valid characters are listed or shown below.

```
a - z (lowercase alphabet)
A - Z (uppercase alphabet)
0 - 9 (digits)
<space>
<tab>
! @ # $ % ^ & * ( ) -
_ = + [ ] { } ; : ' "
` \ | , < > . / ^ %
```

Identifiers

Identifiers are names that identify the following items:

- Devices
- Device pins or nodes
- Functional blocks
- sets
- Input or output signals
- Constants
- Macros
- Dummy arguments

All of these items are discussed later in this chapter. The rules and restrictions for identifiers are the same regardless of what the identifier describes

Identifier Rules

The rules governing identifiers are listed below:

- Identifiers can be up to 31 characters. Longer names are flagged as an error.
- Identifiers must begin with an alphabetic character or with an underscore.
- Other than the first character, identifiers can contain upper- and lowercase characters, digits, tildes (~), and underscores.
- You cannot use spaces in an identifier. Use underscores or uppercase letters to separate words.
- Except for Reserved Identifiers (Keywords), identifiers are case sensitive: uppercase letters and lowercase letters are not the same.
- You cannot use periods in an identifier, except with a supported dot extension.

Constants

You can use constant values in assignment statements, truth tables, and test vectors. You can assign a constant to an identifier, and then use the identifier to specify that value throughout a module. Constant values can be either numeric or one of the non-numeric special constant values. The special constant values are listed below in Table 1.

Table 1: Special constants

Constant	Description
.C.	Clocked input (low-high-low transition)
.D.	Clock down edge (high-low transition)
.F.	Floating input or output signal
.K.	Clocked input (high-low-high transition)
.P.	Register preload
.SVn.	n = 2 through 9. Drive the input to super voltage 2 through 9.
.U.	Clock up edge (low-high transition)
.X.	Don't care condition.
.Z.	Tristate value

When you use a special constant, it must be entered as shown in the above table. Without the periods, .C. is an identifier named C. You can enter special constants in upper- or lowercase.

Blocks

Blocks are sections of text enclosed in braces, { and }. Blocks are used in equations, state diagrams, macros, and directives. The text in a block can be on one line or it can span many lines. Some examples of blocks are shown below.

```
{ this is a block }
{ this is also a block, and it
 spans more than one line. }
```

```
{ A = B # C;
D = [0, 1] + [1, 0];
}
```

Blocks can be nested within other blocks, as shown below, where the block { D = A } is nested within a larger block:

```
{ A = B $ C;
  { D = A; }
  E = C; }
```

Blocks and nested blocks can be useful in macros and when used with directives.

If you need a brace as a character in a block, precede it with a backslash. For example, to specify a block containing the characters { }, write:

```
{ \{ \} }
```

Using Blocks in Logic Descriptions

Using blocks can simplify the description of output logic in equations and state diagrams, and allows more-complex functions than possible without blocks. Blocks can also improve the readability of your design.

Blocks are supported anywhere a single equation is supported. You can use blocks in simple equations, **When-Then-Else**, **If-Then-Else**, **Case**, and **With** statements

When you use equation blocks within a conditional expression (such as **If-Then**, **Case**, or **When-Then**), the logic functions are logically ANDed with the conditional expression.

Blocks in Equations

The following expressions, written without blocks, are limited by the inability to specify more than one output in a **When-Then** expression without using set notation:

Example 1: Without blocks

```
WHEN      (Mode == S_Data) THEN  Out_data := S_in;
ELSE WHEN (Mode == T_Data) THEN  Out_data := T_in;
WHEN      (Mode == S_Data) THEN  S_Valid  := 1;
ELSE WHEN (Mode == T_Data) THEN  T_Valid  := 1;
```

With blocks (delimited with braces { }), the syntax above can be simplified. The logic specified for Out_data is logically ANDed with the WHEN clause:

Example 2: With blocks

```
WHEN      (Mode == S_Data) THEN { Out_data := S_in;
                                S_Valid  := 1; }
ELSE WHEN (Mode == T_Data) THEN { Out_data := T_in;
                                T_Valid  := 1; }
```

Blocks in State Diagrams

Blocks also provide a simpler way to write state diagram output equations. For example, the following two state transition statements are equivalent.

Example 3: Without Blocks

```
IF (Hold) THEN State1 WITH o1 := o1.fb; o2 := o2.fb;
                        ENDWITH
ELSE State2;
```

Example 4: With Blocks

```
IF (Hold) THEN State1 WITH {o1 := o1.fb; o2 := o2.fb;}
ELSE State2;
```

Using Blocks for State Diagram Transitions

Blocks can be used to nest **If-Then** and **If-Then-Else** statements in state diagram descriptions, simplifying the description of complex transition logic.

Blocks for Transition Logic

Example 5: Without Blocks

```
IF (Hold & !Reset) THEN State1;
If (Hold & Error) THEN State2;
If (!Hold) THEN State3;
```

Example 6: With Blocks

```
If (Hold) THEN
{  IF (!Reset) THEN State1;
  IF (Error) THEN State2; }
ELSE State3;
```

Comments

Comments are another way to make a source file easy to understand. Comments explain what is not readily apparent from the source code itself, and do not affect the code. Comments cannot be imbedded within keywords.

➤ You can enter comments two ways:

- Begin with a double quotation mark (") and end with either another double quotation mark or the end of line.
- Begin with a double forward slash (//) and end with the end of the line. This is useful for commenting out lines of ABEL source that contain quote-delineated comments.

Examples of Comments

Examples of comments are shown in boldface below:

```
MODULE Basic_Logic; "gives the module a name
TITLE 'ABEL-HDL design example: simple gates'; "title
"declaration section"
IC4 device 'P10L8'; "declare IC4 to be a P10L8
IC5 "decoder PAL" device 'P10H8';
//IC5 "decoder PAL" device 'p10h8';
```

The information inside single quotation marks (apostrophes) is required and is part of the statement; it is not comments.

Numbers

All numeric operations in ABEL-HDL are performed to 128-bit accuracy, which means the supported numeric values are in the range 0 to 2^{128} minus 1. Numbers are represented in any of five forms. The four most common forms represent numbers in different bases. The fifth form uses alphabetic characters to represent a numeric value.

When one of the four bases other than the default base is chosen to represent a number, the base used is indicated by a symbol preceding the number. Table 2 lists the four bases supported by ABEL-HDL and their accompanying symbols. The base symbols can be upper- or lowercase

Table 2: Number representation in different bases

Base Name	Base	Symbol
Binary	2	[^] b
Octal	8	[^] o
Decimal	10	[^] d (default)
Hexadecimal	16	[^] h

When a number is specified and is not preceded by a base symbol, it is assumed to be in the default base numbering system. The normal default base is base 10. Therefore, numbers are represented in decimal form unless they are preceded by a symbol indicating that another base is to be used.

You can change the default number base. Examples of supported number specifications are shown below in Table 3.

Table 3: Supported base numbers

Specification	Decimal Value
T4 = 75	75
[^] h75	117
[^] b101	5
[^] o17	15
[^] h0F	15

Note: The carat ([^]) is a keyboard character. It is not part of a control-key sequence.

You can also specify numbers by strings of one or more alphabetic characters, using the numeric ASCII code of the letter as the value. For example, the character **a** is decimal 97 and hexadecimal 61 in ASCII coding. The decimal value 97 is used if **a** is specified as a number.

Sequences of alphabetic characters are first converted to their binary ASCII values and then concatenated to form numbers. Some examples are shown below in Table 4.

Table 4: Different number formats

Specification	Hex Value	Decimal Value
a	[^] h61	97
b	[^] h62	98
abc	[^] h616263	6382203

Strings

Strings are series of ASCII characters, including spaces, enclosed by apostrophes. Strings are used in the TITLE, MODULE, and OPTIONS statements, and in pin, node, and attribute declarations, as shown below:

```
'Hello'
' Text with a space in front'
```

```
' '
'The preceding line is an empty string'
'Punctuation? is allowed !!'
```

You can include a single quote in a string by preceding the quote with a backslash, (\).

```
'It\'s easy to use ABEL and DesignDirect'
```

You can include backslashes in a string by using two of them in succession.

```
'He\\she can use backslashes in a string'
```

Note: The grave accent (`) is also accepted as a string delimiter and can be used interchangeably with the apostrophe (').

Operators, Expressions, and Equations

Items such as constants and signal names can be brought together in expressions. Expressions combine, compare, or perform operations on the items they include to produce a single result. The operations to be performed (addition and logical AND are two examples) are indicated by operators within the expression.

You can use the set operator (..) in expressions and equations.

ABEL-HDL operators are divided into four basic types: logical, arithmetic, relational, and assignment. Each of these types are discussed separately below, followed by a description of how they are combined into expressions. Following the descriptions is a summary of all the operators and the rules governing them and an explanation of how equations use expressions

Logical Operators

Logical operators are used in expressions. ABEL-HDL incorporates the standard logical operators listed in Table 5 below. Logical operations are performed bit by bit.

Table 5: Logical Operators

Operator	Description
!	NOT: ones complement
&	AND
#	OR
\$	XOR: exclusive OR
!\$	XNOR: exclusive NOR

Arithmetic Operators

Arithmetic operators define arithmetic relationships between items in an expression. The shift operators are included in this class because each left shift of one bit is equivalent to multiplication by 2 and a right shift of one bit is the same as division by 2. Table 6 lists the arithmetic operators.

Table 6: Arithmetic Operators

Operator	Example	Description
-	-A	twos complement (negation)
-	A-B	subtraction
+	A+B	addition
<i>Not Supported for Sets:</i>		
*	A*B	multiplication
/	A/B	unsigned integer division
%	A%B	modulus: remainder from /
<<	A<<B	shift A left by B bits
>>	A>>B	shift A right by B bits

Note: A minus sign has a different significance, depending on its usage. When used with one operand, it indicates that the twos complement of the operand is to be formed. When the minus sign is found between two operands, the twos complements of the second operand are added to the first.

Division is unsigned integer division: the result of division is a positive integer. Use the modulus operator (%) to get the remainder of a division. The shift operators perform logical unsigned shifts. Zeros are shifted in from the left during right shifts and in from the right during left shifts.

Relational Operators

Relational operators compare two items in an expression. Expressions formed with relational operators produce a Boolean true or false value.

Table 7: Relational Operators

Operator	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

All relational operations are unsigned. For example, the expression `!0 > 4` is true since the complement of `!0` is 1111 (assuming 4 bits of data), which is 15 in unsigned binary, and 15 is greater than 4. In this example, a four-bit representation was assumed; in actual use, `!0`, the complement of 0, is 128 bits all set to 1.

Table 8: Some examples of relational operators in expressions

Expression	Value
<code>2 == 3</code>	False
<code>2 != 3</code>	True
<code>3 < 5</code>	True

-1 > 2 True
 False

The logical values true and false are represented by numbers. Logical true is -1 in twos complement, so all 128 bits are set to 1. Logical false is 0 in twos complement, so all 128 bits are set to 0. This means that an expression producing a true or false value (a relational expression) can be used anywhere a number or numeric expression could be used and -1 or 0 will be substituted in the expression depending on the logical result.

For example,

```
A = D $ (B == C);
```

means that:

- A equals the complement of D if B equals C
- A equals D if B does not equal C.

When using relational operators, always use parentheses to ensure the expression is evaluated in the order you expect. The logical operators & and # have a higher priority than the relational operators (see the priority table later in this chapter).

The following equation:

```
Select = [A15..A0] == ^hD000 # [A15..A0] == ^h1000;
```

needs parentheses to obtain the desired result:

```
Select = ([A15..A0] == ^hD000) # ([A15..A0] == ^h1000);
```

Without the parentheses, the equation would have the default grouping

```
Select = [A15..A0] == (^hD000 # [A15..A0]) == ^h1000;
```

which is not the intended equation.

Assignment Operators

Assignment operators are used in equations rather than in expressions. Equations assign the value of an expression to output signals.

There are four assignment operators (two combinational and two registered). Combinational or immediate assignment occurs, without any delay, as soon as the equation is evaluated. Registered assignment occurs at the next clock pulse from the clock associated with the output.

Table 9: Assignment Operators

Operator	Set	Description
=	ON (1)	Combinational or detailed assignment
:=	ON (1)	Implied registered assignment
?=	DC (X)	Combinational or detailed assignment
?:=	DC (X)	Implied registered assignment

Caution: The := and ?:= assignment operators are used only when writing pin-to-pin registered equations. Use the = and ?= assignment operators for registered equations using detailed dot extensions.

These assignment operators allow you to fully specify outputs in equations. For example, in the following truth table, the output F is fully specified:

```
TRUTH_TABLE ([A,B]->[F]);
           [1,1]-> 0 ; "off-set
           [1,0]-> 1 ; "on-set
           [0,1]-> 1 ; "on-set
```

The equivalent functionality can be expressed in equations:

```
@DCSET
F = A & !B # !A & B; "on-set
F ?= !A & !B;      "dc-set
```

Note: Specifying both the on-set and the don't-care set conditions enhances optimization.

Caution: With equations, @DCSET or ISTYPE 'dc' must be specified or the ?= equations are ignored.

Expressions

Expressions are combinations of identifiers and operators that produce one result when evaluated. Any logical, arithmetic, or relational operators may be used in expressions.

Expressions are evaluated according to the particular operators involved. Some operators take precedence over others, and their operation is performed first. Each operator has been assigned a priority that determines the order of evaluation. Priority 1 is the highest priority, and priority 4 is the lowest. Table 10 summarizes the logical, arithmetic, and relational operators, presented in groups according to their priority.

Table 10: Operator Priority

Priority	Operator	Description
1	-	negate
1	!	NOT
2	&	AND
2	<<	shift left
2	>>	shift right
2	*	multiply
2	/	unsigned division
2	%	modulus
3	++	add
3	-	subtract
3	#	OR
3	\$	XOR: exclusive OR
3	!\$	XNOR: exclusive NOR
4	==	equal

4	!=	not equal
4	<	less than
4	<=	less than or equal
4	>	greater than
4	>=	greater than or equal

Operations of the same priority are performed from left to right. Use parentheses to change the order in which operations are performed. The operation in the innermost set of parentheses is performed first. The following examples of supported expressions in Table 11 show how the order of operations and the use of parentheses affect the evaluated result.

Table 11: How parentheses affect evaluated work

Expression	Result	Comments
$2 * 3/2$	3	operators with same priority
$2 * 3 / 2$	3	spaces are OK
$2 * (3/2)$	2	fraction is truncated
$2 + 3 * 4$	14	multiply first
$(2 + 3) * 4$	20	add first
$2\#4\$2$	4	OR first
$2\#(4\$2)$	6	XOR first
$2 == ^hA$	0	
$14 == ^hE$	-1	

Equations

Equations assign the value of an expression to a signal or set of signals in a logic description. The identifier and expression must follow the rules for those elements.

Equations use the assignment operators =, ?= (combinational) and := ?:=, (registered) described above.

You can use the complement operator (!) to express negative logic. The complement operator precedes the signal name and implies that the expression on the right of the equation is to be complemented before it is assigned to the signal. Use of the complement operator on the left side of equations is provided as an option; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

Equation Blocks

Equation blocks let you specify more complex functions and improve the readability of your equations. An equation block is enclosed in braces { }, and is supported wherever a single equation is supported. When used within a conditional expression, such as **If-Then**, **Case**, or **When-Then**, the logic functions are logically ANDed with the conditional expression that is in effect.

Multiple Assignments to the Same Identifier

When an identifier appears on the left side of more than one equation, the expressions assigned to the identifier are first ORed together, and then the assignment is made. If the identifier on the left side of the equation is complemented, the complement is performed after all the expressions have been ORed.

Table 12:

Equations Found	Equivalent Equation
A = B; A = C;	A = B # C;
A = B; A = C & D;	A = B # (C & D);
A = !B; A = !C;	A = !B # !C;
!A = B; !A = C;	A = !(B # C);
!A = B; A = !C;	A = !C # !B;
!A = B; !A = C;	
A = !D; A = !E;	A = !D # !E # !(B # C);

Note: When the complement operator appears on the left side of multiple assignment equations, the right sides are ORed first, and then the complement is applied.

Sets

A set is a collection of signals and constants. Any operation applied to a set is applied to each element in the set. Sets simplify ABEL-HDL logic descriptions and test vectors by allowing groups of signals to be referenced with one name.

For example, you could collect the outputs (B0-B7) of an eight-bit multiplexer into a set named MULTOUT, and the three selection lines into a set named SELECT. You could then define the multiplexer in terms of MULTOUT and SELECT rather than individual input and output bits.

A set is represented by a list of constants and signals separated by commas or the range operator (..) and surrounded by brackets. The sets MULTOUT and SELECT would be defined as follows

```
MULTOUT = [ B0 , B1 , B2 , B3 , B4 , B5 , B6 , B7 ]
SELECT = [ S2 , S1 , S0 ]
```

The above sets could also be expressed by using the range operator; for example:

```
MULTOUT = [ B0 .. B7 ]
SELECT = [ S2 .. S0 ]
```

Identifiers used to delimit a range must have compatible names: they must begin with the same alphabetical prefix and have a numerical suffix. Range identifiers can also delimit a decremting range or a range which appears as one element of a larger set as shown below:


```
[A7..A0] "decrementing range
[Q1,Q2,..X.,A10..A7] "range within a larger set
```

The brackets are required to delimit the set. ABEL-HDL source file sets are not mathematical sets.

Set Indexing

Set indexing allows you to access elements within a set. The following example uses set indexing to assign four elements of a 16-bit set to a smaller set.

```
declarations
  Set1 = [f15..f0];
  Set2 = [q3..q0];
equations
  Set2 := Set1[7..4];
```

The numeric values used for defining a set index refer to the bit positions of the set, with 0 being the least significant (left-most) element in the set. So Set1[7..4] is Set1, values f8 to f11.

If you are indexing into a set to access a single element, then you can use the following syntax:

```
declarations
  out1 pin istype 'com';
  Set1 = [f15..f0];
equations
  out1 = Set1[4] == 1;
```

In this example, a comparator operator (==) was used to convert the single-element set (Set1[4]) into a bit value (equivalent to f4).

Set Operations

Most operators can be applied to sets, with the operation performed on each element of the set, sometimes individually and sometimes according to the rules of Boolean algebra.

Two-set Operations

For operations involving two or more sets, the sets must have the same number of elements. The expression “[a,b]+[c,d,e]” is not supported because the sets have different numbers of elements.

For example, the Boolean equation:

```
Chip_Sel = A15 & !A14 & A13;
```

represents an address decoder where A15, A14 and A13 are the three high-order bits of a 16-bit address. The decoder can easily be implemented with set operations. First, a constant set that holds the address lines is defined so the set can be referenced by name. This definition is done in the constant declaration section of a module.

The declaration is:

```
Addr = [A15,A14,A13];
```

which declares the constant set Addr. The equation

```
Chip_Sel = Addr == [1,0,1];
```

is functionally equivalent to

```
Chip_Sel = A15 & !A14 & A13;
```

If Addr is equal to [1,0,1], meaning that A15 = 1, A14 = 0 and A13 = 1, then Chip_Sel is set to true. The set equation could also have been written as

```
Chip_Sel = Addr == 5;
```

because 101 binary equals 5 decimal.

In the example above, a special set with the high-order bits of the 16-bit address was declared and used in the set operation. The full address could be used and the same function arrived at in other ways, as shown below:

Example 7:

```
" declare some constants in declaration section
Addr = [a15..a0];
X = .X.; "simplify notation for don't care constant
Chip_Sel = Addr == [1,0,1,X,X,X,X,X,X,X,X,X,X,X,X];
```

Example 8:

```
" declare some constants in declaration section
Addr = [a15..a0];
X = .X.;
Chip_Sel = (Addr >= ^HA000) & (Addr <= ^HBFFF);
```

Both solutions presented in these two examples are functionally equivalent to the original Boolean equation and to the first solution in which only the high order bits are specified as elements of the set:

```
(Addr = [a15, a14, a13])
```

Set Assignment and Comparison

Values and sets of values can be assigned and compared to a set. For example,

```
sigset = [1,1,0] & [0,1,1];
```

results in sigset being assigned the value, [0,1,0]. The set assignment

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

is the same as the two assignments

```
a = c & d;
b = c & d;
```

Numbers in any representation can be assigned or compared to a set. The preceding set equation could have been written as

```
sigset = 6 & 3;
```

When numbers are used for set assignment or comparison, the number is converted to its binary representation and the following rules apply:

- If the number of significant bits in the binary representation of a number is greater than the number of elements in a set, the bits are truncated on the left.
- If the number of significant bits in the binary representation of a number is less than the number of elements in a set, the number is padded on the left with leading zeroes.

Thus, the following two assignments are equivalent:

```
[a,b] = ^B101011;  "bits truncated to the left
[a,b] = ^B11;
```

And so are these two:

```
[d,c] = ^B01;
[d,c] = ^B1;  "compiler will add leading zero
```

Table 13: Supported Set Operations

Operator	Example	Description
=	A = 5	combinational assignment
:=	A := [1,0,1]	registered assignment
!	!A	NOT: ones complement
&	A & B	AND
#	A # B	OR
\$	A \$ B	XOR: exclusive OR
!\$	A!\$ B	XNOR: exclusive NOR
-	-A	negate
-	A - B	subtraction
++	A + B	addition
==	A == B	equal
!=	A != B	not equal
<	A < B	less than
<=	A <= B	less than or equal
>	A > B	greater than
>=	A >= B	greater than or equal

Set Evaluation

How an operator is performed with a set may depend on the types of arguments the operator uses. When a set is written [a , b , c , d], **a** is the MOST significant bit and **d** is the LEAST significant bit.

The result, when most operators are applied to a set, is another set. The result of the relational operators (==, !=, >, >=, <, <=) is a value: TRUE (all ones) or FALSE (all zeros), which is truncated or padded to as many bits as needed. The width of the result is determined by the context of the relational operator, not by the width of the arguments.

The different contexts of the AND (&) operator and the semantics of each usage are described below.

Table 14:

signal & signal a & b	This is the most straightforward use. The expression is TRUE if both signals are TRUE.
signal & number a & 4	The number is converted to binary and the least significant bit is used. The expression becomes a & 0, then is reduced to 0 (FALSE).
signal & set a & [x, y, z]	The signal is distributed over the elements of the set to become [a & x, a & y, a & z]
set & set [a, b] & [x, y]	The sets are ANDed bit-wise resulting in: [a & x, b & y]. An error is displayed if the set widths do not match.
set & number [a, b, c] & 5	The number is converted to binary and truncated or padded with zeros to match the width of the set. The sequence of transformations is [a, b, c] & [1, 0, 1] [a & 1, b & 0, c & 1] [a, 0, c]
number & number 9 & 5	The numbers are converted to binary, ANDed together, then truncated or padded.

Example Equations

```
select = [a15..a0] == ^H80FF
```

select (signal) is TRUE when the 16-bit address bus has the hex value 80FF. Relational operators always result in a single bit.

```
[sel1, sel0] = [a3..a0] > 2
```

The width of **sel** and **a** are different, so the 2 is expanded to four bits (of binary) to match the size of the **a** set. Both **sel1** and **sel2** are true when the value of the four **a** lines (taken as a binary number) is greater than 2.

The result of the comparison is a single-bit result which is distributed to both members of the set on the output side of the equation.

```
[out3..out0] = [in3..in0] & enab
```

If **enab** is TRUE, then the values on **in0** through **in3** are seen on the **out0** through **out3** outputs. If **enab** is FALSE, then the outputs are all FALSE.

Set Operation Rules

Set operations are applied according to Boolean algebra rules. Uppercase letters are set names, and lowercase letters are elements of a set. The letters *k* and *n* are subscripts to the elements and to the sets. A subscript following a set name (uppercase letter) indicates how many elements the set contains. So A_k indicates that set *A* contains *k* elements. a_{k-1} is the (*k*-1)th element of set *A*. a_1 is the first element of set *A*.

Table 15:

Expression	Is Evaluated As...
$!A_k$	[!a _k , !a _{k-1} , ..., !a ₁]
$-A_k$! $A_k + 1$

$A_k.OE$	$[a_k.OE, a_{k-1}.OE, \dots, a_1.OE]$
$A_k \& B_k$	$[a_k \& b_k, a_{k-1} \& b_{k-1}, \dots, a_1 \& b_1]$
$A_k \# B_k$	$[a_k \# b_k, a_{k-1} \# b_{k-1}, \dots, a_1 \# b_1]$
$A_k \$ B_k$	$[a_k \$ b_k, a_{k-1} \$ b_{k-1}, \dots, a_1 \$ b_1]$
$A_k !\$ B_k$	$[a_k !\$ b_k, a_{k-1} !\$ b_{k-1}, \dots, a_1 !\$ b_1]$
$A_k == B_k$	$(a_k == b_k) \& (a_{k-1} == b_{k-1}) \& \dots \& (a_1 == b_1)$
$A_k != B_k$	$(a_k != b_k) \# (a_{k-1} != b_{k-1}) \# \dots \# (a_1 != b_1)$
$A_k + B_k$	D_k where: d_n is evaluated as $a_n \$ b_n \$ c_{n-1}$ c_n is evaluated as $(a_n \$ b_n) \# (a_n \& c_{n-1}) \# (b_n \& c_{n-1})$ c_0 is evaluated as 0
$A_k - B_k$	$A_k + (-B_k)$
$A_k < B_k$	c_k where: c_n is evaluated as $(!a_n \& (b_n \# c_{n-1}) \# a_n \& b_n \& c_{n-1}) != 0$ c_0 is evaluated as 0

Limitations/ Restrictions on Sets

If you have a set assigned to a single value, the value will be padded with 0s and then applied to the set. For example,

$$[A1, A2, A3] = 1$$

is equivalent to

$$\begin{aligned} A1 &= 0 \\ A2 &= 0 \\ A3 &= 1 \end{aligned}$$

which may not be the intended result. If you want 1 assigned to each member of the set, you'd need binary 111 or decimal 7.

The results of using an operator depend on the sequence of evaluation. Without parentheses, operations are performed from left to right. Consider the following two equations. In the first, the constant 1 is converted to a set; in the second, the 1 is treated as a single bit.

Example 9: Constant 1 is converted to a set

The first operation is $[a, b] \& \mathbf{1}$, so 1 is converted to a set [0, 1].

$$\begin{aligned} [x1, y1] &= [a, b] \& 1 \& d \\ &= ([a, b] \& \quad 1 \quad) \& d \\ &= ([a, b] \& [0, 1]) \& d \\ &= ([a \& 0, b \& 1]) \& d \\ &= [\quad 0 \quad , \quad b \quad] \& d \\ &= [0 \& d, b \& d] \\ &= [0, b \& d] \end{aligned}$$

$$\begin{aligned} x1 &= 0 \\ y1 &= b \& d \end{aligned}$$

Example 10: The 1 is treated as a single bit

The first operation is $1 \& d$, so 1 is treated as a single bit.

```
[x2,y2] = 1 & d & [a, b]
= (1 & d) & [a, b]
= d & [a, b]
= [d & a, d & b]
x2 = a & d
y2 = b & d
```

If you are unsure about the interpretation of an equation, try the following:

- Fully parenthesize your equation. Errors can occur if you are not familiar with the precedence rules.
- Write out numbers as sets of 1s and 0s instead of as decimal numbers. If the width is not what you expected, you will get an error message.

Arguments and Argument Substitution

Variable values can be used in macros, modules, and directives. These values are called the arguments of the construct that uses them. In ABEL-HDL, a distinction must be made between two types of arguments: *actual* and *dummy*. Their definitions are given here.

Dummy argument: An identifier used to indicate where an actual argument is to be substituted in the macro, module, or directive.

Actual argument: The argument (value) used in the macro, directive, or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL-HDL.

Dummy arguments are specified in macro declarations and in the bodies of macros, modules, and directives. The dummy argument is preceded by a question mark in the places where an actual argument is to be substituted. The question mark distinguishes the dummy arguments from other ABEL-HDL identifiers occurring in the source file.

Take for example, the following macro declaration arguments:

```
OR_EM MACRO (a,b,c) { ?a # ?b # ?c };
```

This defines a macro named OR_EM that is the logical OR of three arguments. These arguments are represented in the definition of the macro by the dummy arguments, a, b, and c. In the body of the macro, which is surrounded by braces, the dummy arguments are preceded by question marks to indicate that an actual argument is substituted.

The equation

```
D = OR_EM (x,y,z&1);
```

invokes the OR_EM macro with the actual arguments, x, y, and z&1. This results in the equation:

```
D = x # y # z&1;
```

Arguments are substituted into the source file before checking syntax and logic, so if an actual argument contains unsupported syntax or logic, the compiler detects and reports the error only after the substitution.

Spaces in Arguments

Actual arguments are substituted exactly as they appear, so any spaces (blanks) in actual arguments are passed to the expression. In most cases, spaces do not affect the interpretation of the macro. The exception is in functions that compare character strings, such as @IFIDEN and IFNIDEN. For example, the macro

```
iden macro(a,b) {@ifiden(?a,?b)
  {@message 'they are the same';}};
```

compares the actual arguments and prints the message if they are identical. If you enter the macro with spaces in the actual arguments:

```
iden(Q1, Q1);
```

The value is false because the space is passed to the macro.

Argument Guidelines

- Dummy arguments are place-holders for actual arguments.
- A question mark preceding the dummy argument indicates that an actual argument is to be substituted.
- Actual arguments replace dummy arguments before the source file is checked for correctness.
- Spaces in actual arguments are retained.

ABEL-HDL Design Considerations

This section discusses issues you need to consider when you create a design with ABEL-HDL.

Pin-to-pin Language Features

ABEL-HDL is a device-independent language. You do not have to declare a device or assign pin numbers to your signals until you are ready to implement the design into a device. However, when you do not specify a device or pin numbers, you need to specify pin-to-pin attributes for declared signals.

Because the language is device-independent, the ABEL-HDL compiler does not have predetermined device attributes to imply signal attributes. If you do not specify signal attributes or other information (such as the dot extensions, which are described later), your design might not operate consistently if you later transfer it to a different target device.

Device-independence vs. Architecture-independence

The requirement for signal attributes does not mean that a complex design must always be specified with a particular device in mind. You may still have to understand the differences between, for example, a P22V10 PAL and a MACH211 device, but you do not have to specify a particular device when describing your design.

Attributes and dot extensions help you refine your design to work consistently when moving from one class of device architecture to another; for example from devices having inverted outputs to those with a particular kind of reset/preset circuitry. However, the more you refine your design, using these language features, the more restrictive your design becomes in terms of the number of device architectures for which it is appropriate.

Signal Attributes

Signal attributes remove ambiguities that occur when no specific device architecture is declared. If your design does not use device-related attributes (either implied by a `DEVICE` statement or expressed in an `ISTYPE` statement), it may not operate the same way when targeted to different device architectures.

Signal Dot Extensions

Signal dot extensions, like attributes, enable you to more precisely describe the behavior of a circuit that may be targeted to different architectures. Dot extensions remove the ambiguities in equations.

Pin-to-pin vs. Detailed Description Methods for Registered Designs

You can use ABEL-HDL assignment operators when you write high-level equations. The `=` operator specifies a combinational assignment, where the design is written with only the circuit's inputs and outputs in mind. The `:=` assignment operator specifies a registered assignment, where you must consider the internal circuit elements (such as output inverters, presets and resets) related to the memory elements (typically flip-flops). The semantics of these two assignment operators are discussed below.

Using `:=` for Pin-to-pin Descriptions

The `:=` implies that a memory element is associated with the output defined by the equation. For example, the equation;

```
Q1 := !Q1 # Preset;
```

implies that **Q1** will hold its current value until the memory element associated with that signal is clocked (or unlatched, depending on the register type). This equation is a pin-to-pin description of the output signal **Q1**. The equation describes the signal's behavior in terms of desired output pin values for various input conditions. Pin-to-pin descriptions are useful when describing a circuit that is completely architecture-independent.

Language elements that are useful for pin-to-pin descriptions are the `:=` assignment operator, and the `.CLK`, `.OE`, `.FB`, `.CLR`, `.ACLR`, `.SET`, `.ASET` and `.COM` dot extensions. These dot extensions help resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities

In the equation above ($Q1 := !Q1 \# \text{Preset};$), there is an ambiguous feedback condition. The signal **Q1** appears on the right side of the equation, but there is no indication of whether that feedback signal should originate at the register, come directly from the combinational logic that forms the input to the register, or come from the I/O pin associated with **Q1**. There is also no indication of what type of register should be used (although register synthesis algorithms could, theoretically, map this equation into virtually any register type). The equation could be more completely specified in the following manner.

```
Q1.CLK = Clock;           "Register clocked from input
Q1 := !Q1.FB # Preset;    "Reg. feedback normalized to pin value
```

This set of equations describes the circuit completely and specifies enough information that the circuit will operate identically in virtually any device in which you can fit it. The feedback path is specified to be from the register itself, and the **.CLK** equation specifies that the memory element is clocked, rather than latched.

Detailed Circuit Descriptions

In contrast to a pin-to-pin description, the same circuit can be specified in a detailed form of design description in the following manner:

```
Q1.CLK = Clock;           "Register clocked from input
Q1.D   = !Q1.Q # Preset;   "D-type f/f used for register
```

In this form of the design, specifying the D input to a D-type flip-flop and specifying feedback directly from the register restricts the device architectures in which the design can be implemented. Furthermore, the equations describe only the inputs to, and feedback from, the flip-flop and do not provide any information regarding the configuration of the actual output pin. This means the design will operate quite differently when implemented in a device with inverted outputs (a simple P16R4 PAL device, for example), versus a device with non-inverting outputs (such as an M4-32).

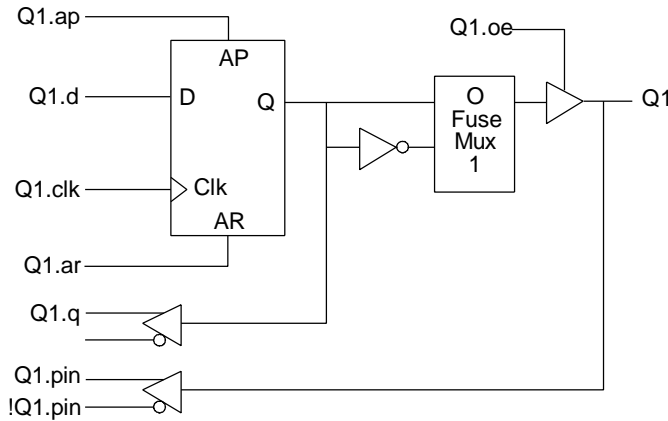
To maintain the correct pin behavior, using detailed equations, one additional language element is required: a 'buffer' attribute (or its complement, an 'invert' attribute). The 'buffer' attribute ensures that the final implementation in a device has no inversion between the specified D-type flip-flop and the output pin associated with **Q1**. For example, add the following to the Declarations section:

```
Q1 pin istype 'buffer';
```

Detailed Descriptions: Designing for Macrocells

One way to understand the difference between pin-to-pin and detailed description methods is to think of detailed descriptions as macrocell specifications. A macrocell is a block of circuitry normally (but not always) associated with a device's I/O pin. The following figure illustrates a typical macrocell associated with signal **Q1**.

Figure 2: Detailed Macrocell

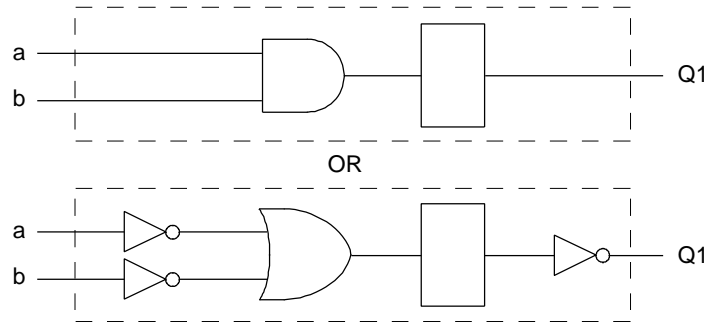


0665-3

Detailed descriptions are written for the various input ports of the macrocell (shown in the figure above with dot extension labels). Note that the macrocell features a configurable inversion between the Q output of the flip-flop and the output pin labeled **Q1**. If you use this inverter (or select a device that features a fixed inversion), the behavior you observe on the **Q1** output pin will be inverted from the logic applied to (or observed on) the various macrocell ports, including the feedback port **Q1.q**.

Pin-to-pin descriptions, on the other hand, allow you to describe your circuit in terms of the expected behavior on an actual output pin, regardless of the architecture of the underlying macrocell. The following figure illustrates the pin-to-pin concept.

Figure 3: Pin-to-pin Macrocell



1748-1

When pin-to-pin descriptions are written in ABEL-HDL, the generic macrocell \t "ABEL 26" shown above is synthesized from whatever type of macrocell actually exists in the target device.

Examples of Pin-to-Pin and Detailed Descriptions

Two equivalent module descriptions, one pin-to-pin and one detailed, are shown below for comparison:

Pin-to-Pin Module Description

```

module Q1_1
    Q1          pin      istype 'reg';
    Clock, Preset pin;

```

```

equations
    Q1.clk = Clock;
    Q1     := !Q1.fb # Preset;

test_vectors ([Clock,Preset] -> Q1)
    [ .c. , 1 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 0 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 1 ] -> 1;
    [ .c. , 1 ] -> 1;

end

```

Detailed Module Description

```

module Q1_2
    Q1          pin      istype 'reg_D,buffer';
    Clock,Preset pin;

equations
    Q1.CLK = Clock;
    Q1.D   = !Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
    [ .c. , 1 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 0 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 1 ] -> 1;
    [ .c. , 1 ] -> 1;

end

```

The first description can be targeted into virtually any device (if register synthesis and device fitting features are available), while the second description can be targeted only to devices featuring D-type flip-flops and non-inverting outputs.

To implement the second (detailed) module in a device with inverting outputs, the source file would need to be modified as shown in the following section.

Detailed Module with Inverted Outputs

```

module Q1_3
    Q1          pin      istype 'reg_D,invert';
    Clock,Preset pin;

equations
    Q1.CLK = Clock;
    !Q1.D  = Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
    [ .c. , 1 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 0 ] -> 1;
    [ .c. , 0 ] -> 0;

```

```

        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;
    end

```

In this version of the module, the existence of an inverter between the output of the D-type flip-flop and the output pin (specified with the 'invert' attribute) has necessitated a change in the equation for **Q1.D**.

As this example shows, device-independence and pin-to-pin description methods are preferable, since you can describe a circuit completely for any implementation. Using pin-to-pin descriptions and generalized dot extensions (such as **.FB**, **.CLK** and **.OE**) as much as possible allows you to implement your ABEL-HDL module into any one of a particular class of devices. (For example, any device that features enough flip-flops and appropriately configured I/O resources.) However, the need for particular types of device features (such as register preset or reset) might limit your ability to describe your design in a completely architecture-independent way.

If, for example, a built-in register preset feature is used in a simple design, the target architectures are limited. Consider this version of the design.

```

module Q1_51
    Q1          pin      istype 'reg,buffer';
    Clock,Preset pin;

    equations
        Q1.CLK = Clock;
        Q1.AP  = Preset;
        Q1     := !Q1.fb ;

    test_vectors ([Clock,Preset] -> Q1)
        [ .c. , 1 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 0 ] -> 1;
        [ .c. , 0 ] -> 0;
        [ .c. , 1 ] -> 1;
        [ .c. , 1 ] -> 1;
    end

```

The equation for **Q1** still uses the **:=** assignment operator and **.FB** for a pin-to-pin description of **Q1**'s behavior, but the use of **.AP** to describe the reset function requires consideration of different device architectures. The **.AP** extension, like the **.D** and **.Q** extensions, is associated with a flip-flop input, not with a device output pin. If the target device has inverted outputs, the design will not reset properly, so this ambiguous reset behavior is removed by using the 'buffer' attribute, which reduces the range of target devices to those with non-inverted outputs.

Using **.ASET** instead of **.AP** can solve this problem if the fitter being used supports the **.ASET** dot extension.

Versions 5 and 7 of the design above and below are unambiguous, but each is restricted to certain device classes.

```

module Q1_71
    Q1          pin      istype 'reg,invert';
    Clock,Preset pin;

```

```

equations
    Q1.CLK = Clock;
    Q1.AR  = Preset;
    Q1     := !Q1.fb ;

test_vectors ([Clock,Preset] -> Q1)
    [ .c. , 1 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 0 ] -> 1;
    [ .c. , 0 ] -> 0;
    [ .c. , 1 ] -> 1;
    [ .c. , 1 ] -> 1;

end

```

When to Use Detailed Descriptions

Although the pin-to-pin description is preferable, there will frequently be situations when you must use a more detailed description. If you are unsure about which method to use for various parts of your design, examine the design's requirements. If your design requires specific features of a device (such as register preset or unusual flip-flop configurations), detailed descriptions are probably necessary. If your design is a simple combinational function, or if it matches the generic macrocell in its requirements, you can probably use simple pin-to-pin descriptions.

Using := for Alternative Flip-flop Types

In ABEL-HDL you can specify a variety of flip-flop types using attributes such as `istype 'reg_D'` and `'reg_JK'`. However, these attributes do not enforce the use of a specific type of flip-flop when a device is selected, and they do not affect the meaning of the `:=` assignment operator.

You can think of the `:=` assignment operator as a memory operator. The type of register that most closely matches the `:=` assignment operator's behavior is the D-type flip-flop.

The primary use for attributes such as `istype 'reg_D'`, `'reg_JK'` and `'reg_SR'` is to control the generation of logic. Specifying one of the `'reg_'` attributes (for example, `istype 'reg_D'`) instructs the AHDL compiler to generate equations using The `.D` extension regardless of whether the design was written using `.D`, `:=` or some other method (for example, state diagrams).

Note: You also need to specify `istype 'invert'` or `'buffer'` when you use detailed syntax.

Using `:=` for flip-flop types other than D-type is only possible if register synthesis features are available to convert the generated equations into equations appropriate for the alternative flip-flop type specified. Since the use of register synthesis to convert D-type flip-flop stimulus into JK or SR-type stimulus usually results in inefficient circuitry, the use of `:=` for these flip-flop types is discouraged. Instead, you should use the `.J` and `.K` extensions (for JK-type flip-flops) or the `.S` and `.R` extensions (for SR-type flip-flops) and use a detailed description method (including `'invert'` or `'buffer'` attributes) to describe designs for these register types.

There is no provision in the language for directly writing pin-to-pin equations for registers other than D-type. State diagrams, however, may be used to describe pin-to-pin behavior for any register type.

Using Active-low Declarations

In ABEL-HDL you can write pin-to-pin design descriptions using implied active-low signals. Active-low signals are declared with a '!' operator, as shown below.

```
!Q1 pin  istype 'reg';
```

If a signal is declared active-low, it is automatically complemented when you use it in the subsequent design description. This complementing is performed for any use of the signal itself, including as an input, as an output, and in test vectors. Complementing is also performed if you use the **.fb** dot extension on an active-low signal.

The following three designs, for example, operate identically.

Example 11: Design 1 — Implied Pin-to-Pin Active-low

```
module act_low2
    !q0,!q1  pin istype 'reg';
    clock    pin;
    reset    pin;

    equations
        [q1,q0].clk = clock;
        [q1,q0] := ([q1,q0].FB + 1) & !reset;

    test_vectors ([clock,reset] -> [ q1, q0])
        [ .c. , 1 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 0 ] -> [ 1 , 0 ];
        [ .c. , 0 ] -> [ 1 , 1 ];
        [ .c. , 0 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 1 ] -> [ 0 , 0 ];

end
```

Example 12: Design 2 — Explicit Pin-to-Pin Active-low

```
module act_low1
    q0,q1    pin istype 'reg';
    clock    pin;
    reset    pin;

    equations
        [q1,q0].clk = clock;
        ![q1,q0] := (![q1,q0].FB + 1) & !reset;

    test_vectors ([clock,reset] -> [!q1,!q0])
        [ .c. , 1 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
```

```

        [ .c. , 0 ] -> [ 1 , 0 ];
        [ .c. , 0 ] -> [ 1 , 1 ];
        [ .c. , 0 ] -> [ 0 , 0 ];
        [ .c. , 0 ] -> [ 0 , 1 ];
        [ .c. , 1 ] -> [ 0 , 0 ];
end

```

Example 13: Design 3 — Explicit Detailed Active-low

```

module act_low3
    q0,q1    pin istype 'reg_d,buffer';
    clock    pin;
    reset    pin;

equations
    [q1,q0].clk = clock;
    ![q1,q0].D := (![q1,q0].Q + 1) & !reset;

test_vectors ([clock,reset] -> [!q1,!q0])
    [ .c. , 1 ] -> [ 0 , 0 ];
    [ .c. , 0 ] -> [ 0 , 1 ];
    [ .c. , 0 ] -> [ 1 , 0 ];
    [ .c. , 0 ] -> [ 1 , 1 ];
    [ .c. , 0 ] -> [ 0 , 0 ];
    [ .c. , 0 ] -> [ 0 , 1 ];
    [ .c. , 1 ] -> [ 0 , 0 ];

end

```

Both of these designs describe an up counter with active-low outputs. The first example inverts the signals explicitly (in the equations and in the test vector header), while the second example uses an active-low declaration to accomplish the same thing.

Polarity Control

Automatic polarity control is a powerful feature in ABEL-HDL where a logic function is converted for both non-inverting and inverting devices.

A single logic function may be expressed with many different equations. For example, all three equations below for F1 are equivalent.

- (1) $F1 = (A \& B);$
- (2) $!F1 = !(A \& B);$
- $!F1 = !A \# !B;$

In the example above, equation (3) uses two product terms, while equation (1) requires only one. This logic function will use fewer product terms in a non-inverting device such as the P10H8 than in an inverting device such as the P10L8. The logic function performed from input pins to output pins will be the same for both polarities.

Not all logic functions are best optimized to positive polarity. For example, the inverted form of F2, equation (3), uses fewer product terms than equation (2).

- (1) $F2 = (A \# B) \& (C \# D);$
- (2) $F2 = (A \& C) \# (A \& D) \# (B \& C) \# (B \& D);$

```
!F2 = (!A & !B) # (!C & !D);
```

Programmable polarity devices are popular because they can provide a mix of non-inverting and inverting outputs to achieve the best fit.

Polarity Control with Istype

In ABEL-HDL, you control the polarity of the design equations and target device (in the case of programmable polarity devices) in two ways:

- Using Istype *neg*, *pos*, and *dc*
- Using Istype *invert* and *buffer*

Using Istype *neg*, *pos*, and *dc* to Control Equation and Device Polarity

The 'neg', 'pos', and 'dc' attributes specify types of optimization for the polarity as follows:

neg	Istype neg optimizes the circuit for negative polarity. Unspecified logic in truth tables and state diagrams becomes a 0.
pos	Istype pos optimizes the circuit for positive polarity. Unspecified logic in truth tables and state diagrams becomes a 1.
dc	Istype dc uses polarity for best optimization. Unspecified logic in truth tables and state diagrams becomes don't care (X).

Using *invert* and *buffer* to Control Programmable Inversion

An optional method for specifying the desired state of a programmable polarity output is to use the 'invert' or 'buffer' attributes. These attributes ensure that an inverter gate either does or does not exist between the output of a flip-flop and its corresponding output pin. When you use the 'invert' and 'buffer' attributes, you can still use automatic polarity selection if the target architecture features programmable inverters located before the associated flip-flop.

These attributes are particularly useful for devices such as the P22V10, where the reset and preset behavior is affected by the programmable inverter.

Note: The 'invert' and 'buffer' attributes do not actually control device or equation polarity — they only enforce the existence or nonexistence of an inverter between a flip-flop and its output pin.

The polarity of devices that feature a fixed inverter in this location, and a programmable inverter before the register, cannot be specified using 'invert' and 'buffer'.

Flip-flop Equations

Pin-to-pin equations (using the `:=` assignment operator) are only supported for D flip-flops. ABEL-HDL does not support the `:=` assignment operator for T, SR or JK flip-flops and has no provision for specifying a particular output pin value for these types.

If you write an equation of the form:

```
Q1 := 1;
```

and the output, **Q1**, has been declared as a T-type flip-flop, the ABEL-HDL compiler will give a warning and convert the equation to

```
Q1.T = 1;
```

Because the T input to a T-type flip-flop does not directly correspond to the value you observed on the associated output pin, this equation will not result in the pin-to-pin behavior you want.

To produce specific pin-to-pin behavior for alternate flip-flop types, you must consider the behavior of the flip-flop you used and write detailed equations that stimulate the inputs of that flip-flop. A detailed equation to set and hold a T-type flip-flop is shown below.

```
Q1.T = !Q1.Q;
```

Feedback Considerations — Dot Extensions

The source of feedback is normally set by the architecture of the target device. If you don't specify a particular feedback path, the design may operate differently in different device types. Specifying feedback paths (with the `.FB`, `.Q` or `.PIN` dot extensions) eliminates architectural ambiguities. Specifying feedback paths also allows you to use architecture-independent simulation.

The following rules should be kept in mind when you are using feedback:

- **No Dot Extension** — A feedback signal with no dot extension (for example, `count := count+1;`) results in pin feedback if it exists in the target device. If there is no pin feedback, register feedback is used, with the value of the register contents complemented (normalized) if needed to match the value observed on the pin.
- **.FB Extension** — A signal specified with the `.FB` extension (for example, `count := count.fb+1;`) results in register feedback normalized to the pin value if a register feedback path exists. If no register feedback is available, pin feedback is used, and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.
- **.COM Extension** — A signal specified with the `.COM` extension (for example, `count := count.com+1;`) results in OR-array (pre-register) feedback, normalized to the pin value if an OR-array feedback path exists. If no OR-array feedback is available, pin feedback is used and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.

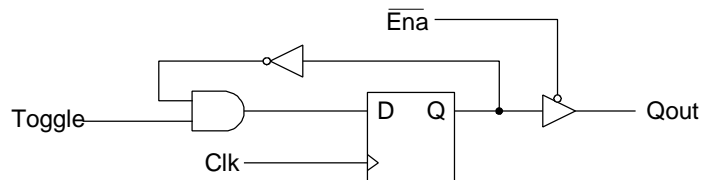
- **.PIN Extension** — If a signal is specified with the .PIN extension (for example, `count := count.pin+1;`), the pin feedback path will be used. If the specified device does not feature pin feedback, an error will be generated. Output enables frequently affect the operation of fed-back signals that originate at a pin.
- **.Q Extension** — Signals specified with the .Q extension (for example, `count.d = count.q + 1;`) will originate at the Q output of the associated flip-flop. The fed-back value may or may not correspond to the value you observe on the associated output pin. If an inverter is located between the Q output of the flip-flop and the output pin (as is the case in most registered PAL-type devices), the value of the fed-back signal will be the complement of the value you observe on the pin.
- **.D Extension** — Some devices, such as the MACH210, allow feedback of the input to the register. To select this feedback, use the .D extension. Some device kits also support .COM for this feedback; refer to your device kit manual for detailed information.

Dot Extensions and Architecture-Independence

To be architecture-independent, you must write your design in terms of its pin-to-pin behavior rather than in terms of specific device features (such as flip-flop configurations or output inversions).

For example, consider the simple circuit shown in the following figure. This circuit toggles high when the Toggle input is forced high, and low when the Toggle is low. The circuit also contains a three-state output enable that is controlled by the active-low Enable input.

Figure 4: Dot Extensions and Architecture-independence: Circuit 1



0770-1

The following simple ABEL-HDL design describes this simple one-bit synchronous circuit. The design description uses architecture-independent dot extensions to describe the circuit in terms of its behavior, as observed on the output pin of the target device. Since this design is architecture-independent, it will operate the same (disregarding initial power-up state), irrespective of the device type.

Example 14: Pin-to-pin One-bit Synchronous Circuit module `pin2pin`.

```

Clk      pin 1;
Toggle   pin 2;
Ena      pin 11;
Qout     pin 19 istype 'reg';

equations
  Qout    := !Qout.FB & Toggle;
  Qout.CLK = Clk;
  Qout.OE  = !Ena;

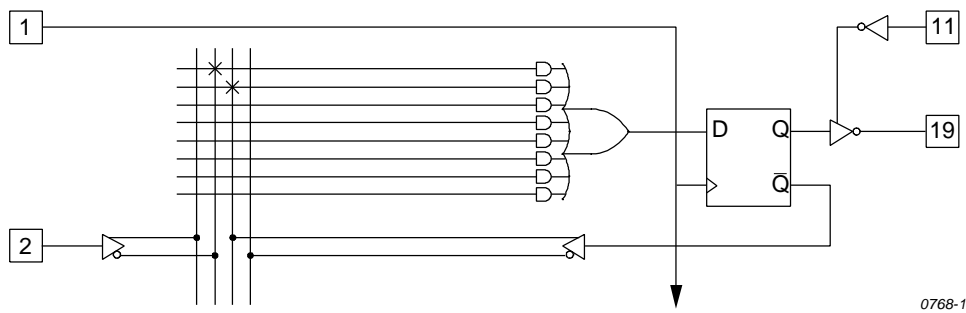
```

```

test_vectors([Clk,Ena,Toggle] -> [Qout])
    [.c., 0 , 0 ] -> 0;
    [.c., 0 , 1 ] -> 1;
    [.c., 0 , 1 ] -> 0;
    [.c., 0 , 1 ] -> 1;
    [.c., 0 , 1 ] -> 0;
    [.c., 1 , 1 ] -> .Z.;
    [ 0 , 0 , 1 ] -> 1;
    [.c., 1 , 1 ] -> .Z.;
    [ 0 , 0 , 1 ] -> 0;
end
    
```

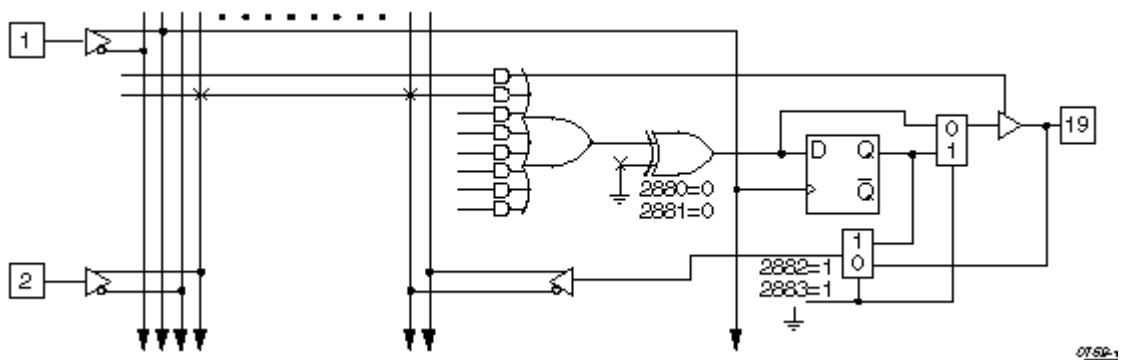
If you implement this circuit in a simple device (either by adding a device declaration statement or by specifying the P16R8 in the Fuseasm process), the result will be a circuit like the one illustrated in the following figure. Since the P16R8 features inverted outputs, the design equation is automatically modified to take the feedback from Q-bar instead of Q.

Figure 5: Dot Extensions and Architecture-independence, Circuit 2



If you implement this design in a device with a different architecture, the resulting circuit could be quite different. But, because this is a pin-to-pin design description, the circuit behavior is the same. The following figure illustrates the circuit that results when you specify a different architecture.

Figure 6: Dot Extensions and Architecture-independence, Circuit 3



Dot Extensions and Detail Design Descriptions

You may need to be more specific about how you implement a circuit in a target device. More complex device architectures have many configurable features, and you may want to use these features in a particular way. You may want a precise power-up and preset operation or, in some cases, you may need to control internal elements.

The circuit previously described (using architecture-independent dot extensions) could be described, for example, using detailed dot extensions in the following ABEL-HDL source file.

Example 15: Detailed One-bit Synchronous Circuit with Inverted Qout

```

module detail1
    d1      device  'P16R8';
    Clk     pin 1;
    Toggle  pin 2;
    Ena     pin 11;
    Qout    pin 19 istype 'reg_D';

    equations
        !Qout.D    = Qout.Q & Toggle;
        Qout.CLK   = Clk;
        Qout.OE    = !Ena;

    test_vectors([Clk,Ena,Toggle] -> [Qout])
        [.c., 0 , 0 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 0 , 1 ] -> 1;
        [.c., 0 , 1 ] -> 0;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 1;
        [.c., 1 , 1 ] -> .Z.;
        [ 0 , 0 , 1 ] -> 0;

end

```

This version of the design will result in exactly the same fuse pattern as indicated in the preceding figure for Circuit 2. As written, this design assumes the existence of an inverted output for the signal Qout. This is why the Qout.D and Qout.Q signals are reversed from the architecture-independent version of the design presented earlier.

Note: *The inversion operator applied to Qout.D does not correspond directly to the inversion found on each output of a P16R8. The equation for Qout.D actually refers to the D input of one of the P16R8's flip-flops; the output inversion found in a P16R8 is located after the register and is assumed rather than specified.*

Using Don't Care Optimization

Use Don't Care optimization to reduce the amount of logic required for an incompletely specified function. The @DCSET directive (used for logic description sections) and ISTYPE attribute 'dc' (used for signals) specify don't care values for unspecified logic.

Consider the following ABEL-HDL truth table:

```
truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
[ 0, 0, 0, 0]->[ 0, 0, 0, 1];
[ 0, 0, 0, 1]->[ 0, 0, 1, 1];
[ 0, 0, 1, 1]->[ 0, 1, 1, 1];
[ 0, 1, 1, 1]->[ 1, 1, 1, 1];
[ 1, 1, 1, 1]->[ 1, 1, 1, 0];
[ 1, 1, 1, 0]->[ 1, 1, 0, 0];
[ 1, 1, 0, 0]->[ 1, 0, 0, 0];
[ 1, 0, 0, 0]->[ 0, 0, 0, 0];
```

This truth table has four inputs, and therefore sixteen (2^4) possible input combinations. The function specified, however, only indicates eight significant input combinations. For each of the design outputs (f3 through f0) the truth table specifies whether the resulting value should be 1 or 0. For each output, then, each of the eight individual truth table entries can be either a member of a set of true functions called the on-set, or a set of false functions called the off-set.

Using output f3, for example, the eight input conditions can be listed as on-sets and off-sets as follows (maintaining the ordering of inputs as specified in the truth table above).

on-set of f3	off-set of f3
0 1 1 1	0 0 0 0
1 1 1 1	0 0 0 1
1 1 1 0	0 0 1 1
1 1 0 0	1 0 0 0

The remaining eight input conditions that do not appear in either the on-set or off-set are said to be members of the dc-set, as follows for f3.

```
dc-set of f3
0 0 1 0
0 1 0 0
0 1 0 1
0 1 1 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 1
```

Expressed as a Karnaugh map, the on-set, off-set and dc-set would appear as follows (with ones indicating the on-set, zeroes indicating the off-set, and dashes indicating the dc-set).

Figure 7: Truth table expressed as a Karnaugh map

		i1 i0			
		00	01	11	10
i3 i2	00	0	0	0	-
	01	-	-	1	-
	11	1	-	1	1
	10	0	-	-	-

1746-1

If the don't-care entries in the Karnaugh map are used for optimization, the function for f3 can be reduced to a single product term ($f3 = i2$) instead of the two ($f3 = i3 \& i2 \& !i0 \# i2 \& i1 \& i0$) otherwise required.

The ABEL-HDL compiler uses this level of optimization if the @DCSET directive or ISTYPE 'dc' is included in the ABEL-HDL source file, as shown below.

Example 16: Source File Showing Don't Care Optimization

```

module dc.
    i3,i2,i1,i0    pin;
    f3,f2,f1,f0    pin istype 'dc,com';

truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
    [ 0, 0, 0, 0]->[ 0, 0, 0, 1];
    [ 0, 0, 0, 1]->[ 0, 0, 1, 1];
    [ 0, 0, 1, 1]->[ 0, 1, 1, 1];
    [ 0, 1, 1, 1]->[ 1, 1, 1, 1];
    [ 1, 1, 1, 1]->[ 1, 1, 1, 0];
    [ 1, 1, 1, 0]->[ 1, 1, 0, 0];
    [ 1, 1, 0, 0]->[ 1, 0, 0, 0];
    [ 1, 0, 0, 0]->[ 0, 0, 0, 0];

end

```

This example results in a total of four single-literal product terms, one for each output. The same example (with no istype 'dc') results in a total of twelve product terms.

For truth tables, Don't Care optimization is almost always the best method. For state machines, however, you may not want undefined transition conditions to result in unknown states, or you may want to use a default state (determined by the type of flip-flops used for the state register) for state diagram simplification.

When using don't care optimization, be careful not to specify overlapping conditions (specifying both the on-set and dc-set for the same conditions) in your truth tables and state diagrams. Overlapping conditions result in an error message.

For state diagrams, you can perform additional optimization for design outputs if you specify the @dcstate attribute. If you enter @dcstate in the source file, all state diagram transition conditions are collected during state diagram processing. These transitions are then complemented and applied to the design outputs as don't-cares. You must use @dcstate in combination with @dcset or the 'dc' attribute.

Exclusive OR Equations

Designs written for exclusive-OR (XOR) devices should contain the 'xor' attribute for architecture-independence.

Optimizing XOR Devices

You can use XOR gates directly by writing equations that include XOR operators, or you can use implied XOR gates. XOR gates can minimize the total number of product terms required for an output or they can emulate alternate flip-flop types.

Using XOR Operators in Equations

If you want to write design equations that include XOR operators, you must either specify a device that features XOR gates in your ABEL-HDL source file, or specify the 'xor' attribute for all output signals that will be implemented with XOR gates. This preserves one top-level XOR operator for each design output. For example

```
module X1
    Q1      pin      istype 'com,xor';
    a,b,c   pin;
equations
    Q1 = a $ b & c;
end
```

Also, when writing equations for XOR PALs, you should use parentheses to group those parts of the equation that go on either side of the XOR. This is because the XOR operator (\$) and the OR operator (#) have the same priority in ABEL-HDL.

Using Implied XORs in Equations

High-level operators in equations often result in the generation of XOR operators. If you specify the 'XOR' attribute, these implied XORs are preserved, decreasing the number of product terms required. For example,

```
module X2
    q3,q2,q1,q0      pin istype 'reg,xor';
    clock             pin;
    count = [q3..q0];
equations
    count.clk = clock;
    count := count.FB + 1;
end
```

This design describes a simple four-bit counter. Since the addition operator results in XOR operators for the four outputs, the 'xor' attribute can reduce the amount of circuitry generated.

Note: The high-level operator that generates the XOR operators must be the top-level (lowest priority) operation in the equation. An equation such as `count := (count.FB + 1) & !reset` does not result in the preservation of top-level XOR operators, since the `&` operator is the top-level operator.

State Machines

A state machine is a digital device that traverses a predetermined sequence of states. State-machines are typically used for sequential control logic. In each state, the circuit stores its past history and uses that history to determine what to do next.

This section provides some guidelines to help you make state diagrams easy to read and maintain and to help you avoid problems. State machines often have many different states and complex state transitions that contribute to the most common problem, which is too many product terms being created for the chosen device. The topics discussed in the following subsections help you avoid this problem by reducing the number of required product terms.

The following subsections provide state machine considerations:

- Use Identifiers Rather Than Numbers for States
- Power-up Register States
- Unsatisfied Transition Conditions, D-Type Flip-Flops
- Unsatisfied Transition Conditions, Other Flip-Flops
- Number Adjacent States for a One-bit Change
- Use State Register Outputs to Identify States
- Use Symbolic State Descriptions

Use Identifiers Rather than Numbers for States

A state machine has different "states" that describe the outputs and transitions of the machine at any given point. Typically, each state is given a name, and the state machine is described in terms of transitions from one state to another. In a real device, such a state machine is implemented with registers that contain enough bits to assign a unique number to each state. The states are actually bit values in the register, and these bit values are used along with other signals to determine state transitions.

As you develop a state diagram, you need to label the various states and state transitions. If you label the states with identifiers that have been assigned constant values, rather than labeling the states directly with numbers, you can easily change the state transitions or register values associated with each state.

When you write a state diagram, you should first describe the state machine with names for the states, and then assign state register bit values to the state names.

For an example, see the following source file for a state machine named "sequence." (This state machine is also discussed in the design examples.) Identifiers (A, B, and C) specify the states. These identifiers are assigned a constant decimal value in the declaration section that identifies the bit values in the state register for each state. A, B, and C are only identifiers: they do not indicate the bit pattern of the state machine. Their declared values define the value of the state register (sreg) for each state. The declared values are 0, 1, and 2.

Example 17: Using identifiers for states

```

module Sequence
title 'State machine example';

    q1,q0                pin    14,15 istype 'reg';
    clock,enab,start,hold,reset pin  1,11,4,2,3;
    halt                 pin    17 istype 'reg';
    in_B,in_C           pin    12,13 istype 'com';
    sreg                 =      [q1,q0];

"State Values...
    A = 0;              B = 1;              C = 2;

equations
    [q1,q0,halt].clk = clock;
    [q1,q0,halt].oe  = !enab;
state_diagram sreg;
    State A:          " Hold in state A until start is active.
        in_B = 0;
        in_C = 0;
        IF (start & !reset) THEN B WITH halt := 0;
        ELSE A WITH halt := halt.fb;

    State B:          " Advance to state C unless reset is active
        in_B = 1;      " or hold is active. Turn on halt indicator
        in_C = 0;      " if reset.
        IF (reset) THEN A WITH halt := 1;
        ELSE IF (hold) THEN B WITH halt := 0;
        ELSE C WITH halt := 0;

    State C:          " Go back to A unless hold is active
        in_B = 0;      " Reset overrides hold.
        in_C = 1;
        IF (hold & !reset) THEN C WITH halt := 0;
        ELSE A WITH halt := 0;
test_vectors([clock,enab,start,reset,hold]-
>[sreg,halt,in_B,in_C])
    [ .p. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
    [ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

    [ .c. , 0 , 1 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
    [ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 1 , 0 ]->[ A , 1 , 0 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 ]->[ A , 1 , 0 , 0 ];

    [ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
    [ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];
end

```

Power-up Register States

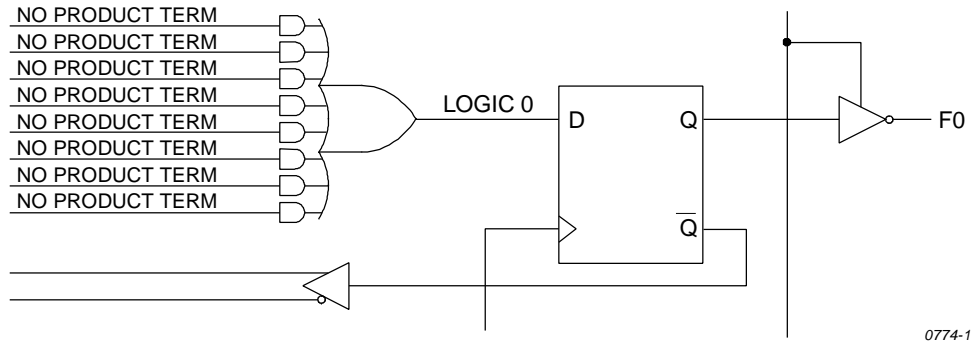
If a state machine has to have a specific starting state, you must define the register power-up state in the state diagram description or make sure your design goes to a known state at power-up. Otherwise, the next state is undefined.

Unsatisfied Transition Conditions

D-Type Flip-Flops

For each state described in a state diagram, you specify the transitions to the next state and the conditions that determine those transitions. For devices with D-type flip-flops, if none of the stated conditions are met, the state register, shown in the following figure, is cleared to all 0s on the next clock pulse. This action causes the state machine to go to the state that corresponds to the cleared state register. This can either cause problems or you can use it to your advantage, depending on your design.

Figure 8: D-type register with false inputs



You can use the clearing behavior of D-type flip-flops to eliminate some conditions in your state diagram, and some product terms in the converted design, by leaving the cleared-register state transition implicit. If no specified transition condition is met, the machine goes to the cleared-register state. This behavior can also cause problems if the cleared-register state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine goes to an undefined state and stays there.

To avoid problems caused by this clearing behavior, always have a state assigned to the cleared-register state. Or, if you don't assign a state to the cleared-register state, define every possible condition so some condition is always met for each state. You can also use the automatic transition to the cleared-register state by eliminating product terms and explicit definitions of transitions. You can also use the cleared-register state to satisfy illegal conditions

Other Flip-flops

If none of the state conditions is met in a state machine that employs JK, RS, and T-type flip-flops, the state machine does not advance to the next state, but holds its present state due to the low input to the register from the OR array output. In such a case, the state machine can get stuck in a state. You can use this holding behavior to your advantage in some designs.

If you want to prevent the hold, you can use the complement array provided in some devices (such as the F105) to detect a "no conditions met" situation and reset the state machine to a known state.

Precautions for Using Don't Care Optimization

When you use don't care optimization, you need to avoid certain design practices. The most common design technique that conflicts with this optimization is mixing equations and state diagrams to describe default transitions. For example, consider the design shown below.

Example 18: State machine description with conflicting logic

```

module TRAFFIC
title 'Traffic Signal Controller'
  Clk,SenA,SenB  pin  1, 8, 7;
  PR             pin  16;           "Preset control
  GA,YA,RA      pin  15..13;
  GB,YB,RB      pin  11..9;

  "Node numbers are not required if fitter is used
  S3..S0        node 31..34 istype 'reg_sr,buffer';
  COMP          node 43;

  H,L,Ck,X      = 1, 0, .C., .X.;
  Count         = [S3..S0];

  "Define Set and Reset inputs to traffic light flip-flops
  GreenA = [GA.S,GA.R];
  YellowA = [YA.S,YA.R];
  RedA = [RA.S,RA.R];
  GreenB = [GB.S,GB.R];
  YellowB = [YB.S,YB.R];
  RedB = [RB.S,RB.R];
  On = [ 1 , 0 ];
  Off = [ 0 , 1 ];

  " test_vectors edited
  equations
    [GB,YB,RB].AP = PR;
    [GA,YA,RA].AP = PR;
    [GB,YB,RB].CLK = Clk;
    [GA,YA,RA].CLK = Clk;
    [S3..S0].AP = PR;
    [S3..S0].CLK = Clk;

  "Use Complement Array to initialize or restart
    [S3..S0].R = (!COMP & [1,1,1,1]);
    [GreenA, YellowA, RedA] = (!COMP & [On ,Off,Off]);
    [GreenB, YellowB, RedB] = (!COMP & [Off,Off,On ]);

  state_diagram Count
    State 0:      if ( SenA & !SenB ) then 0 with COMP = 1;
                  if (!SenA & SenB ) then 4 with COMP = 1;

```

```

                                if ( SenA == SenB ) then 1 with COMP = 1;
State 1:                        goto 2 with COMP = 1;
State 2:                        goto 3 with COMP = 1;
State 3:                        goto 4 with COMP = 1;
State 4:                        GreenA = Off;
                                YellowA = On ;
                                goto 5 with COMP = 1;
State 5:                        YellowA = Off;
                                RedA    = On ;
                                RedB    = Off;
                                GreenB  = On ;
                                goto 8 with COMP = 1;
State 8:                        if (!SenA & SenB ) then 8 with COMP = 1;
                                if ( SenA & !SenB ) then 12 with COMP = 1;
                                if ( SenA == SenB ) then 9 with COMP = 1;
State 9:                        goto 10 with COMP = 1;
State 10:                       goto 11 with COMP = 1;
State 11:                       goto 12 with COMP = 1;
State 12:                       GreenB = Off;
                                YellowB = On ;
                                goto 13 with COMP = 1;
State 13:                       YellowB = Off;
                                RedB    = On ;
                                RedA    = Off;
                                GreenA  = On ;
                                goto 0 with COMP = 1;

end

```

This design uses the complement array feature of the Signetics FPLA devices to perform an unconditional jump to state [0,0,0,0]. If you use the **@DCSET** directive, the equation that specifies this transition

$$[S3,S2,S1,S0].R = (!COMP \& [1,1,1,1]);$$

will conflict with the dc-set generated by the state diagram for S3.R, S2.R, S1.R, and S0.R. If equations are defined for state bits, the **@DCSET** directive is incompatible. This conflict would result in an error and failure when the logic for this design is optimized.

To correct the problem, you must remove the **@DCSET** directive so the implied dc-set equations are folded into the off-set for the resulting logic function. Another option is to rewrite the module as shown below.

Example 19: @DCSET-compatible state machine description

```

module TRAFFIC1
title 'Traffic Signal Controller'

    Clk,SenA,SenB    pin    1, 8, 7;
    PR              pin    16;          "Preset control
    GA,YA,RA       pin    15..13;
    GB,YB,RB       pin    11..9;

    S3..S0         node 31..34 istype 'reg_sr,buffer';

```

```

H,L,Ck,X      = 1, 0, .C., .X.;
Count         = [S3..S0];

"Define Set and Reset inputs to traffic light flip flops
GreenA = [GA.S,GA.R];
YellowA = [YA.S,YA.R];
RedA    = [RA.S,RA.R];
GreenB  = [GB.S,GB.R];
YellowB = [YB.S,YB.R];
RedB    = [RB.S,RB.R];
On      = [ 1 , 0 ];
Off     = [ 0 , 1 ];

" test_vectors edited
equations
  [GB,YB,RB].AP = PR;
  [GA,YA,RA].AP = PR;
  [GB,YB,RB].CLK = Clk;
  [GA,YA,RA].CLK = Clk;
  [S3..S0].AP = PR;
  [S3..S0].CLK = Clk;

@DCSET
state_diagram Count
  State 0:      if ( SenA & !SenB ) then 0;
                if (!SenA & SenB ) then 4;
                if ( SenA == SenB ) then 1;

  State 1:      goto 2;
  State 2:      goto 3;
  State 3:      goto 4;

  State 4:      GreenA = Off;
                YellowA = On ;
                goto 5;

  State 5:      YellowA = Off;
                RedA    = On ;
                RedB    = Off;
                GreenB  = On ;
                goto 8;

  State 6:      goto 0;
  State 7:      goto 0;

  State 8:      if (!SenA & SenB ) then 8;
                if ( SenA & !SenB ) then 12;
                if ( SenA == SenB ) then 9;

  State 9:      goto 10;
  State 10:     goto 11;
  State 11:     goto 12;
  State 12:     GreenB = Off;
                YellowB = On ;
                goto 13;

  State 13:     YellowB = Off;
                RedB    = On ;
                RedA    = Off;
                GreenA  = On ;
                goto 0;

```

```

State 14:      goto 0;
State 15:      "Power up and preset state
                RedA   = Off;
                YellowA = Off;
                GreenA  = On ;
                RedB    = On ;
                YellowB = Off;
                GreenB  = Off;
                goto 0;

end

```

Number Adjacent States for One-bit Change

You can reduce the number of product terms produced by a state diagram by carefully choosing state register bit values. Your state machine should be described with symbolic names for the states, as described above. Then, if you assign the numeric constants to these names so the state register bits change by only one bit at a time as the state machine goes from state to state, you will reduce the number of product terms required to describe the state transitions.

As an example, take the states A, B, C, and D, which go from one state to the other in alphabetical order. The simplest choice of bit values for the state register is a numeric sequence, but this is not the most efficient method. To see why, examine the following bit value assignments. The preferred bit values cause a one-bit change as the machine moves from state B to C, whereas the simple bit values cause a change in both bit values for the same transition. The preferred bit values produce fewer product terms.

Table 16: Simple vs. preferred bit values

State	Simple Bit Values	Preferred Bit Values
A	00	00
B	01	01
C	10	11
D	11	10

If one of your state register bits uses too many product terms, try reorganizing the bit values so that state register bit changes in value as few times as possible as the state machine moves from state to state.

Obviously, the choice of optimum bit values for specific states can require some tradeoffs; you may have to optimize for one bit and, in the process, increase the value changes for another. The object should be to eliminate as many product terms as necessary to fit the design into the device.

Use State Register Outputs to Identify States

Sometimes it is necessary to identify specific states of a state machine and signal an output that the machine is in one of these states. Fewer equations and outputs are needed if you organize the state register bit values so one bit in the state register determines if the machine is in a state of interest. Take, for example, the following sequence of states in which identification of the C_n states is required.

Table 17: State register bit values

State Name	Q3	Q2	Q1
A	0	0	0
B	0	0	1
C1	1	0	1
C2	1	1	1
C3	1	1	0
D	0	1	0

This choice of state register bit values allows you to use Q3 as a flag to indicate when the machine is in any of the C_n states. When Q3 is high, the machine is in one of the C_n states. Q3 can be assigned directly to an output pin on the device. Notice also that these bit values change by only one bit as the machine cycles through the states, as is recommended in the section above.

Using Symbolic State Descriptions

Symbolic state descriptions describe a state machine without having to specify actual state values. A symbolic state description is shown below.

Example 20: Symbolic state description

```

module SM
  a,b,clock      pin;           " inputs
  a_reset,s_reset pin;         " reset inputs
  x,y            pin istance 'com'; " simple outputs

  sreg1         state_register;
  S0..S3       state;

equations
  sreg1.clk = clock;

state_diagram sreg1
  state S0:
    goto S1 with {x = a & b;
                  y = 0; }
  state S1: if (a & b)
    then S2 with {x = 0;
                  y = 1; }
  state S2: x = a & b;
    y = 1;
    if (a) then S1 else S2;
  state S3:
    goto S0 with {x = 1;
                  y = 0; }

  async_reset S0: a_reset;
  sync_reset S0: s_reset;
end

```

Symbolic state descriptions use the same syntax as non-symbolic state descriptions; the only difference is the addition of the **State_register** and **State** declarations, and the addition of symbolic synchronous and asynchronous reset statements.

Symbolic Reset Statements

In symbolic state descriptions, the **Sync_Reset** and **Async_Reset** statements specify synchronous or asynchronous state machine reset logic. For example, to specify that a state machine must asynchronously reset to state Start when the Reset input is true, you write:

```
ASYNC_RESET Start : (Reset) ;
```

Symbolic Test Vectors

You can also write test vectors to refer to symbolic state values by entering the symbolic state register name in the test vector header (in the output sections), and the symbolic state names in the test vectors as output values.

ABEL-HDL and Truth Tables

Truth Tables in ABEL-HDL represent a very easy and straightforward description method, well suited in a number of situations involving combinational logic.

The principle of the Truth Table is to build an exhaustive list of the input combinations (referred to as the ON-set) for which the outputs become active.

The following list summarizes design considerations for Truth Tables. Following the list are more detailed examples.

- The **OFF-set** lines in a Truth Table are necessary when more than one output is assigned in the Truth Table. In this case, not all Outputs are fired under the same conditions, and therefore OFF-set conditions do exist.
- OFF-set lines are ignored because they represent the default situation, unless the output variable is declared **dc**. In this case, a third set is built, the DC-set and the Output inside it is assigned proper values to achieve the best logic reduction possible.
- If output type **dc** (or **@dcset**) is not used and multiple outputs are specified in a Truth table, consider the outputs one by one and ignore the lines where the selected output is not set.
- Don't Cares (.X.) used on the right side of a Truth Table have no optimization effect.
- When dealing with multiple outputs of different kind, avoid general settings like **@DCSET** which will affect all your outputs. Use **istype '.....DC'** on outputs for which this reduction may apply.
- Beware of Outputs for which the ON-set might be empty.
- As a general guideline, it is important not to rely on first impression or simple intuition to understand Truth tables. The way they are understood by the compiler is the only possible interpretation. This means that Truth Tables should be presented in a clear and understandable format, should avoid side effects, and should be properly documented (commented).

Basic Syntax - Simple Examples

In the following example, the lines commented as L1 and L2 are the **ON-set**. Lines **L3** and **L4** are **ignored** because Out is type *default* (meaning '0' for unspecified combinations). The resulting equation does confirm this.

```

MODULE DEMO1
TITLE 'Module 1'
" Inputs
  A, B, C pin;
"Output
  Out pin istype 'com';
Truth_Table
  ([A,  B,  C]    -> Out )
  [0,  1,  0]    -> 1;  // L1
  [1,  1,  1]    -> 1;  // L2
  [0,  0,  1]    -> 0;  // L3
  [1,  0,  0]    -> 0;  // L4
END
// Resulting Reduced Equation :
// Out = (!A & B & !C) # (A & B & C);

```

Module 2 (below) differs from Module 1 (above) because Out is now type 'COM, DC'. (optimizable don't care)

In this case, the lines commented as L1 and L2 are the ON-set, L3 and L4 are the *OFF-set* and other combinations become *don't care* (DC-set) meaning 0 or 1 to produce the best logic reduction. As a result in this example, the equation is very simple.

@DCSET instruction would have produced the same result as to declare Out of type **dc**. But @DCSET must be used with care when multiple outputs are defined : they all become dc.

```

MODULE DEMO1
TITLE 'Module 2'
" Inputs
  A, B, C pin;
"Output
  Out pin istype 'com, dc';
Truth_Table
  ([A,  B,  C]    -> Out )
  [0,  1,  0]    -> 1;  // L1
  [1,  1,  1]    -> 1;  // L2
  [0,  0,  1]    -> 0;  // L3
  [1,  0,  0]    -> 0;  // L4
END
// Resulting Reduced Equation :
// Out = (B);

```

Influence of Signal Polarity

We'll see now with Module 3 how the polarity of the signal may influence the truth table :

In this example, **Out1** and **Out2** are strictly equivalent. For !Out1, note that the ON-set is the 0 values. *The third line L3 is ignored.*

```

MODULE DEMO2
TITLE 'Module 3'
" Inputs
  A, B, C  pin;
"Output
  Out1  pin  istype 'com, neg';
  Out2  pin  istype 'com, neg';
  Out3  pin  istype 'com, neg'; // BEWARE

Truth_Table
  ([A,  B,  C]    -> [!Out1, Out2, Out3] )
  [0,  0,  1]    -> [ 0,    1,    0 ];//L1
  [0,  1,  1]    -> [ 0,    1,    0 ];//L2
  [1,  1,  0]    -> [ 1,    0,    1 ];//L3
END
// Resulting Equations :
//      !Out1 = !Out2 = (A # !C);
// or:   Out1 = Out2 = (!A & C);
// BUT:  Out3 = (A & B & !C); <<what you wanted ?

```

For active-low outputs, one must be careful to specify **1** for the active state if the Output appears without the exclamation point (!).

0 must be used when !output is defined in the table header.

We recommend the style used for Out1.

For Out3, line used is L3, L1 and L2 are ignored.

Using .X. in Truth Tables Conditions

Don't Care used on the left side in Truth tables have no optimization purpose; they only serve as a shortcut to write several conditions in one single line.

Be careful when using .X. in conditions. This can lead to overlapping conditions which look not consistent (see Module 4 below). *Due to the way the compiler work, this type of inconsistency is not checked nor reported.* In fact, only the ON-set condition is taken into account, the OFF-set condition is ignored. The following example illustrates this:

```

MODULE DEMO3
TITLE 'Module 4'
"Inputs
  A, B, C      pin;
"Output
  Out  pin  istype 'com';
"Equivalence
  X = .X.;
Truth_Table
  ([A,  B,  C]    -> Out )
  [0,  0,  1]    -> 0; //L1 ignored in fact
  [0,  1,  0]    -> 1; //L2
  [1,  X,  X]    -> 1; //L3
  [0,  0,  1]    -> 1; //L4 incompatible

```

```

    [1, 1, 0] -> 0; //L5 incompatible
END
// Result : Out = A # (B & !C) # (!B & C)

```

L1 is in fact ignored. Out is active high, therefore only line L4 is taken into account.

Likewise, L5 intersects L3, but is ignored since it is not in the ON-set for Out.

Globally, only L2, L3 and L4 are taken into account, as we can check in the resulting equation, without any error reported.

Using .X. on the right side

The syntax allows to use .X. as a target value for an output. In this case, the condition is simply **ignored**.

Note: This is *not* the method to specify optimizable don't care states. See example 2 for such an example.

Module 5 shows that-> .X. states **are not optimized** if DC type or @DCSET are not used.

These lines are ALWAYS ignored.

```

MODULE DEMO6
TITLE 'Module 5'
"Inputs
  A, B, C      pin;
"Output
  Out  pin istype 'com';
"Equivalence
  X = .X.;
Truth_Table
  ( [A,  B,  C]      -> Out )
    [0,  0,  0]      -> 0;
    [0,  0,  1]      -> X;
    [0,  1,  0]      -> 1;
    [0,  1,  1]      -> X;
    [1,  X,  X]      => X;
END
// As is : Out = (!A & B & !C);
// With istype 'com,DC' : Out = (B);

```

They are in fact of no use, except maybe as a way to document that output does not matter.

Special case: Empty ON-set

There is a special case which is unlikely to happen, but may sometimes occurs. Consider this example:

```

MODULE DEMO5
TITLE 'Module 6'
"Inputs

```

```

    A, B, C pin;
"Output
    Out pin istype 'com, pos';
Truth_Table
    ( [A, B, C] -> Out )
    [0, 0, 1] -> 0;
    [0, 1, 0] -> 0;
    [1, 0, 0] -> 0;
// [0, 0, 0] -> 1;//changes everything!
END
// Without the last line L4 :
// !Out=(A & !B & !C)# (!A & B & !C)# (!A & !B & C);
// WITH L4 : Out = (!A & !B & !C);

```

What we obtain is slightly unexpected. This table should produce **Out=0**; as the result. (We enumerated only OFF conditions, and the polarity is POS (or default), so unlisted cases should **also** turn into zeroes.)

One reason to build such a table could be when multiple outputs are defined, and when Out needs to be shut off for whatever reason.

In the absence of the line L4, the **result is not intuitive**. The output is **0 only** for the listed cases (L1, L2, L3), and is **1 for all other cases**, even if dc or pos is used.

When line L4 is restored, then the output equation becomes $Out = (!A \& !B \& !C)$; because we fall in the general situation where the ON-set is not empty.

Registered Logic in Truth tables

Truth Tables can specify registered outputs. In this case, the assignment become `:=` (instead of `->`).

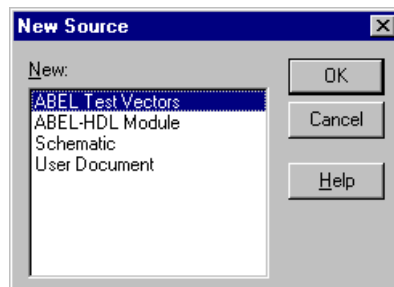
Using a Template to Create an ABEL-HDL Source

This section describes how to enter an ABEL-HDL design description using a template to create a single ABEL-HDL source.

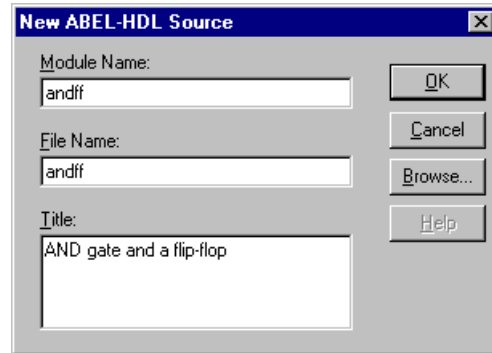
Creating a Template

➤ **To create a template for an ABEL-HDL source file:**

1 In the Project Manager, choose **Source > New** to open the New Source dialog.



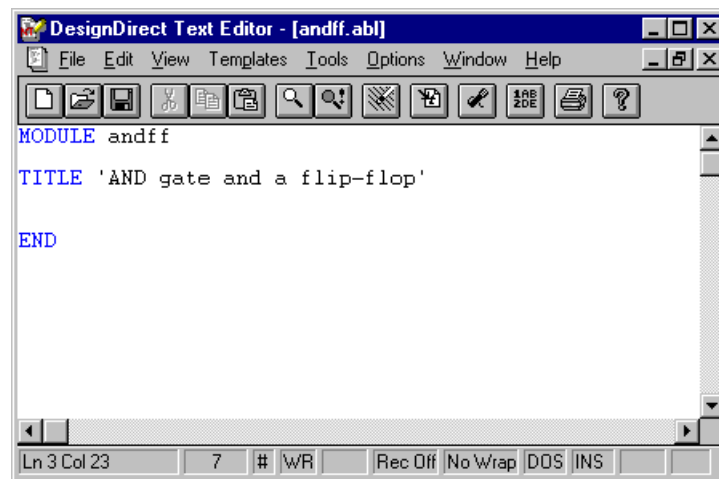
- 2 Select **ABEL-HDL Module** and click **OK**. The Text Editor opens and a dialog prompts you for a module name, file name, and title.



- 3 Type a Module Name, which is the name of the MODULE statement. The MODULE statement is required. It defines the beginning of the module and must be paired with an END statement. The MODULE statement also indicates whether any module arguments are used.
- 4 Type a File Name. The file extension can be omitted.

Note: The module name and file name should have the same base name. (The base name is the name without the 3-character extension.) If the module and file names are different, some automatic functions in the Project Manager might fail to run properly.

- 5 Optionally, type a descriptive Title for the module.
- 6 When you have finished typing the information, click **OK**. You now have a template for an ABEL-HDL source file.



Entering Declarations

The Declarations section specifies the names and attributes of signals used in the design; defines constants, macros, and states; declares lower-level modules and schematics; and optionally declares a device. Each module must have at least one DECLARATIONS section, and declarations affect only the module in which they are defined. If a TITLE statement exists in the template file, enter these statements after the TITLE statement.

Using the *andff* module above as an example, the following describes the DECLARATIONS statement for the three inputs (two AND gate inputs and the clock) and the output.

```
DECLARATIONS
    input_1, input_2, Clk      pin;
    output_q                  pin istype 'reg';
```

These two statements declare four signals (*input_1*, *input_2*, *Clk*, and *output_q*).

Note: ABEL-HDL does not have an explicit declaration for inputs and outputs; whether a given signal is an input or an output depends on how it is used in the design description that follows. The signal *output_q* is declared to be type 'reg', which implies that it is a registered output pin. The actual behavior of *output_q*, however, is specified using one or more equations.

Entering Logic Descriptions

You can use Equations, a State diagram, or a Truth table to describe your logic design. The following EQUATIONS statement describes the actual behavior of the *andff* example module.

```
EQUATIONS
    output_q      := input_1 & input_2;
    output_q.clk  = Clk;
```

These two equations define the data to be loaded on the registered output, and define the clocking function for the output.

Entering Test Vectors

The traditional method for testing ABEL-HDL designs is to use test vectors. Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with both Equation and JEDEC simulators.

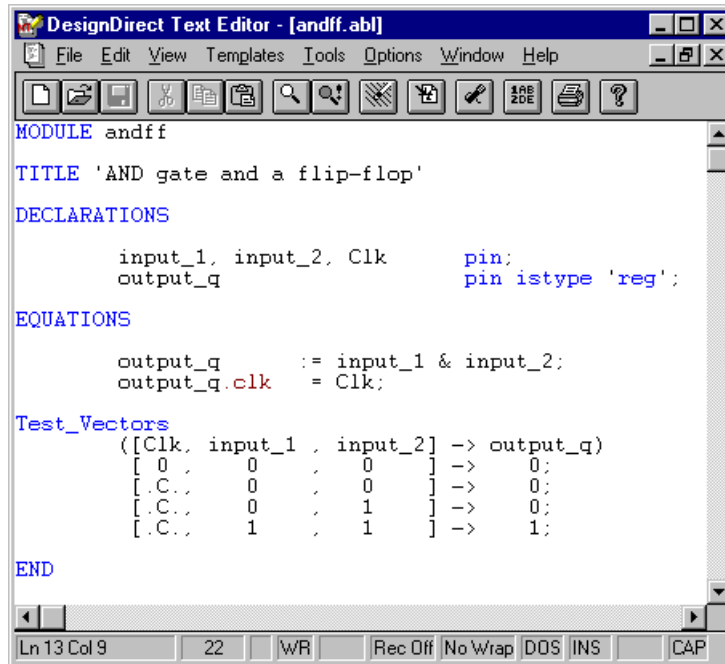
You can specify test vectors in two ways: in the ABEL-HDL source file, or in an external ABEL Test Vector file (.ABV). When you specify the test vectors in the ABEL-HDL source file, DesignDirect creates a "dummy" .ABV file that points to the ABEL-HDL source containing the vectors. This file is necessary because an .ABV file is required in order to have access to the Equation and JEDEC simulation processes.

To continue with the example *andff* module, the following describes the TEST_VECTORS statement.

```
TEST_VECTORS
    ([Clk, input_1 , input_2] -> output_q)
    [ 0 ,      0      , 0      ] -> 0;
    [.C.,      0      , 0      ] -> 0;
    [.C.,      0      , 1      ] -> 0;
    [.C.,      1      , 1      ] -> 1;
```

The following figure shows the complete ABEL-HDL source file describing the example *andff* module.

Figure 9: ABEL-HDL source file describing the andff module



```
DesignDirect Text Editor - [andff.abl]
File Edit View Templates Tools Options Window Help
[Icons]
MODULE andff
TITLE 'AND gate and a flip-flop'
DECLARATIONS
    input_1, input_2, Clk      pin;
    output_q                  pin istype 'reg';
EQUATIONS
    output_q      := input_1 & input_2;
    output_q.clk  = Clk;
Test_Vectors
    ([Clk, input_1 , input_2] -> output_q)
    [ 0 , 0 , 0 ] -> 0;
    [.C., 0 , 0 ] -> 0;
    [.C., 0 , 1 ] -> 0;
    [.C., 1 , 1 ] -> 1;
END
Ln 13 Col 9 22 WR Rec Off No Wrap DOS INS CAP
```