ABEL-HDL
Reference

SYNARIO

Universal FPGA Design System

Synario is a Data I/O Product

Table of Contents

1. Introduction

2. Language Structure

3. Design Considerations

4. Designing with FPGAs

5. Source File Examples

6. Language Reference

Chapter 1
Introduction

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. The requirements for ABEL-HDL are described in the following chapters.

- **Chapter 2,** "**Language Structure**" — provides the basic syntax and structure of an ABEL-HDL design description. For information on specific elements, refer to Chapter 6, "Language Reference."
- **Chapter 3,** "**Design Considerations**" — discusses issues to consider when creating an ABEL-HDL module, such as architecture-independent language features, active low declarations, flip-flop equations, feedback considerations, and polarity control.
- **Chapter 4,** "**Designing with FPGAs**" — discusses issues to consider when designing for FPGA devices. The information in this chapter is only applicable to Synario.
- **Chapter 5,** "**Source File Examples**" — contains ABEL-HDL module examples. These examples are representative of programmable logic applications and illustrate significant ABEL features. They also help you create your own source files.
- **Chapter 6,** "**Language Reference**" — gives detailed information about ABEL-HDL language elements. This chapter assumes you are familiar with the basic syntax discussed in Chapter 2, "Language Structure."

Chapter 2
Language Structure

This chapter provides the basic syntax and structure of a design description in ABEL-HDL. For information on specific elements, refer to Chapter 6, "Language Reference." You can write a source file using any editor that produces ASCII files; you are not limited to the ABEL or Synario Text Editor.

Summary

This chapter contains the following sections:

Introduction to ABEL-HDL and to the idea of architecture-independent and architecture-specific logic descriptions.

Basic syntax of a source file, including

- 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

Basic Structure of a design description, including

- Header
  - Module
  - Interface (lower-level)
  - Title
- Declarations
  - Declarations keyword
  - Interface and Functional_block declarations
  - Constant declarations
  - Signal declarations
  - Device declarations
- Logic description
  - Equations
  - Truth tables
  - State descriptions
  - Fuses
  - XOR factors
- Test vectors (for PLD JEDEC simulation only)
- End

Introduction to ABEL-HDL

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 ABEL and Synario versions of the ABEL-HDL compiler (and supporting software) functionally verify ABEL-HDL designs through simulation. The compilers then implement the designs in PLDs or FPGAs. ABEL-HDL designs can also be transferred to other design environments through standard-format design transfer files.

You can enter designs in ABEL-HDL and verify them with little or no concern for the architecture of the target device.

Architecture-independent design descriptions (those that do not include device declarations and pin number declarations) require more comprehensive descriptions than their architecture-specific counterparts. Assumptions that can be made when a particular device is specified are not possible when no device is specified. See the section "Architecture-independent Language Features" in Chapter 3, "Design Considerations" for a more detailed discussion.

Basic Syntax

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 ↵ .
- 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.

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.

Some supported identifiers are listed below:

```
HELLO
hello
_K5input
P_h
This_is_a_long_identifier
AnotherLongIdentifier
```

Some unsupported identifiers are listed below:

```
7_      Does not begin with a letter or underscore
$4      Does not begin with a letter or underscore
HEL.LO  Contains a period (.LO is not a valid dot extension)
b6 kj   Contains a space
```

The last of these identifiers is interpreted as two identifiers, b6 and kj.

Reserved Identifiers (Keywords)

The keywords listed below are reserved identifiers. Keywords cannot be used to name devices, pins, nodes, constants, sets, macros, or signals. If a keyword is used in the wrong context, an error is flagged.

async_reset	fuses	state
case	goto	state_diagram
declarations	if	state_register
device	in	sync_reset
else	interface	test_vectors
enable (obsolete)	istype	then
end	library	title
endcase	macro	trace
endwith	module	truth_table
equations	node	when
external	options	with
flag (obsolete)	pin	
functional_block	property	

Choosing Identifiers

Choosing the right identifiers can make a source file easy to read and understand. The following suggestions can help make your logic descriptions self-explanatory, eliminating the need for extensive documentation.

- Choose identifiers that match their function. For example, the pin you're going to use as the carry-in on an adder could be named Carry_In. For a simple OR gate, the two input pins might be given the identifiers IN1 and IN2, and the output might be named OR.
- Avoid large numbers of similar identifiers. For example, do not name the outputs of a 16 bit adder: ADDER_OUTPUT_BIT_1 ADDER_OUTPUT_BIT_2 and so on.
- Use underscores or mixed-case characters to separate words in your identifier.

  ```
  THIS_IS_AN_IDENTIFIER
  ThisIsAnIdentifier
  ```

  is much easier to read than

  ```
  THISISANIDENTIFIER
  ```

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 (see "Declarations" and "Module Statement" later in this chapter). Constant values can be either numeric or one of the non-numeric special constant values. The special constant values are listed in Table 2-1.

Table 2-1
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 Table 2-1. 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. (See "Macro Declarations" later in this chapter and in Chapter 6, "Language Reference.")

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 allow more-complex functions than possible without blocks. Blocks can 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:

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:

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:

Without Blocks:

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

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

Without Blocks:

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

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 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) are required strings, not comments, and are part of the statement.

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 2-2 lists the four bases supported by ABEL-HDL and their accompanying symbols. The base symbols can be upper- or lowercase.

Table 2-2
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. See **@RADIX** in Chapter 6, "Language Reference," for more information. Examples of supported number specifications are shown below. The default base is base ten (decimal).

Specification	Decimal Value
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:

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 Synario'
```

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 2-3. Logical operations are performed bit by bit. For alternate operators, refer to the @ALTERNATE directive in Chapter 6, "Language Reference."

Table 2-3
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 2-4 lists the arithmetic operators.

Table 2-4
Arithmetic Operators

Operator	Example	Description
-	-A	twos complement (negation)
-	A-B	subtraction
+	A+B	addition
Not Supported for Sets:		
*	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 2-5 lists the relational operators.

Table 2-5
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.

Some examples of relational operators in expressions are listed below:

Expression	**Value**
2 == 3	False
2 != 3	True
3 < 5	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. For more information, see the "Equations" section later in this chapter.

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. Refer to Chapter 3, "Design Considerations." Table 2-6 shows the assignment operators.

Table 2-6
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 2-7 summarizes the logical, arithmetic and relational operators, presented in groups according to their priority.

Table 2-7
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 show how the order of operations and the use of parentheses affect the evaluated result.

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.

See Also

"Equations" and "When-Then-Else" in Chapter 6, "Language Reference."

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.

See Also

If-Then-Else, When-Then-Else, and CASE in Chapter 6, "Language Reference."

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.

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 decrementing 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).

See **multiply.abl** for more examples of set indexing. See also the @Setsize directive.

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. Table 2-8 lists the operators you can use with sets. "Set Operations," found later in this chapter, describes how these operators are applied to sets.

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 1

```
" 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];
```

Example 2

```
" 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. Supported set operations are given in Table 2-8. 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 2-8
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.

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 s**el2** 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.

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, ..., a_1.OE$]$
$A_k \& B_k$	$[a_k \& b_k, a_{k-1} \& b_{k-1}, ..., a_1 \& b_1]$
$A_k \# B_k$	$[a_k \# b_k, a_{k-1} \# b_{k-1}, ..., a_1 \# b_1]$
$A_k \$ B_k$	$[a_k \$ b_k, a_{k-1} \$ b_{k-1}, ..., a_1 \$ b_1]$
A_k !$ B_k	$[a_k$!$ b_k, a_{k-1} !$ b_{k-1}, ..., a_1 !$ b_1]$
$A_k == B_k$	$(a_k == b_k) \& (a_{k-1} == b_{k-1}) \& ... \& (a_1 == b_1)$
$A_k != B_k$	$(a_k != b_k) \# (a_{k-1} != b_{k-1}) \# ... \# (a_1 != b_1)$
$A_k + B_k$	D_k where: d_n is evaluated as $a_n \$ b_n \$ c_{n-1}$ c_n is evaluated as $(a_n \$ b_n) \# (a_n \& c_{n-1}) \# (b_n \& c_{n-1})$ c0 is evaluated as 0
$A_k - B_k$	$A_k + (-B_k)$
$A_k < B_k$	c_k where: c_n is evaluated as $(!a_n \& (b_n \# c_{n-1}) \# a_n \& b_n \& c_{n-1}) != 0$ c0 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

```
A1 = 0
A2 = 0
A3 = 1
```

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.

Equation 1:
The first operation is **[a, b] & 1**, so 1 is converted to a set [0, 1].

```
[x1, y1] = [a, b] & 1 & d
         = ([a, b] &    1  ) & d
         = ([a, b] & [0, 1]) & d
         = ([a & 0,  b & 1]) & d
         =  [  0  ,    b  ]  & d
         =  [0 & d, b & d]
         =  [0, b & d]

 x1 = 0
 y1 = b & d
```

Equation 2:
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 in Table 2-7.
- 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 (see "Macro Declarations" later in this chapter and the design example file **macro.abl**):

```
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.

Further discussion and examples of argument use are given in Chapter 6, "Language Reference" under "Module," "Macro," and "*@directive*."

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:

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

The elements of the source file are shown in the template in Figure 2-1. There are also directives that can be included in any of the middle three sections. The sections are presented briefly below, then each element is introduced. You can find complete information in Chapter 6, "Language Reference."

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. 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:

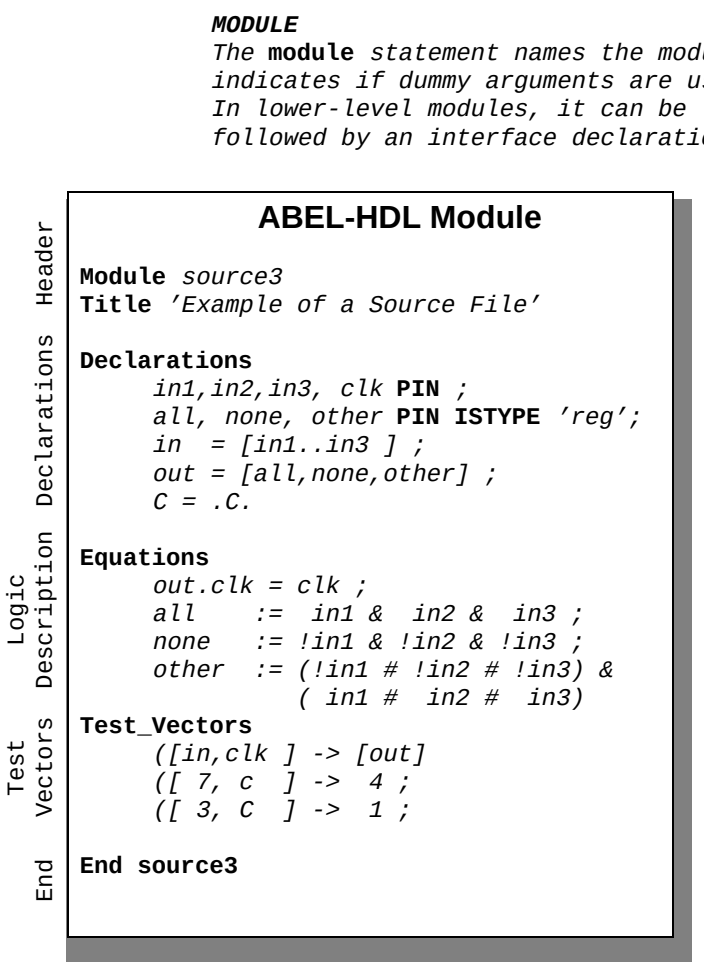
- Module (required)
- Interface (lower level, optional)
- Title

Declarations

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)

Figure 2-1
ABEL-HDL Module Structure



Logic Description

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

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

Test Vectors Section

Test vectors are only used forEquation or JEDEC Simulation. See the *Equation and JEDEC Simulators User Manual* for information on simulating other devices. A Test Vectors section can consist of the following elements:

- Test Vectors
- Trace Statement
- Test Script

End Statement

A module is closed with the end statement:

- End Statement

Other Elements

Directives can be placed anywhere you need them:

- Directives

Header

Module

Keyword: 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.

Interface

Keyword: interface

The interface statement is used in lower-level sources to indicate signals used in upper-level files. The interface statement is optional.

Title

Keyword: title

The title is optional. The title appears as a header in some output files.

Declarations

The declarations section of a module 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. There are several types of declaration statements:

- Constant (see =)
- Device
- Hierarchy
- Library
- Macro
- Signal (see Pin, Node and Istype)
- State
- State register

The syntax and use of each of these types is presented in Chapter 6, "Language Reference." Some are discussed briefly below.

Declarations Keyword

Keyword: declarations

This keyword allows declarations (such as sets or other constants) in any part of the source file.

Device Declaration

Keyword: device

device_id **DEVICE** *real_device* ;

The Device declaration is optional, and only one can be made per module. It associates a device identifier with a specific programmable logic device.

Hierarchy Declarations

Interface Declarations

Top-level Interface Declarations

Keyword: interface

```
low-level module_name INTERFACE (inputs[=value] -> outputs :>
bidirs ...)
```

The **interface** keyword declares lower-level modules that are used by the current module. This declaration is used in conjunction with a **functional_block** declaration for each instantiation of a module.

When you instatiate a functional block, you must map port names to signal names with equations. See **functional_block** for more information.

Lower-level Interface Declarations

Keyword: interface

MODULE *module_name*
INTERFACE (*input/set=value . . . -> output/set* :> *bidir/set*) ;

Use the **interface** declaration in lower-level modules to assign a default port list and input values for the module when instantiated in higher-level ABEL-HDL sources. In the higher-level source, you must declare signals and sets in the same order and grouping as given in the interface statement in the instantiated module.

The -> and :> delimiters are used to indicate the direction of each port of a functional block.

CAUTION: Interface declarations cannot contain dot extensions. If you need a specific dot extension across a source boundary (to resolve feedback ambiguities, for example), you must introduce an intermediate signal into the lower-level module to provide the connection to the higher-level source. All dot extension equations for a given output signal must be located in the ABEL-HDL module in which the signal is defined. No references to that signal's dot extensions can be made outside of the ABEL-HDL module.

Note: *Bidirectional interface statements may not be supported in all device kits.*

Functional_block Statement

Keyword: functional_block

DECLARATIONS
instance_name FUNCTIONAL_BLOCK *module_name* ;

EQUATIONS
instance_name.port_name = signal_name;

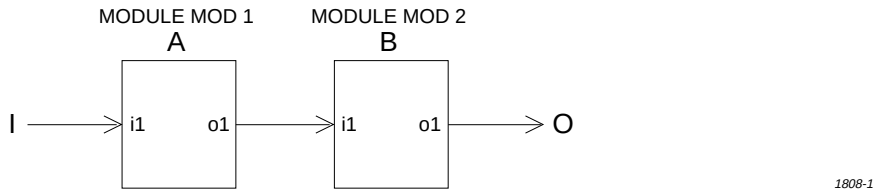
Use a **functional_block** declaration to instantiate a declared source within a higher-level ABEL-HDL source. You must declare a source with an **interface** declaration before instantiating it with **functional_block**.

Example of Functional Block Instantiation

To declare the two ABEL-HDL sources shown in Figure 2-2 would require the following syntax:

```
module FUNC ;
   mod1 INTERFACE (i1 -> o1);
        A FUNCTIONAL_BLOCK mod1;
   mod2 INTERFACE (i1 -> o1);
        B FUNCTIONAL_BLOCK mod2;
   I pin ;
   O pin istype 'com';

Equations
   O = B.o1;
   B.i1 = A.o1;
   A.i1 = I;

end Func
```

Figure 2-2
Functional Block Instantiation



Note that the output of an equation must always be on the left side of the equations.

See Also

"Hierarchy" in Chapter 3, "Design Considerations."

Signal Declarations

The **Pin** and **Node** declarations are made to declare signals used in the design, and optionally to associate pin and/or node numbers with those signals. Actual pin and node numbers do not have to be assigned until you want to map the design into a device. Attributes can be assigned to signals within pin and node declarations with the **Istype** statement. Dot extensions can also be used in equations to precisely describe the signals; see "Dot Extensions" under "Logic Descriptions" later in this chapter.

Note: *Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later by a fitter.*

Pin Declarations

Keyword: **pin**

```
[ ! ]pin_id [,[ ! ]pin_id...] PIN [pin# [,pin# ] ]
[ISTYPE 'attributes' ] ;
```

See "Attribute Assignment" below, and "Using Active-low Declarations" in Chapter 2, "Language Structure."

Node Declarations

Keyword: **node**

```
[ ! ]node_id [, [ ! ]node_id...] NODE [node# [,node# ] ]
[ISTYPE 'attributes' ] ;
```

See "Attribute Assignment" below, and "Using Active-low Declarations" in Chapter 2, "Language Structure."

Attribute Assignment

Keyword: **istype**

```
signal [,signal]...  ISTYPE  'attributes';
```

The ISTYPE statement defines attributes (characteristics) of signals for devices with programmable characteristics or when no device and pin/node number has been specified for a signal. Even when a device has been specified, using attributes will make it more likely that the design operates consistently if the device is changed later. ISTYPE can be used after pin or node declarations.

Attributes may be entered in uppercase, lowercase, or mixed-case letters. Table 2-9 summarizes the attributes. Each attribute is discussed in more detail in Chapter 6, "Language Reference" under Istype.

Table 2-9
Attributes

Dot Ext.	Arch. Indep.	Description
'buffer'		No Inverter in Target Device.
'collapse'		Collapse (remove) this signal. [1]
'com'	✓	Combinational output.
'dc'	✓	Unspecified logic is don't care. [2]
'invert'		Inverter in Target Device.
'keep'		Do not collapse this signal from equations. [1]
'neg'	✓	Unspecified logic is 1. [2]
'pos'	✓	Unspecified logic is 0. [2]
'retain'	✓	Do not minimize this output. Preserve redundant product terms. [3]
'reg'	✓	Clocked Memory Element.
'reg_d'		D Flip-flop Clocked Memory Element.
'reg_g'		D Flip-flop Gated Clock Memory Element.
'reg_jk'		JK Flip-flop Clocked Memory Element.
'reg_sr'		SR Flip-flop Clocked Memory Element.
'reg_t'		T Flip-flop Clocked Memory Element.
'xor'		XOR Gate in Target Device.

[1] *If neither 'keep' nor 'collapse' is specified, the optimization or fitter programs can keep or collapse the signal, as needed, to optimize the circuit.*

[2] *The 'dc,' 'neg,' and 'pos' attributes are mutually exclusive.*

[3] *The 'retain' attribute only controls optimization performed by ABEL-HDL Compile Logic. To preserve redundant product terms, you must also specify no reduction for the Reduce Logic and fitting (place and route) programs.*

Constant Declarations

Keyword: **=**

id [, *id*]... **=** *expr* [, *expr*]... ;

A constant is an identifier that retains a constant value in a module, and is specified with the = sign. Constant declarations must be in a declarations section or after a @CONST directive.

See Also

"Special Constants" in this chapter.

Symbolic State Declarations

The State_register and State declarations are made to declare a symbolic state machine name, and to declare symbolic state names.

See Also

"State Descriptions" under "Logic Descriptions" later in this chapter.

State_register Declarations

Keyword: **state_register**

```
statereg_id STATE_REGISTER [ISTYPE 'attributes'];
```

State Declarations

Keyword: **state**

```
state_id [, state_id ...] STATE [state_value
[, state_value ...]];
```

Macro Declarations

Keyword: **macro**

macro_id **MACRO** [(*dummy_arg* [,*dummy_arg*]...)] {*block*} ;

The macro declaration statement defines a macro. Use macros to include functions in a source file without repeating the code.

Library Declaration

Keyword: **library**

LIBRARY *'name'* ;

The LIBRARY statement extracts the contents of the indicated file from the ABEL-HDL library and inserts it into your file.

Logic Description

One or more of the following elements can be used to describe your design.

- Equations
- Truth Tables
- State Descriptions
- 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.

Dot Extensions

Syntax *signal_name.ext*

Dot extensions can be specific for certain devices (device-specific) or generalized for all devices (architecture-independent). Device-specific dot extensions are used with detailed syntax; architecture-independent dot extensions are used with pin-to-pin syntax. Detailed and pin-to-pin syntax is described in more detail in Chapter 3, "Design Considerations." Dot extensions can be applied in complex language constructs such as nested sets or complex expressions.

The ABEL-HDL dot extensions are listed in Table 2-10.

Table 2-10
Dot Extensions

Dot Extension	Description
Pin-to-Pin Syntax, Architecture-independent	
.ACLR	*Asynchronous clear
.ASET	*Asynchronous set
.CLK	Clock input to an edge-triggered flip-flop
.CLR	*Synchronous clear
.COM	*Combinational feedback normalized to the pin value
.FB	Register feedback
.OE	Output enable
.PIN	Pin feedback
.SET	*Synchronous set

Dot Extension	Description
Detailed Syntax, Device-specific	
.AP	Asynchronous register preset
.AR	Asynchronous register reset
.CE	Clock-enable input to a gated-clock flip-flop
.D	Data input to a D-type flip-flop
.FC	Flip-flop mode control
.J	J input to a JK-type flip-flop
.K	K input to a JK-type flip-flop
.LD	Register load input
.LE	Latch-enable input to a latch
.LH	Latch-enable (high) to a latch
.PR	Register preset
.Q	Register feedback
.R	R input to an SR-type flip-flop
.RE	Register reset
.S	S input to an SR-type flip-flop
.SP	Synchronous register preset
.SR	Synchronous register reset
.T	T input to a T-type (toggle) flip-flop

* *The .CLR, .ACLR, .SET, .ASET and .COM dot extensions are not recognized by device fitters released prior to ABEL 5.0. If you are using a fitter that does not support these reset/preset dot extensions, specify istype 'invert' or istype 'buffer' and the compiler converts the new dot extensions to .SP, .AP, .SR, .AR, and .D, respectively.*

Equations

Keyword: **equations**

```
Equations
[ WHEN condition    THEN ] [  ! ] element=expression;
[ ELSE equation ];
          or
[ WHEN condition    THEN ] equation;  [ ELSE equation];
```

The EQUATIONS statement defines the beginning of a group of equations that specify the logic functions of a device. See "Operators, Expressions and Equations" earlier in this chapter and "When-Then-Else" in the "Language Reference."

Truth Tables

Keyword: **truth_table**

```
TRUTH_TABLE (inputs -> outputs )
inputs -> outputs ;
 :
    or
TRUTH_TABLE (inputs [:> registered outputs] [-> outputs ] )
```

Truth tables specify outputs as functions of input combinations in tabular form. See also "@DCSET" under "*@directive*" in Chapter 6, "Language Reference."

State Descriptions

Keyword: state_diagram

```
STATE_DIAGRAM state_reg
                        [-> state_out]
[STATE state_exp : [equation]
                          [equation]
  :
  :
  :
trans_stmt  ...]
```

The **State_Diagram** section contains state descriptions that describe the logic design.

The specification of a state description requires the use of the **State_diagram** syntax, which defines the state machine, and the **If-Then-Else, Case,** and **Goto** statements that determine the operation of the state machine.

See also

"With" in the "Language Reference."

Fuse Declarations

Keyword: fuses

```
FUSES
fuse_number = fuse value ;
    or
fuse_number_set = fuse value ;
```

The FUSES section explicitly declares the state of fuses in the associated device. A device must be declared before a fuses declaration.

XOR Factors

Keyword: XOR_Factors

```
XOR_Factors
signal name = xor_factors
```

The XOR_Factors section allows you to specify a Boolean expression that is to be factored out of and XORed with the sum-of-products reduced equations. This factoring can result in smaller reduced equations when the design is implemented in a device featuring XOR gates.

Test Vectors Section

Note: *Test vectors are supported only for Equation and JEDEC simulation. The Verilog Simulator uses test fixtures. See the Verilog Simulator User Manual.*

Test Vectors

Keyword: test_vector

```
Test_vectors [note ]
(inputs -> outputs)
[invalues -> outvalues ; ] ...
```

Test vectors specify the expected operation of a logic device by defining its outputs as a function of its inputs.

Trace Statement

Keyword: trace

```
trace (inputs -> outputs) ;
```

The Trace statement limits which inputs and outputs are displayed in the simulation report.

End Statement

Keyword: end

```
end module_name
```

The **End** statement ends the module, and is required.

Other Elements

Directives

Keyword: @*directive*

@directive [*options]*

Directives provide options that control the contents or processing of a source file. Sections of ABEL-HDL source code can be included conditionally, code can be brought in from another file, and messages can be printed during processing.

Some directives take arguments that determine how the directive is processed. These arguments can be actual arguments or dummy arguments preceded by a question mark. The rules applying to actual and dummy arguments are presented under "Arguments and Argument Substitution" earlier in this chapter.

Available directives are listed below. See @ in Chapter 6, "Language Reference" for complete information.

@ALTERNATE	@IFNDEF
@CARRY	@IFNIDEN
@CONST	@INCLUDE
@DCSET	@IRP
@DCSTATE	@IRPC
@EXPR	@MESSAGE
@EXIT	@ONSET
@IF	@PAGE
@IFB	@RADIX
@IFDEF	@REPEAT
@IFIDEN	@SETSIZE
@IFNB	@STANDARD

Chapter 3
Design Considerations

This chapter discusses issues you need to consider when you create a design with ABEL-HDL. The topics covered are listed below:

- Hierarchy in ABEL-HDL
- Pin-to-Pin Architecture-independent Language Features
- Pin-to-Pin Vs. Detailed Descriptions for Registered Designs
- Using Active-low Declarations
- Polarity Control
- Istypes and Attributes
- Flip-flop Equations
- Feedback Considerations — Using Dot Extensions
- @DCSET Considerations and Precautions
- Exclusive OR Equations
- State Machines
- Using Complement Arrays

Hierarchy in ABEL-HDL

You use hierarchy declarations in an upper-level ABEL-HDL source to refer to (instantiate) an ABEL-HDL module. To instantiate an ABEL-HDL module:

In the lower-level module: (optional)

1. Identify lower-level I/O Ports (signals) with an **Interface** statement.

In the top-level source:

2. Declare the lower-level module with an **Interface** declaration.
3. Instantiate the lower-level module with **Functional_block** declarations.

Note: *Hierarchy declarations are not required when instantiating an ABEL-HDL module in a Synario schematic. For instructions on instantiating lower-level modules in schematics, refer to your schematic reference.*

Instantiating a Lower-level Module in an ABEL-HDL Source

Identifying I/O Ports in the Lower-level Module

The way to identify an ABEL-HDL module's input and output ports is to place an **Interface** statement immediately following the **Module** statement. The **Interface** statement defines the ports in the lower-level module that are used by the top-level source.

You must declare all input pins in the ABEL-HDL module as ports, and you can specify default values of 0, 1, or Don't-care.

You do not have to declare all output pins as ports. Any undeclared outputs become No Connects or redundant nodes. Redundant nodes can later be removed from the designs during post-link optimization.

The following source fragment is an example of a lower-level **interface** statement.

```
module lower
interface (a=0, [d3..d0]=7 -> [z0..z7]) ;
title 'example of lower-level interface statement ' ...
```

This statement identifies input **a**, **d3**, **d2**, **d1** and **d0** with default values, and outputs **z0** through **z7**. For more information, see "Interface (lower-level)" in Chapter 6, "Language Reference."

Specifying Signal Attributes

Attributes specified for pins in a lower-level module are propagated to the higher-level source. For example, a lower-level pin with an 'invert' attribute affects the higher-level signal wired to that pin (it affects the pin's preset, reset, preload, and power-up value).

Output Enables (OE)

Connecting a lower-level tristate output to a higher-level pin results in the output enable being specified for the higher-level pin. If another OE is specified for the higher-level pin, it is flagged as an error. Since most tristate outputs are used as bidirectionals, it might be important to keep the lower-level OE.

Buried Nodes

Buried nodes in lower-level sources are handled as follows:

Dangling Nodes	Lower-level nodes that do not fanout are propagated to the higher-level module and become dangling nodes. Optimization may remove dangling nodes.
Combinational nodes	Combinational nodes in a lower-level module become collapsible nodes in the higher-level module.
Registered nodes	Registered nodes are preserved with hierarchical names assigned to them.

Declaring Lower-level Modules in the Top-level Source

To declare a lower-level module, you match the lower-level module's interface statement with an interface declaration. For example, to declare the lower-level module given above, you would add the following declaration to your upper-level source declarations:

```
lower interface (a, [d3..d0] -> [z0..z7]) ;
```

You could specify different default values if you want to override the values given in the instantiated module, otherwise the instantiated module must exactly match the lower-level interface statement. See "Interface (top-level)" in Chapter 6, "Language Reference" for more information.

Instantiating Lower-level Modules in Top-level Source

Use a **functional_block** declaration in an top-level ABEL-HDL source to instantiate a declared lower-level module and make the ports of the lower-level module accessible in the upper-level source. You must declare sources with an **interface** declaration before you instantiate them.

To instantiate the module declared above, add an interface declaration and signal declarations to your top-level declarations, and add port connection equations to your top-level equations, as shown in the source fragment below:

```
DECLARATIONS
   low1 FUNCTIONAL_BLOCK lower ;
   zed0..zed7 pin ;                "upper-level inputs
   atop pin istype 'reg,buffer';   "upper-level output
   d3..d0 pin istype 'reg,buffer'; "upper-level ouputs
EQUATIONS
   atop = low1.a;                  "wire this source's outputs
   [d3..d0] = low1.[d3..d0] ;      "  to lower-level inputs
   low1.[z0..z7] = [zed0..zed7];   "wire this source's inputs to
                                   "  lower-level outputs
```

See "Functional_block" in Chapter 6, "Language Reference" for more information.

Hierarchy and Retargeting and Fitting

Redundant Nodes

When you link multiple sources, some unreferenced nodes may be generated. These nodes usually originate from lower-level outputs that are not being used in the top-level source. For example, when you use a 4-bit counter as a 3-bit counter. The most significant bit of the counter is unused and can be removed from the design to save device resources. This step also removes trivial connections. In the following example, if out1 is a pin and t1 is a node:

```
out1 = t1;
t1 =  a86;
```

would be mapped to

```
out1 = a86;
```

Merging Feedbacks

Linking multiple modules can produce signals with one or more feedback types, such as .FB and .Q. You can tell the optimizer to combine these feedbacks to help the fitting process.

Post-linked Optimization

If your design has a constant tied to an input, you can re-optimize the design. Re-optimizing may further reduce the product terms count.

For example, if you have the equation

```
out = i0 & i1 || !i0 & i2;
```

and i0 is tied to 1, the resulting equation would be simplified to

```
out = i1;
```

Hierarchy and Test Vectors (PLD JEDEC Simulation)

If you are targeting a PLD device and want to do JEDEC simulation of your project, you must specify your test vectors in the top-level source. If you have existing test vectors in lower-level sources, you can merge the inputs stimulus of blocks that are connected to the top-level pins with the expected values of blocks that are connected to the top-level outputs. The test vectors in the lower-level modules can still be used for individual JEDEC simulation.

Node Collapsing

All combinational nodes are collapsible by default . Nodes that are to be collapsed (or nodes that are to be preserved) are flagged through the use of signal attributes in the language. The signal attributes are:

Istype 'keep'	Do not collapse this node.
'collapse'	Collapse this node.

Collapsing provides multi-level optimization for combinational logic. Designs with arithmetic and comparator circuits generally generate a large number of product terms that will not fit to any programmable logic device. Node collapsing allows you to describe equations in terms of multi-level combinational nodes, then collapse the nodes into the output until it reaches the product term you specify. The result is an equation that is optimized to fit the device constraints.

Selective Collapsing

In some instances you may want to prevent the collapsing of certain nodes. For example, some nodes may help in the simulation process. You can specify nodes you do not want collapsed as Istype 'keep' and the optimizer will not collapse them.

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 an EP600 EPLD, 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. See "Pin Declaration," "Node Declaration" and "Istype" in Chapter 6, "Language Reference."

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.

Refer to "Dot Extensions" later in this chapter and in Chapter 2, "Language Structure" or *.ext* in Chapter 6, "Language Reference" for more information.

Pin-to-pin vs. Detailed Descriptions 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 described in Chapter 6, "Language Reference." These dot extensions help resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities

In the equation above (Q1 := !Q1 # 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 fed-back 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 EP600).

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:

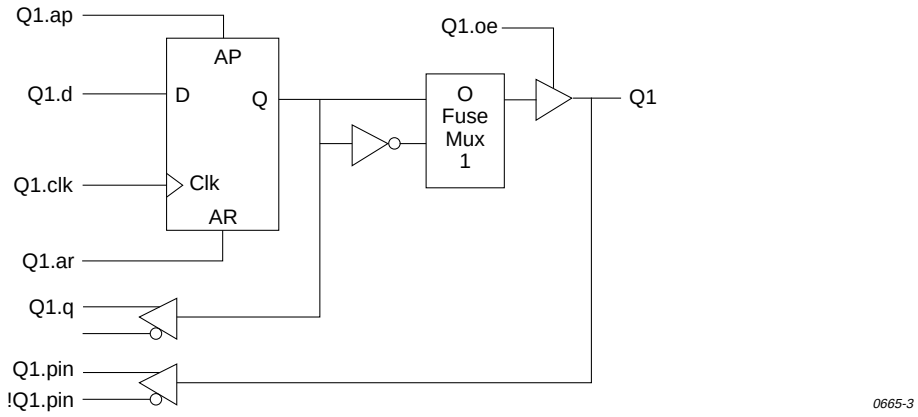
```
Q1 pin istype 'buffer';
```

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. Figure 3-1 illustrates a typical macrocell associated with signal **Q1**.

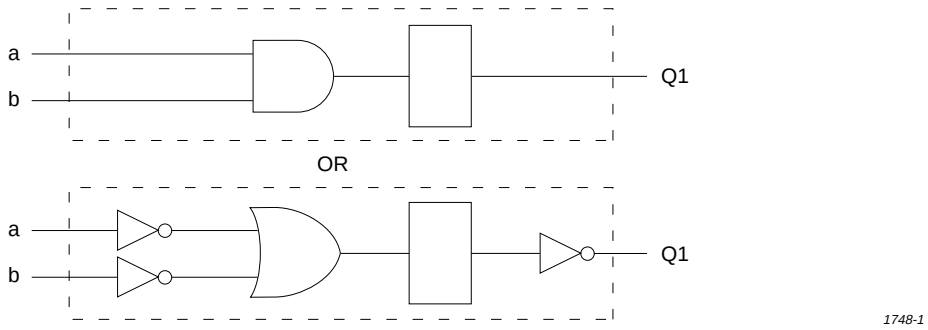
Detailed descriptions are written for the various input ports of the macrocell (shown in Figure 3-1 with dot extension labels). Note that the macrocell features a configurable inversion between the Q output of the flip-flop and the output pin labeled **Q1**. If you use this inverter (or select a device 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**.

Figure 3-1
Detail Macrocell



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. Figure 3-2 illustrates the pin-to-pin concept:

Figure 3-2
Pin-to-pin Macrocell



When pin-to-pin descriptions are written in ABEL-HDL, the "generic macrocell" 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 in the following manner:

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_5
        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_7
        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:

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
```

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
```

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);

(3)               !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);

(3)         !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;
```

Since 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 — Using 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.

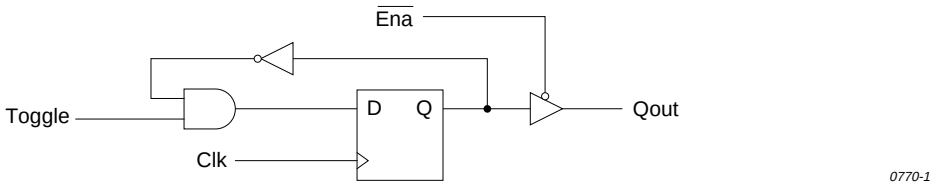
- **.COM Extension** — A signal specified with the .COM extension (for example, count := count.com+1;) results in OR-array (pre-register) feedback, normalized to the pin value if an OR-array feedback path exists. If no OR-array feedback is available, pin feedback is used and the fuse mapper checks that 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 and P18CV8, 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 Figure 3-3. 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 3-3
Dot Extensions and Architecture- Independence: Circuit 1



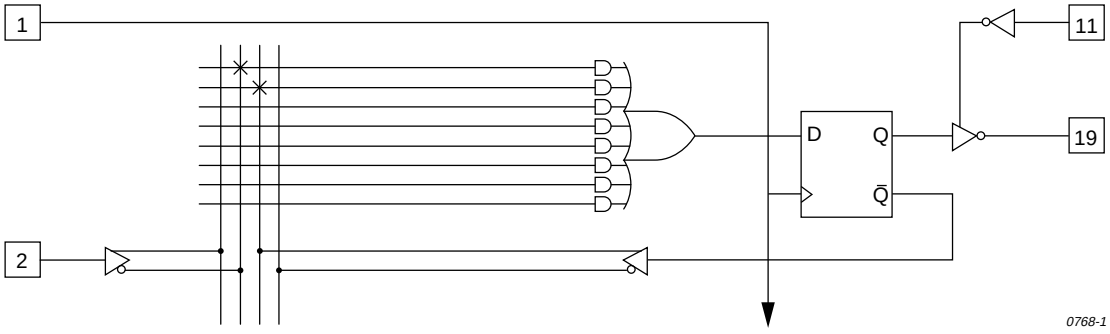
The following simple ABEL-HDL design (Figure 3-4) describes this simple one-bit synchronous circuit. The design description uses architecture-independent dot extensions to describe the circuit in terms of its behavior, as observed on the output pin of the target device. Since this design is architecture-independent, it will operate the same (disregarding initial powerup state), irrespective of the device type.

Figure 3-4
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
```

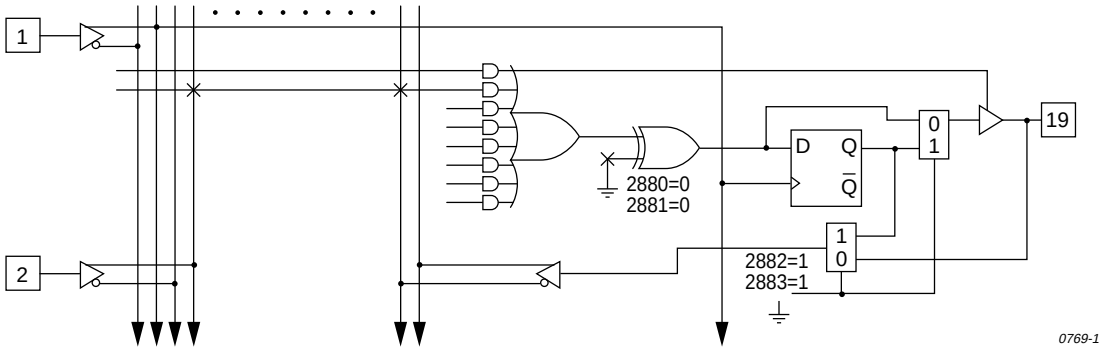
If you implement this circuit in a simple P16R8 PAL device (either by adding a device declaration statement or by specifying the P16R8 in the Fuseasm process), the result will be a circuit like the one illustrated in Figure 3-5. Since the P16R8 features inverted outputs, the design equation is automatically modified to take the feedback from Q-bar instead of Q.

Figure 3-5
Dot Extensions and Architecture-Independence: Circuit 2



If you implement this design in a device with a different architecture, such as an E0320, the resulting circuit could be quite different. But, because this is a pin-to-pin design description, the circuit behavior is the same. Figure 3-6 illustrates the circuit that results when you specify an E0320.

Figure 3-6
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 powerup 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 (Figure 3-7):

Figure 3-7
Detail 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 Figure 3-5. 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.*

To implement this design in a device that does not feature inverted outputs, the design description must be modified. The following example (Figure 3-8) shows how to write this detailed design for the E0320 device:

Figure 3-8
Detail One-bit Synchronous Circuit with Non-inverted Qout

```
module detail2
        d2       device  'E0320';
        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 design would result in the same circuit and E0320 fuse pattern previously illustrated in Figure 3-6.

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.

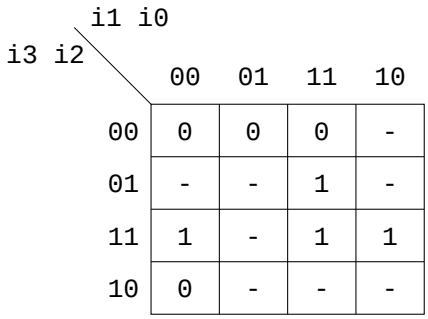
Using output f3, for example, the eight input conditions can be listed as on-sets and off-sets as follows (maintaining the ordering of inputs as specified in the truth table above):

```
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):



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 in Figure 3-9.

Figure 3-9
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. See example **octalf.abl**.

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.*

Using XORs for Flip-flop Emulation

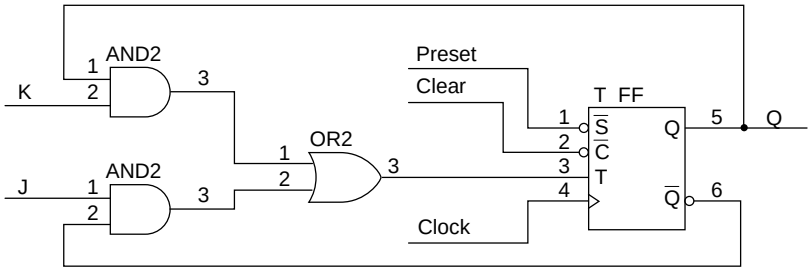
Another way to use XOR gates is for flip-flop emulation. If you are using an XOR device that has outputs featuring an XOR gate and D-type flip-flops, you can write your design as if you were going to be implementing it in a device with T-type flip-flops. The XOR gates and D-type flip-flops emulate the specified T-type flip-flops. When using XORs in this way, you should not use the 'xor' attribute for output signals unless the target device has XOR gates.

JK Flip-Flop Emulation

You can emulate JK flip-flops using a variety of circuitry found in programmable devices. When a T-type flip-flop is available, you can emulate JK flip-flops by ANDing the Q output of the flip-flop with the K input. The !Q output is then ANDed with the J input. This specific approach is useful in devices such as the Intel / Altera E0600 and E0900.

Figure 3-10 illustrates the circuitry and the Boolean expression.

Figure 3-10
JK Flip-flop Emulation Using T Flip-flop

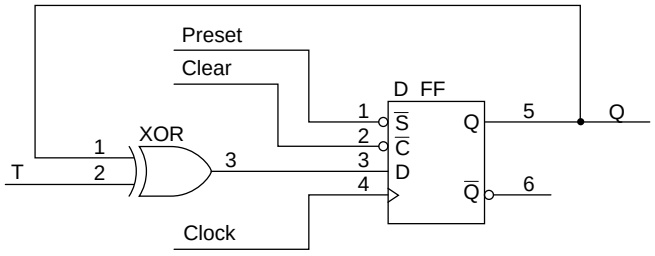


Q : = (J & !Q) # (K & Q)

0777-1

You can emulate a JK flip-flop with a D flip-flop and an XOR gate. This technique is useful in devices such as the P20X8. The circuitry and Boolean expression is shown below in Figure 3-11.

Figure 3-11
T Flip-flop Emulation Using D Flip-flop

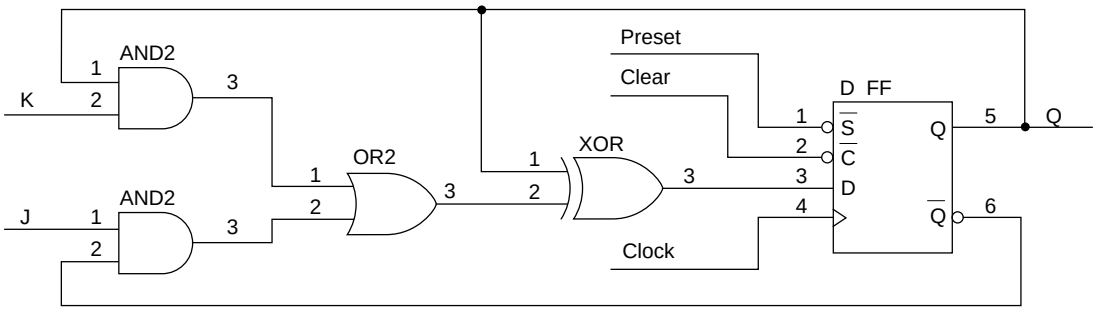


Q : = T $ Q

0755-1

Finally, you can also emulate a JK flip-flop by combining the D flip-flop emulation of a T flip-flop, in Figure 3-11, with the circuitry of Figure 3-10. Figure 3-12 illustrates this concept.

Figure 3-12
JK Flip-flop Emulation, D Flip-flop with XOR



Q := (Q) $ (J & !Q # K & Q)

Integrated Circuit in Digital Electronics
Arpad Barna and Dan Porat
John Whiley & Sons 1973

0756-1

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
- Powerup 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 Figure 3-13, which lists the 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.

Figure 3-13
Using Identifiers for States

```
module Sequence
title 'State machine example  D. B. Pellerin  Data I/O Corp';

   sequence        device  'p16r4';

   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
```

Powerup Register States

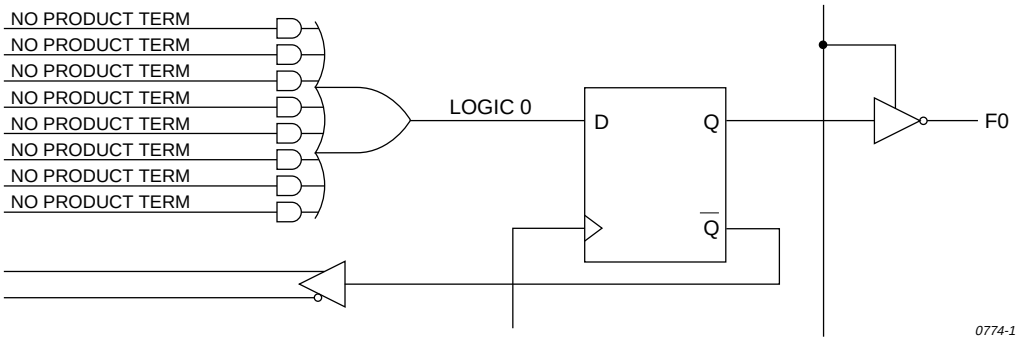
If a state machinehas to have a specific starting state, you must define the register powerup state in the state diagram description or make sure your design goes to a known state at powerup. Otherwise, the next state is undefined.

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 Figure 3-14, 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 3-14
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 in Figure 3-15.

Figure 3-15
State Machine Description with Conflicting Logic

```
module TRAFFIC
title 'Traffic Signal Controller Kim-Fu Lim  Data I/O Corp'

        traffic device 'F167';
        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]);
```

Figure 3-16
@DCSET-compatible State Machine Description

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 incompatable. 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 in Figure 3-16.

```
module TRAFFIC1
title 'Traffic Signal Controller, M. McClure Data I/O Corp'

        traffic1        device 'F167';

        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.

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:

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

Figure 3-17
Symbolic State Description

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 in Figure 3-17.

```
module SM
  a,b,clock         pin;                  " inputs
  a_reset,s_reset   pin;                  " reset inputs
  x,y               pin istype '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.

Using Complement Arrays

The complement array is a unique feature found in some logic sequencers. This section shows a typical use ending counter sequence.

You can use transition equations to express the design of counters and state machines in some devices with JK or SR flip-flops. A transition equation expresses a state of the circuit as a variation of, or adjustment to, the previous state. This type of equation eliminates the need to specify every node of the circuit; you can specify only those that require a transition to the opposite state.

An example of transition equations is shown in Figure 3-18, a source file for a decade counter having a single (clock) input and a single latched output. This counter divides the clock input by a factor of ten and generates a 50% duty-cycle squarewave output. The device used is an F105 FPLS. In addition to its registered outputs, this device contains a set of "buried" (or feedback) registers whose outputs are fed back to the product term inputs. These nodes must be declared, and can be given any names.

Node 49, the complement array feedback, is declared (as COMP) so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero if an illegal counter state is encountered. Any illegal state (and also state 9) will result in the absence of an active product term to hold node 49 at a logic low. When node 49 is low, Figure 3-19 shows that product term 9 resets each of the feedback registers so the counter is set to state zero. (To simplify the following description of the equations in Figure 3-18, node 49 and the complement array feedback are temporarily ignored.)

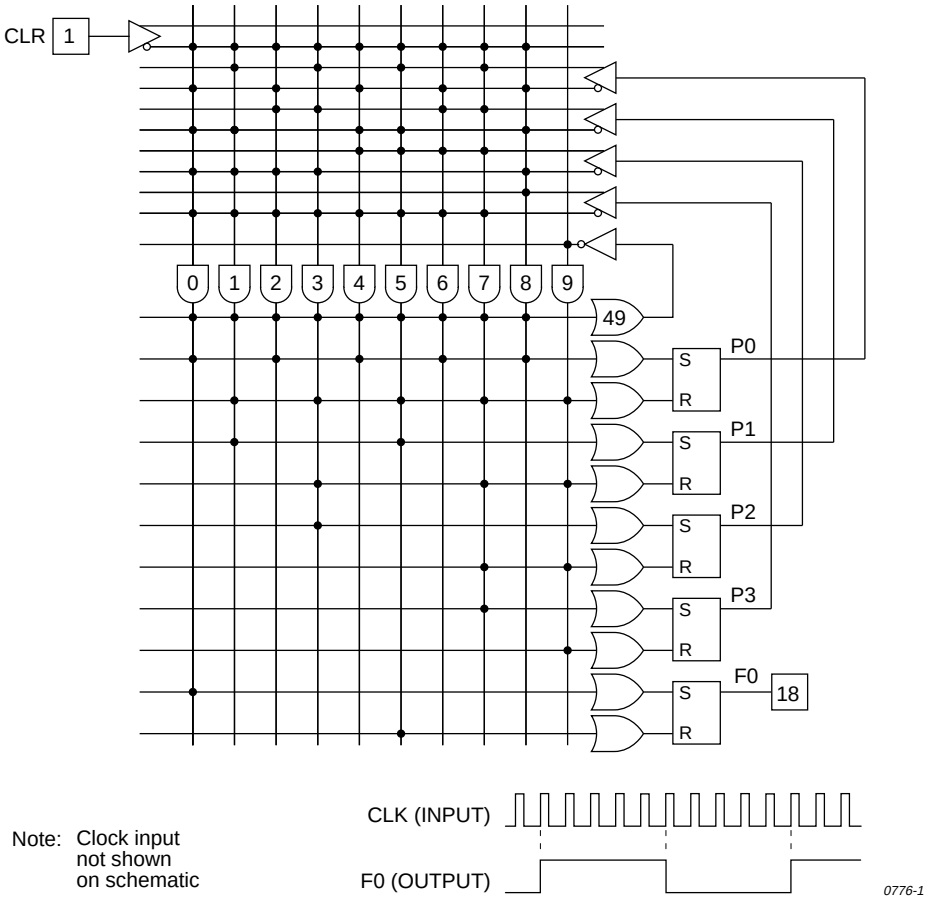
Figure 3-18
Transition Equations for a Decade Counter

The first equation states that the F0 (output) register is set (to provide the counter output) and the P0 register is set when registers P0, P1, P2, and P3 are all reset (counter at state zero) and the clear input is low. Figure 3-19 shows how the fuses are blown to fulfill this equation; the complemented outputs of the registers (with the clear input low) form product term 0. Product term 0 sets register P0 to increment the decade counter to state 1, and sets register F0 to provide an output at pin 18.

```
module DECADE
title 'Decade Counter   Uses Complement Array
Michael Holley   Data I/O Corp'

        decade          device 'F105';

        Clk,Clr,F0,PR   pin  1,8,18,19;
        P3..P0     node 40..37;
        COMP            node 49;

        F0,P3..P0  istype 'reg_sr,buffer';

        _State          = [P3,P2,P1,P0];
        H,L,Ck,X        = 1, 0, .C., .X.;

equations
    [P3,P2,P1,P0,F0].ap  = PR;
    [F0,P3,P2,P1,P0].clk = Clk;

 "Output          Next State             Present State        Input
 [F0.S, COMP,            P0.S] = !P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "0 to 1
 [     COMP,        P1.S,P0.R] = !P3.Q & !P2.Q & !P1.Q &  P0.Q & !Clr; "1 to 2
 [     COMP,            P0.S] = !P3.Q & !P2.Q &  P1.Q & !P0.Q & !Clr; "2 to 3
 [     COMP,   P2.S,P1.R,P0.R] = !P3.Q & !P2.Q &  P1.Q &  P0.Q & !Clr; "3 to 4
 [     COMP,            P0.S] = !P3.Q &  P2.Q & !P1.Q & !P0.Q & !Clr; "4 to 5
 [F0.R, COMP,       P1.S,P0.R] = !P3.Q &  P2.Q & !P1.Q &  P0.Q & !Clr; "5 to 6
 [     COMP,            P0.S] = !P3.Q &  P2.Q &  P1.Q & !P0.Q & !Clr; "6 to 7
 [   COMP,P3.S,P2.R,P1.R,P0.R] = !P3.Q &  P2.Q &  P1.Q &  P0.Q & !Clr; "7 to 8
 [   COMP                P0.S] =  P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "8 to 9
 [        P3.R,P2.R,P1.R,P0.R] =                                 !COMP; "Clear

"After Preset, clocking is inhibited until High-to-Low clock transition.
test_vectors   ([Clk,PR,Clr] -> [_State,F0  ])
                [ 0 , 0, 0 ] -> [   X  , X];
                [ 1 , 1, 0 ] -> [^b1111, H]; " Preset high
                [ 1 , 0, 0 ] -> [^b1111, H]; " Preset low
                [ Ck, 0, 0 ] -> [   0  , H]; " COMP forces to State 0
                [ Ck, 0, 0 ] -> [   1  , H];
"               ..vectors edited...
                [ Ck, 0, 1 ] -> [   0  , H]; " Clear
end
```

The second equation performs a transition from state 1 to state 2 by setting the P1 register and resetting the P0 register. (The .R dot extension is used to define the reset input of the registers.) In state 2, the F0 register remains set, maintaining the high output. The third equation again sets the P0 register to achieve state 3 (P0 and P1 both set), while the fourth equation resets P0 and P1, and sets P2 for state 4, and so on.

Figure 3-19
Abbreviated F105 Schematic



Wraparound of the counter from state 9 to state 0 is achieved by means of the complement array node (node 49). The last equation defines state 0 (P3, P2, P1, and P0 all reset) as equal to !COMP, that is, node 49 at a logic low. When this equation is processed, the fuses are blown as indicated in Figure 3-19. Figure 3-19 shows that state 9 (P0 and P3 set) provides no product term to pull node 49 high. As a result, the !COMP signal is true to generate product term 9 and reset all the "buried" registers to zero.

Chapter 4
Designing with FPGAs

The information in this chapter is only applicable to Syanario. ABEL-HDL allows you to generate source files with efficient logic for FPGAs, including Logic Cell Arrays (LCAs), ACT1, and HIPER devices.

FPGA Design Strategies

The following design strategies are helpful when designing for FPGAs. You will find more detailed information in later sections.

- Define external and internal signals with **pin** and **node** statements, respectively.
- For state machines and truth tables, include **@DCSET** (or 'dc' attributes) if possible, since it usually reduces logic.
- Use only dot extensions that are appropriate for FPGA designs. You can find information about using dot extensions in the specific FPGA fitter user manuals.
- Use intermediate signals to create multi-level logic to match FPGA architectures.

Declaring Signals

The first step in creating a logic module for an FPGA is to declare the signals in your design. In ABEL-HDL, you do this with pin and node statements.

Pin — Pin Statements indicate external signals (used as inputs and outputs to the functional block). Pin numbers are optional in ABEL-HDL, and are not recommended for FPGAs, since pin statements don't actually generate pins on the device package. If you declare an external signal as a node instead of a pin, the device fitter may interpret the signal incorrectly and delete it.

Node — Node Statements indicate internal signals (not accessible by circuitry outside the functional block). Signals declared as nodes are expected to have a source and loads.

For example, Figure 4-1 shows a state machine as a functional block. State bits S1 through S7 are completely internal; all other signals are external.

Figure 4–1
Hypothetical State Machine as a Functional Block

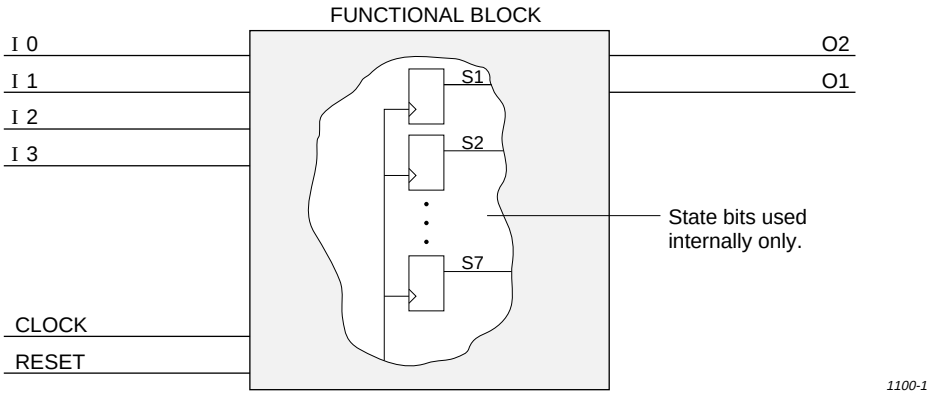


Figure 4-2 shows the corresponding signal declarations. The CLOCK, RESET, input, and output signals must connect with circuitry outside the functional block, so they are declared as pins. The state bits are not used outside the functional block, so they are declared as nodes.

Figure 4-2
Signal Declarations

```
CLOCK, RESET              Pin;
I0,I1,I2,I3               Pin;
O1,O2                     Pin;

S7,S6,S5,S4,S3,S2,S1      Node;
```

Using Intermediate Signals

An intermediate signal is a combinatorial signal that is declared as a node and used as a component of other more complex signals in a design. Intermediate signals minimize logic by forcing it to be factored. Creating intermediate signals in an ABEL-HDL logic description has the following benefits:

- Reduces the amount of optimization a device fitter has to perform
- Increases the chances of a fit
- Simplifies the ABEL-HDL source file

Figure 4-4 shows a schematic of combinational logic. Signals A, B, C, D, and E are inputs; X and Y are outputs. There are no intermediate signals; every declared signal is an input or an output to the subcircuit.

Figure 4-3 shows the ABEL-HDL declarations and equations that would generate the logic shown in Figure 4-4.

Figure 4-3
Declarations and Equations

```
"declarations
      A, B, C, D, E  pin;
        X, Y         pin;

equations
        X = (A&B&C) # (B$C);
        Y = (A&D) # (A&E) # (A&B&C);
```

Figure 4-4
Schematic without Intermediate Signal

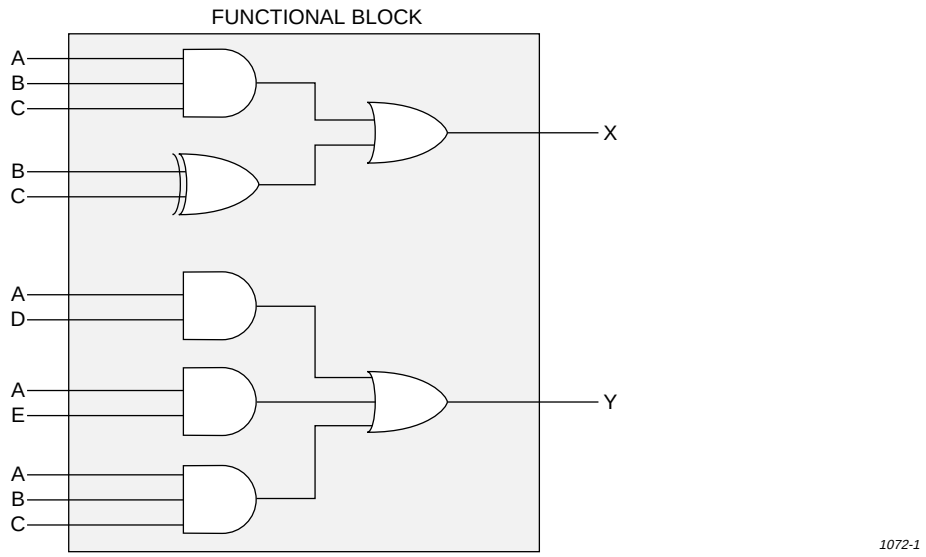


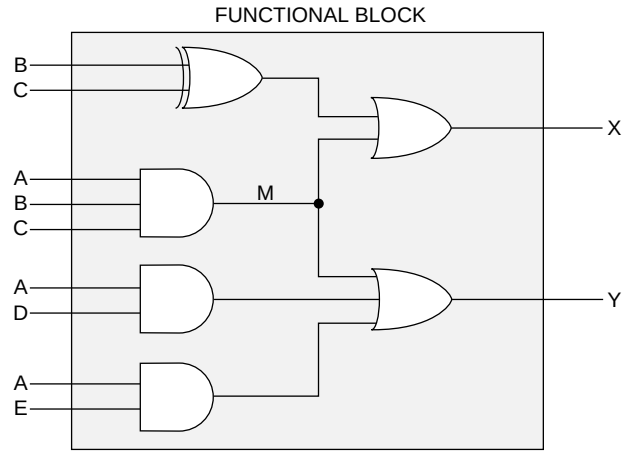
Figure 4-6 shows the same logic using an intermediate signal, M, which is declared as a node and named, but is used only inside the subcircuit as a component of other, more complex signals.

Figure 4-5 shows the declarations and equations that would generate the logic shown in Figure 4-6.

Figure 4-5
Declarations and Equations

```
"declarations
      A, B, C, D, E  pin;
        X, Y         pin;
        M            node;

equations
"intermediate signal equations
        M = A&B&C;
        X = M # (B$C);
        Y = (A&D) # (A&E) # M;
```

Figure 4-6
Schematic with Intermediate Signal M



Both design descriptions are functionally the same. Without the intermediate signal, compilation generates the AND gate associated with A&B&C twice, and the device fitter must filter out the common term. With the intermediate signal, this sub-signal is generated only once as the intermediate signal, M, and the fitter has less to do.

Using intermediate signals in a large design, targeted for a complex PLD or FPGA, can save fitter optimization effort and time. It also makes the design description easier to interpret. As another example, compare the state machine descriptions in Figures 4-7 and 4-8. Note that Figure 4-8 is easier to read.

Figure 4-7
State Machine Description without Intermediate Signals

```
CASE
which_code_enter==from_disarmed_ready:

     CASE

     (sens_code==sens_off) & (key_code!=key_pound)
                           & (key_code!=key_star)
                           & (key_code!=key_none):
     code_entry_?X  WITH {
                    which_code_enter := which_code_enter; }

     (key_code==sens_off) & (key_code==key_none):
     code_entry_?Y  WITH {
                    which_code_enter := which_code_enter; }

     (key_code==key_pound) # (key_code==key_star):
     error;

     (sens_code!=sens_off):
     error;

     ENDCASE

which_code_enter==from_armed:

     CASE

     (key_code!=key_pound)
     & (key_code!=key_star)
     & (key_code!=key_none):
     code_entry_?X  WITH {
                    which_code_enter := which_code_enter; }

     ((key_code==key_pound) # (key_code==key_star)):
     armed          WITH {
                    which_code_enter := which_code_enter; }

     (key_code==key_none):
     code_entry_?Y  WITH {
                    which_code_enter := which_code_enter; }
     ENDCASE

ENDCASE
```

Figure 4-8
State Machine Description with Intermediate Signals

```
CASE

enter_from_disarmed_ready:

     CASE

     sensors_off & key_numeric:
     code_entry_?X  WITH {
                    which_code_enter := which_code_enter; }
     sensors_off & key_none:
     code_entry_?Y  WITH {
                    which_code_enter := which_code_enter; }

     key_pound_star:
     error;

     !sensors_off:
     error;

     ENDCASE

enter_from_armed:

     CASE

     key_numeric:
     code_entry_?X  WITH {
                    which_code_enter := which_code_enter; }

     key_pound_star:
     armed          WITH {
                    which_code_enter := which_code_enter; }

     key_none:
     code_entry_?Y  WITH {
                    which_code_enter := which_code_enter; }

     ENDCASE

ENDCASE
```

The declarations and equations required to create the intermediate signals used in Figure 4-8 are shown in Figure 4-9.

Figure 4-9
Intermediate Signal Declarations and Equations

```
"pin and node declarations
     sens_code_0, sens_code_1,
     sens_code_2, sens_code_3      pin;

     key_code_0, key_code_1,
     key_code_2, key_code_3        pin;

     which_code_enter_0,
     which_code_enter_1,
     which_code_enter_2            node istype 'reg';


"set declarations
     which_code_enter = [which_code_enter_0..which_code_enter_2];
     sens_code = [sens_code_0..sens_code_3];
     key_code  = [key_code_0 ..key_code_3];

"code-entry sub-states
     from_disarmed_ready     = [1, 0, 0];
     from_armed              = [0, 0, 0];
     sens_off                = [0, 0, 0, 0];

"key encoding
     key_pnd   = [1, 1, 0, 0];
     key_str   = [1, 0, 1, 1];
     key_non   = [0, 0, 0, 0];


"intermediate signals
     enter_from_disarmed_ready     node;
     enter_from_armed              node;
     sensors_off                   node;
     key_numeric                   node;
     key_none                      node;
     key_pound_star                node;
```

```
equations
"intermediate equations
  enter_from_disarmed_ready =
           (which_code_enter==from_disarmed_ready);
  enter_from_armed = (which_code_enter==from_armed);
     sensors_off    = (sens_code==sens_off);
     key_numeric    = (key_code!=key_pnd)
                    & (key_code!=key_str)
                    & (key_code!=key_non);

     key_none       = (key_code==key_non);

     key_pound_star = (key_code==key_pnd)
                    # (key_code==key_str);
```

For large designs, using intermediate signals can be essential. An expression such as

IF (input==code_1) . . .

generates a product term (AND gate). If the input is 8 bits wide, so is the AND gate. If the expression above is used 10 times, the amount of logic generated will cause long run times during compilation and fitting, or may cause fitting to fail.

If you write the expression as an intermediate equation,

code_1_found node;

equations
code_1_found = (input==code_1);

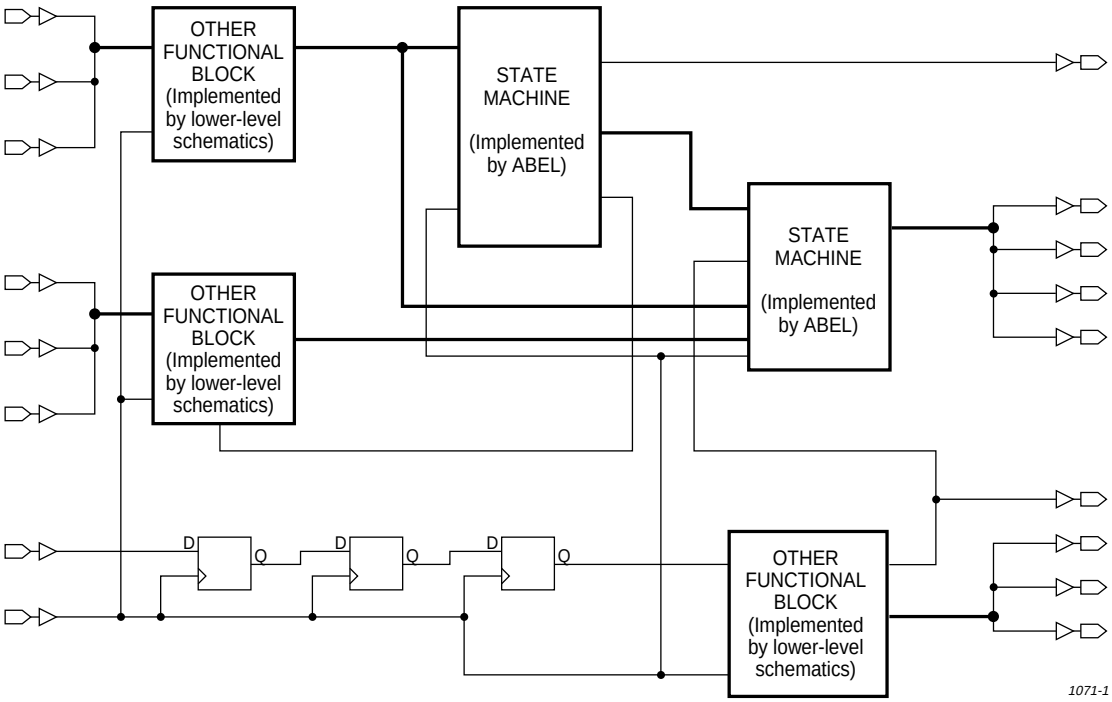
you can use the intermediate signal many times without creating an excessive amount of circuitry.

IF code_1_found . . .

Another way to create intermediate equations is to use the @CARRY directive. The @CARRY directive causes comparators and adders to be generated using intermediate equations for carry logic. This results in an efficient multilevel implementation.

You should design for multi-level FPGAs in a multi-level fashion, using intermediate signals as much as possible. An FPGA device fitter is capable of transforming two-level PLD designs into multi-level FPGA designs, but it takes a lot of time and occasionally fails. Rewriting your PLD designs to reflect the multi-level nature of the FPGA architecture often reduces the time for fitting, increases the chance of a fit, and simplifies your design descriptions.

Figure 4-10
Typical FPGA Design



Using FPGA Device Kits

This section provides information on selecting FPGA device kits and on integrating ABEL-HDL designs into larger circuits.

Integrating ABEL-HDL Designs into Larger Circuits

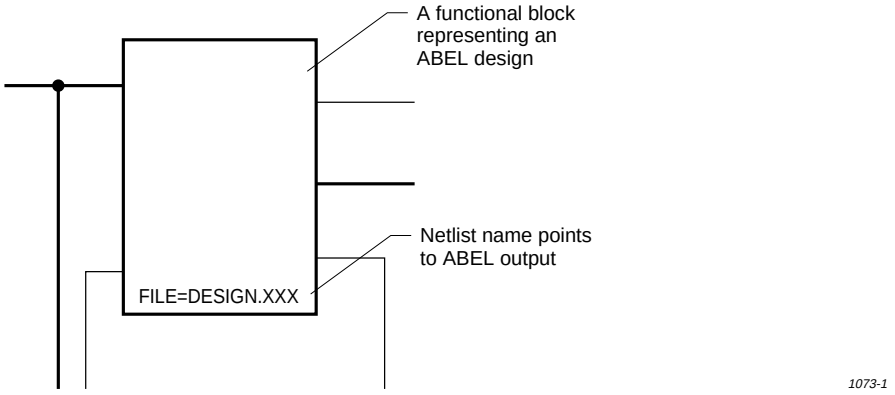
Some FPGA architectures are appropriate for behavioral entry (for example, Mach and Max). Other architectures may have portions that are appropriate for behavioral entry and portions best suited to schematic entry (for example, LCAs and ACT1 devices).

A typical FPGA design might have a top-level schematic (showing the device's pin-out and lower-level function blocks), and a collection of functional blocks. Some functional blocks point to lower-level schematics, and others point to behaviorally-described subcircuits (see Figure 4-10). If the design is large, some functional blocks may have sub-blocks.

To integrate an ABEL-HDL subcircuit into a schematic, the functional block in the higher-level drawing representing the subcircuit must point to the ABEL-HDL logic description. How that is done depends on the architecture and schematic capture system you are using, but the basic principle is similar in most cases.

To reference an ABEL-HDL logic description, label the functional blocks representing ABEL-HDL subcircuits with the name of the ABEL-HDL design (see Figure 4-11).

Figure 4-11
Functional Block Labeled with ABEL Module Name



The FPGA Device Kit manuals contain more detailed information on the kinds of subcircuits ABEL-HDL is good at implementing for specific architectures.

Chapter 5
Source File Examples

The following examples are representative of programmable logic applications and serve to illustrate significant ABEL-HDL features. You can use these examples to get started creating your own source files. For complete information on creating a source file, refer to Chapter 2, "Language Structure" and Chapter 6, "Language Reference."

All the examples in this section are installed with your software, and you can use them without making any changes, or modify them in your designs.

The examples are divided into sections that demonstrate how to use the following programmable logic applications:

- Equations
- State Diagrams
- Truth Tables
- Combined Logic Descriptions
- Hierarchy
- ABEL or Synario Projects

Equations

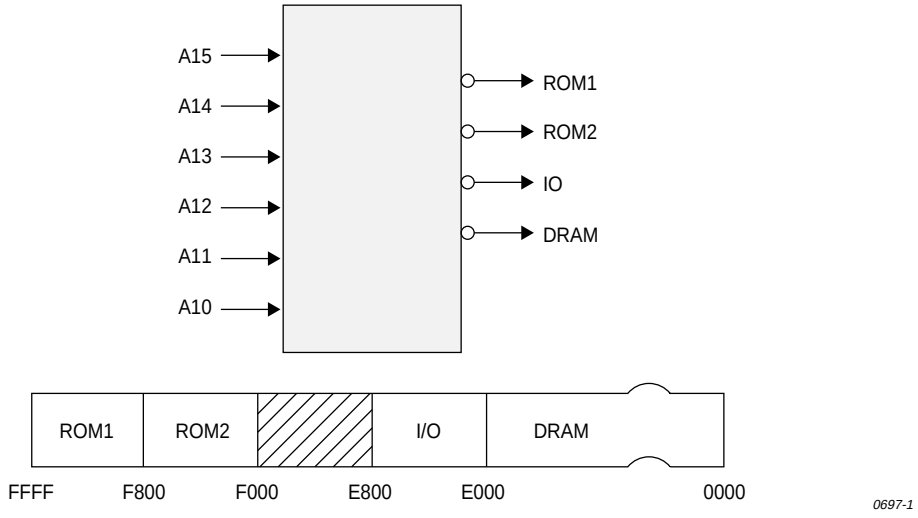
Memory Address Decoder

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL-HDL implementation of such a design.

Design Specification

Figure 5-1 shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (DRAM), I/O (IO), and two sections of ROM (ROM1 and ROM2). The purpose of this decoder is to monitor the 6 high-order bits (A15-A10) of a sixteen-bit address bus and select the correct section of memory based on the value of these address bits. To perform this function, a simple decoder with six inputs and four outputs is designed for implementation in a simple PLD.

Figure 5-1
Block Diagram: Memory Address Decoder



The address ranges associated with each section of memory are shown below. These address ranges can also be seen in the source file in Figure 5-3.

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

Design Method

Figure 5-2 shows a simplified block diagram for the address decoder. The decoder is implemented with equations employing relational and logical operators as shown in Figure 5-3.

Significant simplification is achieved by grouping the address bits into a set named Address. The ten address bits that are not used for the address decode are given no-connect values in the set, indicating that the address in the overall design (that beyond the decoder) contains 16 bits, but that bits 0 to 9 do not affect the decode of that address and are not monitored. In contrast, defining the set as

```
Address = [A15,A14,A13,A12,A11,A10]
```

ignores the existence of the lower-order bits. Specifying all 16 address lines as members of the address set allows full 16-bit comparisons of the address value against the ranges shown above.

Figure 5-2
Simplified Block Diagram: Memory Address Decoder

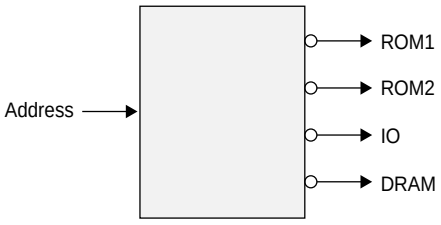


Figure 5-3
Memory Address Decoder Source File

```
module decode
title 'memory decode  Jean Designer  Data I/O Corp Redmond WA'

        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17 istype 'com';
        H,L,X   = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
        !DRAM   = (Address <= ^hDFFF);
        !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);
        !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
        !ROM1   = (Address >= ^hF800);

test_vectors
             (Address -> [ROM1,ROM2,IO,DRAM])
              ^h0000  -> [  H,  H,  H,   L ];
              ^h4000  -> [  H,  H,  H,   L ];
              ^h8000  -> [  H,  H,  H,   L ];
              ^hC000  -> [  H,  H,  H,   L ];
              ^hE000  -> [  H,  H,  L,   H ];
              ^hE800  -> [  H,  H,  H,   H ];
              ^hF000  -> [  H,  L,  H,   H ];
              ^hF800  -> [  L,  H,  H,   H ];
end
```

Test Vectors

In this design, the test vectors are a straightforward listing of the values that must appear on the output lines for specific address values. The address values are specified in hexadecimal notation.

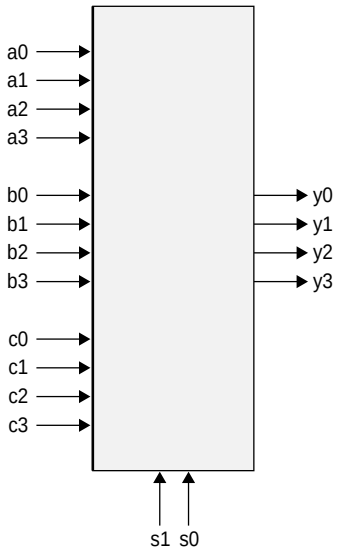
12-to-4 Multiplexer

The following describes the implementation of a 12-input to 4-output multiplexer using high level equations.

Design Specification

Figure 5-4 shows the block diagram for this design. The multiplexer selects one of the four inputs and routes that set to the output. The inputs are a0-a3, b0-b3, and c0-c3. The outputs are y0-y3. The routing of inputs to outputs is straightforward: a0 or b0 or c0 is routed to the output y0, a1 or b1 or c1 is routed to the output y1, and so on with the remaining outputs. The select lines, s0 and s1, control the decoding that determines which set is routed to the output.

Figure 5-4
Block Diagram: 12-to-4 Multiplexer



Design Method

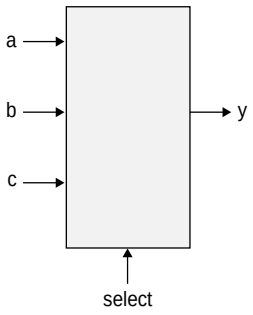
Figure 5-5 shows a block diagram for the same multiplexer after sets have been used to group the signals. All of the inputs have been grouped into the sets a, b, and c. The outputs and select lines are grouped into the sets, *y* and *select*, respectively. This grouping of signals into sets takes place in the declaration section of the source file listed in Figure 5-6.

When the sets have been declared, specification of the design is made with the following four equations that use WHEN-THEN statements.

```
when (select == 0) then y = a;
when (select == 1) then y = b;
when (select == 2) then y = c;
when (select == 3) then y = c;
```

The relational expression (==) inside the parentheses produces an expression that evaluates to true or false value, depending on the values of s0 and s1.

Figure 5-5
Simplified Block Diagram: 12-to-4 Multiplexer



0705-1

In the first equation, this expression is then ANDed with the set a which contains the four bits, a0-a3, and could be written as

```
y = (select == 0) & a
```

Assume select is equal to 0 (s1 = 0 and s0 = 0), so a true value is produced. The true is then ANDed with the set a on a bit by bit basis, which in effect sets the product term to a. If select were not equal to 0, the relational expression inside the parentheses would produce a false value. This value, when ANDed with anything, would give all zeroes.

The other product terms in the equation work in the same manner. Because select takes on only one value at a time, only one of the product terms pass the value of an input set along to the output set. The others contribute 0 bits to the ORs.

Test Vectors

The test vectors for this design are specified in terms of the input, output, and select sets. Note that the values for a set can be specified by decimal numbers and by other sets. The constants H and L, used in the test vectors, were declared as four bit sets containing all ones or all zeroes.

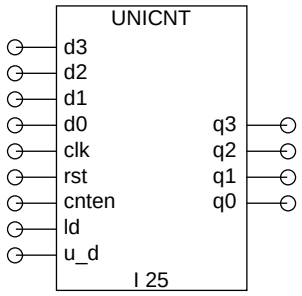
Figure 5-6
Source File: 12-to-4 Multiplexer

```
module Mux12T4
title '12 to 4 multiplexer
Dave Pellerin   Data I/O Corp.  Redmond WA'

        mux12t4         device  'P16V8S';

        a0..a3     pin     1..4;
        b0..b3     pin     5..8;
        c0..c3     pin     9..13;
        s1,s0     pin     18,19;
        y0..3     pin     14..17;

        H       =       [1,1,1,1];
        L       =       [0,0,0,0];
        X       =       .x.;
        select  =       [s1, s0];
        y       =       [y3..y0];
        a       =       [a3..a0];
        b       =       [b3..b0];
        c       =       [c3..c0];

equations
        when (select == 0) then y = a;
        when (select == 1) then y = b;
        when (select == 2) then y = c;
        when (select == 3) then y = c;

test_vectors ([select, a, b, c] -> y)
              [0     , 1, X, X] -> 1;"select = 0, gates lines a to output
              [0     ,10, H, L] -> 10;
              [0     , 5, H, L] -> 5;
              [1     , H, 3, H] -> 3;"select = 1, gates lines b to output
              [1     ,10, 7, H] -> 7;
              [1     , L,15, L] -> 15;
              [2     , L, L, 8] -> 8;"select = 2, gates lines c to output
              [2     , H, H, 9] -> 9;
              [2     , L, L, 1] -> 1;
              [3     , H, H, 0] -> 0;"select = 3, gates lines c to output
              [3     , L, L, 9] -> 9;
              [3     , H, L, 0] -> 0;
end
```

4-Bit Universal Counter

The following design describes the implementation of a 4-bit up / down counter with parallel load and count enable. The design is described using high-level ABEL-HDL equations. Figure 5-7 shows a block diagram of the counter and its signals. Figure 5-8 shows the source file for this design.

Figure 5-7
Block Diagram: 4-bit Universal Counter



2193-1

The outputs q3, q2, q1, and q0 contain the current count. The least significant bit (LSB) is q0 the most significant bit (MSB) is q3.

Using Sets to Create Modes

The counter has four different modes of operation: Load Data From Inputs, Count Up, Count Down, and Hold Count. You select the modes by applying various combinations of values to the inputs **cnten**, **ld**, and **u_d**, as described below. The four modes have different priorities, which are defined in the ABEL-HDL description.

The Load mode has the highest priority. If the **ld** input is high, then the **q** outputs reflect the value on the **d** inputs after the next clock edge.

The Hold mode has the next highest priority. Provided **ld** is low, then when the **cnten** input is low, the **q** outputs maintain their current values upon subsequent clock edges, ignoring any other inputs.

The Up and Down modes have the same priority, and by definition are mutually exclusive. Provided **cnten** is high and **ld** is low, then when **u_d** is high, the counter counts up and when **u_d** is low, the counter counts down.

Counter Reset

The counter is reset asynchronously by assertion of the input **rst**.

Using Range Operators

Because this design uses range operators and sets, you can modify the counter to be any width by making changes in the declarations section. You could create a 9-bit counter by changing the lines which read "d3..d0" and "q3..q0" to "d8..d0" and "q8..q0," respectively. The range expressions are expanded out and create register sets of corresponding width.

Design Description

Hierarchical Interface Declaration

Directly after the module name, the design contains a hierarchical interface declaration which is used by the ABEL-HDL compiler and linker if another ABEL-HDL source instantiates this source. The interface list shows all of the input, output, and bidirectional signals (if any) in the design.

Declarations

The declarations contain sections that make the design easier to interpret. The sections are as follows:

Constants	Constant values are defined.
Inputs	Design inputs are declared.
Outputs	The output pin list contains an **istype** declaration for retargetability.
Sets	The names data and count are defined as sets (groups) containing the inputs d3, d2, d1, and d0, and the outputs q3, q2, q1, and q0, respectively.
Modes	The "Mode equations" are actually more contant declarations. First MODE is defined as the set containing **cnten**, **ld**, and **u_d**, in that order. Next, LOAD is defined as being true when the members of MODE are equal to X, 1, and X, respectively. HOLD, UP, and DOWN are defined similarly.

Equations

The design of the counter equations enables you to easily define modes and your actual register equations will be easily readable. The counter equation uses **when-then-else** syntax. The first line

```
when LOAD then count := data
```

uses the symbolic name LOAD, defined earlier in the source file as

```
LOAD = (MODE == [X, 1, X])
```

and MODE itself is a set of inputs in a particular order, defined previously as

```
MODE = [cnten, ld, u_d]
```

The first line of the equation could have been written as follows

```
when ((cnten == X) & (ld == 1) & (u_d == X)) then count := data
```

which is functionally the same, but the intermediate definitions used instead makes the source file more readable and easier to modify.

Figure 5-8
Source file: 4-bit Universal Counter

```
module unicnt
interface (d3..d0, clk,rst,ld, u_d -> q3..q0) ;

title '4 bit universal counter with parallel load
       Tom Bowns   Data I/O Corporation' ;

"Constants
    X,C,Z = .X., .C., .Z. ;

"Inputs
    d3..d0  pin ;              "Data inputs, 4 bits wide
    clk     pin ;              "Clock input
    rst     pin ;              "Asynchronous reset
    cnten   pin ;              "Count enable
    ld      pin ;              "Load counter with input data value
    u_d     pin ;              "Up/Down selector: HIGH selects up

"Outputs
    q3..q0  pin istype 'reg';  "Counter outputs

"Sets
    data  = [d3..d0];          "Data set
    count = [q3..q0];          "Counter set

"Mode equations
            MODE =  [cnten,ld,u_d];   "Mode set composed of control pins.
    LOAD = (MODE == [  X  , 1, X ]);  "Various modes are defined by
    HOLD = (MODE == [  0  , 0, X ]);  "values applied to control pins.
    UP   = (MODE == [  1  , 0, 1 ]);  "Symbolic name may be defined as
    DOWN = (MODE == [  1  , 0, 0 ]);  "a set equated to a value.

Equations
         when LOAD then count := data        "Load counter with data
    else when UP   then count := count + 1   "Count up
    else when DOWN then count := count - 1   "Count down
    else when HOLD then count := count ;     "Hold count

    count.clk = clk;                         "Counter clock input
    count.ar  = rst;                         "Counter reset input

"Test_vectors edited...

End
```

Note: *You can also see the advantages of set notation in the test vector section (which has been edited in this manual, but can be seen in the actual .abl file). In the test vectors, the input data is applied as a decimal value, and the output count is a decimal value rather than a set of binary bits.*

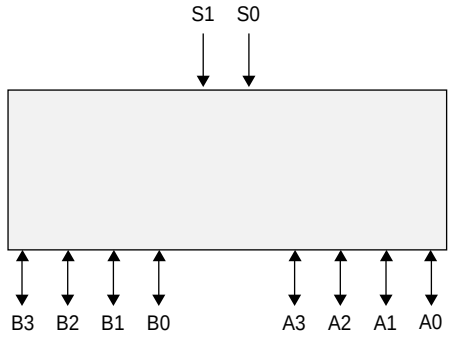
Bidirectional Three-state Buffer

A four-bit bidirectional buffer with tristate outputs is presented here. The design is implemented in an F153 FPLA with bidirectional inputs/outputs and programmable output polarity. Simple Boolean equations are used to describe the function.

Design Specification

Figure 5-9 shows a block diagram for this four-bit buffer. Signals A0-A3 and B0-B3 function both as inputs and outputs, depending on the value on the select lines, S0-S1. When the select value (the value on the select lines) is 1, A0-A3 are enabled as outputs. When the select value is 2, B0-B3 are enabled as outputs. (The choice of 1 and 2 for select values is arbitrary.) For any other values of the select lines, both the A and B outputs are at high impedance. Output polarity for this design is positive.

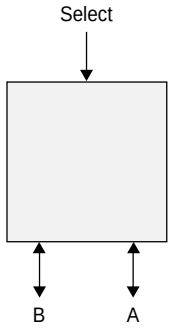
Figure 5-9
Block Diagram: Bidirectional Three-state Buffer



Design Method

A simplified block diagram for the buffer is shown in Figure 5-10. The A and B inputs/outputs are grouped into two sets, A and B. The select lines are grouped into the select set. Figure 5-11 shows the source file that describes the design.

Figure 5-10
Simplified Block Diagram: Bidirectional Three-state Buffer



0717-1

High-impedance and don't-care values are declared to simplify notation in the source file. The equations section describes the full function of the design. What appear to be unresolvable equations are written for A and B, with both sets appearing as inputs and outputs. The enable equations, however, enable only one set at a time as outputs; the other set functions as inputs to the buffer.

Test vectors are written to test the buffer when either set is selected as the output set, and for the case when neither is selected. The test vectors are written in terms of the previously declared sets so the element values do not need to be listed separately.

Figure 5-11
Source File: Bidirectional Three-state Buffer

```
module tsbuffer
title 'bidirectional three state buffer  Brenda French & Mary Bailey   Data I/O Corp'
        TSB1    device 'F153';
        S1,S0           Pin 1,2;         Select = [S1,S0];
        A3,A2,A1,A0     Pin 12,13,14,15;   A    = [A3,A2,A1,A0];
        B3,B2,B1,B0     Pin 16,17,18,19;   B    = [B3,B2,B1,B0];

        X,Z     = .X., .Z.;

equations
        A = B;
        B = A;

        A.oe = (Select == 1);
        B.oe = (Select == 2);

test_vectors
        ([Select,  A,  B]-> [  A,  B])
         [  0   ,  0,  0]-> [  Z,  Z];
         [  0   , 15, 15]-> [  Z,  Z];

         [  1   ,  X,  5]-> [  5,  X];
         [  1   ,  X, 10]-> [ 10,  X];

         [  2   ,  5,  X]-> [  X,  5];
         [  2   , 10,  X]-> [  X, 10];

         [  3   ,  0,  0]-> [  Z,  Z];
         [  3   , 15, 15]-> [  Z,  Z];
end
```

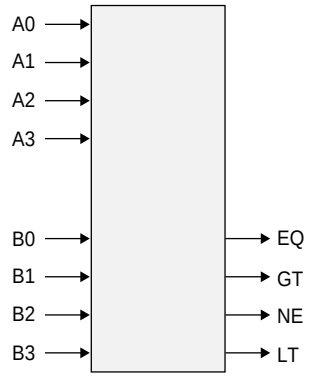
4-Bit Comparator

This is a design for a 4-bit comparator that provides an output for "equal to," "less than," "not equal to,"and "greater than" (as well as intermediate outputs). The design is implemented with high level equations.

Design Specification

The comparator, as shown in Figure 5-12, compares the values of two four-bit inputs (A0-A3 and B0-B3) and determines whether A is equal to, not equal to, less than, or greater than B. The result of the comparison is shown on the output lines, EQ, GT, NE, and LT.

Figure 5-12
Block Diagram: 4-bit Comparator



0740-2

Design Method

Figure 5-13 and Figure 5-14 show the simplified block diagram and source file listing for the comparator. The inputs A0-A3 and B0-B3 are grouped into the sets A and B. YES and NO are defined as 1 and 0, to be used in the test vectors.

The equations section of the source file contains the following equations:

```
EQ  = A == B;
NE  = !(A == B);
GT  = A > B;
LT  = !((A > B) # (A == B));
```

Figure 5-13
Simplified Block Diagram: 4-bit Comparator

You could also use the following equations for the design of this comparator. However, many more product terms are used in the FPLA:

```
EQ  = A == B;
NE  = A != B;
GT  = A > B;
LT  = A < B;
```

The first set of equations takes advantage of product term sharing within the target FPLA, while the latter set requires a different set of product terms for each equation. For example, the equation

```
NE  = !(A == B);
```

uses the same 16 product terms as the equation

```
EQ  = A == B;
```

thereby reducing the number of product terms. In a similar manner, the equation

```
LT  = !((A > B) # (A == B));
```

uses the same product terms as equations

```
EQ  = A == B;
GT  = A > B;
```

whereas the equation

```
LT  = A < B;
```

(in the second set of equations) requires the use of additional product terms. Sharing product terms in devices that allow this type of design architecture can serve to fit designs into smaller and less expensive logic devices.

Figure 5-14
Source File: 4-bit Comparator

```
module comp4a
title '4-bit look-ahead comparator
Steve Weil & Gary Thomas  Data I/O Corp.'

      comp4a  device 'F153';
   A3..A0         pin  1..4;
   A              =    [A3..A0];
   B3..B0         pin  5..8;
   B              =    [B3..B0];

   NE,EQ,GT,LT    pin 16..19 istype 'com';

   No,Yes  = 0,1;

equations
   EQ  =   A == B;
   NE  = !(A == B);
   GT  =   A > B;
   LT  = !((A > B) # (A == B));

" test_vectors deleted...

end
```

Test Vectors

Three separate test vectors sections are written to test three of the four possible conditions. (The fourth and untested condition of NOT EQUAL TO is simply the inverse of EQUAL TO.) Each test vectors table includes a test vector message that helps make report output from the compiler and the simulators easier to read.

The three tested conditions are not mutually exclusive, so one or more of them can be met by a given A and B. In the test vectors table, the constants YES and NO (rather than 1 and 0) are used for for ease of reading. YES and NO are declared in the declaration section of the source file.

Truth Table Examples

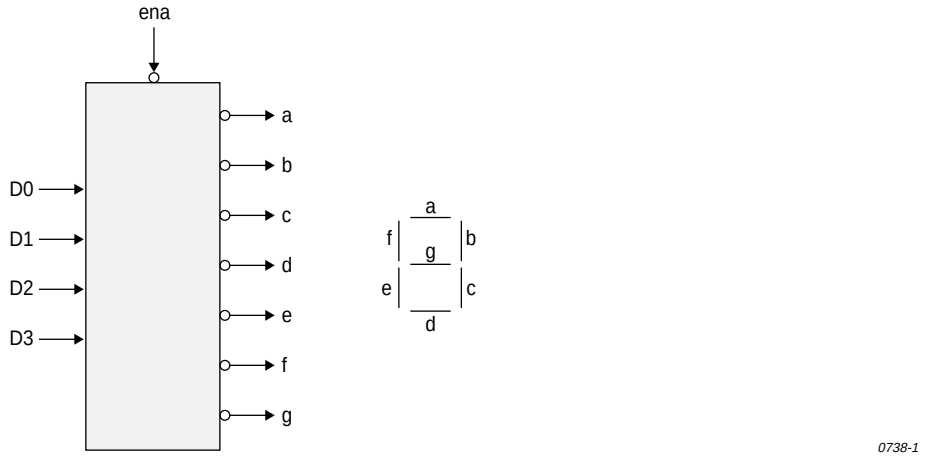
Seven-segment Display Decoder

This display decoder decodes a four-bit binary number to display the decimal equivalent on a seven-segment LED display. The design incorporates a truth table.

Design Specification

Figure 5-15 shows a block diagram for the design of a seven-segment display decoder and a drawing of the display with each of the seven segments labeled to correspond to the decoder outputs. To light a segment, the corresponding line must be driven low. Four input lines D0-D3 are decoded to drive the correct output lines. The outputs are named a, b, c, d, e, f, and g corresponding to the display segments. All outputs are active low. An enable, ena, is provided. When ena is low, the decoder is enabled; when ena is high, all outputs are driven to high impedance.

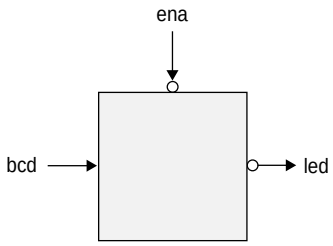
Figure 5-15
Block Diagram: Seven-segment Display Decoder



Design Method

Figure 5-16 and Figure 5-17 show the simplified block diagram and the source file for the ABEL-HDL implementation of the display decoder. The binary inputs and the decoded outputs are grouped into the sets **bcd** and **led**. The constants ON and OFF are declared so the design can be described in terms of turning a segment on or off. To turn a segment on, the appropriate line must be driven low, thus we declare ON as 0 and OFF as 1.

Figure 5-16
Simplified Block Diagram: Seven-segment Display Decoder



The design is described in two sections, an equations section and a truth table section. The decoding function is described with a truth table that specifies the outputs required for each combination of inputs. The truth table header names the inputs and outputs. In this example, the inputs are contained in the set named bcd and the outputs are in led. The body of the truth table defines the input to output function.

Because the design decodes a number to a seven segment display, values for bcd are expressed as decimal numbers, and values for led are expressed with the constants ON and OFF that were defined in the declarations section of the source file. This makes the truth table easy to read and understand; the incoming value is a number and the outputs are on and off signals to the LED.

The input and output values could have just as easily been described in another form. Take for example the line in the truth table:

```
5 -> [ ON, OFF, ON , ON, OFF, ON, ON]
```

This could have been written in the equivalent form:

```
[ 0, 1, 0, 1 ] -> 36
```

In this second form, 5 was simply expressed as a set containing binary values, and the LED set was converted to decimal. (Remember that ON was defined as 0 and OFF was defined as 1.) Either form is supported, but the first is more appropriate for this design. The first form can be read as, "the number five turns on the first segment, turns off the second, . . ." whereas the second form cannot be so easily translated into meaningful terms.

Figure 5-17
Source file: 4-bit Counter with 2-input Mux

```
module  bcd7
title 'seven segment display decoder    1 Aug 1990
Walter Bright  Data I/O Corp  Redmond WA'
"          a
"         ---       BCD-to-seven-segment decoder similar to the 7449
"       f| g |b
"         ---           segment identification
"       e| d |c
"         ---
        bcd7 device  'P16P8';

        D3,D2,D1,D0,Ena pin 2,3,4,5,6;
        a,b,c,d,e,f,g   pin 13,14,15,16,17,18,19 istype 'com';

        bcd     = [D3,D2,D1,D0];
        led     = [a,b,c,d,e,f,g];

        ON,OFF  = 0,1;                  " for common anode LEDs
        L,H,X,Z = 0,1,.X.,.Z.;

equations
        led.oe = !Ena;
@dcset
truth_table (bcd -> [ a ,  b ,  c ,  d ,  e ,  f ,  g ])
              0  -> [ ON,  ON,  ON,  ON,  ON,  ON, OFF];
              1  -> [OFF,  ON,  ON, OFF, OFF, OFF, OFF];
              2  -> [ ON,  ON, OFF,  ON,  ON, OFF,  ON];
              3  -> [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
              4  -> [OFF,  ON,  ON, OFF, OFF,  ON,  ON];
              5  -> [ ON, OFF,  ON,  ON, OFF,  ON,  ON];
              6  -> [ ON, OFF,  ON,  ON,  ON,  ON,  ON];
              7  -> [ ON,  ON,  ON, OFF, OFF, OFF, OFF];
              8  -> [ ON,  ON,  ON,  ON,  ON,  ON,  ON];
              9  -> [ ON,  ON,  ON,  ON, OFF,  ON,  ON];
" test_vectors edited
end
```

Test Vectors

The test vectors for this design test the decoder outputs for the ten valid combinations of input bits. The enable is also tested by setting ena high for the different combinations. All outputs should be at high impedance whenever ena is high.

State Diagram Examples

Three-state Sequencer

The following design is a simple sequencer that demonstrates the use of ABEL-HDL state diagrams. The design is implemented in a P16R4 device. The number of State Diagram states that can be processed depends on the number of transitions and the path of the transitions. For example, a 64-state counter uses fewer terms (and smaller equations) than a 63-state counter. For large counter designs, use the syntax CountA:= CountA + 1 to create a counter rather than using a state machine. See also example COUNT116.abl for further information on counter implementation.

Design Specification

Figure 5-18 shows the sequencer design with a state diagram that shows the transitions and desired outputs. The state machine starts in state A and remains in that state until the 'start' input becomes high. It then sequences from state A to state B, from state B to state C, and back to state A. It remains in state A until the 'start' input is high again. If the 'reset' input is high, the state machine returns to state A at the next clock cycle. If this reset to state A occurs during state B, a 'halt' synchronous output goes high, and remains high until the machine is again started.

During states B and C, asynchronous outputs 'in_B' and 'in_C' go high to indicate the current state. Activation of the 'hold' input will cause the machine to hold in state B or C until 'hold' is no longer high, or 'reset' goes high.

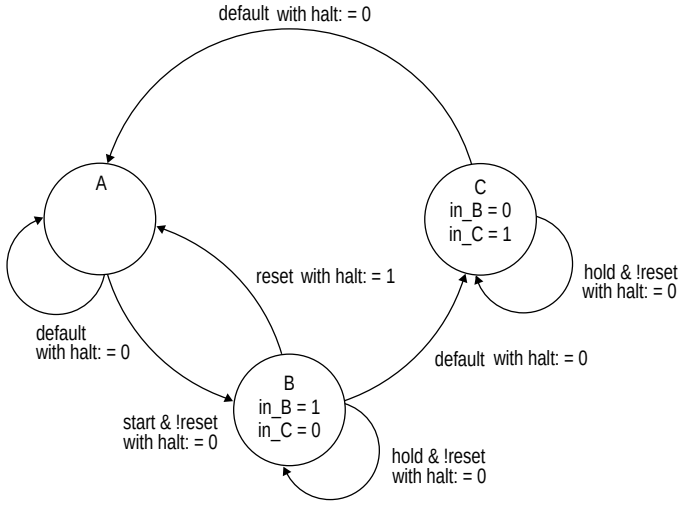
Design Method

The sequencer is described by using a STATE_DIAGRAM section in the source file. Figure 5-19 shows the source file for the sequencer. In the source file, the design is given a title, the device type is specified, and pin declarations are made. Constants are declared to simplify the state diagram notation. The two state registers are grouped into a set called 'sreg' and the three states (A, B, and C) are declared, with appropriate values specified for each.

The state values chosen for this design allow the use of register preload to ensure that the machine starts in state A. For larger state machines with more state bits, careful numbering of states can dramatically reduce the logic required to implement the design. Using constant declarations to specify state values saves time when you make changes to these values.

The state diagram begins with the STATE_DIAGRAM statement that names the set of signals to use for the state register. In this example, 'sreg' is the set of signals to use.

Figure 5-18
State Diagram: Three-state Sequencer



Within the STATE_DIAGRAM, IF-THEN-ELSE statements are used to indicate the transitions between states and the input conditions that cause each transition. In addition, equations are written in each state that indicate the required outputs for each state or transition.

For example, state A reads:

```
State A:
  in = 0;
  in_C = 0;
  if (start & !reset) then B with
halt := 0;
  else A with halt := halt;
```

This means that if the machine is in state A, and **start** is high but **reset** is low, it advances to state **B**. In any other input condition, it remains in state **A**.

The equations for **in_B** and **in_C** indicate those outputs should remain low while the machine is in state **A**. The equations for **halt,** specified with the **with** keyword, indicate that **halt** should go low if the machine transitions to state **B**, but should remain at its previous value if the machine stays in state **A**.

Test Vectors

The specification of the test vectors for this design is similar to other synchronous designs. The first vector is a preload vector, to put the machine into a known state (state A), and the following vectors exercise the functions of the machine. The A, B, and C constants are used in the vectors to indicate the value of the current state, improving the readability of the vectors.

Figure 5-19
Source File: Three-state Sequencer

```
module sequence
title 'State machine example  D. B. Pellerin  Data I/O Corp';

        sequence        device  'p16r4';

        q1,q0                         pin   14,15 istype 'reg,invert';
        clock,enab,start,hold,reset   pin   1,11,4,2,3;
        halt                          pin   17;
        in_B,in_C                     pin   12,13;
        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 edited...
end
```

Combined Logic Descriptions

This section contains an advanced logic design and builds on examples and concepts presented in the earlier sections of this manual. This design, a blackjack machine, is the combination of more than one basic logic design. Design specification, methods, and complete source files are given for all parts of the blackjack machine example, which contains the following logic designs:

- Multiplexer
- 5-bit adder
- Binary to BCD converter
- State machine

This example is a classic blackjack machine based on C.R. Clare's design in Designing Logic Systems Using State Machines (McGraw Hill, 1972). The blackjack machine plays the dealer's hand, using typical dealer strategies to decide, after each round of play, whether to draw another card or to stand.

The blackjack machine consists of these functions: a card reader that reads each card as it is drawn, control logic that tells it how to play each hand (based on the total point value of the cards currently held), and display logic that displays scores and status on the machine's four LEDs. For this example, we are assuming that the two digital display devices used to display the score have built-in seven-segment decoders.

To operate the machine, insert the dealer's card into the card reader. The machine reads the value and, in the case of later card draws, adds it to the values of previously read cards for that hand. (Face cards are valued at 10 points, non-face cards are worth their face value, and aces are counted as either 1 or 11, whichever count yields the best hand.) If the point total is 16 or less, the GT16 line will be asserted (active low) and the Hit LED will light up. This indicates that the dealer should draw another card. If the point total is greater than 16 but less than 22, no LEDs will light up (indicating that the dealer should draw no new cards). If the point total is 22 or higher, LT22 will be asserted (active low) and the Bust LED will light (indicating that the dealer has lost the hand).

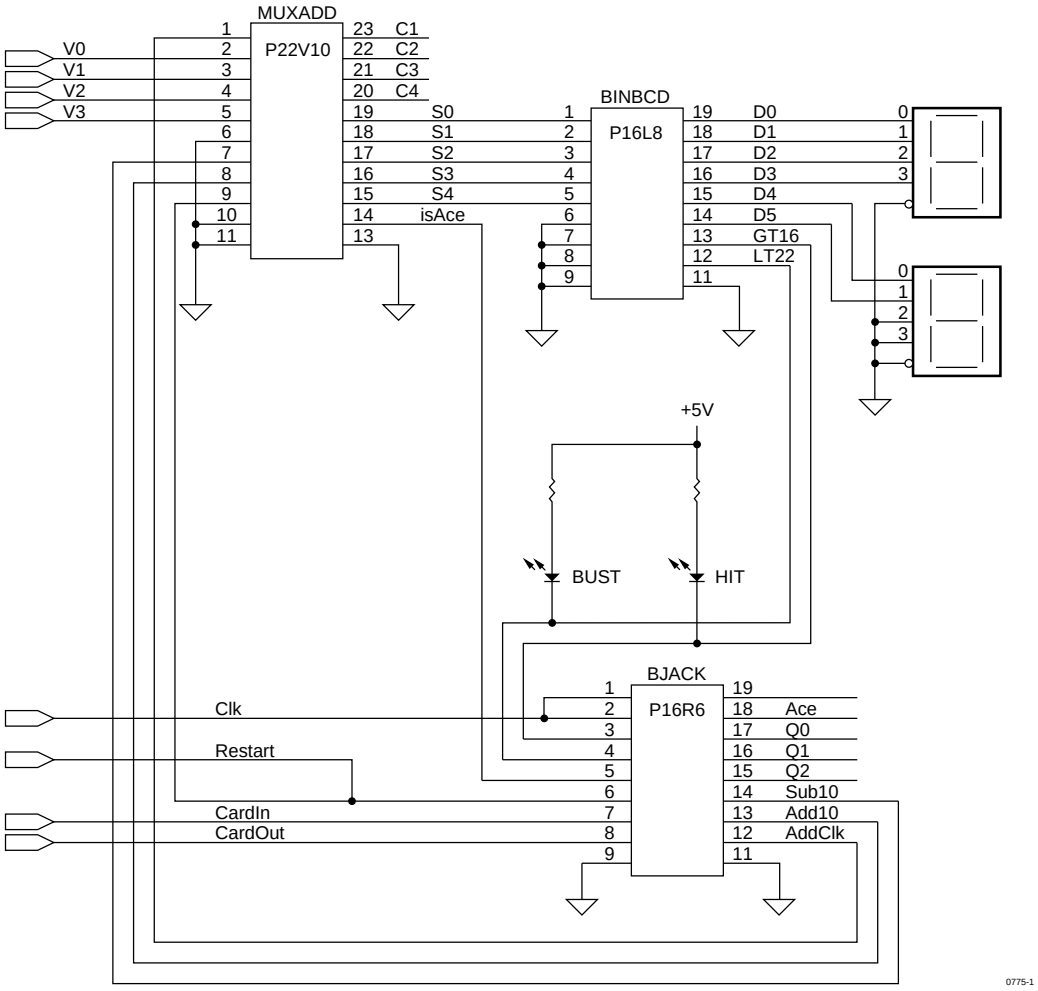
As Figure 5-20 shows, the blackjack machine is implemented in three PLDs:

1. A multiplexer-adder-comparator, which adds the value of the newly drawn card to the existing hand (and indicates an ace to the state machine);

2. A binary to binary-coded-decimal (BCD) converter, which takes in the five-bit binary score and converts it to two-digits of BCD for the digital display.

3. The blackjack controller (a state machine that contains the game logic). This logic includes instructions that determine when to add a card value, when to count an ace as 1, and when to count an ace as 11.

Figure 5-20
Schematic of a Blackjack Machine Implemented in Three PLDs



Circuits that are a straightforward function of a set of inputs and outputs are often most easily expressed in equations; the adder is such a circuit. The PLD for the adder function (identified as MUXADD in Figure 5-20) includes three elements: a multiplexer, the adder itself, and a comparator.

The multiplexer selects either the value of the newly dealt card or one of the two fixed values used for the ace (ADD10 or SUB10). The adder adds the value selected by the multiplexer to the previous score when triggered by the clock signal, ADDCLK. The comparator detects when an ace is present and passes this information on to the blackjack controller, BJACK.

Outputs that do not follow a specific pattern are most easily expressed as truth tables. This is the case with the binary-to-BCD converter that is identified in the schematic (Figure 5-20) as BINBCD. This PLD converts five bits of binary input to BCD output for two digital display elements.

The following text describes the internal logic design necessary to keep the card count, to control the play sequence, and to show the count on the digital display (or the state on the Hit and Bust LEDs). Neither the card reader nor the physical design is discussed here. Assume that the card reader provides a binary value that is representative of the card read.

The design has eight inputs (four of which are the binary encoded card values, V0-V3). The remaining four inputs are signals that indicate the following:

- Restart (the machine is to be restarted)
- CardIn (a card is in the reader)
- CardOut (no card is in the reader)
- CLK (a clock signal to synchronize the design to the card reader)

CardIn, CardOut, and Clk are provided by the card reader. Restart is provided by a switch on the exterior of the machine.

Device	Function in the Blackjack Machine
P22V10	Multiplexer / Adder / Comparator
P16L8	Binary-BCD converter
P16R6	State machine

Design Specification — MUXADD

MUXADD consists of an input multiplexer, an adder, and a comparator. The multiplexer determines what value is added to the current score (by the adder) The added value consists of the contents of the external card reader (V0-V1 declared as Card), a numeric value of +10, or a numeric value of -10.

The inputs Add10 and Sub10 from the controller (state machine) BJACK determine which of the three values the multiplexer selects for application to the adder. Card is applied to the adder when Add10 and Sub10 are active high, as generated by the BJACK controller. When Add10 becomes active low, 10 is added to the current score (to count an ace as 11 instead of 1), and when Sub10 is active low, -10 is added to the current score (to count an ace as 1 instead of 11).

The adder provides an output named Score (S0-S4) which is the sum of the current adder contents and the value selected by the input multiplexer (the card reader contents, +10, or -10). The comparator monitors the contents of the external card reader (Card) and generates an output, is_Ace, to the BJACK controller that signifies that an ace is present in the card reader.

Design Method — MUXADD

MUXADD is implemented in a P22V10, and consists of a three-input multiplexer, a five-bit ripple adder, and a five-bit comparator. These circuit elements are defined in the equations shown in Figure 5-21. For the multiplexer inputs, a set named Card defines inputs V0 through V4 as the value of the card reader, while inputs Add10 and Sub10 are used directly in the following equations to define the multiplexer. The multiplexer output to the adder is named Data and is defined by the equations

```
Data     =  Add10 &  Sub10 & Card
         # !Add10 &  Sub10 & ten
         #  Add10 & !Sub10 & minus_ten;
```

The adder (MUXADD) contained in the P22V10 is a five-bit binary ripple adder that adds the current input from the multiplexer to the current score, with carry. The adder is clocked by a signal (AddClk) from the BJACK controller and is described with the following equations:

```
Score    := Data $ Score.FB $ CarryIn;
CarryOut  = Data & Score.FB # (Data # Score.FB) & CarryIn;
Reset     = !Clr;
```

In the above equations, Score is the sum of Data (the card reader output, value of ten, or value of minus ten), Score (the current or last calculated score), and CarryIn (the shifted value of CarryOut, described below). The new value of Score appears at the S0 through S4 outputs of MUXADD at the time of the AddClk pulse generated by the BJACK controller (state machine).

Before the occurrence of the AddClk clock pulse, an intermediate adder output appears at combinatorial outputs of the P22V10, labeled C0 through C4 and defined as the set named CarryOut (shown below). A second set named CarryIn defines the same combinatorial outputs as CarryOut, but the outputs are shifted one bit to the left, as shown below.

```
CarryIn   = [C4..C1, 0];
CarryOut  = [ X,C4..C1];
```

The set declarations define CarryIn as CarryOut with the required shift to the left for application back to adder input. At the time of the AddClk pulse from the BJACK controller, CarryIn is added to Score and Data by an exclusive-or operation.

The comparator portion of MUXADD is defined with

```
is_Ace = Card == 1;
```

which provides an input to the BJACK controller whenever the value provided by the card reader is 1.

Test Vectors — MUXADD

The test vectors shown in Figure 5-21 verify operation of MUXADD by first clearing the adder (so Score is zero), then adding card reader values 7 and 10. The test vectors then input an ace (1) from the card reader (Card) to produce a Score of 1 and pull the is_Ace output high. Subsequent vectors verify the -10 function of the input multiplexer and adder. The **trace** statement lets you observe the carry signals in simulation.

Figure 5-21
Source File: Multiplexer / Adder / Comparator

```
module MuxAdd
title '5-bit ripple adder with input multiplex  Michael Holley   Data I/O Corp.'

   muxadd  device  'P22V10';

   AddClk,Clr,Add10,Sub10,is_Ace   pin  1,9,8,7,14;
   V4..V0                  pin 6..2;
   S4..S0                  pin 15..19;
   C4..C1                  pin 20..23;

   X,C,L,H   = .X., .C., 0, 1;

   Card      = [V4..V0];
   Score     = [S4..S0];
   CarryIn   = [C4..C1, 0];
   CarryOut  = [ X,C4..C1];
   ten       = [ 0, 1, 0, 1, 0];
   minus_ten = [ 1, 0, 1, 1, 0];

   S4..S0 istype 'reg' ;

" Input Multiplexer
        Data     =  Add10 &  Sub10 & Card
                 # !Add10 &  Sub10 & ten
                 #  Add10 & !Sub10 & minus_ten;

equations
   Score   := Data $ Score.FB $ CarryIn;
   CarryOut = Data & Score.FB # (Data # Score.FB) & CarryIn;
   Score.ar  = !(Clr # Clr);
   Score.clk = AddClk;
   is_Ace   = Card == 1;

" test_vectors edited...
end MuxAdd
```

Design Specification — BINBCD

To display the Score, appearing at the output of MUXADD, a binary to bcd converter is implemented in a P16L8. It is the function of the converter to accept the four lines of binary data generated by MUXADD and provide two sets of binary coded decimal outputs for two bcd display devices; one to display the units of the current score, and the other to display the tens. The four-bit output bcd1 (D0-D3) contains the units of the current score, and is connected to the high-order display digit. The two-bit output bcd2 (D4 and D5) contains the tens, and is fed to the low-order display digit.

BINBCD also provides a pair of outputs to light the Bust and Hit LEDs. Bust is lit whenever Score is 22 or greater; while Hit is lit whenever Score is 16 or less.

Design Method — BINBCD

The design of BINBCD is shown in the source file of Figure 5-22. The design of the converter is easily expressed with a truth table that lists the value of Score (inputs S0 through S4 are declared as Score) for values of bcd1 and bcd2. bcd1 and bcd2 are sets that define the outputs that are fed to the two digital display devices. The truth table lists Score values up to decimal 31.

The truth table represents a method of expressing the design "manually." You could use a macro to create a truth table in the following manner:

```
clear(binary);
@repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

As indicated in Figure 5-22 (and described in "Test Vectors — BINBCD"), this macro is used to generate the test vectors for the converter. The generated *.lst file shows the truth table created from the macro.

The BINBCD design also provides the outputs LT22 and GT16 to control the Bust and Hit LEDs. A pair of equations generate an active-high LT22 signal to turn off the Bust LED when Score is less than 22, and an active-high GT16 signal to turn off the Hit LED when Score is greater than 16.

Test Vectors — BINBCD

The test vectors shown in Figure 5-22 verify operation of the LT22 and GT16 outputs of the converter by assigning various values for Score and checking for the corresponding outputs.

The test vectors for the binary to bcd converter are defined by means of the following macro:

```
test_vectors ( score - [bcd2,bcd1])
  clear(binary);
  @repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

This macro generates a test vector with the variable binary set to 0 by the macro (a) {@const ?a=0}; (in the binbcd.abl source file shown in Figure 5-22), followed by 31 vectors provided by the @repeat directive. The 31 vectors are generated by incrementing the value of the variable binary by a factor of 1 for each vector. Refer to the inc macro (a) {@const ?a=?a+1;}; line in Figure 5-22. On the output side of the test vectors, division is used to create the output for bcd2 (tens display digit), while the remainder (modulus) operator is used to create the output for bcd1 (units display digit).

Figure 5-22
Source file: 4-bit Counter with 2-input Mux

```
module BINBCD
title 'comparator and binary to bcd decoder for Blackjack Machine
Michael Holley  Data I/O Corp  '

" The 5 -bit binary (0 - 31) score is converted into two BCD outputs.
" The integer division '/' and the modulus operator '%' are used to
" extract the individual digits from the two digit score.
" 'Score % 10' will yield the 'units' and
" 'Score / 10' will yield the 'tens'
"
" The 'GT16' and 'LT22' outputs are for the state machine controller.

        binbcd  device  'P16L8';

   S4..S0  pin 5..1;
   score           = [S4..S0];
   LT22,GT16       pin 12,13 istype 'com';

   D5,D4      pin 14,15 istype 'com';
   bcd2       = [D5,D4];
   D3..D0     pin 16..19 istype 'com';
   bcd1       = [D3..D0];

" Digit separation macros
   binary          = 0;            "scratch variable
   clear   macro (a) {@const ?a=0};
   inc     macro (a) {@const ?a=?a+1;};

equations
        LT22    = (score < 22);         "Bust
        GT16    = (score > 16);         "Hit / Stand

" test_vectors edited...
```

```
truth_table ( score -> [bcd2,bcd1])
                 0  -> [  0 ,  0 ];
                 1  -> [  0 ,  1 ];
                 2  -> [  0 ,  2 ];
                 3  -> [  0 ,  3 ];
                 4  -> [  0 ,  4 ];
                 5  -> [  0 ,  5 ];
                 6  -> [  0 ,  6 ];
                 7  -> [  0 ,  7 ];
                 8  -> [  0 ,  8 ];
                 9  -> [  0 ,  9 ];
                10  -> [  1 ,  0 ];
                11  -> [  1 ,  1 ];
                12  -> [  1 ,  2 ];
                13  -> [  1 ,  3 ];
                14  -> [  1 ,  4 ];
                15  -> [  1 ,  5 ];
                16  -> [  1 ,  6 ];
                17  -> [  1 ,  7 ];
                18  -> [  1 ,  8 ];
                19  -> [  1 ,  9 ];
                20  -> [  2 ,  0 ];
                21  -> [  2 ,  1 ];
                22  -> [  2 ,  2 ];
                23  -> [  2 ,  3 ];
                24  -> [  2 ,  4 ];
                25  -> [  2 ,  5 ];
                26  -> [  2 ,  6 ];
                27  -> [  2 ,  7 ];
                28  -> [  2 ,  8 ];
                29  -> [  2 ,  9 ];
                30  -> [  3 ,  0 ];
                31  -> [  3 ,  1 ];

" This truth table could be replaced with the following macro.
"       clear(binary);
"       @repeat 32 {
"               binary -> [binary/10,binary%10]; inc(binary);}
"
" The test vectors will demonstrate the use of the macro.
test_vectors ( score -> [bcd2,bcd1])
        clear(binary);
        @repeat 32 {
                binary -> [binary/10,binary%10]; inc(binary);}
end
```

Design Specification — BJACK

BJACK, the blackjack controller, is technically a state machine (a circuit capable of storing an internal state reflecting prior events). State machines use sequential logic, branching to new states and generating outputs on the basis of both the stored states and external inputs.

In the case of the controller, the state machine stores states that reflect the following blackjack machine conditions:

- The value of Score in one of the decimal value ranges (0 to 16, 17 to 21, or 22+).
- The status of the card reader (card in or card out).
- The presence of an ace in the card reader.

On the basis of these stored states (and input from each new card), the controller decides whether or not a +10 or -10 value is sent to the adder.

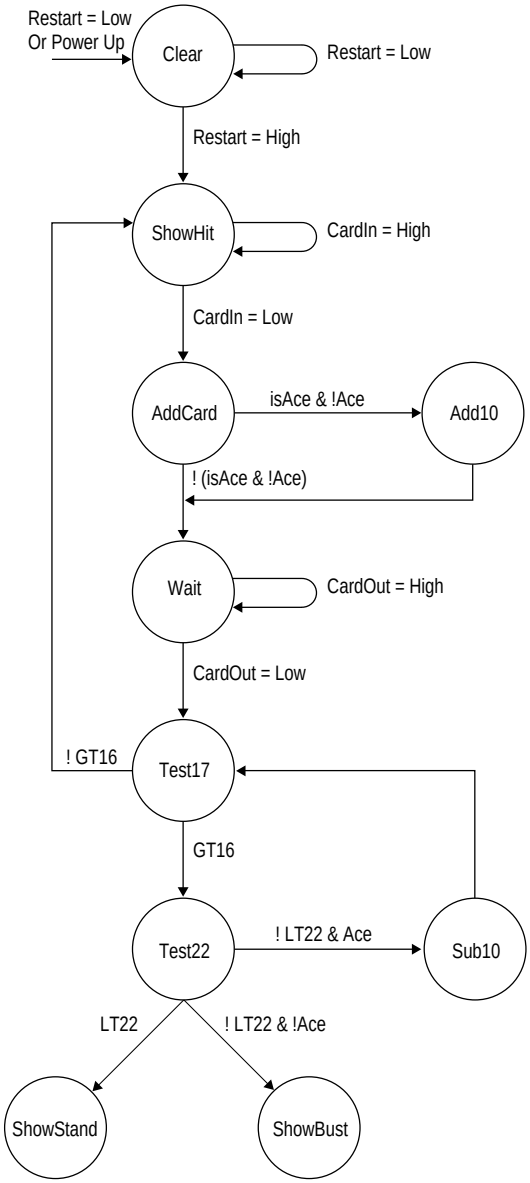
Design Method — BJACK

Developing a bubble diagram is the first step in describing a state machine. Figure 5-23 shows a bubble diagram (pictorial state diagram) for the controller. This bubble diagram indicates state transitions and the conditions that cause those transitions. Transitions are represented by arrows and the conditions causing the transitions are written alongside the arrow.

You must express the bubble diagram in the form shown in the state_diagram in Figure 5-24. There is a one-to-one correlation between the bubble diagram and the state diagram described in the source file (Figure 5-24). The table below describes the state identifiers (state machine states) illustrated in the bubble diagram and listed in the source file.

State Identifier	Description
Clear	Clear the state machine, adder, and displays.
ShowHit	Indicate that another card is needed. Hit indicator is lit.
AddCard	Add the value at the adder input to the current count.
Add10	Add the fixed value 10 to the current count, effectively giving an ace a value of 11.
Wait	Wait until a card is taken out of the reader.
Test17	Test the current count for a value less than 17.
Test22	Test the current count for a value less than 22.
Sub10	Add the fixed value -10 to the current count, effectively subtracting 10 and restoring an ace to 1.
ShowBust	Indicate that no more cards are needed. Bust indicator is lit.
ShowStand	Indicate that no more cards are needed. Neither Hit nor Bust indicators are lit.

Figure 5-23
Pictorial State Diagram: Blackjack Machine



Note that in Figure 5-24, each of the state identifiers (for example, ShowHit) is defined as sets having binary values. These values were chosen to minimize the number of product terms used in the P16R6.

Operation of the state machine proceeds as follows if no aces are drawn:

- If a card is needed from the reader, the state machine goes to state ShowHit.
- When CardIn goes low, meaning that a card has been read, a transition to state AddCard is made. The card value is added to the current score.
- The machine goes to Wait state until the card is withdrawn from the reader.
- The machine goes to Test17 state.
- If the score is less than 17, another card is drawn.
- If the score is greater than or equal to 17, the machine goes to state Test22.
- If the score is less than 22, the machine goes to the ShowStand state.
- If the score is 22 or greater, a transition is made to the ShowBust state.
- In either ShowStand or ShowBust state, a transition is made to Clear (to clear the state register and adder) when Restart goes low.
- When Restart goes back to high, the state machine returns to ShowHit and the cycle begins again.

Operation of the state machine when an ace is drawn is essentially the same. A card is drawn and the score is added. If the card is an ace and no ace has been drawn previously, the state machine goes to state Add10, and ten is added to the count (in effect making the ace an 11). Transitions to and from Test17 and Test22 proceed as before. However, if the score exceeds 21 and an ace has been set to 11, the state machine goes to state Sub10, 10 is subtracted from the score, and the state machine goes to state Test17.

Test Vectors — BJACK

Figure 5-24 shows three sets of test vectors; each set represents a different "hand" of play (as described above the set of vectors) and tests the different functions of the design. The Restart function is used to set the design to a known state between each hand, and the state identifiers are used instead of the binary values (which they represent).

Figure 5-24
Source File: State Machine (Controller)

```
module bjack
title 'BlackJack state machine controller
Michael Holley  Data I/O Corp.'

        bjack   device  'P16R6';

"Inputs
   Clk,ClkIN       pin 1,2;      "System clock
   GT16,LT22       pin 3,4;      "Score less than 17 and 22
   is_Ace          pin 5;        "Card is ace
   Restart         pin 6;        "Restart game
   CardIn,CardOut  pin 7,8;      "Card present switches
   Ena             pin 11;

   Sensor          = [CardIn,CardOut];
   _In             = [  0   ,   1   ];
   InOut           = [  1   ,   1   ];
   Out             = [  1   ,   0   ];

"Outputs
   AddClk          pin 12 istype 'com';                  "Adder clock
   Add10           pin 13 istype 'reg_D,invert';         "Input Mux control
   Sub10           pin 14 istype 'reg_D,invert';         "Input Mux control
   Q2,Q1,Q0        pin 15,16,17 istype 'reg_D,invert';
   Ace             pin 18 istype 'red_D,invert';         "Ace Memory

   High,Low        = 1,0;
   H,L,C,X         = 1,0,.C.,.X.;  "test vector characters

   Qstate          =  [Add10,Sub10,Q2,Q1,Q0];
   Clear           =  [  1  ,  1  , 1, 1, 1]; "31
   ShowHit         =  [  1  ,  1  , 1, 1, 0]; "30
   AddCard         =  [  1  ,  1  , 0, 0, 0]; "24
   Add_10          =  [  0  ,  1  , 0, 0, 0]; "16
   Wait            =  [  1  ,  1  , 0, 0, 1]; "25
   Test_17         =  [  1  ,  1  , 0, 1, 0]; "26
   Test_22         =  [  1  ,  1  , 0, 1, 1]; "27
   ShowStand       =  [  1  ,  1  , 1, 0, 0]; "28
   ShowBust        =  [  1  ,  1  , 1, 0, 1]; "29
   Sub_10          =  [  1  ,  0  , 0, 0, 1]; "17
   Zero            =  [  0  ,  0  , 0, 0, 0]; "0

equations
   [Qstate,Ace].c  = Clk;
   [Qstate,Ace].oe = !Ena;
```

```
@dcset
state_diagram Qstate

        State Clear:    AddClk    = !ClkIN;
                        Ace      := Low;
                        if (Restart==Low) then Clear else ShowHit;

        State ShowHit:  AddClk    = Low;
                        Ace      := Ace;
                        if (CardIn==Low) then AddCard else ShowHit;

        State AddCard:  AddClk    = !ClkIN;
                        Ace      := Ace;
                        if (is_Ace & !Ace) then Add_10 else Wait;

        State Add_10:   AddClk    = !ClkIN;
                        Ace      := High;
                        goto      Wait;

        State Wait:     AddClk    = Low;
                        Ace      := Ace;
                        if (CardOut==Low) then Test_17 else Wait;

        State Test_17:  AddClk    = Low;
                        Ace      := Ace;
                        if !GT16 then ShowHit else Test_22;

        State Test_22:  AddClk    = Low;
                        Ace      := Ace;
                        case     LT22           : ShowStand;
                                !LT22 & !Ace    : ShowBust;
                                !LT22 &  Ace    : Sub_10;
                        endcase;

        State Sub_10:   AddClk    = !ClkIN;
                        Ace      := Low;
                        goto      Test_17;

        State ShowBust: AddClk    = Low;
                        Ace      := Ace;
                        if (Restart==Low) then Clear else ShowBust;

       State ShowStand: AddClk    = Low;
                        Ace      := Ace;
                        if (Restart==Low) then Clear else ShowStand;

       State Zero:      goto Clear;

@page
" test_vectors edited...
```

Hierarchy Examples

The following ABEL-HDL source files show how to combine the three blackjack examples into one top-level source for implementation in a larger device.

The three lower-level modules are unchanged from the non-hierarchical versions and still include their original device declarations. The device declarations in this example provide the ABEL-HDL compiler with information about device-specific requirements, such as implied logic and default signal attributes.

Note: *To process this design, you must enable the "compatible" and "implied" properties of the Compile Logic process.*

The test vectors have been removed, since this design is not targeted to a PLD.

```
module bjacktop;
title 'instantiating bjack, muxadd, binbcd; By Kim-Fu Lim, Data I/O Corp.'

// Sub-module prototypes...
    bjack  interface (Clk,ClkIN,GT16,LT22,is_Ace,Restart,CardIn,CardOut,Ena
                  :> AddClk, Add10,Sub10,Q2..Q0, Ace);

    muxadd interface (V0..V4,AddClk,Clr,Add10,Sub10 :> S0..S4 -> is_Ace);

    binbcd interface (S0..S4, LT22, GT16 -> D0..D5);

// Sub-module instantiations...
    BJ functional_block bjack;
    MA functional_block muxadd;
    BB functional_block binbcd;

// Top level inputs...
    Clk             pin;      "System clock          -- bjack
    CardIn,CardOut  pin;      "Card present switches -- bjack
    Restart         pin;      "Restart game          -- bjack
    V4..V0          pin;      "                      -- muxadd
    Ace                         pin;
    Card            = [V4..V0];
    Sensor          = [CardIn,CardOut];
    _In             = [  0   ,   1   ];
    InOut           = [  1   ,   1   ];
    Out             = [  1   ,   0   ];

// Top level outputs...

    D5..D0          pin istype 'com';           "            -- binbcd
    BCD1            = [D3..D0];
    BCD2            = [D5,D4];
```

```
// Top level pins (for observing state machine)...
    Q2..Q0          pin istype 'com';
    Add10,Sub10     pin istype 'com';
    AddClk          node istype 'com,keep';

    Qstate          =  [Add10,Sub10,Q2,Q1,Q0];
    Clear           =  [  1  ,  1  , 1, 1, 1]; "31
    ShowHit         =  [  1  ,  1  , 1, 1, 0]; "30
    AddCard         =  [  1  ,  1  , 0, 0, 0]; "24
    Add_10          =  [  0  ,  1  , 0, 0, 0]; "16
    Wait            =  [  1  ,  1  , 0, 0, 1]; "25
    Test_17         =  [  1  ,  1  , 0, 1, 0]; "26
    Test_22         =  [  1  ,  1  , 0, 1, 1]; "27
    ShowStand       =  [  1  ,  1  , 1, 0, 0]; "28
    ShowBust        =  [  1  ,  1  , 1, 0, 1]; "29
    Sub_10          =  [  1  ,  0  , 0, 0, 1]; "17
    Zero            =  [  0  ,  0  , 0, 0, 0]; "0

    Hit         = !BB.GT16;
    Bust        = !BB.LT22;

    C,X,L,H           = .c.,.x.,0,1;

equations

// Describe the input connections...
    MA.[V4,V3,V2,V1,V0] = Card;
    MA.Clr      = Restart;
    BJ.Clk      = Clk;
    BJ.ClkIN    = Clk;
    BJ.Restart  = Restart;
    BJ.CardIn   = CardIn;
    BJ.CardOut  = CardOut;
    BJ.Ena = 0;

// Describe the output connections...
    [D5,D4,D3,D2,D1,D0]    = BB.[D5,D4,D3,D2,D1,D0];
    Add10       = BJ.Add10;
    Sub10       = BJ.Sub10;
    Q0          = BJ.Q0;
    Q1          = BJ.Q1;
    Q2          = BJ.Q2;
        Ace             = BJ.Ace;

// Describe inter-module connections...
    MA.Sub10    = BJ.Sub10;
    MA.Add10    = BJ.Add10;
    AddClk      = BJ.AddClk;
    MA.AddClk   = AddClk;
    BB.[S0,S1,S2,S3,S4] = MA.[S0,S1,S2,S3,S4];
    BJ.is_Ace   = MA.is_Ace;
    BJ.GT16     = BB.GT16;
    BJ.LT22     = BB.LT22;

end;
```

```
module bjack   ;
title 'BlackJack state machine controller
Michael Holley  Data I/O Corp.  9 Aug 1990'

        bjack   device  'P16R6';

"Inputs
        Clk,ClkIN       pin 1,2;      "System clock
        GT16,LT22       pin 3,4;      "Score less than 17 and 22
        is_Ace          pin 5;        "Card is ace
        Restart         pin 6;        "Restart game
        CardIn,CardOut  pin 7,8;      "Card present switches
        Ena             pin 11;

        Sensor          = [CardIn,CardOut];
        _In             = [  0   ,   1   ];
        InOut           = [  1   ,   1   ];
        Out             = [  1   ,   0   ];

"Outputs
        AddClk          pin 12;         "Adder clock
        Add10           pin 13;         "Input Mux control
        Sub10           pin 14;         "Input Mux control
        Q2,Q1,Q0        pin 15,16,17;
        Ace             pin 18;         "Ace Memory

        High,Low        = 1,0;
        H,L,C,X         = 1,0,.C.,.X.;  "test vector charactors

        Qstate          =  [Add10,Sub10,Q2,Q1,Q0];
        Clear           =  [  1  ,  1  , 1, 1, 1]; "31
        ShowHit         =  [  1  ,  1  , 1, 1, 0]; "30
        AddCard         =  [  1  ,  1  , 0, 0, 0]; "24
        Add_10          =  [  0  ,  1  , 0, 0, 0]; "16
        Wait            =  [  1  ,  1  , 0, 0, 1]; "25
        Test_17         =  [  1  ,  1  , 0, 1, 0]; "26
        Test_22         =  [  1  ,  1  , 0, 1, 1]; "27
        ShowStand       =  [  1  ,  1  , 1, 0, 0]; "28
        ShowBust        =  [  1  ,  1  , 1, 0, 1]; "29
        Sub_10          =  [  1  ,  0  , 0, 0, 1]; "17
        Zero            =  [  0  ,  0  , 0, 0, 0]; "0
```

```
equations
        [Qstate,Ace].c  = Clk;
        [Qstate,Ace].oe = !Ena;

@dcset
state_diagram Qstate

        State Clear:    AddClk    = !ClkIN;
                        Ace      := Low;
                        if (Restart==Low) then Clear else ShowHit;

        State ShowHit:  AddClk    = Low;
                        Ace      := Ace;
                        if (CardIn==Low) then AddCard else ShowHit;

        State AddCard:  AddClk    = !ClkIN;
                        Ace      := Ace;
                        if (is_Ace & !Ace) then Add_10 else Wait;

        State Add_10:   AddClk    = !ClkIN;
                        Ace      := High;
                        goto      Wait;

        State Wait:     AddClk    = Low;
                        Ace      := Ace;
                        if (CardOut==Low) then Test_17 else Wait;

        State Test_17:  AddClk    = Low;
                        Ace      := Ace;
                        if !GT16 then ShowHit else Test_22;

        State Test_22:  AddClk    = Low;
                        Ace      := Ace;
                        case     LT22           : ShowStand;
                                !LT22 & !Ace    : ShowBust;
                                !LT22 &  Ace    : Sub_10;
                        endcase;

        State Sub_10:   AddClk    = !ClkIN;
                        Ace      := Low;
                        goto      Test_17;

        State ShowBust: AddClk    = Low;
                        Ace      := Ace;
                        if (Restart==Low) then Clear else ShowBust;

       State ShowStand: AddClk    = Low;
                        Ace      := Ace;
                        if (Restart==Low) then Clear else ShowStand;

       State Zero:      goto Clear;

end
```

```
module muxadd ;
title '5-bit ripple adder with input multiplex
Michael Holley   Data I/O Corp.  26 Mar 1990'

        muxadd  device  'P22V10';

        AddClk,Clr,Add10,Sub10,is_Ace   pin  1, 9, 8, 7,14;
        V4,V3,V2,V1,V0                  pin  6, 5, 4, 3, 2;
        S4,S3,S2,S1,S0                  pin 15,16,17,18,19;
        C4,C3,C2,C1                     pin 20,21,22,23;

        X,C,L,H   = .X., .C., 0, 1;

        Card      = [V4,V3,V2,V1,V0];
        Score     = [S4,S3,S2,S1,S0];
        CarryIn   = [C4,C3,C2,C1, 0];
        CarryOut  = [ X,C4,C3,C2,C1];
        ten       = [ 0, 1, 0, 1, 0];
        minus_ten = [ 1, 0, 1, 1, 0];

        S4,S3,S2,S1,S0 istype 'reg' ;

" Input Multiplexer
        Data     =  Add10 &  Sub10 & Card
                 # !Add10 &  Sub10 & ten
                 #  Add10 & !Sub10 & minus_ten;

equations
        Score   := Data $ Score $ CarryIn;

        CarryOut = Data & Score # (Data # Score) & CarryIn;

        Score.ar = !Clr;

        Score.c  = AddClk;

        is_Ace   = Card == 1;

end;
```

```
module binbcd;
title 'comparator and binary to bcd decoder for Blackjack Machine
Michael Holley   Data I/O Corp   12 Oct 1992'

" The 5 -bit binary (0 - 31) score is converted into two BCD outputs.
" The interger division '/' and the modulus operator '%' are used to
" extract the individual digits from the two digit score.
" 'Score % 10' will yield the 'units' and
" 'Score / 10' will yield the 'tens'
"
" The 'GT16' and 'LT22' outputs are for the state machine controller.

        binbcd  device  'P16L8';

        S4,S3,S2,S1,S0  pin 5,4,3,2,1;
        score           = [S4,S3,S2,S1,S0];

        LT22,GT16       pin 12,13       istype 'com';

        D5,D4           pin 14,15       istype 'com';
        bcd2            = [D5,D4];

        D3,D2,D1,D0     pin 16,17,18,19 istype 'com';
        bcd1            = [D3,D2,D1,D0];

" Digit separation macros
        binary    = 0;      "scratch variable
        clear   macro (a) {@const ?a=0};
        inc     macro (a) {@const ?a=?a+1;};

equations
        LT22    = (score < 22);  "Bust
        GT16    = (score > 16);  "Hit / Stand

test_vectors ( score -> [GT16,LT22])
                 1   -> [ 0  , 1  ];
                 6   -> [ 0  , 1  ];
                 8   -> [ 0  , 1  ];
                16   -> [ 0  , 1  ];
                17   -> [ 1  , 1  ];
                18   -> [ 1  , 1  ];
                20   -> [ 1  , 1  ];
                21   -> [ 1  , 1  ];
                22   -> [ 1  , 0  ];
                23   -> [ 1  , 0  ];
                24   -> [ 1  , 0  ];
```

```
truth_table ( score -> [bcd2,bcd1])
                 0  -> [  0 ,  0 ];
                 1  -> [  0 ,  1 ];
                 2  -> [  0 ,  2 ];
                 3  -> [  0 ,  3 ];
                 4  -> [  0 ,  4 ];
                 5  -> [  0 ,  5 ];
                 6  -> [  0 ,  6 ];
                 7  -> [  0 ,  7 ];
                 8  -> [  0 ,  8 ];
                 9  -> [  0 ,  9 ];
                10  -> [  1 ,  0 ];
                11  -> [  1 ,  1 ];
                12  -> [  1 ,  2 ];
                13  -> [  1 ,  3 ];
                14  -> [  1 ,  4 ];
                15  -> [  1 ,  5 ];
                16  -> [  1 ,  6 ];
                17  -> [  1 ,  7 ];
                18  -> [  1 ,  8 ];
                19  -> [  1 ,  9 ];
                20  -> [  2 ,  0 ];
                21  -> [  2 ,  1 ];
                22  -> [  2 ,  2 ];
                23  -> [  2 ,  3 ];
                24  -> [  2 ,  4 ];
                25  -> [  2 ,  5 ];
                26  -> [  2 ,  6 ];
                27  -> [  2 ,  7 ];
                28  -> [  2 ,  8 ];
                29  -> [  2 ,  9 ];
                30  -> [  3 ,  0 ];
                31  -> [  3 ,  1 ];

" This truth table could be replaced with the following macro.
"       clear(binary);
"       @repeat 32 {
"               binary -> [binary/10,binary%10]; inc(binary);}
"
"
" The test vectors will demonstrate the use of the macro.
"
test_vectors ( score -> [bcd2,bcd1])
        clear(binary);
        @repeat 32 {
                binary -> [binary/10,binary%10]; inc(binary);}
end
```

ABEL and Synario Projects

The following ABEL-HDL source, **p6top.abl**, (Figure 5-25) instantiates variable instances of **prep6.abl** (Figure 5-26).

Figure 5-25
Top-level ABEL-HDL Source

```
module p6top (rep)
title 'Variable instances of PREP6 described in
       Hierarchical ABEL-HDL By Kim-Fu Lim,  Data I/O Corp.'

   @ifb (?rep)
   { @message 'Must specify -arg N'
     @exit  }
   D15..D0 pin;
   Q15..Q0 pin istype 'reg';
   Clk, Rst pin;

   Q = [Q15..Q0];
   D = [D15..D0];

   @const N = ?rep - 1;

prep6 interface (D15..D0, Clk, Rst -> Q15..Q0);

   ACC macro (i)
   { @expr {ACC}?i; }

   @const i = 0;
   @repeat ?rep
   { ACC(i) functional_block prep6;
     @const i = i + 1; }

equations
   ACC0.[D15..D0] = D;

      @const i = 0;
      @repeat ?rep
      { ACC(i).[Clk, Rst] = [Clk, Rst];
        @const i = i+1; }

      @const i = 0;
      @repeat ?rep-1
      { ACC(i+1).[D15..D0] = ACC(i).[Q15..Q0];
        @const i = i+1; }
   Q = ACC(N).[Q15..Q0];
end
```

Lower-level Sources

Figure 5-26 shows the lower-level ABEL-HDL file instantiated by **p6top.abl**. This file does not contain an **interface** statement, which is optional in lower-level files.

Figure 5-26
Lower-level ABEL-HDL Source

```
MODULE prep6

TITLE '16-Bit Accumulator
       By Kim-Fu Lim, Data I/O Corp.'

D15..D0 pin;
Q15..Q0 pin istype 'reg';
Clk, Rst pin;

Q = [Q15..Q0];
D = [D15..D0];

@carry 2;
EQUATIONS
  Q := D + Q;
  Q.C = Clk;
  Q.AR = Rst;
END
```

Chapter 6
Language Reference

This chapter provides detailed information about each of the language elements in ABEL-HDL. It assumes you are familiar with the basic syntax discussed in the Chapter 2, "Language Structure." Each entry contains the following sections (if applicable):

- **Syntax** — is the required syntax for the element.
- **Purpose** — is a brief description of the intended use of the element.
- **Use** — is a discussion of the potential uses of the element, including any special considerations.
- **Examples** — are examples of the element as it is used in a design description.
- **See Also** — refers to other elements and discussions, and to design examples that demonstrate the use of an element.

Basic syntax information (on subjects such as blocks, strings, sets and arguments) is provided in Chapter 2, "Language Structure."

.ext — Dot Extensions

Syntax

signal_name.ext

Purpose

Dot extensions provide a way to refer specifically to internal signals and nodes that are associated with a primary signal in your design.

Use

Signal dot extensions describe, more precisely, the behavior of signals in a logic description, and remove the ambiguities in equations.

You can use ABEL-HDL dot extensions in complex language constructs, such as nested sets or complex expressions.

Using Pin-to-Pin Vs. Detailed Dot Extensions:

Dot extensions allow you to refer to various circuit elements (such as register clocks, presets, feedback and output enables) that are related to a primary signal.

Some dot extensions are general purpose and are intended for use with a wide variety of device architectures. These dot extensions are therefore referred to as pin-to-pin (or "architecture-independent"). Other dot extensions are intended for specific classes of device architectures, or require specific device configurations. These dot extensions are referred to as detailed (or "architecture-dependent" or "device-specific") dot extensions.

In most cases, you can describe a circuit using either pin-to-pin or detailed dot extensions. Which form you use depends on the application and whether you want to implement the application in a variety of architectures. The advantages of each method are discussed later in this section.

Table 6-1 lists the ABEL-HDL dot extensions. Pin-to-pin dot extensions are indicated with a check in the **Pin-to-Pin** column.

Table 6-1
Dot Extensions

Dot Ext.	Pin-to-pin	Description
.ACLR [3,4]	✓	A device-independent asynchronous register reset, equivalent to .AR with ISTYPE 'buffer' (or .AP with ISTYPE 'invert').
.AP		Asynchronous register preset
.AR		Asynchronous register reset
.ASET [2,3]	✓	A device-independent asynchronous register preset, equivalent to .AP with ISTYPE 'buffer' (or .AR with ISTYPE 'invert').
.CE		Clock-enable input to a gated-clock flip-flop
.CLK [1]	✓	Clock input to an edge-triggered flip-flop
.CLR [2,3]	✓	A device-independent synchronous register reset, equivalent to .SR with ISTYPE 'buffer' (or .SP with ISTYPE 'invert').
.COM [3]	✓	A combinational feedback from the flip-flop data input, normalized to the pin value and used to distinguish between pin (.PIN) and internal logic array (.COM) feedback.
.D [1]		When on the left side of an equation, .D is the data input to a D-type flip-flop; on the right side, .D is combinational feedback.
.FB	✓	Register feedback
.FC		Flip-flop mode control
.J		J input to a JK-type flip-flop
.K		K input to a JK-type flip-flop
.LD		Register load input
.LE		Latch-enable input to a latch
.LH		Latch-enable (high) to a latch
.OE [1]	✓	Output enable

Dot Ext.	Pin-to-pin	Description
.PIN	✓	Pin feedback
.PR [1]		Register preset (synchronous or asynchronous)
.Q		Register feedback
.R		R input to an SR-type flip-flop
.RE [1]		Register reset (synchronous or asynchronous)
.S		S input to an SR-type flip-flop
.SET [2,3]	✓	A device-independent synchronous register preset, equivalent to .SP with ISTYPE 'buffer' (or .SR with ISTYPE 'invert').
.SP		Synchronous register preset
.SR		Synchronous register reset
.T		T input to a T-type (toggle) flip flop

1 Example follows.

2 If ISTYPE 'buffer' or 'invert' is specified, the compiler converts these dot extensions to the equivalent detailed dot extension.

3 Some fitters do not support these dot extensions.

Detailed Design Dot Extensions

Table 6-2 shows the dot extensions that are supported (and which of those are required) for different register types in detailed design descriptions. The required dot extensions are indicated with a check in the **Extension Required** column.

Table 6-2
Dot Extensions for Device-specific (detailed) Designs

Register Type	Extension Required	Supported Extensions	Definition
combinational		.oe	output enable
(no register)		.pin	pin feedback
		.com	combinational feedback
D-type flip-flop	✓	.clk	clock
	✓	.d	data input
		.fc	flip-flop mode control
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin
JK-type flip-flop	✓	.clk	clock
	✓	.j	j input
	✓	.k	k input
		.fc	flip-flop mode control
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin feedback

Register Type	Extension Required	Supported Extensions	Definition
SR-type flip-flop	✓	.clk	clock
	✓	.s	set input
	✓	.r	reset input
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous preset
		.pin	pin feedback
T-type flip-flop	✓	.clk	clock
	✓	.t	toggle input
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin feedback
L-type latch	✓	.d	data input
	✓	.le	latch enable input to a latch
		.lh	latch enable (high) input to a
		.oe	latch
		.q	output enable
		.pin	flip-flop feedback
			pin feedback
Gated clock D	✓	.clk or .ce	clock or clock enable
flip-flop	✓	.d	data input
		.oe	output enable
		.q	flip-flop feedback
		.pin	pin feedback

Pin-to-Pin Design Dot Extensions

Table 6-2 shows the dot extensions that are allowable (and which of those are required) for pin-to-pin design descriptions. The required dot extensions are indicated with a check in the **Required** column.

Table 6-3
Dot Extensions for Architecture-independent (pin-to-pin) Designs

Register Type	Required	Allowable Extensions	Definition
combinational		none	output
(no register)		.oe	output enable
		.pin	pin feedback
registered logic		.clr	synchronous preset
		.aclr	asynchronous preset
		.set	synchronous set
		.aset	asynchronous set
	✓	.clk	clock
		.com	combinational feedback
		.fb	registered feedback
		.pin	pin feedback

Figures 6-1 through 6-8 show the effect of each dot extension. The actual source of the feedback may vary from that shown.

Figure 6-1
Pin-to-pin Dot Extensions in an Inverted Output Architecture

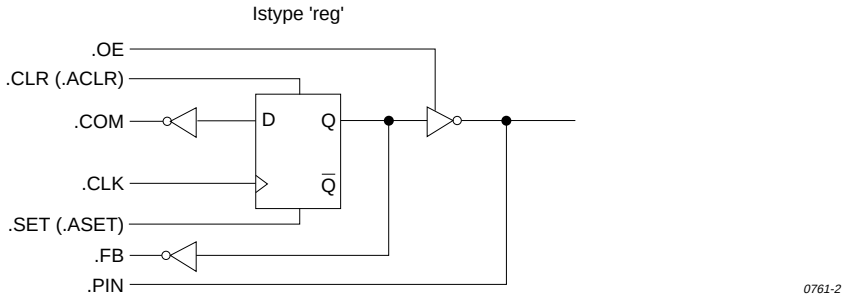


Figure 6-2
Pin-to-pin Dot Extensions in a Non-inverted Output Architecture

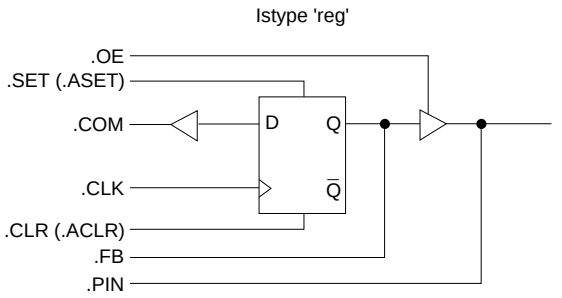


Figure 6-3
Detailed Dot Extensions for a D-type Flip-flop Architecture

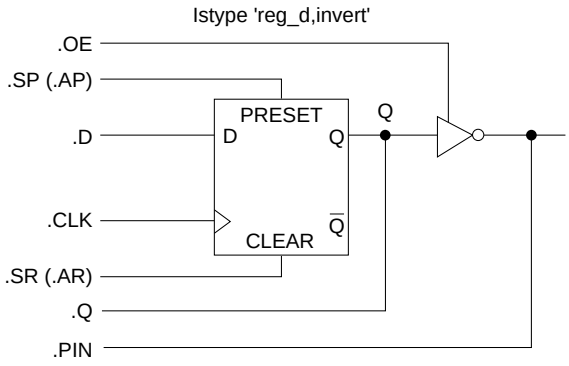


Figure 6-4
Detailed Dot Extensions for a T-type Flip-flop Architecture

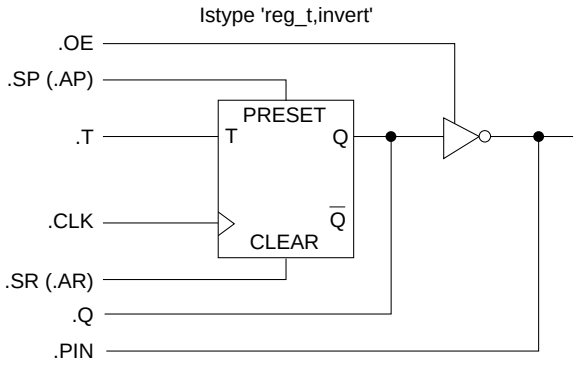


Figure 6-5
Detailed Dot Extensions for an RS-type Flip-flop Architecture

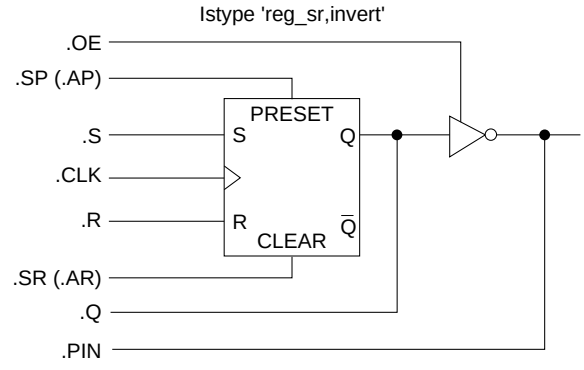
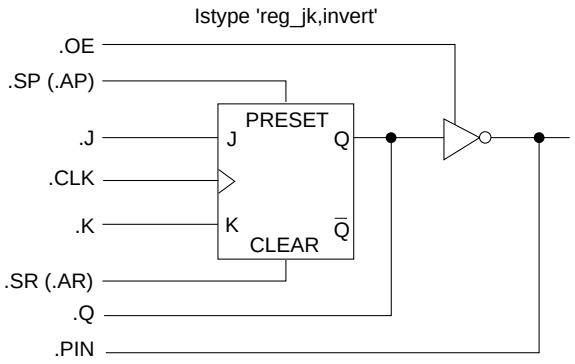
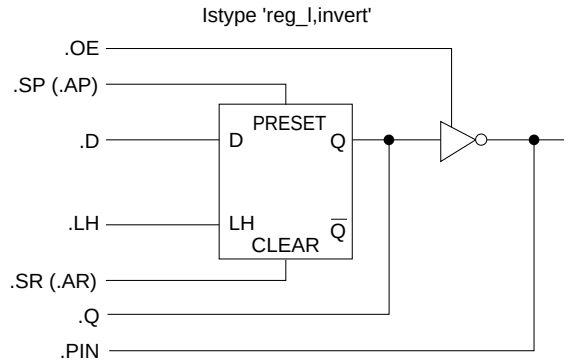


Figure 6-6
Detailed Dot Extensions for a JK-type Flip-flop Architecture



0765-2

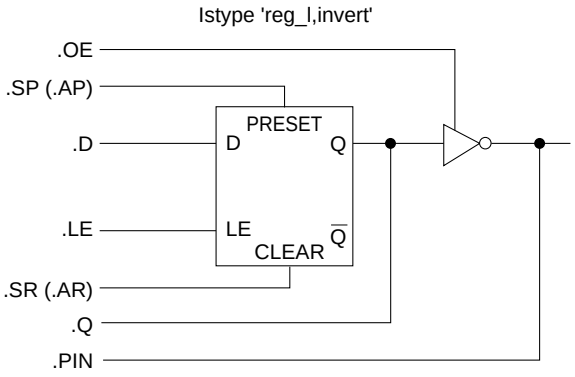
Figure 6-7
Detailed Dot Extensions for a Latch with Active High Latch Enable



LH	D	Q
1	0	0
1	1	1
0	X	Last Q

0766-2

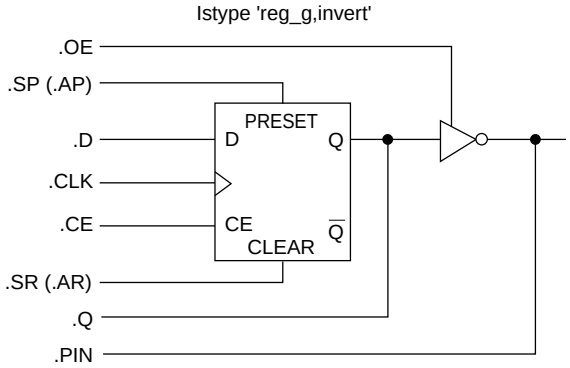
Figure 6-8
Detailed Dot Extensions for a Latch with Active Low Latch Enable



LE	D	Q
0	0	0
0	1	1
1	X	Last Q

0767-2

Figure 6-9
Detailed Dot Extensions for a Gated-clock D Flip-flop



CE	CLK	D	Q
X	0	0	0
1	0	1	1
1	1	X	Last Q

1803-2

Examples

These equations precisely describe the desired circuit as a toggling D-type flip-flop that is clocked by the input Clock, assuming ISTYPE 'reg_D,buffer':

```
Q1.clk = Clock;
Q1.D = !Q1.Q # Preset;
```

Register preset:

```
Q2.PR = S & !T;
```

Register reset:

```
Q2.RE = R & !T;
```

The same circuit can be described without ISTYPE 'buffer' as:

```
Q1.clk = Clock;
Q1 := !Q1.FB # Preset;
Q2.SET = S & !T; Q2.CLR = R & !T;
```

Three-state Output Enables

Output enables are described in ABEL-HDL with the **.oe** dot extension applied to an output signal name. For example,

```
Q1.oe = !enab;
```

The equation specifies that the input signal **enab** controls the output enable for an output signal named **Q1**.

Note: *If you explicitly state the value of a fixed output enable, you restrict the device fitters' ability to map the indicated signal to a simple input pin instead of a three-state I/O pin.*

See Also

Istype
"Attributes" in the Chapter 2, "Language Structure"
"Dot Extensions" in the Chapter 3, "Design Considerations"

= — Constant Declarations

Syntax

id [,*id*]... = *expr* [,*expr*]... ;

Purpose

A constant declaration that defines constants used in a module.

Use

id	An identifier naming a constant to be used within a module.
expr	An expression defining the constant value.

Note: *The equal sign (=) used for constant declarations in the Declarations section is also used for equations in the Equations section. See "Operators" in Chapter 2, "Language Structure."*

A constant is an identifier that retains a constant value throughout a module.

The identifiers on the left side of the equals sign are assigned the values listed on the right side. There is a one-to-one correspondence between the identifiers and the expressions listed. There must be one expression for each identifier.

The ending semicolon is required after each declaration.

Constants are helpful when you use a value many times in a module, especially when you may be changing the value during the design process. Constants allow you to change the value once in the declaration of the constant, rather than changing the value throughout the module.

Constant declarations may not be self-referencing. The following examples will cause errors:

```
X = X;
```

```
a = b;
b = a;
```

An include file, **constant.inc**, in the ABEL-HDL library file contains definitions for the most frequently used ABEL-HDL constants. To include this file, enter

```
Library 'constant' ;
```

Examples

```
ABC = 3 * 17;          " ABC is assigned the value 51
Y = 'Bc' ;             " Y =  ^h4263 ;
X =.X.;                " X means 'don't care'
ADDR = [1,0,15];       " ADDR is a set with 3 elements
A,B,C = 5,[1,0],6;     " 3 constants declared here
D pin 6;               " see next line
E = [5 * 7,D];         " signal names can be included
G = [1,2]+[3,4];       " set operations are legal
A = B & C;             " operations on identifiers are valid
A = [!B,C];            " set and identifiers on right
```

Using Intermediate Expressions

You can use intermediate (constant) expressions in the declarations section to reduce the number of output pins required to implement multi-level functions. Intermediate expressions can be useful when a module has repeated expressions. In general, intermediate expressions

- decrease the number of output pins required, but
- increase the amount of logic required per output

A constant expression is interpreted as a string of characters, not as a function to be implemented. For example, for the following Declarations and Equations

```
Declarations
  TMP1 = [A3..A0] == [B3..B0];
  TMP2 = [A7..A4] == [B7..B4];
Equations
  F = TMP1 & TMP2;
```

the compiler substitutes the declarations into the equations, creating

```
F = (A7 !$ B7) & (A6 !$ B6) & (A5 !$ B5) & (A4 !$ B4) &
    (A3 !$ B3) & (A2 !$ B2) & (A1 !$ B1) & (A0 !$ B0);
```

In contrast, if you move the constant declarations into the equations section:

```
Declarations
  TMP1,TMP2 pin 18,19
Equations
  TMP1 = [A3..A0] == [B3..B0];
  TMP2 = [A7..A4] == [B7..B4];
  F = TMP1 & TMP2;
```

the compiler implements the equations as three discrete product terms, with the result

```
TMP1 =(A3 !$ B3) & (A2 !$ B2) & (A1 !$ B1) & (A0 !$ B0);
TMP2 =(A7 !$ B7) & (A6 !$ B6) & (A5 !$ B5) & (A4 !$ B4);
  F = TMP1 & TMP2;
```

The first example (using intermediate expressions) requires one output with 16 product terms, the second example (using equations) requires three outputs with less than 8 product terms per output. In some cases, the number of product terms required for both methods can be reduced during optimization.

Note: *As an alternate method for specifying multi-level circuits such as this, you can use the @CARRY directive. See "@directive" later in this chapter.*

See Also

Declarations
Equations
"Constants" in the Chapter 2, "Language Structure"

'attr' — Signal Attributes

See Istype.

@*directive* — Directives

Purpose Directives control the contents or processing of a source file. You can use directives to conditionally include sections of ABEL-HDL source code, bring code in from another file, and print messages during processing. The available directives are given on the following pages.

Use Some of the directives use arguments to determine how the directive is processed. The arguments can be actual arguments, or dummy arguments preceded by question marks (?). The rules applying to actual and dummy arguments are presented in Chapter 2, "Language Structure."

@Alternate — Alternate Operator Set

Syntax `@alternate`

Use @Alternate enables an alternate set of operators. If you are more familiar with the alternate set, you may want to use this directive.

The alternate operators remain in effect until the @Standard directive is used or the end of the module is reached.

Using the alternate operator set precludes use of the ABEL-HDL addition, multiplication, and division operators because they represent the OR, AND, and NOT logical operators in the alternate set. The standard operators !, &, #, $, and !$ still work when **@Alternate** is in effect.

The alternate operator set is listed in Table 6-4.

Table 6-4
Alternate Operator Set

ABEL-HDL Operator	Alternate Operator	Description
!	/	NOT
&	*	AND
#	+	OR
$	: + :	XOR
!$	: * :	XNOR

See Also

@STANDARD

@Carry — Maximum Bit-width for Arithmetic Functions

Syntax

```
@carry expression ;
```

expression A numeric expression.

Use

The @Carry directive allows you to reduce the amount of logic required for processing large arithmetic functions by specifying how adders, counters, and comparators are generated. The number generated by the expression indicates the maximum bit-width to use when performing arithmetic functions.

For example, for an 8-bit adder, a @Carry statement with an expression which results in 2 would divide the 8-bit adder into four 2-bit adders, creating intermediate nodes. This would reduce the amount of logic generated.

The statement:

```
@carry 1;
```

generates chains of one-bit adders and comparators for all subsequent adder and comparator equations (instead of the full look-ahead carry equations normally generated).

This directive automatically generates additional combinational nodes. Use different values for the @CARRY statement to specify the types of adders and comparators required for the design. To specify that full lookahead carry should be generated (the default if no @CARRY has been specified) use the statement:

```
@carry 0
```

Examples

```
@carry 2; "generate adder chain
[s8..s0] = [.x.,a7..a0]+[.x.,b7..b0]
```

See Also

= (Constant Declarations)
"Constants" in Chapter 2, "Language Structure"

@Const — Constant Declarations

Syntax

```
@const id = expression ;
```

id An identifier.

expression An expression.

Use

@Const allows new constant declarations to be made in a source file outside the normal (and required) declarations section.

The @Const directive defines internal constants inside macros. Constants defined with @Const override previous constant declarations. You cannot use @Const to redefine an identifier that was used earlier in the source file as something other than a constant (for example, a macro or pin).

Examples

```
@CONST count = count + 1;
```

See Also

= (Constant Declarations)
"Constants" in Chapter 2, "Language Structure"

@Dcset — Don't Care Set

Syntax `@dcset`

Use ABEL-HDL uses don't-care conditions to help optimize partially-specified logic functions. Partially-specified logic functions are those that have less than 2^n significant input conditions, where *n* is the number of input signals. The **@Dcset** directive allows the optimization to use either 1 or 0 for don't cares to optimize these functions.

CAUTION: The @Dcset directive overrides Istype attributes 'dc', 'neg' and 'pos'.

See Also

@Onset
Istype 'dc'
?:= and ?= Assignment Operators

Truth_table
"@DCSET Considerations and Precautions" in the chapter "Design Considerations"

@Dcstate — State Output Don’t Cares

Syntax `@dcstate`

Use When @dcstate is specified, all unspecified state diagram states and transitions are applied to the design outputs as don’t cares. You must use this option in combination with @dcset or with the ’dc’ attribute.

See Also

@DCSET
Istype ’dc’

@Exit — Exit Directive

Syntax `@exit`

Use The **@Exit** directive stops processing of the source file with error bits set. (Error bits allow the operating system to determine that a processing error has occurred.)

@Expr — Expression Directive

Syntax

```
@expr [ {block} ] expression ;
```

block A block.

expression An expression.

Use

@Expr evaluates the given expression and converts it to a string of digits in the default base numbering system. This string and the block are then inserted into the source file at the point where the @Expr directive occurs. The expression must produce a number.

@Expr can contain variable values and you can use it in loops with @Repeat.

Examples

```
@expr {ABC} ^B11 ;
```

Assuming that the default base is base ten, this example causes the text ABC3 to be inserted into the source file.

@If — If Directive

Syntax `@if` *expression* {*block* }

expression An expression.

block A block of text.

Use @IF includes or excludes sections of code based on the value of an expression. If the expression is non-zero (logical true), the block of code is included.

Dummy argument substitution is supported in the expression.

Examples

```
@if (A > 17) { C = D $ F ; }
```

@Ifb — If Blank Directive

Syntax

```
@IFB (arg) {block }
```

arg An actual argument, or a dummy argument preceded by a "?"

block A block of text.

Use

@IFB includes the text contained within the block if the argument is blank (if it contains 0 characters).

Examples

```
@IFB ()
{text here is included with the rest of the source file.}
@IFB ( hello ) { this text is not included }
@IFB (?A) {this text is included if no value is substituted for
A.  }
```

See Also

"Arguments and Argument Substitution" in the Chapter 2, "Language Structure"

@Ifdef — If Defined Directive

Syntax

```
@ifdef id {block }
```

id An identifier.

block A block of text.

Use

@IFDEF includes the text contained within the block, if the identifier is defined.

Examples

```
A pin 5 ;
@ifdef A { Base = ^hE000 ; }
"the above assignment is made because A was defined
```

@Ifiden — If Identical Directive

Syntax

```
@ifiden (arg1,arg2 ) {block }
```

arg1,2	Actual arguments, or dummy argument names preceded by a "?"
block	A block of text.

Use

The text in the block is included if arg1 and arg2 are identical.

Examples

```
@ifiden (?A,abcd) { ?A device 'P16R4'; }
```

A device declaration for a P16R4 is made if the actual argument substituted for A is identical to abcd.

@Ifnb — If Not Blank Directive

Syntax

`@ifnb (`*`arg`*`) {`*`block`*` }`

arg	An actual argument, or a dummy argument name preceded by a "?"
block	A block of text.

Use

@IFNB includes the text contained within the block if the argument is not blank (if it contains more than 0 characters).

Examples

```
@IFNB () { ABEL-HDL source here is not included with the rest
of the source file.  }
@IFNB ( hello ) { this text is included }
@IFNB (?A) {this text is included if a value is substituted
for A}
```

@Ifndef — If Not Defined Directive

Syntax

```
@ifndef id {block }
```

id An identifier.

block A block of text.

Use

@IFNDEF includes the text contained within the block, if the identifier is undefined. Thus, if no declaration (pin, node, device, macro, or constant) has been made for the identifier, the text in the block is inserted into the source file.

Examples

```
@ifndef A{Base=^hE000;}
"if A is not defined, the block is inserted in the text
```

@Ifniden — If Not Identical Directive

Syntax

```
@ifniden (arg1,arg2 ) {block }
```

arg1,2 Actual arguments, or dummy argument names preceded by a "?"

block A block of text.

Use

The text in the block is included in the source file if arg1 and arg2 are not identical.

Examples

```
@ifniden (?A,abcd) { ?A device 'P16R8'; }
```

A device declaration for a P16R8 is made if the actual argument substituted for A is not identical to abcd.

@Include — Include Directive

Syntax

```
@include filespec
```

filespec A string specifying the name of a file.

Use

@INCLUDE causes the contents of the specified file to be placed in the ABEL-HDL source file. The inclusion begins at the location of the @INCLUDE directive. The file specification can include an explicit drive or path specification that indicates where the file is found. If no drive or path specification is given, the default drive or path is used.

Examples

```
@INCLUDE 'macros.abl' "file specification
@INCLUDE '\\incs\\macros.inc'  "DOS paths require 2 slashes
```

See Also

Library

@Irp — Indefinite Repeat Directive

Syntax

```
@irp dummy_arg ( arg [,arg ]...  ) {block }
```

dummy_arg	A dummy argument.
arg	An actual argument, or a dummy argument name preceded by a "?"
block	A block of text.

Use

@IRP causes the block to be repeated in the source file n times, where n equals the number of arguments contained in the parentheses. Each time the block is repeated, the dummy argument takes on the value of the next successive argument.

Examples

```
@IRP A (1, ^H0A,0)
{B = ?A ; }
```

results in:

```
B = 1 ;
B = ^H0A ;
B = 0 ;
```

which is inserted into the source file at the location of the @IRP directive. Note that multiple assignments to the same identifier result in an implicit OR.

Note that if the directive is specified like this:

```
@IRP A (1,^H0A,0)
{B = ?A ; }
```

the resulting text would be:

```
B = 1 ; B = ^H0A ; B = 0 ;
```

The text appears all on one line because the block in the @IRP definition contains no end-of-lines. Remember that end-of-lines and spaces are significant in blocks.

@Irpc — Indefinite Repeat, Character Directive

Syntax

```
@irpc dummy_arg (arg) {block }
```

dummy_arg	A dummy argument.
arg	An actual argument, or a dummy argument name preceded by a "?"
block	A block.

Use

@IRPC causes the block to be repeated in the source file *n* times, where *n* equals the number of characters contained in *arg*. Each time the block is repeated, the dummy argument takes on the value of the next character.

Examples

```
@IRPC A (Cat)
{B = ?A ;
}
```

results in:

```
B = C ;
B = a ;
B = t ;
```

which is inserted into the source file at the location of the @IRPC directive.

@Message — Message Directive

Syntax

```
@message 'string'
```

string Any string.

Use

@Message sends the message specified in *string* to your monitor. You can use this directive to monitor the progress of the parsing step of the compiler, or as an aid to debugging complex sequences of directives.

Examples

```
@message 'Includes completed'
```

@Onset — No Don’t Care’s

Syntax `@onset`

Use The **@onset** directive disables the use of don’t care input conditions for optimization.

See Also

@Dcset
ISTYPE ’dc’

@Page — Page Directive

Syntax `@page`

Use Send a form feed to the listing file. If no listing is being created, **@page** has no effect.

@Radix — Default Base Numbering Directive

Syntax

```
@radix expr ;
```

expr An expression that produces the number 2, 8, 10 or 16 to indicate a new default base numbering.

Use

The **@Radix** directive changes the default base. The default is base 10 (decimal). This directive is useful when you need to specify many numbers in a base other than 10. All numbers that do not have their base explicitly stated are assumed to be in the new base. (See "Numbers" in Chapter 2, "Language Structure.")

The newly-specified default base stays in effect until another **@radix** directive is issued or until the end of the module is reached. Note that when a new @radix is issued, the specification of the new base must be in the current base format.

When the default base is set to 16, all numbers in that base that begin with an alphabetic character must begin with leading zeroes.

Examples

```
@radix 2 ;    "change default base to binary
@radix 1010 ;  "change from binary to decimal
```

@Repeat — Repeat Directive

Syntax

```
@repeat expr {block }
```

expr A numeric expression.

block A block.

Use

@REPEAT causes the block to be repeated *n* times, where *n* is specified by the constant expression.

Examples

The following use of the repeat directive,

```
@repeat 5 {H,}
```

results in the insertion of the text "H,H,H,H,H," into the source file. The @REPEAT directive is useful in generating long truth tables and sets of test vectors. Examples of @REPEAT can be found in the example files.

@Setsize — Set Indexing

Syntax `@setsize [`*`expression`*`];`

Purpose The **@setsize** directive generates a number corresponding to the number of elements in the expression, which must be a set. This directive is useful for set indexing operations.

Example `@SETSIZE [a,b,c]`

generates the number 3.

For set indexing, you can use the @SETSIZE directive in macros in the following manner:

```
high macro (s) {?S[@SETSIZE(?S);-1..@SETSIZE(?S);/2-1]};
```

The **high** macro returns the upper half of a set of any size (the high 4 bits of an 8-bit set, for example).

Note: *The terminating semicolons are required.*

See Also

"Set Indexing" in Chapter 2, "Language Structure"

@Standard — Standard Operators Directive

Syntax `@standard`

Use The **@standard** option resets the operators to the ABEL-HDL standard. The alternate set is chosen with the **@alternate** directive.

Async_reset and Sync_reset

Syntax

```
SYNC_RESET symbolic_state_id : input_expression ;
ASYNC_RESET symbolic_state_id : input_expression ;
```

Purpose

In symbolic state descriptions, the **SYNC_RESET** and **ASYNC_RESET** statements specify synchronous or asynchronous state machine reset logic in terms of symbolic states.

Use

symbolic_state_id	An identifier used for reference to a symbolic state.
input_expression	Any expression.

Examples

```
ASYNC_RESET Start : Reset ;
```

```
SYNC_RESET Start : Reset ;
```

See Also

State
State_diagram
"Using Symbolic State Descriptions" in Chapter 3, "Design Considerations"

Case

Syntax

```
CASE  expression : state_exp;
[ expression : state_exp; ] ...
ENDCASE ;
```

Purpose

Use the CASE statement in a **State_diagram** to indicate transitions of a state machine when multiple conditions affect the state transitions.

Use

expression An expression.

state_exp An expression identifying the next state, optionally followed by WITH transition equations.

You can nest CASE statements with If-Then-Else, GOTO, and other CASE statements, and you can use equation blocks.

Note: *Equation blocks used within a conditional expression such as IF-THEN, CASE, or WHEN-THEN result in logic functions that are logically ANDed with the conditional expression that is in effect.*

The state machine advances to the state indicated by *state_exp* (following the expression that produces a true value). If no expression is true, the result is undefined, and the resulting action depends on the device being used. (For devices with D flip-flops, the next state is the cleared register state.) For this reason, you should be sure to cover all possible conditions in the CASE statement expressions. If the expression produces a numeric rather than a logical value, 0 is false and any non-zero value is true. The expressions contained within the Case-endcase keywords must be mutually exclusive (only one of the expressions can be true at any given time). If two or more expressions within the same Case statement are true, the resulting equations are undefined.

Examples

```
"Mutually exclusive Case statement
case a == 0 : 1 ;
     a == 1 : 2 ;
     a == 2 : 3 ;
     a == 3 : 0 ;
endcase ;

"Not mutually exclusive Case statement
case (a == 0) : 1 ;
     (a == 0) & (B == 0) : 0 ;
endcase ;
```

See Also

State_diagram
Goto
If-then-else
With

Constant Declarations

See = (Constant Declarations).

Declarations

Syntax

`Declarations` *`declarations`*

Purpose

The declarations keyword allows you to declare declarations (such as sets or other constants) in any part of the ABEL-HDL source file.

Use

declarations You can use any declarations after the **Declarations** keyword.

The Declarations keyword is not necessary for declarations immediately following the module and / or title statement(s).

Examples

An example of declared equations is shown below:

```
module castle
  moat device  'P16V8C'; "declarations implied
  A,B        pin 1,2;
  Out1       pin 15 istype 'com';
Equations
  Out1 = A & B;

Declarations "declarations keyword required
  C,D,E,F    pin 3,4,5,6;
  Out2   pin 16 istype 'com';
  Temp1 = C & D;
  Temp2 = E & F;

Equations
  Out2 = Temp1 # Temp2;
end;
```

See Also

demo1800.abl

Device

Syntax

device_id **DEVICE** *real_device* ;

Purpose

The device declaration statement associates the device name used in a module with an actual programmable logic device on which designs are implemented.

Use

device_id An identifier used for the programmer to load filenames.

real_device A string describing the architecture name of the real device represented by *device_id*.

The device declaration is optional.

You should give device identifiers, used in device declarations, valid filenames since JEDEC files are created by appending the extension .jed to the identifier. The architecture name of the programmable logic device is indicated by the string, *real_device*.

The ending semicolon is required.

Examples

```
D1 DEVICE 'P16R4' ;
```

End

Syntax

end *module_name*

Purpose

The **end** statement denotes the end of the module.

Use

The end statement can be followed by the module name. For multi-module source files, the module name is required.

Equations

Syntax

```
equations
element [?]= condition ;
element [?]:= condition ;
when-then-else_statement ;
```

Purpose

The equations statement defines the beginning of a group of equations associated with a device.

Use

condition	An expression.
element	An identifier naming a signal, set of signals, or actual set to which the value of the expression is assigned.
expression	An expression.
=, :=, ?= and ?:=	Combinational and registered (pin-to-pin) on-set and dc-set assignment operators.
when-then-else	When-then-else statements.

Equations specify logic functions with an extended form of Boolean algebra.

A semicolon is required after each equation.

The equations following the equation statement are equations as described in Chapter 2, "Language Structure."

CAUTION: Use the := and ?:= operators only when writing pin-to-pin registered equations. Use the = and ?= assignment operators for registered equations with detailed dot extensions.

Examples

A sample equations section follows:

```
equations
A = B & C # A ;
[W,Y] = 3 ;
!F = (B == C) ;
Output.D = In1 # In2
```

See Also

When-Then-Else
Module
State_diagram
Truth_table
"Operators, Expressions and Equations" in Chapter 2, "Language Structure"

Functional_block

Syntax

```
DECLARATIONS
instance_name FUNCTIONAL_BLOCK source_name ;

EQUATIONS
instance_name.port_name = signal_name;
```

Purpose

You can use a **functional_block** declaration in an upper-level ABEL-HDL source to instantiate a declared lower-level module and make the ports of the lower-level module accessible in the upper-level source. You must declare modules with an **interface** declaration before you can instantiate them with a **functional_block** statement.

Use

instance_name — A unique identifier for this instance of the functional block in the current source.

source_name — The name of the lower-level module that is being instantiated.

Note: *When a module is instanced by an upper-level source, any signal attributes (explicit or implied) are inherited by the upper-level source signals. Therefore, you do not need to specify ISTYPEs in higher-level sources for instantiated signals.*

Creating Multiple Instances

You can use the range operator (..) to instantiate multiple instance names of the module. For example,

```
CNT0..CNT3 functional_block cnt4 ;
```

creates 4 instances of the lower-level module cnt4.

Mapping Ports to Signals

Signal names are mapped to port names, using equations (similar to wiring the signals on a schematic). You need to specify only the signals used in the upper-level source, if default values have been specified in the lower-level module interface statement. See "Interface (lower-level)" in this chapter for more information on setting default values.

To specify the signal wiring, map signal names to the lower-level module port names with dot extension notation. There are three kinds of wire: input, output, and interconnect.

Input Wire Connects lower-level module inputs to upper-level source inputs.

```
instance.port = input
```

Output Wire Connects upper-level source outputs to lower-level module outputs.

```
output = instance.port
```

Interconnect Wire Connects the outputs of one instance of a lower-level module to another instance's inputs.

```
instance0.port = instance1.port
```

Examples

```
module counter;

cnt4 interface (ce, ar, clk, [q0..q3]); // cnt4's top-level inter
face declaration.
CNT0..CNT3 functional_block cnt4;            // Four instances of
 cnt4.

Clk, AR, CE pin;
Q0..Q3 pin;

equations
        CNT0.[clk, ar, ce] = [Clk; AR, CE];    // Connecting to C
lk, AR, and CE inputs.
        CNT0.[q0..q3] = [Q0..Q3];          // Connecting to Q0..Q
3 outputs.
end
```

Figure 6-10 shows how the above ABEL-HDL file wires the upper-level source's signals to the lower-level module's ports. Note that the above file instantiates four instances of **cnt4**, but only one (CNT0) is wired.

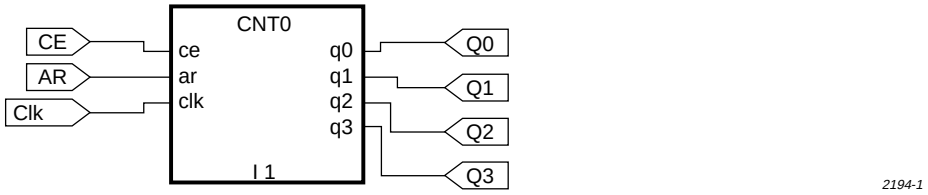
Overriding Default Values

You can override the default values given in a lower-level module's interface statement by specifying default equations in the higher-level source. For example, if you have specified a default value of 1 for the signal **ce** in interface **cnt4** (but in instance **CNT0,** you want **ce** to be 0), you would write:

```
CNT0.ce = 0 ;
```

This equation overrides the 1 with a 0. If you override the default values, you may want to re-optimize the post-linked design.

Figure 6-10
Wiring of CNT0



Unused Outputs (No Connects)

If you do not want to use a lower-level module's outputs, specify them as No Connects (NC) by not wiring up the port to a physical pin. For example, to make a 3-bit counter out of a 4-bit counter in the upper-level source, you might use the following wiring equations:

```
q2..q0 pin ; "upper-level signals
Equations
  [q2..q0] = CNT_A.[q2..q0]
```

See Also

Interface (top-level)
"Hierarchy" in Chapter 3, "Design Considerations
hiermult.abl
hmult2.abl

Fuses

Syntax

```
FUSES
fuse_number = fuse_value ;
       or
FUSES
[ fuse_number_set ] = fuse_value ;
```

Purpose

The **fuses** section explicitly declares the state of any fuse in the targeted device.

Use

fuse_number	The fuse number obtained from the logic diagram of the device.
fuse_number_set	The set of fuse numbers (contained in square brackets).
fuse_value	The number indicating the state of fuse(s).

The **fuses** statement provides device-specific information, and precludes changing devices without editing the statement in the source file.

Fuse values that appear on the right side of the = symbol can be any number. If a single fuse number is specified on the left side of the = symbol, the least significant bit (LSB) of the fuse value is assigned to the fuse. A 0 indicates an intact fuse and a 1 indicates a blown fuse. In the case of multiple fuse numbers, the fuse value is expanded to a binary number and truncated or given leading zeros to obtain fuse values for each fuse number.

CAUTION: When fuse states are specified using the FUSES section, the resulting fuse values supersede the fuse values obtained through the use of equations, truth tables and state diagrams, and affect device simulation accordingly.

ABEL-HDL has a limit of 128 fuses per statement, due to the set size limitations.

Examples

```
FUSES
3552 = 1 ;
[3478...3491] = ^Hff;
```

See Also

cnt10rom.abl

Goto

Syntax

```
GOTO state_exp ;
```

Purpose

The **GOTO** statement is used in the **State_diagram** section to cause an unconditional transition to the state indicated by *state_exp*.

Use

state_exp An expression identifying the next state, optionally followed by WITH transition equations.

GOTO statements can be nested with If-Then-Else and CASE statements.

Examples

```
GOTO 0 ; "goto state 0
GOTO x+y ;  "goto the state x + y
```

See Also

State_diagram
Case
If-then-else
With

If-Then-Else

Syntax

```
IF exp THEN state_exp
[ ELSE state_exp ] ;
```

Chained IF-THEN-ELSE:

```
IF expr THEN state_exp
             ELSE IF exp THEN state_exp
                  ELSE state_exp ;
```

Nested IF-THEN-ELSE:

```
IF exp THEN state_exp
   ELSE   IF exp THEN
                IF exp THEN state_exp
                ELSE state_exp
          ELSE state_exp ;
```

Nested IF-THEN-ELSE with Blocks:

```
IF exp THEN
{       IF exp THEN state_exp
        IF exp THEN state_exp
}
ELSE state_exp ;
```

Purpose

The **If-then-else** statements are used in the **State_diagram** section to describe the progression from one state to another.

Use

exp An expression.

state_exp An expression or block identifying the next state, optionally followed by WITH transition equations.

CAUTION: If-Then-Else is only supported within a state_diagram description. Use When-Then-Else for equations.

Note: *Equation Blocks used within a conditional expression (such as If-then, Case, or When-then) results in logic functions that are logically ANDed with the conditional expression that is in effect.*

The expression following the **If** keyword is evaluated, and if the result is true, the machine goes to the state indicated by the *state_exp*, following the **Then** keyword. If the result of the expression is false, the machine jumps to the state indicated by the **Else** keyword.

Any number of **If** statements may be used in a given state, and the **Else** clause is optional. The indenting and formatting of an **If-then-else** statement is not significant: breaking a complex transition statement across many lines (and indenting) improves readability.

If-then-else statements can be nested with **Goto**, **Case**, and **With** statements. **If-then-else** and **Case** statements can also be combined and nested.

Chained IF-THEN-ELSE Statements:

Any number of **If-then-else** statements can be chained, but the final statement must end with a semicolon. The chained **If-then-else** statement is intended for situations where the conditions are not mutually exclusive. The **Case** statement more clearly expresses the same function as chained mutually-exclusive **If-then-else** statements.

Chained **If-then-else** statements can provide multiway branching transition logic. For example, multiple **If-then-else** statements can be chained to describe a three-way branch in the following manner:

```
STATE S0:
   IF (address < ^h0400)
        THEN S0
   ELSE
   IF (address <= ^hE100)
        THEN S2
   ELSE
        S1;
```

Examples

```
if A==B then 2 ;   "if A equals B goto state 2
if x-y then j else k; "if x-y is not 0 goto j, else goto k
if A then b*c;     "if A is true (non-zero) goto state b*c
```

Chained IF-THEN-ELSE

```
if a then 1
  else
if b then 2
  else
if c then 3
  else 0 ;
```

Nested IF-THEN-ELSE with Blocks

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

Nested IF-THEN-ELSE Statements

A complex state transition could be written with nested transitions in the following manner:

```
STATE S0:
    CASE (select == 1): IF (address == ^h0100)
                             THEN S16
                        ELSE
                        IF (address > ^hE100)
                             THEN S17
                        ELSE
                             S0;

         (select == 2): S2;

         (select == 3): IF (address <= ^hE100)
                             THEN IF (reset)
                                       THEN S3
                                  ELSE S0;
                        ELSE S17;

         (select == 0): S0;
    ENDCASE;
```

See Also

State_diagram
Case
Goto
With

Interface (top-level)

Syntax

source_name `INTERFACE (input/set[=value] -> output/set :> bidir/set);`

Purpose

The **interface** keyword declares lower-level modules and their ports (signals) that are used in the current source. This declaration is used in conjunction with a **functional_block** declaration for each instantiation of the module.

Use

`module_name`	The name of the module being declared.
`inputs->outputs:>bidirs`	A list of signals in the lower-level module used in the current source. Signal names are separated by commas. Use -> and :> to indicate the direction of each port of a functional block.
`value`	An optional default value for an input that overrides defaults in the lower-level module.

If the lower-level module uses the **interface** keyword to declare signals, the upper-level source interface statement must exactly match the signal listing.

CAUTION: Interface declarations cannot contain dot extensions. If you need a specific dot extension across a source boundary (to resolve feedback ambiguities, for example), you must introduce an intermediate signal into the lower-level module to provide the connection to the higher-level source. All dot extension equations for a given output signal must be located in the ABEL-HDL module in which the signal is defined. No references to the signal's dot extensions can be made outside of the ABEL-HDL module.

Note: *When you instantiate a lower-level module in an upper-level source, any signal attributes (either explicit or implicit) are inherited by the higher-level source signals that map to the lower-level signals. Do not specify ISTYPEs for instantiated signals.*

Examples

```
module top;
cnt4 interface (ce,ar,clk -> [q3..q0])
```

Map port names to signal names with equations. See **functional_block.**

See Also

Functional_block
"Hierarchy" in Chapter 3, "Design Considerations"

bjacktop.abl	hiermult.abl
cnt4.abl	hmult2.abl

Interface (lower-level)

Syntax

```
MODULE module_name
INTERFACE (input/set[=port_value] -> output/set [:> bidir/set]);
```

Purpose

The **interface** declaration is optional for lower-level modules. Use the **interface** declaration in lower-level modules to assign a default port list and input values for the module when instantiated in higher-level ABEL-HDL sources. If you use the interface statement in an instantiated module, you must declare the signals and sets in the upper-level source in the same order and grouping as given in the **interface** statement in the lower-level module.

Declaring signals in the lower-level module, although optional, does allow the compiler to check for signal declaration mismatches and therefore reduces the possibility of wiring errors.

Use

module_name	The standard module statement.
signal/set	Signals or sets in the lower-level module used as ports to higher-level sources. Use -> and :> to indicate the direction of each port of a functional block. Use commas to separate groups of signals
port_value	The default value for the port for input signals only. Default values do not apply to output and bidirectional signals.

Declared Signals

Declared signals can be a list of lower-level pins, sets, or a combination of both. The following constraints apply to the different signal types:

Signal Type	Constraints
Input	Default values must be binary if applied to an individual bit, or any positive integer applied to a set. All inputs must be listed.
Output	Unlisted outputs are interpreted as No connects (NC). Unlisted, fed-back outputs are interpreted as nodes in the upper-level source, following the naming convention *instance_name/node_name*
Bidirectional	Listing bidirectional signals is optional, except for those with output enable (OE). If you specify bidirectional signals, the compiler checks for invalid wire connections.

CAUTION: Interface declarations cannot contain dot extensions. If you need a specific dot extension across a source boundary (to resolve feedback ambiguities, for example), you must introduce an intermediate signal into the lower-level module to provide the connection to the higher-level source. All dot extension equations for a given output signal must be located in the ABEL-HDL module in which the signal is defined. No references to the signal's dot extensions can be made outside of the ABEL-HDL module.

Note: *When you instantiate a lower-level module in a higher-level source, any signal attributes (explicit or implicit) are inherited by the higher-level source signals that map to the lower-level signals. Do not specify ISTYPEs for instantiated signals.*

Unlisted Signals

If you do not list some signals of the lower-level module in the interface statement, the following rules apply:

Unlisted Pins Are:	**The Compiler Interprets Them As:**
Inputs or Bidirectionals with OE	Errors
Outputs	No Connects (NC), and they can be removed
Feedback outputs	Nodes in the upper-level source, following the naming convention: *instance_name/node_name*

Examples

The following interface statement declares inputs ce, ar, and clk (giving default values for two of them) and outputs q3 through q0.

```
module cnt4 interface (ce=1,ar=1,clk -> [q3..q0]) ;
```

Specifying default values allows you to instantiate **cnt4** without declaring the **ce** and **ar** inputs in the upper-level source. If you do not declare these inputs, they are replaced with the constants 1 and 0, respectively. Since these constants may affect optimization, you may need to re-optimize the lower-level module with the constants.

Note: *Supported default values are 1, 0, or X (don't care). You can give default values for a set with a positive integer, and each digit of the integer's binary form supplies the default value for the corresponding signal in the set.*

See Also

Interface (top-level)
Functional_block
"Hierarchy" in Chapter 3, "Design Considerations"
hiermult.abl hmult2.abl

Istype _ Attribute Declarations

Syntax

```
signal [, signal...] [PIN | NODE [##s ] ]  ISTYPE 'attr [,attr  ]...';
```

Purpose

The ISTYPE statement defines attributes (characteristics) of signals (pins and nodes). You should use signal attributes to remove ambiguities in architecture-independent designs. Even when a device has been specified, using attributes ensures that the design operates consistently if the device is changed later.

Use

signal A pin or node identifier.

attr A string that specifies attributes for the signal(s). Supported attributes are described below.

Signal attributes are specified with the ISTYPE statement, which can be combined with pin or node declarations in a single declaration. The attributes defined with ISTYPE specify the architectural constraints for signals that have not been assigned to a specific device, pin, or node number, or a specified device (and / or pin number) that has programmable characteristics.

All attributes listed on the right side of the ISTYPE statement are applied to each signal specified on the left side.

Declarations of the pin and node names used in the ISTYPE statement must be made before or with the ISTYPE statement.

Table 6-5 summarizes the available attributes.

CAUTION: If you do not specify signal attributes with Istype, the compiler makes assumptions about signal attributes that may or may not be what you intended.

See Also

.ext—Dot Extensions

Table 6-5
Attributes

Dot Ext.	Arch. Indep.	Description
'buffer'		No Inverter in Target Device
'collapse'		Collapse (remove) this signal. [1]
'com'	✓	Combinational output
'dc'	✓	Unspecified logic is don't care. [2]
'invert'		Inverter in Target Device
'keep'		Do not collapse this signal from equations. [1]
'neg'	✓	Unspecified logic is 1. [2]
'pos'	✓	Unspecified logic is 0. [2]
'retain'	✓	Do not minimize this output. Preserve redundant product terms. [3]
'reg'	✓	Clocked Memory Element.
'reg_d'		D Flip-flop Clocked Memory Element
'reg_g'		D Flip-flop Gated Clock Memory Element
'reg_jk'		JK Flip-flop Clocked Memory Element
'reg_sr'		SR Flip-flop Clocked Memory Element
'reg_t'		T Flip-flop Clocked Memory Element
'xor'		XOR Gate in Target Device

[1] *If neither 'keep' nor 'collapse' is specified, the optimization or fitter programs can keep or collapse the signal as needed to optimize the circuit.*

[2] *The 'dc,' 'neg,' and 'pos' attributes are mutually exclusive.*

[3] *The 'retain' attribute only controls optimization performed by ABEL-HDL Compile Logic. To preserve redundant product terms, you must also specify no reduction for the Reduce Logic and fitting (place and route) programs.*

'buffer' The target architecture does not have an inverter between the associated flip-flop (if any) and the actual output pin.

'invert' The target architecture has an inverter between the associated flip-flop (if any) and the actual output pin.

Control of output inversion in devices is accomplished through the use of the 'invert' or 'buffer' attributes. These attributes enforce the existence ('invert') or non-existence ('buffer') of a hardware inverter at the device pin associated with the output signal specified.

In registered devices, the 'invert' attribute ensures that an inverter is located between the output pin and its associated register output.

Note: *Ensuring an inverter is important for both pin-to-pin and detailed design descriptions because the location of the inverter affects a register's reset, preset, preload, and powerup behavior as observed on the associated output pin.*

'collapse' Collapse (remove) this combinational node. If neither 'keep' nor 'collapse' is specified, the optimization and fitter programs will keep or collapse the node for best optimization. In the following example, signal **b** is given the 'collapse' attribute:

```
module coll_b
a,c,d,e   pin ;
b         node istype 'collapse'

equations
a = b & e;
b = c & d;
end
```

The resulting equation collapses b out of the equations:

```
a = c & d & e ;
```

'keep' Do not collapse this combinational node from equations. In the example under 'collapse,' **b** would be retained.

'com' Specifies a combinational symbol.

'dc,' 'neg,' and 'pos'

These attributes control the value of unspecified logic in your design, and are mutually exclusive. The values they specify are shown below:

Istype Attribute	Unspecified Logic is
'dc'	X (don't care)
'neg'	1
'pos'	0

The 'dc' attribute is equivalent to the @DCSET directive, except it operates on signals instead of applying to a whole section.

Note: *The 'neg' or 'pos' attribute is implied if a device is specified. For example, 'neg' is implied if the device output is inverted (for example, with a 16L8).*

CAUTION: The @DCSET directive overrides 'dc,' 'neg,' and 'pos'.

'reg' The signal specified is a registered output. Equations, state diagrams, and truth tables will generate logic for a D-type flip-flop, normalized to take into account any inverters in the target device.

'reg_d' The signal specified is a registered output. Equations, state diagrams, and truth tables will generate logic for a D-type flip-flop, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer').

'reg_g' The signal specified is a registered output. Equations, state diagrams, and truth tables will generate logic for a D-type flip-flop, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Write equations and truth tables using the .D and .CE dot extensions when you use this attribute.

'reg_jk' The signal specified is a JK-type registered output. State diagrams generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Write Equations and truth tables using the .J and .K dot extensions when you use this attribute.

'reg_sr' The signal specified is an SR-type registered output. State diagrams will generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Write equations and truth tables using the .S and .R dot extensions when you use this attribute.

'reg_t' The signal specified is a T-type registered output. State diagrams will generate logic for this register type, but you must specify if the output is inverted in the target device (with attribute 'invert' or 'buffer'). Write equations and truth tables using the .T dot extension when you use this attribute.

'retain' Do not minimize this output. Preserve redundant product terms for the signal.

'xor' The signal specified will be implemented using an XOR gate fed by two sum-of-products logic circuits. If you use XOR operators in the design equations for this output (or if you use high-level operators that result in XOR operations), then one XOR operator is retained through optimization. Use this attribute if you are implementing your design in an architecture featuring XOR gates.

Examples

```
F0, A istype 'invert, reg' ;
```

This declaration statement defines **F0** and **A** as inverted registered outputs. You must define both **F0** and **A** earlier in the module. The following signal declarations are all supported

```
q3,q2,q1,q0   NODE ISTYPE 'reg_SR';
Clk,a,b,c     PIN 1,2,3,4;
reset         PIN;
reset         ISTYPE 'com';
Output        PIN 15 ISTYPE 'reg,invert';
```

See Also

.ext
Pin
Node
"Dot Extensions" and "Attribute Assignment" in the Chapter 2, "Language Structure"
"Architecture-independent Designs" and "XOR Factor" in the Chapter 3, "Design Considerations"

Library

Syntax

```
LIBRARY 'name' ;
```

Purpose

The LIBRARY statement causes the contents of the indicated file to be inserted in the ABEL-HDL source file. The insertion begins at the LIBRARY statement.

Use

name A string that specifies the name of the library file, excluding the file extension.

The file extension of '.inc' is appended to the name specified, and the resulting filename is searched for. If no file is found, the abel5lib.inc library file is searched.

See Also

Module
@Include

Macro

Syntax

macro_id MACRO [(*dummy_arg* [,*dummy_arg*]...)] {*block* } ;

Purpose

The macro declaration statement defines a macro. Macros are used to include ABEL-HDL code in a source file without typing or copying the code everywhere it is needed.

Use

macro_id An identifier naming the macro.

dummy_arg A dummy argument.

block A block.

A macro is defined once in the declarations section of a module and then used anywhere within the module as frequently as needed. Macros can be used only within the module in which they are declared.

Wherever the *macro_id* occurs, the text in the block associated with that macro is substituted. With the exception of dummy arguments, all text in the block (including spaces and end-of-lines) is substituted exactly as it appears in the block.

When debugging your source file, you can use the **-list** expand option to examine macro statements. The **-list expand** option causes the parsed and expanded source code (and the macros and directives that caused code to be added to the source) to be written to the listing file.

Macros and Declared Equations

Use declared equations for constant expressions (instead of macros) for faster processing. The file, **mac.abl**, in Figure 6-11 demonstrates the difference:

Figure 6-11
Differences Between MACRO and Declared Equations

```
module mac
title 'Demonstrates difference between MACRO and declared equations'
        mac       device 'P16H8';
        A,B,C,    pin 1,2,3;
        X1,X2,X3  pin 14,15,16 istype 'com';
        Y1  macro {B # C};
        Y2 = B # C;

equations
        X1 = A & Y1;
        X2 = A & (Y1);
        X3 = A & Y2;
```

```
" Note: Because Y1 is a text replacement macro the equation
" for X1 will expand to  A & B # C. If the desired function
" was A & (B # C) use parentheses around the macro or use
" a subexpression (Y1=B#C) instead of the macro in the declarations.

" The macro could also be written       Y1 macro {(B # C)};

test_vectors   ([A,B,C] -> [X1,X2,X3])
                [0,0,0] -> [ 0, 0, 0];
                [0,0,1] -> [ 1, 0, 0];
                [0,1,0] -> [ 0, 0, 0];
                [0,1,1] -> [ 1, 0, 0];
                [1,0,0] -> [ 0, 0, 0];
                [1,0,1] -> [ 1, 1, 1];
                [1,1,0] -> [ 1, 1, 1];
                [1,1,1] -> [ 1, 1, 1];
end
```

Examples

The dummy arguments used in the macro declaration allow different actual arguments to be used each time the macro is referenced. Dummy arguments are preceded by a "?" to indicate that an actual argument is substituted for the dummy by the compiler.

The equation,

```
NAND3 MACRO (A,B,C) { !(?A & ?B & ?C) } ;
```

declares a macro named NAND3 with the dummy arguments A, B, and C. The macro defines a three-input NAND gate. When the macro identifier occurs in the source, actual arguments for A, B, and C are supplied.

For example, the equation

```
D = NAND3 (Clock,Hello,Busy) ;
```

brings the text in the block associated with NAND3 into the code, with Clock substituted for ?A, Hello for ?B, and Busy for ?C.

This results in:

```
D = !( Clock & Hello & Busy ) ;
```

which is the three-input NAND.

The macro NAND3 has been specified by a Boolean equation, but it could have been specified using another ABEL-HDL construct, such as the truth table shown here:

```
NAND3 MACRO (A,B,C,Y)

{ TRUTH_TABLE ( [?A ,?B ,?C ] -> ?Y )

                [ 0 ,.X.,.X.] ->  1 ;
                [.X., 0 ,.X.] ->  1 ;
                [.X.,.X., 0 ] ->  1 ;
                [ 1 , 1 , 1 ] ->  0 ; } ;
```

In this case, the line,

```
NAND3 (Clock,Hello,Busy,D)
```

causes the text,

```
TRUTH_TABLE ( [Clock,Hello,Busy] -> D )
               [  0  , .X. ,.X. ] -> 1 ;
              [ .X. ,  0  ,.X. ] -> 1 ;
              [ .X. , .X. , 0  ] -> 1 ;
              [  1  ,  1  , 1  ] -> 0 ;
```

to be substituted into the code. This text is a truth table definition of D, specified as the function of three inputs, Clock, Hello, and Busy. This is the same function as that given by the Boolean equation above. The truth table format is discussed under Truth_table.

Other examples of macros:

```
"macro with no dummy arguments
nodum macro { W = S1 & S2 & S3 ; } ;

onedum MACRO (d) { !?d } ; "macro with 1 dummy argument
```

and when macros are called in logic descriptions:

```
nodum
X = W + onedum(inp) ;
Y = W + onedum( )C ; "note the blank actual argument
```

resulting in:

```
"note leading space from block in nodum
 W = S1 & S2 & S3 ;
X = W + ! inp ;
Y = W + ! C ;
```

Recursive macro references (when a macro definition refers to itself) are not supported, and the compiler halts abnormally. If errors appear after the first use of a macro, and the errors cannot be easily explained otherwise, check for a recursive macro reference by examining the listing file.

See Also

= (Constant Declarations)
"Arguments and Argument Substitution" in Chapter 2, "Language Structure"

Module

Syntax

```
MODULE modname [ ( dummy_arg [,dummy_arg ] ...  ) ]
```

Purpose

The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.

Use

modname An identifier naming the module.

dummy_arg Dummy arguments.

The optional dummy arguments allow actual arguments to be passed to the module when it is processed. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value is substituted.

Examples

```
MODULE MY_EXAMPLE (A,B)
  :
  C = ?B + ?A
```

In the module named MY_EXAMPLE, C takes on the value of "A + B" where A and B contain actual arguments passed to the module when the language processor is invoked.

See Also

Title
Interface (submodule)
End
"Arguments and Argument Substitution" in Chapter 2, "Language Structure"

Node

Syntax

```
[!]node_id [,[!]node_id...] NODE [node# [,node# ]] [ISTYPE 'attributes' ];
```

Purpose

The NODE keyword declares signals assigned to buried nodes.

Use

node_id An identifier used for reference to a node in a logic design.

node# The node number on the real device.

attributes A string that specifies node attributes for devices with programmable nodes. Any number of attributes can be listed, separated by commas. Attributes are listed in Table 2-9 under "Attributes" in Chapter 2,"Language Structure."

Note: *Using the NODE keyword does not restrict a signal to a buried node. A signal declared with NODE can be assigned to a device I/O pin by a device fitter.*

You can use the range operator (..) to declare sets of nodes. The ending semicolon is required after each declaration.

When lists of *node_id* and *node #* are used in one node declaration, there is a one-to-one correspondence between the identifiers and numbers.

The following example declares three nodes A, B, and C.

```
A, B, C NODE  ;
```

The node attribute string, **Istype *'attributes,'*** should be used to specify node attributes. Since a node declaration is only required in a detailed description, use detailed attributes, not pin-to-pin attributes. The ISTYPE statement and attributes are discussed under Istype.

The node declaration,

```
B NODE istype 'reg' ;
```

specifies that node B is a buried flip-flop.

Example

```
a0..a3 node 22..25;
```

assigns a0, a1, a2 and a3 to nodes 22, 23, 24 and 25, respectively.

See Also

Istype
Pin
Module
"Attribute Assignment" in the Chapter 2,"Language Structure
"Architecture-independent Designs" in the Chapter 3, "Design Considerations"

Pin

Syntax

```
[!]pin_id [,[!]pin_id...] PIN  [pin# [, pin# ]] [ISTYPE 'attr' ];
```

Purpose

The PIN keyword declares input and output signals that must be available on a device I/O pin.

Use

pin_id An identifier that refers to a pin in a module.

pin# The pin number on the physical device.

attr A string that specifies pin attributes for devices with programmable pins. Attributes are listed in ISTYPE.

When lists of *pin_id*s and *pin#*s are used in a pin declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one pin number associated with each identifier listed.

You can use the range operator (..) to declare sets of pins. The ending semicolon is required after each declaration.

Note: *Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later by a fitter.*

The **!** operator in pin declarations indicates that the pin is active-low, and is automatically negated when the source file is compiled.

The pin attribute string, **Istype *'attributes,'*** should be used to specify pin attributes. The ISTYPE statement and attributes are discussed under Istype. Istype attribute statements are recommended for all pins.

Examples

```
Clock, !Reset, S1 PIN 1,15,3;
```

Clock is assigned to pin 1, Reset to pin 15, and S1 to pin 3.

```
a0..a3 PIN 2..5 istype 'reg,buffer';
```

Assigns a0, a1, a2 and a3 to pins 2, 3, 4 and 5, respectively.

See Also

Istype
Node
Module
"Architecture-independent Designs" in the chapter 3, "Design Considerations"

Property

Syntax

```
property_id PROPERTY 'string' ;
```

Purpose

The **property** declaration statement allows you to specify additional design information associated with an external processing module (such as a device kit).

The format of the string depends on the fitter to which the property is being passed. See your device kit user manuals for syntax descriptions.

Note: *You can specify properties for any number of fitters in your design, since all fitters process only properties with their property ID and ignore all other properties.*

Use

property_id	Identifies properties relevant to specific external modules, such as fitters.
string	Argument containing the actual property data.

CAUTION: Property IDs and strings can be case-sensitive. Check your vendor's fitter documentation.

CAUTION: Property Information will not be present in pre-route/functional simulation. Consider using schematics (in Synario) to access property features that affect simulation

Example

AMDMACH property 'GROUP A Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0';.

See Also

amd_cm8.abl

State (Declaration)

Syntax

state_id [, *state_id* ...] **STATE** [IN *statereg_id*] ;

Purpose

The State declaration is made to declare a symbolic state name and, optionally, associate it with a state register.

Use

state_id A symbolic state name to be referenced in a symbolic state description.

statereg_id An identifier for a state register.

If your design includes more than one symbolic state machine, use the IN keyword to associate each state with the corresponding state register.

Each state you declare corresponds to one flip-flop in a one-hot machine.

See Also

Async_reset
State_register
Sync_reset
"Symbolic State Declarations" in Chapter 2, "Language Structure"
"Using Symbolic State Descriptions" on page 3-39 in Chapter 3, "Design Considerations"

State (in State_diagram)

Syntax

```
[STATE state_exp : [equation ]
       [equation ]
  :
  :
  :
trans_stmt ; ...]
```

Purpose

The state keyword and the associated section describes one state of a state diagram. It includes a state value (or a symbolic state name), a state transition statement, and optional state output equations.

Use

state_exp — An expression, value, or symbolic state name giving the current state.

equation — An equation that defines the state machine outputs.

trans_stmt — IF-THEN-ELSE, CASE, or GOTO statements, optionally followed by WITH transition equations.

The specification of a state description requires the use of the **State_diagram** syntax (which defines the state machine) and the **If-Then-Else, Case, Goto**, and **With** statements (which determine the operation of the state machine). Symbolic state machines (machines for which the actual state registers and state values are unspecified) require additional declarations for the symbolic state register and state names. See "Symbolic State Declarations" in Chapter 2, "Language Structure" page 2-35.

A semicolon is required after each transition statement.

See Also

Async_reset
Case
@Dcset
Equations
Goto
If-then-else
Module
State
State_diagram
Sync_reset
Truth_table
With
Chapter 3, "Design Considerations"

State_diagram

Syntax

```
State_diagram state_reg
       [-> state_out ]
[STATE state_exp : [equation ]
       [equation ]
  :
trans_stmt ; ...]
```

Purpose

The state description describes the operation of a sequential state machine implemented with programmable logic.

Use

state_reg — An identifier or set of identifiers specifying the signals that determine the current state of the machine. For symbolic state diagrams, this identifier is a symbolic state register name that has been declared with a State_register declaration.

state_out — An identifier or set of identifiers that determines the next state of the machine (for designs with external registers).

state_exp — An expression or symbolic state name giving the current state.

equation — An equation that defines the state machine outputs.

trans_stmt — IF-THEN-ELSE, CASE, or GOTO statements, optionally followed by WITH transition equations.

A semicolon is required after each transition statement.

Use **State_diagram** syntax to define a state machine, and the **If-Then-Else, Case, Goto,** and **With** statements to determine the operation of the state machine. Symbolic state machines (machines for which the actual state registers and state values are unspecified) require additional declarations for the symbolic state register and state names (see "Symbolic State Declarations" in Chapter 2, "Language Structure" page 2-35).

The syntax for the IF-THEN-ELSE, CASE, GOTO, WITH, SYNC_RESET, and ASYNC_RESETstatements are presented here briefly, and are discussed further in their respective sections.

A state machine starts in one of the states defined by *state_exp*. The equations listed after that state are evaluated, and the transition statement (*trans_stmt*) is evaluated after the next clock, causing the machine to advance to the next state.

Equations associated with a state are optional; however, each state must have a transition statement. If none of the transition conditions for a state is met, the next state is undefined. (For some devices, undefined state transitions cause a transition to the cleared register state.)

Transition Statements

Transition statements describe the conditions that cause transition from one state to the next. Each state in a state diagram must contain at least one transition statement. Transition statements can consist of GOTO statements, IF-THEN-ELSE conditional statements, CASE statements, or combinations of these different statements.

GOTO Syntax

```
GOTO state_exp ;
```

The GOTO statement unconditionally jumps to a different state. When GOTO is used, it is the only transition for the current state. Example:

```
STATE S0:
   GOTO S1;  "unconditional branch to state S1
```

CASE Syntax

```
CASE  expression : state_exp ;
[ expression : state_exp ; ] ...
ENDCASE ;
```

The CASE statement is used to list a sequence of mutually-exclusive transition conditions and corresponding next states. Example:

```
STATE S0:
   CASE (sel == 0): S0 ;
        (sel == 1): S1 ;
   ENDCASE
```

CASE statement conditions must be mutually exclusive. No two transition conditions can be true at the same time, or the resulting next state is unpredictable.

IF-THEN-ELSE Syntax

```
IF expression THEN state_exp
[ ELSE state_exp ] ;
```

IF-THEN-ELSE statements specify mutually-exclusive transition conditions. Example:

```
STATE S0:
   IF (address > ^hE100) THEN S1 ELSE S2;
```

You can use blocks in IF-THEN-ELSE statements, for example,

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

The ELSE clause is optional. A sequence of IF-THEN statements with no ELSE clauses is equivalent to a sequence of CASE statements. IF-THEN-ELSE statements can be chained and nested. See IF-THEN-ELSE for more information.

WITH Syntax

```
state_exp WITH equation ;
[equation  ; ]
```

You can use the WITH statement in any of the above transition statements (the GOTO, IF-THEN-ELSE, or CASE statements) in place of a simple state expression. For example, to specify that a set of registered outputs are to contain a specific value after one particular transition, specify the equation using a WITH statement similar to the one shown below:

```
STATE S0:
   IF (reset)
        THEN S9 WITH {
                  ErrorFlag := 1;
                  ErrorAddress := address;
                }
   ELSE
   IF (address <= ^hE100)
        THEN S2
   ELSE
        S0;
```

The WITH statement is also useful when you describe output behavior for registered outputs (since registered outputs written only for a current state would lag by one clock cycle).

SYNC_RESET and ASYNC_RESET Syntax

In symbolic state descriptions the SYNC_RESET and ASYNC_RESET statements are used to specify synchronous or asynchronous state machine reset logic in terms of symbolic states. For example, to specify that a state machine must asynchronously reset to state Start when the Reset input is true, you would write

```
ASYNC_RESET Start : Reset ;
```

See "Symbolic State Declarations" in Chapter 2, "Language Structure," and "State Machines" in Chapter 3, "Design Considerations."

State Descriptions and Pin-to-pin Descriptions

Sequential circuits described with ABEL-HDL's state diagram language are normally written with a pin-to-pin behavior in mind, regardless of the flip-flop type specified.

The state machine shown below operates the same (in terms of the behavior seen on its outputs) no matter what type of register is substituted for 'reg' in the signal declarations. To allow this flexibility, the specification of 'buffer' or 'invert' is required when a state diagram is written for a register type other than 'reg.'

Figure 6-12
Architecture-independent State Machine

```
module statema
title 'State machine example';
  clock,hold,reset    pin;
  P1,P0     pin  istype 'reg,buffer';
  C = .c.;

equations
  [P1,P0].clk   = clock;
  [P1,P0].ar    = reset;

" state declarations...
declarations
  statema = [P1,P0]
  stateA  = [0,0];
  stateB  = [1,0];
  stateC  = [1,1];
  stateD  = [0,1];

state_diagram statema
  state stateA:
       goto stateB;
  state stateB:
       goto stateC;
  state stateC:
       goto stateD;
  state stateD:
       goto stateA;
"test_vectors edited
end
```

See Also

Async_reset	State
Case	State_register
@Dcset	Sync_reset
Equations	Truth_table
Goto	With
If-then-else	"Symbolic State Declarations" in Chapter 2,
Module	"Language Structure"

State_register

Syntax

statereg_id **STATE_REGISTER** ;

Purpose

For symbolic state diagrams, the State_register is made to declare a symbolic state machine name.

Use

statereg_id An identifier naming the state register.

See Also

State
State_diagram
"Symbolic State Declarations" in Chapter 2, "Language Structure"
"Using Symbolic State Descriptions" in Chapter 3, "Design Considerations"

Sync_reset

See **async_reset**

Test_vectors

Syntax

```
Test_vectors [note ]
(input [,input ]... -> output[,output ]...)

[invalues -> outvalues ; ]
  :
```

Purpose

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs.

Note: *Test_vectors are only used with PLD JEDEC file simulation. For functional simulation with the ABEL or Synario Simulators, use a test stimulus file.*

Use

note	An optional string that describes the test vectors.
inputs	An identifier or set of identifiers specifying the names of the input signals, or feedback output signals.
outputs	An identifier or set of identifiers specifying the output signals.
invalues	An input value or set of input values.
outvalues	A pin-to-pin output value or set of output values resulting from the given inputs.

Test vectors are used for simulation of an internal model of the device and functional testing of the design and device. The number of test vectors is unlimited.

The format of the test vectors is determined by the header. Each vector is specified in the format described within the parentheses in the header statement. An optional note string can be specified in the header to describe what the vectors test, and is included as output in the simulation output file, the document output file, and the JEDEC programmer load file.

The table lists input combinations and their resulting outputs. All or some of the possible input combinations can be listed. All values specified in the table must be constants, either declared, numeric, or a special constant (for example, .X. and .C.). Each line of the table (each input/output listing) must end with a semicolon. Test vector output values always represent the pin-to-pin value for the output signals.

Test vectors must be sequential for state machines. Test vectors must go through valid state transitions.

The **Trace** keyword can be used to control simulator output from within the source file.

Functional testing of the physical device is performed by a logic programmer after a device has been programmed. The test vectors become part of the programmer load file.

Examples

Following is a simple test vectors section:

```
TEST_VECTORS
( [A,B] -> [C, D] )

  [0,0] -> [1,1] ;
  [0,1] -> [1,0] ;
  [1,0] -> [0,1] ;
  [1,1] -> [0,0] ;
```

The following test vectors are equivalent to those specified above because values for sets can be specified with numeric constants.

```
TEST_VECTORS
( [A,B] -> [C, D] )

    0   ->   3   ;
    1   ->   2   ;
    2   ->   1   ;
    3   ->   0   ;
```

If the signal identifiers in the test vector header are declared as active-low in the declaration section, then constant values specified in the test vectors are inverted accordingly (interpreted pin-to-pin).

See Also

Module
Trace

Title

Syntax

`title 'string'`

Purpose

The title statement gives a module a title that appears as a header in both the programmer load file and documentation file created by the language processor.

Use

The title is specified in the string following the keyword, **title**. The string is opened and closed by an apostrophe and is limited to 324 characters.

The title statement is optional.

Asterisks in the title string do not appear in the programmer load file header in order to conform with the JEDEC standard.

Examples

An example of a title statement that spans three lines and describes the logic design is shown below:

```
module m6809a
title '6809 memory decode
Jean Designer
Data I/O Corp Redmond WA'
```

See Also

Module

Trace

Syntax

```
TRACE (inputs -> outputs) ;
```

Purpose

The TRACE statement controls which inputs and outputs are displayed in the simulation output.

Use

TRACE statements can be placed before a test vector section, or embedded within a sequence of test vectors.

Note: *Test_vectors are only used with Equation and JEDEC file simulation. For functional simulation with the Verilog Simulator, use a test stimulus file.*

Examples

```
TRACE        ( [A,B] -> [C  ] );
TEST_VECTORS ( [A,B] -> [C,D] )
                 0   ->   3    ;
                 1   ->   2    ;
TRACE        ( [A,B] -> [  D] );
                 2   ->   1    ;
                 3   ->   0    ;
```

See Also

Test_vectors

Truth_table

Syntax

```
TRUTH_TABLE ( in_ids -> out_ids )
              inputs -> outputs ;
    or
TRUTH_TABLE ( in_ids :> reg_ids )
              inputs :> reg_outs ;
    or
TRUTH_TABLE
( in_ids :> reg_ids  -> out_ids )
  inputs :> reg_outs -> outputs ;
```

Purpose

Truth tables specify outputs as functions of input combinations, in a tabular form.

Use

in_ids	Input signal identifiers.
out_ids	Output signal identifiers.
reg_ids	Registered signal identifiers.
inputs	The inputs to the logic function.
outputs	The outputs from the logic function.
reg_outs	The registered (clocked) outputs.
-> :>	Indicates the input to output function for combinational (->) and registered (:>) outputs.

Truth tables are another way to describe logic designs with ABEL-HDL and may be used in lieu of (or in addition to) equations and state diagrams. A truth table is specified with a header describing the format of the table and with the table itself.

A semicolon is required after each line in the truth table.

The truth table header can have one of the three forms shown above, depending on whether the device has registered or combinational outputs or both.

The inputs and outputs (both registered and combinational) of a truth table are either single signals, or (more frequently) sets of signals. If only one signal is used as either the input or output, its name is specified. Sets of signals used as inputs or outputs are specified in the normal set notation with the signals surrounded by brackets and separated by commas (see "Sets" in Chapter 2, "Language Structure").

The syntax shown in the first form defines the format of a truth table with simple combinational outputs. The values of the inputs determine the values of the outputs.

The second form describes a format for a truth table with registered outputs. The symbol ":" preceding the outputs distinguishes these outputs from the combinational outputs. Again the values of the inputs determine the values of the outputs, but now the outputs are registered or clocked: they will contain the new value (as determined by the inputs) after the next clock pulse.

The third form is more complex, defining a table with both combinational and registered outputs. It is important in this format to make sure the different specification characters "-" and ":" are used for the different types of outputs.

Truth Table Format

The truth table is specified according to the form described within the parentheses in the header. The truth table is a list of input combinations and resulting outputs. All or some of the possible input combinations may be listed.

All values specified in the table must be constants, either declared, numeric, or the special constant .X. Each line of the table (each input/output listing) must end with a semicolon.

The header defines the names of the inputs and outputs. The table defines the values of inputs and the resulting output values.

Examples

This example shows a truth table description of a simple state machine with four states and one output. The current state is described by signals A and B, which are put into a set. The next state is described by the registered outputs C and D, which are also collected into a set. The single combinational output is signal E. The machine simply counts through the different states, driving the output E low when A equals 1 and B equals 0.

```
TRUTH_TABLE ( [A,B] :> [C,D] -> E )
                0   :>   1   -> 1 ;
                1   :>   2   -> 0 ;
                2   :>   3   -> 1 ;
                3   :>   0   -> 1 ;
```

Note that the input and output combinations are specified by a single constant value rather than by set notation. This is equivalent to:

```
[0,0] :> [0,1] -> 1 ;
[0,1] :> [1,0] -> 0 ;
[1,0] :> [1,1] -> 1 ;
[1,1] :> [0,0] -> 1 ;
```

When writing truth tables in ABEL-HDL (particularly when describing registered circuits) follow the same rules for dot extensions, attributes, and pin-to-pin / detailed descriptions described earlier for writing equations. The only difference between equations and truth tables is the ordering of the inputs and outputs.

The following two fragments of source code, for example, are functionally equivalent:

Fragment 1:

```
equations

   q := a & load # !q.fb & !load;
```

Fragment 2:

```
truth_table   (  [a ,q.fb,load] :> q)
                 [0 ,  0 ,  0 ] :> 1;
                 [0 ,  1 ,  0 ] :> 0;
                 [1 ,  0 ,  0 ] :> 1;
                 [1 ,  1 ,  0 ] :> 0;
                 [0 ,  0 ,  1 ] :> 0;
                 [1 ,  0 ,  1 ] :> 1;
                 [0 ,  1 ,  1 ] :> 0;
                 [1 ,  1 ,  1 ] :> 1;
```

As an example, the following truth table defines an exclusive-OR function with two inputs (A and B), one enable (en), and one output (C):

```
TRUTH_TABLE ( [en, A , B ] ->  C )
              [ 0,.X.,.X.] -> .X.;" don't care w/enab off
              [ 1, 0 , 0 ] ->  0 ;
              [ 1, 0 , 1 ] ->  1 ;
              [ 1, 1 , 0 ] ->  1 ;
              [ 1, 1 , 1 ] ->  0 ;
```

See Also

Module
Equations
State_diagram
@Dcset
led1.abl
led7.abl

When-Then-Else

Syntax

```
[ WHEN condition    THEN ] [  ! ] element=expression;
[ ELSE equation ];
          or
[ WHEN condition    THEN ] equation;  [ ELSE equation];
```

Purpose

The **When-then-else** statement is used in equations to describe a logic function.

Use

condition Any valid expression.

element An identifier naming a signal or set of signals, or an actual set, to which the value of the expression is assigned.

expression Any valid expression.

=, :=, ?= and ?:= Combinational and registered (pin-to-pin) assignment operators.

Equations use the assignment operators = and ?= (combinational), and := and ?:= (registered) described in Chapter 2, "Language Structure."

The complement operator, "!", expresses 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. Using the complement operator on the left side of equations is also supported; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

CAUTION: When-Then-Else is only supported in equations. Use If-Then-Else in state_diagram descriptions.

Note: *Equation blocks in conditional expressions such as WHEN-THEN result in logic functions that are logically ANDed with the conditional expression that is in effect.*

Examples

```
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;  }
```

See Also

"Equations" in Chapter 2, "Language Structure"

With

Syntax

trans_stmt state_exp WITH *equation*
[*equation*]..;

Purpose

The WITH statement is used in the **State_diagram** section. When used in conjunction with the IF-THEN or CASE statement, it allows output equations to be written in terms of transitions.

Use

trans_stmt The IF-THEN-ELSE, GOTO, or CASE statement.

state_exp The next state.

equation An equation for state machine outputs.

You can use the WITH statement in any transition statement, in place of a simple state expression.

The WITH statement is also useful when you are describing output behavior for registered outputs, since registered outputs written only for a current state would lag by one clock cycle.

To specify that a set of registered outputs should contain a specific value after one particular transition, specify the equation using a WITH statement similar to the one shown below:

```
STATE S0:
   IF (reset)  THEN S9 WITH { ErrorFlag := 1;
                              ErrorAddress := address;}
   ELSE   IF (address <= ^hE100)
          THEN S2
          ELSE   S0;
```

Examples

```
State 5 :
           IF a == 1 then 1   WITH { x := 1 ;
                                     y := 0 ;}
           ELSE 2    WITH { x := 0 ;
                            y := 1 ;}
```

See Also

State_diagram
Case
Goto
If-then-else

XOR_Factors

Syntax

```
XOR_Factors
signal name = xor_factors  ;
```

Purpose

Use XOR_factors to specify a Boolean expression to be factored out of (and XORed with) the sum-of-products reduced equations. Factors can dramatically reduce the reduced equations if you use a device featuring XOR gates.

Use

XOR_factors converts a sum of products (SOP) equation into an exclusive OR (XOR) equation. The resulting equation contains the sum of product functions that, when exclusive ORed together, have the same function as the original. The XOR_Factors equation is divided into the original equation, with the factor (or its complement) on one side of the XOR and the remainder on the other.

After deciding the best XOR_Factors, remember to revise the source file to use an XOR device for the final design.

Note: *The assignment operator you use in XOR_Factors equations must match the assignment operator in the Equations section.*

Examples

```
!Q16 =    A &  B &     !D
       #  A &  B & !C
       #      !B &  C &  D
       # !A &       C &  D;
```

Reordering the product terms indicates that (A & B) and (C & D) are good candidate factors, as shown below:

```
!Q16 =  A &  B & (!C # !D)
    # (!A # !B) &  C &  D;
```

If we process the following source file, the program reduces the equations according to the XOR_Factors, A & B.

```
module xorfact
   xorfact     device 'P20X10';
   Clk,OE      pin 1,13;
   A,B,C,D     pin 2,3,4,5;
   Q16         pin 16 istype 'reg,xor';
XOR_Factors
   Q16 := A & B;
equations
   !Q16   :=   A &  B &      !D
           #       !B &  C &  D
           #  !A &       C &  D
           #   A &  B & !C;
end
```

Using A & B as the XOR_Factors, the reduced equations are

```
!Q16 := ((A & B) $ (C & D));
```

Example 2

The example octalf.abl uses a more complex high-level equation:

```
module OCTALF
title 'Octal counter with xor factoring
Adam Zilinskas  Data I/O Corp.'

   octalf device 'P20X8';
   D0..D7                 pin  3..10;
   Q7..Q0                 pin  15..22 istype 'reg,xor';
   CLK,I0,I1,OC,,CarryIn  pin  1,2,11,13,23;
   CarryOut               pin  14 istype 'com';
   H,L,X,Z,C = 1, 0, .X., .Z., .C.;

   Data  = [D7..D0];
   Count = [Q7..Q0];

   Mode  = [I1,I0];
   Clear = [ 0, 0];
   Hold  = [ 0, 1];
   Load  = [ 1, 0];
   Inc   = [ 1, 1];

xor_factor
        Count.FB := Count & I0;

" ..comments removed..
equations
   Count  := (Count.FB + 1) & (Mode == Inc) & !CarryIn
           # (Count.FB    ) & (Mode == Inc) &  CarryIn
           # (Count.FB    ) & (Mode == Hold)
```

```
            # (Data     ) & (Mode == Load)
            # (0        ) & (Mode == Clear);

   !CarryOut       = !CarryIn & (Count.FB == ^hFF);

   Count.C  = CLK;
   Count.OE = !OC;
"..test vectors removed..
"..comments removed..
end OCTALF;
```

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Z