Appendix C

Introduction to ABEL Hardware Description Language

This appendix provides a brief overview of the Advanced Boolean Equation Language (ABEL) which is an industry–standard Hardware Description Language (HDL) used in Programmable Logic Devices (PLDs).

C.1 Introduction

The Advanced Boolean Equation Language (ABEL) is an easy–to–understand and use programming language that allows the user to describe the function of logic circuits. It is now an industry–standard allows you to enter behavior–like descriptions of a logic circuit. Developed by Data I/O Corporation for Programmable Logic Devices (*PLDs*), ABEL is now an industry–standard *Hardware Description Language (HDL)*. An example of how ABEL is used, was given in an example in Chapter 11. There are other hardware description languages such as the *VHSIC** *Hardware Description Language (VHDL)* and *Verilog*. ABEL is a simpler language than VHDL and Verilog. ABEL can be used to describe the behavior of a system in a variety of forms, including logic equations, truth tables, and state diagrams using C–like statements. The ABEL compiler allows designs to be simulated and implemented into PLDs such as PALs, CPLDs and FPGAs.

We will not discuss ABEL in detail. We will present a brief overview of some of the features and syntax of ABEL. For more advanced features, the reader may refer to an ABEL manual or the Xilinx on–line documentation.

C.2 Basic Structure of an ABEL Source File

An ABEL source file consists of the following elements:

- Header: including Module, Options, and Title
- Declarations: Pin, Constant, Node, Sets, States, Library
- Logic Descriptions: Equations, Truth_table, State_diagram
- Test Vectors: Test_vectors
- End

Keywords (words recognized by ABEL as commands, e.g. **goto**, **if**, **then**, **module**, etc.) are not case sensitive. User–supplied names and labels (identifiers) can be uppercase, lowercase or mixed–case, but are case–sensitive, for instance `output1` is different from `Output1`.

* *VHSIC is an acronym for Very High Speed Integrated Circuits.*

Example C.1

A source file is given below.

module *module name*

[**title** *string*] (optional)

[deviceID **device** deviceType;] (optional)

pin declarations (optional)

other declarations (optional)

comments (optional)

equations

equations

[**Test_Vectors**] (optional)

test vectors

end *module name*

Using the structure of this source file, write a source file that implements a 4–bit full adder.

Solution:

```
module circuit_05;
title ' EE124 Lab Experiment 05'
Four_bit_adder device '74HC283';
" input pins
A1, B1, A2, B2, A3, B3, A4, B4, CIN pin 5, 6, 3, 2, 14, 15, 12,
11, 7;

 " output pins
 S1, S2, S3, S4, COUT pin 4, 1, 13, 10 9 istype 'com';
equations
S1 = A1 $ B1 $ CIN ;
COUT1 = (A1 & B1) # (A1 & CIN) # (B1 & CIN) ;
S2 = A2 $ B2 $ COUT1 ;
COUT2 = (A2 & B2) # (A2 & COUT1) # (B2 & COUT1) ;
S3 = A3 $ B3 $ COUT2 ;
COUT3 = (A3 & B3) # (A3 & COUT2) # (B3 & COUT2) ;
S4 = A4 $ B4 $ COUT3 ;
COUT = (A4 & B4) # (A4 & COUT3) # (B4 & COUT3) ;
```

```
end circuit_05;
```

A brief explanation of the statements follows.

module: is a module statement followed by the module name (identifier) `circuit_05.` The module statement is mandatory.

title: is optional but in this example here it is used to identify the project. The title name must be between single quotes. It is ignored by the compiler but is recommended for documentation purposes.

string: is a series of ASCII characters enclosed by single quotes. Strings are used for **title**, **options** statements, and in pin, node and attribute declarations.

device: this declaration is optional and associates a device identifier with a specific programmable logic device. The device statement must end with a semicolon. The format is as follows:

device_id **device** 'real_device';

For this example it is: `Four_bit_adder` **device** `'74HC283';`

comments: comments can be inserted anywhere in the file and begin with a double quote and end with another double quote or the end of the line, whatever comes first. For this example, `" input pins and " output pins` are comments.

equations: We use this keyword to write the Boolean expressions using the following operators:

TABLE C.1 Operators used with Boolean expressions in ABEL

Operator	Description
&	AND
#	OR
!	NOT
$	XOR
!$	XNOR

end: End of the source file whose name was specified in module. For this example it is

```
end circuit_05;
```

C.3 Declarations

In this section we provide a general description of the ABEL declarations listed below.

module *module name*

[**title** *string*]

[deviceID **device** deviceType;]

comments: Comments can be inserted anywhere in the file and begin with a double quote and end with another double quote or the end of the line, whatever comes first.

pin declarations

other declarations

equations

equations

[**Test_Vectors**]

test vectors

end *module name*

module: A source files starts with a module statement followed by a module name (identifier). Large source files can be broken down to a group of smaller modules with their own individual title, equations, end statements, etc.

title: The title is optional and it is ignored by the complier but it is recommended for easy identification. If used, its name must be written between single quotes.

string: Strings, a series of ASCII characters, are used for **title**, **options** statements, and in **pin**, **node** (defined below), and attribute declarations. Strings must be enclosed in single quotes.

device: Device is another optional declaration and if used, it associates a device identifier with a specific programmable logic device. The device statement must end with a semicolon. The format is as follows:

device_id **device** 'real_device';

Example: Quad_2_input_Multiplexer **device** '74HC157';

pin: Pin declarations tell the compiler which symbolic names are associated with the external pins of the device. For active Low, we use the (!) symbol meaning that the signal will be inverted. The `istype` is an optional attribute assignment for a pin such as `com` to indicate that the output is a combinational signal or `reg` for a clocked signal (registered with a flip flop). The `istype` attribute is only used for output pins. Thus, for active High the format is:

```
pin_id pin [pin#] [istype 'attributes'] ;
```

and for active Low the format is:

```
[!]pin_id pin [pin#] [istype 'attributes'] ;
```

We can specify more than one pin per line with the following format:

```
pin_id , pin_id, pin_id pin [pin#, [pin#, [pin#]]] [istype
'attributes'];
```

Examples:

```
IN1, IN2, A1, B1 pin 2, 3, 4, 5;
OUT1 pin 7 istype 'reg';
ENABLE pin 10;
!Chip_select pin 12 istype 'com';
!S0..!S6 pin istype 'com';
```

It is not mandatory to specify the pin numbers, but it is a good practice for identification purposes.

node: The node declarations have the same format as the pin declaration. Nodes are internal signals, that is, they are not connected to external pins.

Example:

```
tmp1 node [istype 'com'];
```

Other declarations allow us to define constants, sets, macros and expressions that can simplify the program. As an example a constant declaration has the following format:

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

Examples:

```
A = 15;
B=3*8;
ADDRESS = [0,1,14];
PRODUCT = B & C;
```

The following lines are examples of a vector notation and they are equivalent.

```
A = [A5, A4, A3, A2, A1, A0];
A = [A5..A0];
```

The last two equations are equivalent. The use of ".." in the last line above is convenient way to specify a range. Thus, whenever we use A in an equation, it will refer to the vector [A5, A4, A3, A2, A1, A0].

C.4 Numbers

ABEL recognizes numbers in binary, octal, decimal and hexadecimal formats. The default base is decimal, that is, when no symbol is specified, it is assumed to be in the decimal base. Table C.2 shows the symbols (upper or lower case allowed) we can use to specify the base. We can replace the default base with another base using the directive **@Radix** as discussed in Section C.5.

TABLE C.2 Number bases used in ABEL

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

Example C.2

The first column in Table C.3 shows the numbers specified in ABEL. Provide the equivalent decimal numbers in the second column.

TABLE C.3 Table for Example C.2

Specified in ABEL	Decimal Equivalent
^b10111	
^o110111	
45	
^hF3A5	

Solution:

The decimal equivalents are shown in Table C.4.

TABLE C.4 Table for Example C.2 – Solution

Specified in ABEL	Decimal Equivalent
^b10111	23
^o110111	67
45	45
^hDB0A	3,504,625

C.5 Directives

Directives allow manipulation of the source file and processing. Directives can be placed anywhere in the source file.

C.5.1 The @alternate Directive

Syntax : **`@alternate`**

The **`@alternate`** directive allows us to specify an alternate set of operators. Please refer to Table C.4 in Section C.7.1, for a list of alternate operators. As shown in Table C.4, we cannot

use (+) for addition, (*) for multiplication, and (/) for division because they represent the OR, AND and NOT logical operators respectively in the alternate set. The standard operator still work when **@alternate** is in effect. The alternate operators remain in effect until the @STANDARD directive is used or the end of the module is reached.

C.5.2 The @radix Directive

Syntax : **@radix** expr ;

where expr is a valid expression that produces the number 2, 8, 10 or 16 to indicate a new default base number. The **@radix** directive changes the default base. The default is base 10 (decimal). 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.

Example C.3

Use the **@radix** directive to convert the decimal number 3,504,625 to its equivalent hexadecimal. Use this directive again to convert from hexadecimal back to decimal.

Solution:

@radix 16; "change default base to hexadecimal

@radix DB0A; "change back from hexadecimal to decimal

C.5.3 The @standard Directive

Syntax : **@standard**

The **@standard** directive resets the operators to the ABEL–HDL standard. As stated in subsection C.5.1, the alternate set is chosen with the **@alternate** directive.

C.6 Sets

A *set* is a collection of variables or constants that allows us to reference a group by one name. A set provides a convenient way simplify logic expressions. Any operation that is applied to a set is applied to each element of the set. A set is separated by commas or the range operator (..) and must be enclosed in square brackets. For example, suppose a set is defined as

```
[A0, A1, A2, A3, A4]
```

We can increment this range with the set

```
[A0..A7]
```

As another example, suppose that a set is defined as

```
[a0..a8]
```

We can decrement this range with the set

```
[a4..a0]
```

We can also redefine the range

```
[A0..A7]
```

as

```
[A8..A15]
```

A set can also have a range within a larger set. For instance,

```
[A0..A7, B2..B5]
```

The options above apply also to active Low variables, for instance

```
[!B8..!B0]
```

A set of the form

```
[X0, Y]
```

is invalid. However, if

```
Y = [Y0..Y3]
```

we can express it in valid for as

```
[X0, Y0..Y3]
```

C.6.1 Indexing or Accessing a Set

We can access elements within a set using *indexing*. We must use numerical values to indicate the set index. The numerical values refer to the bit position in the set starting with 0 for the least significant bit of the set. Some examples follow where the operator := in lines 3 and 4 below is an assignment operator. We will discuss assignment operators in Subsection C.7.4.

```
A1 = [A9..A0]; "set declaration
B2 = [B3..B0]; "set declaration
B2 := A1[5..0]; "makes X2 equal to [A5, A4, A3, A2, A1, A0]
B2 := A1[8..5]; "makes X2 equal to [A8, A7, A6, A5]
```

In our previous discussion, we have used the operator (=) as an assignment operator. In ABEL, the assignment operator is used in equations whereas the equality operator is denoted as (==).

To access one element as an output in a set, we use the following syntax:

```
OUTPUT = (B[2] == 1);
```

The equality sign (==) gives a logical 1 or a logical 0 depending on the output being True or False.

A.6.2 Set Operations

Set Operations are operations applied to a set. Unless we use parentheses to specify precedence, operators with the same priority are performed from left to right. Some simple examples follow to illustrate set operations.

Example C.4

Let

```
VOUT = [V2, V1, V0]; "Declaration of an output voltage set
```

What is VOUT if:

a. `VOUT = [1,0,1] & [0,1,1];`

b. `VOUT = [1,0,1] # [0,1,1];`

Solution:

a. Each element of the first set is ANDed with the corresponding element of the second set, and thus

```
VOUT = [1 & 0, 0 & 1, 1 & 1]=[0, 0, 1]
```

b. Each element of the first set is ORed with the corresponding element of the second set, and thus

```
VOUT =  [1 # 0, 0 # 1, 1 # 1] = [1, 1, 1]
```

Example C.5

What is `!V` if `V = [V1, V2, V3];` ?

Solution:

```
!V = [!V1, !V2, !V3];
```

The statement

```
[A,B]= C & D;
```

is equivalent to the two statements

```
A= C & D;

B= C & D;
```

Example C.6

What is the statement

```
[X1,Y1] = [A1,A2] & [B2,B3]
```

equivalent to?

Solution:

```
[X1,Y1] = [A1 & B2, A2 & B3];
```

Thus,

```
[X1] = [A1 & B2];
```

and

```
[Y1] = [A2 & B3];
```

Example C.7

What is the statement

```
VIN = A1 & [B1,C1,D1]
```

equivalent to?

Solution:

```
VIN = A1 & B1, A1 & C1, A1  & D1];
```

Consider the statement

```
VOUT = 3 & [V1, V2, V3, V4];
```

This is a Boolean expression and the constant 3 must be expressed in binary form. Also, because the bracketed term contains 4 variables, the constant 3 must be padded with 2 leading 0s, that is, it must be expressed as 0011. Therefore,

```
VOUT = 0011 & [V1, V2, V3, V4];
```

or

```
VOUT = [0 & V1, 0 & V2, 1 & V3, 1 & V4]; = [V3, V4]
```

Example C.8

Consider the statement

```
VOUT = 3 & [!V1, !V2, !V3, V4];
```

This is a Boolean expression and the constant 3 must be expressed in binary form. Thus,

```
VOUT = 0011 & [!V1, !V2, !V3, V4];
```

But this cannot be true because V3 is complemented, that is, !V3 = 0. Therefore, the binary number 0011 is truncated to 0001 and thus the expression

```
VOUT = 3 & [!V1, !V2, !V3, V4];
```

is the same as

```
VOUT = 1 & [!V1, !V2, !V3, V4];
```

Example C.9

What is the statement

```
[X3, X2, X1, X0] = 2
```

equivalent to?

Solution:

As in Example C.7, the constant 2 is converted into binary and padded with zeros, that is, 0010. Then,

```
X3 = 0     X2 = 0     X1 = 1     X0 = 0
```

Sets are also being used with logic expressions. The logic expressions are very helpful when working with a large number of variables, such as a 16–bit address For instance, we can use sets to express the logic expression

```
Input = A3 & !A2 & !A1 & A0
```

as the declaration

```
VIN = [A3, A2, A1, A0];
```

Now, we can use the following equation to specify VIN as follows:

```
Input = VIN == [1, 0, 0, 1]
```

Thus, if A3 = 1, A2 = 0, A1 = 0, and A0 =1, the expression VIN == [1, 0, 0, 1] is logic 1 (True) and Input will also be logic 1. We can also use the equation

```
Input = VIN == 9;
```

where $(9)_{10} = (1001)_2$.

C.7 Operators

There are four basic types of operators: logical, arithmetic, relational and assignment.

C.7.1 Logical Operators

The *logical operators* are shown in Table C.5. Operations are performed bit by bit. We can use the alternate operators shown on the right column of Table C.5 with the **@alternative** directive.

TABLE C.5 Default and Alternate Logic Operators

Default Operator	Description	Alternate Operator
!	NOT (inversion)	/
&	AND	*
#	OR	+
$	XOR	:+:
!$	XNOR	:*:

C.7.2 Arithmetic Operators

The table below gives the *arithmetic operators*. Note that the last four operators are not allowed with sets. The minus (–) sign can have different meanings: used between two operands it indicates subtraction (or adding the twos complement), while used with one operator it indicates the twos complement. The arithmetic operators *, /, %, <<, and >> shown on the last 5 rows of Table C.6 cannot be used with sets.

TABLE C.6 Arithmetic Operators

Operator	Description	Examples
–	Twos complement	–A
–	Subtraction	A1–B1
+	Addition	C1+D1
*	Multiplication	A*B
/	Integer Division (unsigned numbers)	C/D
%	Modulus (Remainder of Division)	A%B (Remainder of A/B)
<<	Shift Left	A<<B (Shift Left A by B bits)
>>	Shift Right	C>>D (Shift Right C by D bits)

C.7.3 Relational Operators

The *relational operators* are shown in Table C.7. These operators produce a Boolean value of True (logic 1) or False (logic 0).

TABLE C.7 Relational Operators

Operator	Description	Assumptions	Examples
= =	Equal	A=5+7, B=3+9	A = = B (True)
!=	Not equal	C=17, D=12	C!=D (True)
<	Less than	a = 8, b = 4	a < b (False)
<=	Less than or equal to	c = 3, d = 5	c <= d (True)
>	Greater than	e = 11, f = 8	f > e (False)
>=	Greater than or equal to	g = 2, h = 3	h >= g (True)

We recall that a binary number 8 bits long can be represented in the range from –128 (11111111) to +127 (01111111), and a binary number 16 bits long can be represented in the range –32768 (1111111111111111) to +32767 (0111111111111111). The logical true value of –1 in twos complement in a 16 bit word is as 1111 1111 1111 1111.* It is important, however, to remember that *relational operators are unsigned*. Thus, `–1>3` and likewise !0 is the one complement of 0 or 11111111 (8 bits data) which is 255 in unsigned binary. Thus !0 > 7 is true.

Relational expressions are very useful in specifying conditional logical expressions. For instance, if we want X to be equal to Y if A is not equal to B and produce a logic 1 (True) when this condition is satisfied, and to produce a logic 0 (False) otherwise, we can use the statement

```
X = Y !$ (A!=B)
```

C.7.4Assignment Operators

Assignment operators are those used in equations to assign the value of an expression to output variables. There are two types of assignment operators: *combinational* and *registered*. For a combinational assignment operator the syntax is

```
[!]pin_id pin [pin#] [istype 'com'];
```

and for a registered assignment operator the syntax is

```
[!]pin_id pin [pin#] [istype 'reg'];
```

In a combinational operator the assignment occurs immediately without any delay. However, the registered assignment occurs at the next clock pulse associated with the output. For instance, we can define the output of a flip–flop with either of the following statements:

```
Q1_out pin istype 'reg';

Q1 := D1;
```

The first statement defines the output Q1 of a flip–flop by using the 'reg' as istype (registered output). The second statement assumes that Q1 is the output of a D–type flip flop and the state of this output will be the same as the state of flip flop D1 at the next clock transition.

C.7.5 Operator Priorities

The precedence (priority) of each operator is as shown in Table C.8 with priority 1 the highest and 4 the lowest. When operators have the same priority, the operations are performed from left to right.

* *This can be verified by taking the ones complement of 1111111111111111 which is 0000000000000000 and add 1 to the lsb.*

TABLE C.8

Priority	Operator	Description
1	–	Negation (twos complement)
1	!	NOT
2	&	AND
2	<<	Shift left
2	>>	Shift right
2	*	Multiplication
2	/	Division (Unsigned)
2	%	Modulus (remainder)
3	+	Addition
3	–	Subtraction
3	#	OR
3	$	XOR
3	!$	XNOR
4	= =	Equal
4	!=	Not equal
4	<	Less than
4	<=	Less or equal
4	>	Greater than
4	>=	Greater or equal

C.8 Logic Description

A logic design can be described in the following way.

- Equations
- Truth Table
- State Description

C.8.1 Equations

We begin the logic description with the word **`equations`**. The equations specify logic expressions using the operators listed in Table C.8, or with the `When-Then-Else` statement. This statement is used for the description of logic functions. The `If-Then-Else` statement is used to describe state progression and it will be discussed in Subsection C.8.3.

The format of the `When-Then-Else` statement is as follows:

```
WHEN condition THEN element=expression;
ELSE equation;
```

or

```
WHEN condition THEN equation;
```

Some simple examples follow.

1.
```
SUM0 = (A0 & !B0) # (!A0 & B0);
```

2.
```
X0 := CLK & A2 & !A1 & !A0;
```

3.
```
WHEN (A1 == B1) THEN Q1 = A2;
  ELSE WHEN (A1 == C1) THEN Q1 = A0;
```

We can use the braces { } to group sections together in blocks. For instance,

```
WHEN (D>C) THEN {Q1 :=D1; Q2 :=D2;}
```

The text in a block can be on one line as above, or it can be expressed in two or more lines. Blocks are used in equations, state diagrams and directives.

C.8.2 Truth Tables

With *truth tables* we use the words **truth_table** and the syntax can be written in either of the three expressions below where –> is used for outputs of combinational circuits, and :> is used for outputs of sequential (registered) circuits.

1.
```
TRUTH_TABLE (in_ids -> out_ids)
inputs -> outputs ;
```

2.
```
TRUTH_TABLE (in_ids :> reg_ids)
inputs :> reg_outs ;
```

3
```
TRUTH_TABLE (in_ids :> reg_ids -> out_ids)
inputs :> reg_outs -> outputs ;
```

The first line of a truth table (between parentheses) defines the inputs and the output signals. The following lines show the values of the inputs and outputs. Each line must end with a semicolon. The inputs and outputs can be single signals or sets. When sets are used as inputs or outputs, we use the normal set notation, i.e. signals surrounded by square brackets and separated by commas. A don't care is represented by an .X.

Example C.10

Write the truth table for a full adder.

Solution:

We wrote the truth table for a full adder in Chapter 7. In ABEL the first line is

```
TRUTH_TABLE ([ A, B, CIN] -> [SUM, COUT])

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

The truth table above can be simplified if we define the set

$$IN = [A,B];$$

and

$$OUT = [SUM, COUT]$$

Then, the truth table is written as

```
TRUTH_TABLE ( IN -> OUT)

     0 -> 0;
     1 -> 2;
     2 -> 2;
     3 -> 1;
     4 -> 2;
     5 -> 1;
     6 -> 1;
     7 -> 3;
```

Example C.11

The output of an XOR gate is connected to an ON/OFF switch. Write the truth table.

Solution:

Let us denote the switch as SW where the OFF position is represented as logic 0, and the ON position as logic 1, and the inputs to the XOR gate as A and B. Using a short notation as in Example C.10, the truth table is written as follows where `.X.` indicates a don't care.

```
TRUTH_TABLE ([ SW, A, B ] -> OUT)

     [ 0, .X., .X. ] -> [ .X. ];
     [ 1, 0, 0 ] -> 0;
     [ 1, 0, 1 ] -> 1;
     [ 1, 1, 0 ] -> 1;
     [ 1, 1, 1 ] -> 0;
```

Truth tables can also be used to define sequential circuits. In Chapter 8, Example 8.5, we designed a BCD up counter. In the following example, we will implement this counter in ABEL.

Example C.12

Implement a BCD up counter which counts from 0000, 0001, ... to 1001 and back to 0000. Make provisions that the counter will generate an output OUT whenever the counter reaches the state 1001. Assume that the flip flops can be simultaneously reset to 0000 by the reset direct (RD) signal.

Solution:

The state table shown as Table C.9 lists the present and next states where Q_4 is the most significant bit (msb) and Q_1 as the least significant bit (lsb). We can generate the output OUT (combinational signal) by adding a 4–input AND gate that produces a logic 1 output when the input is $Q_4\overline{Q}_3\overline{Q}_2Q_1$.

TABLE C.9 State table for Example C.12

Present State				Next State			
Q_4	Q_3	Q_2	Q_1	Q_4	Q_3	Q_2	Q_1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

The implementation in ABEL is shown below where have used the dot extensions `.CLK` and `.AR`. We will discuss dot extensions in Subsection C.8.4.

```
module BCD_COUNTER;

CLOCK pin; " input signal

RESET pin; " input signal
```

```
OUT pin istype 'com'; " output signal (combinational)
Q4, Q3, Q2, Q1 pin istype 'reg'; " output signal (registered)

[Q4, Q3, Q2, Q1].CLK = CLOCK; "FF clocked on the CLOCK input
[Q4, Q3, Q2, Q1].AR = RESET; "asynchronous reset by DIRECT RESET

TRUTH_TABLE ([Q4, Q3, Q2, Q1] :> [Q4, Q3, Q2, Q1] -> OUT)

                [ 0 0 0 0 ] :> [ 0 0 0 1 ] -> 0;
                [ 0 0 0 1 ] :> [ 0 0 1 0 ] -> 0;
                [ 0 0 1 0 ] :> [ 0 0 1 1 ] -> 0;
                [ 0 0 1 1 ] :> [ 0 1 0 0 ] -> 0;
                [ 0 1 0 0 ] :> [ 0 1 0 1 ] -> 0;
                [ 0 1 0 1 ] :> [ 0 1 1 0 ] -> 0;
                [ 0 1 1 0 ] :> [ 0 1 1 1 ] -> 0;
                [ 0 1 1 1 ] :> [ 1 0 0 0 ] -> 0;
                [ 1 0 0 0 ] :> [ 1 0 0 1 ] -> 0;
                [ 1 0 0 1 ] :> [ 0 0 0 0 ] -> 1;
            [ 1 0 1 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 0 1 1 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 0 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 0 1 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 1 0 ] :> [.X. .X. .X. .X.] -> .X.;
            [ 1 1 1 1 ] :> [.X. .X. .X. .X.] -> .X.;

end BCD_COUNTER;
```

C.8.3 State Diagram

This subsection describes the state_diagram that contains the state description for the logic design. We can declare symbolic state names in the declaration section, and this makes the reading easier. We will discuss the *state_diagram* syntax and the *If–Then–Else*, *Goto*, *Case*, and *With* statements.

In the declaration section, the *state declaration* syntax is

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

Suppose we have an 8–bit register and we assign the state name REG8 to it. We can associate this state name with the outputs Q_0, Q_1, …Q_7 as

$$REG8 = [Q_0 .. Q_7]$$

The syntax for State_diagram is as follows:

```
State_diagram state_reg
STATE state_value : [equation;]
[equation;]
:
:
trans_stmt ; ...
```

The word **state_diagram** in the first line above is used to indicate the beginning of a state description.

The STATE word and following statements describe one state of the state diagram and includes a state value or symbolic state name, state transition statement, and an optional output equation. Thus, in the second line, after STATE we may write either

1. state_value: this can be an expression, a value or a symbolic state name of the current state.

2. state_reg: is an identifier that defines the signals that determine the state of the machine. This can be a symbolic state register that has been declared earlier in the declaration section.

The [equation;] in the second line is optional.

The [equation;] in the third line is mandatory; it is an equation that defines the outputs.

In the last line, trans_stmt, can be used with the If-Then-Else, Goto, or Case statements to defines the next state, followed with optional With transition equations.

The **If-Then-Else** statement is used to determine the next state depending on mutually exclusive transition conditions. Its syntax is

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

The state_exp can be a logic expression or a symbolic state name. We must remember that the If-Then-Else statement can only be used in the state_diagram section.* The ELSE† clause is optional. The If-Then-Else statements can be nested with Goto, Case, and With statements. As an example, in the declaration section we first define the state registers:

```
REG8=[Q0..Q7]; " Define 8-bit register

SUM = (A & !B) # (!A & B) ;
SUM0 = [0, 0];
```

* *We recall from Subsection C.8.1 that the When–Then–Else statement is used with equations.*

† *As stated earlier, words recognized by ABEL as commands, e.g.* ***goto, if, then, module****, are not case–sensitive.*

```
SUM1 = [1, 1];
state_diagram REG8
state SUM0: OUT1 = 1;
    if A then S1
       else S0;
state SUM1: OUT2 =1;
    if A then S0
       else S1;
```

The **with** statement is used with the syntax:

```
trans_stmt state_exp WITH equation
[equation ] ... ;
```

where the trans_stmt can be If-then-else, Goto, or a Case statement, state_exp is the next state, and equation is an equation for the machine outputs. This statement can be used with the If-then-else, Goto, or Case statements in place of a simple state expression. The with statement allows the output equations to be written in terms of transitions. For example, in the statement

```
if A0 & B0 ==1 then SUM0 with C0=1 else SUM1
```

the output C0 will be true if A0 & B0 ==1 is True (logic 1). Any expression after with can be an equation and it will be evaluated when the if condition is true. The following statement is also valid.

```
if A0 # B0 ==1 then SUM0 with C0=A0 & B0 else SUM1 with A0 != B0
```

The **Case** statement is used with the syntax

```
case expression : state_exp;
[ expression : state_exp; ]
:
endcase ;
```

where expression is any valid expression and state_exp is an expression indicating the next state. For example,

```
State SUM0:
     case ( A == 0) : SUM1;
          ( A == 1) : SUM0;
   endcase;
```

The `case` statement is used to list a sequence of mutually-exclusive conditions and corresponding next states. The `case` statement conditions must be mutually exclusive, that is, two conditions cannot be true at the same time.

C.8.4 Dot Extensions

We can use *dot extensions* to describe the behavior of a digital circuit more precisely. The extensions are very convenient and provide a means to refer specifically to internal states and nodes associated with a primary condition. Dot extensions are not case sensitive. Dot extensions are used with the syntax

```
signal_name.ext
```

Several dot extensions are listed in Table C.10.

TABLE C.10 Dot extensions

Extension	Description
.ACLR	Asynchronous register reset
.ASET	Asynchronous register preset
.CLK	Clock input to an edge-triggered flip flop
.CLR	Synchronous register reset
.COM	Cominbational feedback from flip flop data input
.FG	Register feedback
.OE	Output enable
.PIN	Pin feedback
.SET	Synchronous register preset
.D	Data input to a D Flip flop
.J	J input to a JK flip-flop
.K	K input to a JK flip-flop
.S	S input to a SR flip-flop
.R	R input to a SR flip-flop
.T	T input to a T flip-flop
.Q	Register feedback
.PR	Register preset
.RE	Register reset
.AP	Asynchronous register preset
.AR	Asynchronous register reset
.SP	Synchronous register preset
.SR	Synchronous register reset

As an example, the statements

```
Q.AR = reset;

[Q1.ar, Q2.ar] = reset;
```

reset the flip flop outputs when `reset` is logic 1. As another example, the statement

```
[LD7..LD0].OE = ACTIVE;
```

will enable the outputs of line drivers `LD7` through `LD0` when `ACTIVE` is logic 1, otherwise the outputs will be in a high-Z state.

C.9 Test Vectors

Test vectors are optional but are useful for providing a means to verify the correct operation of a state. The vectors specify the expected logical operation of a logic device by explicitly giving the outputs as a function of the inputs. Test vectors are used with the syntax

```
Test_vectors [note]
(input [, input ].. -> output [, output ] .. )
[invalues -> outvalues ; ]

:

:
```

As an example, we can use test vectors to verify the outputs of a full adder as follows:

```
  Test_vectors

   ([ A, B, CIN] -> [SUM, COUT])

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

As with truth tables, we can also specify the values for the set with numeric constants as shown below.

```
  Test_vectors

([ A, B, CIN] -> [SUM, COUT])

       0 -> 0;
       1 -> 2;
```

```
0 -> 0;
1 -> 2;
2 -> 2;
3 -> 1;
4 -> 2;
5 -> 1;
6 -> 1;
7 -> 3;
```

Don't cares (.X.), clock inputs (.C.), and symbolic constants are also allowed as shown in the following example.

```
test_vectors

( [CLK, RESET, QA, QB ] -> [ X0, X1, X2] )
  [.X., 1, .X.,.X.] -> [ A0, 0, 0];
  [.C., 0, 0, 1 ]   -> [ B0, 0, 0];
  [.C., 1, 1, 0 ]    ->[ C0, 0, 1];
```

C.10 Property Statements

Property statements allow us to assign device specific statements. For field programmable devices the property statements include

- Slew rates
- Power settings
- Preload values

C.11 Active–Low Declarations

As we've learned, active low signals are defined with a "!" operator. Consider, for example, the declaration

```
!OUT pin istype 'com' ;
```

When this signal is used in a subsequent design description, it will be automatically complemented. As an example consider the following description,

```
module EXAMPLE
A, B pin ;
!OUT pin istype 'com';
equations
OUT = A & !B # !A & B ;
end
```

In this example, the signal OUT is an XOR of A and B, i.e. OUT will be "1" (High, or ON) when only one of the inputs is "1", otherwise OUT is "0". However, the output pin is defined as !OUT ,

i.e. as an active–low signal, which means that the pin will go low "0" (Active–low or ON) when only one of the two inputs are "1". We could have obtained the same result by inverting the signal in the equations and declaring the pin to be OUT, as is shown in the following example. This is called explicit pin–to–pin active–low (because one uses active–low signals in the equations).

```
module EXAMPLE
A, B pin ;
OUT pin istype 'com';
equations
!OUT = A & !B # !A & B ;
end
```

Active low can be specified for a set as well. As an example lets define the sets A, B, and C.

```
A = [A2,A1,A0]; "set declaration
B = [B2,B1.B0]; "set declaration
C = [X2,X1.X0];  "set declaration
!C = A & !B # !A & B;
```

The last equation is equivalent to

```
!C0 = A0 & !B0 # !A0 & B0;
!C1 = A1 & !B1 # !A1 & B1;
!C2 = A2 & !B2  # !A2 & B2;
```