The Abel Hardware Description Language (HDL)

Introduction

This chapter is based on the easyABEL HDL version 4.3 and the Xilinx™-Abel Software Design which describe the XABEL environment incorporated in the Foundation Series version 2.2.

The ABEL HDL was made for PLD circuit design by DATA I/O and with other hardware descriptor languages come to easy the PLD design. Modern computer languages are almost invariably composed of declarative and executable statements, and HDL languages are particularly rich in the former. Comparing the result of a High level programming language (as C++) the result of a HDL program will be a hardware and not an executable program.

ABEL hardware description language allows to one to describe digital designs with equations, truth tables, state diagrams, or any combinations of the three, optimize and simulate the design without specifying a device or assigning pins. The output files produced by the ABEL environment are in standard formats to interface with other tools (in our case with the Xilinx™ design environment) JEDEC format output files download directly the PLD programmers and PROM format files to allow PROM programming. ABEL hardware description language like behavioral description languages describe the hardware structure on its functional run (the output signals are function of the variation of the input signals). The ABEL HDL permit to write hardware independent programs, and after program verification and optimization one can choose the PLD device. In this operation the so called SmartPart help the user to choose the best PLD in which the program fit in with optimal parameters.

Basic Structure of an ABEL HDL File

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

- A line may be up to 150 characters long.
- Lines are ended by a line feed character (0AH), by a vertical tab (0BH), or by a form feed (0CH). On most computers, an input line is ended simply by pressing `ENTER'.
- Keywords, identifiers, and numbers must be separated by at least one space. Exceptions to this rule are lists of identifiers separated by commas, in expressions where identifiers or numbers are separated by operators, or in places 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.
- 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.
- Although identifiers in ABEL-HDL are case sensitive, the Xilinx™ XNF netlist is not case sensitive. So, identifiers consisting of the same letters which differ only in case should not be used. Otherwise an error will appear during synthesis.

Identifiers

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>

! @ # $ ? + & * ( )- _ = +

[ { } ] ; : ` " ` ~ \ | , < > . / ^ %
```

The rules and restrictions for identifiers are the same regardless of what the identifier describes. The rules governing identifiers are:

- Identifiers can be up to 31 characters long.
- Identifiers must begin with an alphabetic character or with an underscore.
- The caracter 'tilde' (~) is also supported in signal names.
- Other than the first character, identifiers can contain upper- and lowercasecharacters, digits and underscores.
- Spaces cannot be used in an identifier. Use underscores or uppercase letters to separate between words.
- Except for Reserved Keywords, identifiers are case sensitive: uppercase letters and lowercase letters are not the same.
- Periods cannot be used in an identifier, except when using a valid dot extension.

Reserved 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.

```
DECLARATIONS            IF                  STATE_DIAGRAM
DEVICE                  IN (obs)            TEST_VECTORS
ELSE                    ISTYPE              THEN
ENABLE (obs)            LIBRARY             TITLE
END                     MACRO               TRACE
ENDCASE                 MODULE              TRUTH_TABLE
ENDWITH                 NODE                WHEN
FUSES                   OPTIONS             WITH
EQUATIONS               PIN
FLAG (obs)              PROPERTY
```

Constants

Numerical systems: ABEL support the binary- (2), octal- (8), decimal- (10), hexadecimal systems. the default numerical system (10) can be changed by the declaration @RADIX. the following examples show the symbolization of each numerical system:

```
56              decimal 56

^h56            hexadecimal 56 (its decimal value is 86)

^b1001          binary 1001 (its decimal value is 9)

^o56            octal 56 (its decimal value is 46)

^h0E            hexadecimal 0E (its decimal value is 14)
```

Characters could have also a value, for example `d'=100

Logical values

True (logical 1) and false (logical 0) are represented as numbers (32 bit integer), true = -1 (all the 32 bits are 1) and false =0 (all the 32 bits are 0).

Special Constants

```
.C.     Clocked input (low-high-low transition)

.K.     Clocked input (high-low-high transition)

.U.     Clock up edge (low-high transition)

.D.     Clock down edge (high-low transition)

.F.     Floating input or output signal

.P.     Register preload

.SVn.   n = 2 .. 9. Drive the input to super voltage 2 through 9V.

.X.     Don't care condition

.Z.     Tristate value
```

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.

```
`Hello this is a string!'

`Punctuation? is even allowed !!'
```

A single quote can be included in a string by preceding it with a backslash, "\."

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

Backslashes can be put in a string by using two of them in succession.

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

Operators

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 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.

Logical Operators: Logical operators are used in expressions and operations are performed bit by bit.

!	!A	NOT: ones complement
&	A & B	AND
#	A # B	OR
$	A $ B	XOR: exclusive OR

!$	A !& B	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.

```
-      -A      Twos complement (negation)
-    A - B     Subtraction
+    A + B     Addition
```

The following operators are not valid for sets:

```
*      A*B     Multiplication
/      A/B     Unsigned integer division
%      A%B     Modulus remainder from division
<<     A<<B    Shift A left by B bits
>>     A>>B    Shift A right by B bits
```

Relational Operators: Relational operators compare two items in an expression. Relational operations are unsigned operations. Expressions formed with relational operators produce a Boolean true or false value. Relational operations should be put in brackets, in this way the higher priority of logical operators will not disturb their evaluation.

```
==  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
```

Assignment Operators:

Assignment operators are special class of operators used in equations rather than in expressions. Equations assign the value of an expression to output signals. There are four assignment operators, two combinatorial and two registered. Combinatorial 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. These assignment operators allow you to fully specify outputs in equations

```
 =   ON(1)  Combinatorial assignment or detailed equation
?=   DC(X)  Combinatorial assignment or detailed equation
:=   ON(1)  Implied registered assignment
?:=  DC(X)  Registered  assignment
```

Operator Priority:

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. Operations of the same priority are performed from left to right. Use parentheses to change the order operations are performed. The operation in the innermost set of parentheses performed first.

- - (negate), ! NOT
- & (AND), <<, >> (shift left, shift right), * (multiply), / (unsigned division),% (modulus)
- + (add) - (subtract), # (OR), $ (XOR), !$ (XNOR)
- ==, != (equal, not equal), < (less than), <= (less than or equal) > (greater than), >= (greater than or equal)

Sets

A set is a collection of signals and constants. You can refer to the signals and constants with a common identifier. In other language these sets are referred as bus. The elements of a set are written between [and] brackets, separated by coma (,). also they can be written separated by the range `..' operator. For example:

address =[A8, A7, A6, A5, A4, A3, A2, A1, A0]

or

address =[A8..A0]

Operations like multiplication, division, modulus, shift are not allowed between sets.

Blocks

Blocks are sections of text enclosed in braces,"{" and "}." Blocks are used in equations, state diagrams, macros and directives. The text contained in a block can be all on one line or can span many lines. Blocks can be nested within other blocks.

Some examples of blocks follow:

```
{ this is a block }

{this is also a block, and it

spans more than one line.}

{ A = B # C;

D = [0, 1] + [l, 0];

}
```

Comments

A comment begin with " and it is terminated with another ".

```
 "You can use comments to write observation in your program."
```

Arguments and Dummy arguments

Dummy arguments have values only in macros. In module or in directives they point to the parameter on which it is applied the specified operation. In the body of macro declaration before the dummy parameters it is written the question mark (?), and they are separated by `space' characters:

```
EXP MACRO (a, b, c, d)

      {?a & ?b # ?c & ?d};

      `This was the macro declaration'

      `The call of the defined macro:'

      F = EXP (x, y, z, w);

      `The evaluation of the macro will be:'

      F= x & y # z & w;
```

Directives

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.

Some of the available directives are presented below.

@ALTERNATE (Alternate operator Set)

The @ALTERNATE directive enables alternate set of operators:

Standard	Alternate	Description
!	/	NOT
&	*	AND
#	+	OR
$	:+:	XOR
!$	:*:	XNOR

Standard operators still work when @ALTERNATE is in effect. Directive @STANDARD resets standard set of operators. Naturally you should switch back to standard operator set when using arithmetical operations.

@STANDARD (Standard operator Set):

The @STANDARD directive resets the operators to the ABEL standard.

@CONST (Constant Declaration)

The @CONST directive allows new constant declaration to be made in a source file outside normal (and required) declaration sections. It is used to define internal constants inside macros.

Syntax:

```
@CONST identifier = expression;
```

Example:

```
@CONST OK = 1;

@CONST count = count + 1;
```

@DCSET (Don't Care Set)

The @DCSET directive allows the optimization to use don't cares to optimize partially-specified logic functions. This directive overrides attributes `dc', `neg' and `pos'.

The @ONSET directive neutralize it's effect.

@IF

The @IF directive includes or excludes sections of source code based on the value of an expression. If the expression is true, the block of code is included.

Syntax:

```
@IF expression {block}
```

Example:

```
@IF (A>B) {C = D # E}
```

@INCLUDE

The @INCLUDE directive causes the contents of the specified file to be placed in the ABEL source file. The inclusion begins at the location of the directive.

Syntax: @INCLUDE `file_name.abl` (DOS path require two slashes in the string)

Example:

```
@INCLUDE `macros.abl`

@INCLUDE `C:\\ABEL\\INCLUDE\\CONVERT.ABL`
```

@PAGE

The @PAGE directive sends a form feed to the listing file.

@RADIX (Default Base Numbering Directive)

The @RADIX directive changes the default base. The default base before the first @RADIX directive is 10. The radix value can be an expression.

Syntax:

```
@ RADIX expression
```

Example:

```
A = 10;          `A is 10 (decimally)'

@RADIX 2;        `changes default base to 2'

A = 10;          `now A is 2 (decimally)

@RADIX 10000;    `changes default base to 16'

A = 10;          `now A is 16 (decimally)'

@RADIX 0A;       `changes default base to 10'

A = 10;          `now A is 10 (decimally) again
```

@REPEAT

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

Syntax:

```
@REPEAT expression { block };
```

@IRP (Indefinite Repeat Directive)

The @IRP (Indefinite Repeat) directive 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.

Syntax:

```
@IRP dummy_argument ( argument [,argument] . )  { block }
```

The ABEL have other directives, which permit other `tricks', but is not the case to present all the directives.

```
@EXPR    (Expression Directive)

@IFB     (If Blank Directive)

@IFDEF   (If Defined Directive)

@IFIDEN  (If Identical Directive)

@IFNB    (If Not Blank Directive)

@IFNDEF  (If Not Defined Directive)

@IFNIDEN (If Not Identical Directive)
```

Description of ABEL HDL Elements

The ABEL HDL Source File Structure

The complete functional description of the design it is included in the so called module. An ABEL program can have more than one module, but the environment will translate only the first module, the other modules will be checked syntactically. In one module can be specified only one PLD device.

A module consist of the following sections:

Header

Declarations

Logic Description

Test Vectors

End

The following three 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 either immediately follow the header or the Declarations keyword.
- No symbol (identifier) can be referenced before it is declared.

Header (MODULE)

Any ABEL HDL program must begin with MODULE header which indicate the beginning of the program

The Header Section can consist of the following elements:

- **MODULE** ***name*** (The identifier is the beginning of module statement (required))
- **OPTIONS** (Optional element that can influence the run of the program)
- **TITLE** ***`string'*** (Optional element that it is written in the header of JEDEC file)

The order of the identifiers must be the order presented

Declarations

A Declarations Section can consist of the following elements:

- **DECLARATIONS** Keyword
- **DEVICE** PLD device declaration (one per module)
- **PIN** External signal declarations (input/output pin/signaldeclarations)
- **NODE** Internal signal/node declarations
- **ISTYPE** Optional attribute of PIN/NODE declaration
- **Constant Declarations**
- **MACRO** Macro declarations
- **LIBRARY** References

All the declarations of an object have to be pre-definite for the first occurrence of the object. Corresponding to this if there exist a DEVICE declaration then should be placed before all of the other declarations.

Logic Description (EQUATIONS)

The logic description section define the functional and structural description of the design. All the variables and elements of this section were declared by the declaration section.

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

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

Test Vectors

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs. Test vectors are used for simulation of an internal model of the device and functional testing of the programmed device.

If the signal identifiers used in the test vector header were declared as active-low in the declaration section, then constant values specified in the test vectors will be inverted accordingly.

A Test Vectors Section can consist of the following elements:

- TEST_VECTORS Describe how have `to work' the design
- TRACE The TRACE statement is used to control the display features. TRACE statements can be placed before a test vector section, or imbedded within a sequence of test vectors.

End Statement

A module is closed with the end statement.

Header (MODULE)

The MODULE statment defines the beginning of an HDL program and must be paired with END statment that defines the functional description end.

The Header Section can consist of the following elements:

MODULE *name* (The identifier is the beginning of module statement (required))

OPTIONS (Optional element that can influence the run of the program)

TITLE *`string'* (Optional element that it is written in the header of JEDEC file)

The order of the identifiers must be the order presented

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 the PLD device. Each module must have at least one declarations section, and declarations affect only the module in which they are defined.

The device it is declared with the following syntax:

```
device_identifier DEVICE real_device
```

The device declaration is optional. It associates the device name used in a module with an actual programmable logic device on which designs are implemented. Device identifiers used in device declarations should be valid filenames since JEDEC files are created by appending the extension .jed to the identifier. The ending semicolon is required.

- **XABEL:** Do not specify the name of a Xilinx™ device with the device statement. Device is specified in the Project Manager.
- **XABEL EPLD Design:** The following ABEL-HDL device statement should be specified in the header of the source ABEL file used as a top-level design or as a single file design. This statement tells XABEL that this file represents complete stand-alone design. It has the following syntax:

  ```
  modulename DEVICE;
  ```

  In an included file(s) the device statement should not appear.

Examples:

```
D1 device `P22V10' ;

module UART;    "Xilinx™ XEPLD design"
```

```
UART device;
```

Signal Declarations

There are several types of declaration statements:

```
ATTRIBUTE
CONSTANT
LIBRARY
MACRO
NODE
PIN
```

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;

Note that 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 in the process by a fitter.

Pin and Node Declarations

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

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

where pin# and node# are the pin number on the real device, and
attributes a string that specifies pin attributes for devices with programmable
pins

Attributes may be centered in uppercase, lowercase or mixed-case letters.

Signal Attributes:

If the signals are pine numbers of real device then this declaration
will define the signal type and it is not necessary to use attributes for their
definition.

signal_name, [signal_name] ISTYPE `attr'

General or Architecture Independent Attributes:

`com'           The signal is combinatorial. Implies pin-to-pin equations. the signal is
not registered output

`reg'           A clocked memory element (generic flip-flop). Implies pin-to-pin syntax.
If `invert' or `buffer' is specified, the compiler converts .D and .Q to .reg
and .fb.

`neg'           The input/output signal is inverted, the reduced-fixed option will
optimize to this attribute. Unspecified logic is 1.

`pos'           The input/output signal isn't inverted, the reduced-fixed option will
optimize to this attribute. Unspecified logic is 0.

Architecture Dependent Attributes

`buffer'        The target architecture does not have an inverter between the
associated flip-flop (if any) and the actual output pin.

`dc'            Unspecified logic is don't care.
```

`invert' The target architecture has an inverter between the associated flip-flop (if any) and the actual output pin.

`reg_D'  A clocked memory element (D-type flip-flop). Implies detailed syntax. If `invert' or `buffer' is specified, the compiler converts := equations (and .fb) to .D and .Q.

`reg_T' A clocked memory element (T-type flip-flop). Implies detailed syntax.

`reg_SR'A clocked memory element (SR-type flip-flop). Implies detailed syntax.

`reg_JK'A clocked memory element (JK-type flip-flop). Implies detailed syntax.

`reg_G' A memory element (D-type flip-flop with a gated clock). Implies detailed syntax.

`retain' Do not minimize this output. Preserve redundant product terms for the signal. Must be used with the reduce none option in PLAOpt.

`xor' The target architecture has an XOR gate, so one top-level exclusive-OR operator is retained in the design equations.

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 makes it more likely that the design operates consistently if the device is changed later. ISTYPE can be used after pin, node or state register declarations.

Constant Declarations

Syntax:

identifier [, identifier]. .= expression [, expression]. . ,

A constant is an identifier that retains a constant value throughout a module. The identifiers listed on the left side of the equals sign are assigned the values on the right side. There is a one-to-one correspondence between the identifiers and the expressions listed and there must be one expression for each identifier. The ending semicolon is required.

Symbolic State Declarations

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

Syntax:

state_identifier [, state_identifier.] STATE state_value [,state_value, ..];

Macro Declarations

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.

Syntax:

macro_identifier MACRO [(dummy_argument[, dummy_argument])] {block};

Library Declarations

The LIBRARY statement extracts the contents of the indicated file from the abel5lib.inc library and inserts it into the ABEL-HDL source file at the location of the LIBRARY statment

Syntax:

LIBRARY `library_name'

Logic Description

The following elements can be used to describe your design:

Equations Keyword which is compulsory.

Boolean Logic Equations

Truth Tables

State Descriptions

Fuses

XOR Factors

Dot extensions:

Signal dot extensions describe more precisely the behavior of a circuit in a logic description that may be targeted to a variety of different architectures.

Syntax:

signal_name.dot_extension

Dot extensions can be architecture independent and can be specific for certain devices. Device specific dot extensions are used with detailed syntax and architecture independent dot extensions are used with pin-to-pin syntax.

Architecture independent extensions:

.CLK Clock input to an edge triggered flip-flop

.OE Output enable

.PIN Pin feedback

.FB Register feedback

Architecture specific dot extensions:

.D On the right side of an equation .D is combinatorial feedback from the D input to a flip-flop. On the left side of an equation is data input to the D-type flip-flop

.J J input to an JK-type flip-flop

.K K input to an JK-type flip-flop

.R R input to an SR-type flip-flop

.S S input to an SR-type flip-flop

.T T input to an T (toggle)-type flip-flop

.Q Register output feedback

.PR Register preset (synchronous or asynchronous)

.RE Register reset (synchronous or asynchronous)

.ACLR Asynchronous register reset

.ASET Asynchronous register preset

.CLR Synchronous register reset

.SET Synchronous register preset

.AR Asynchronous register reset

.AP Asynchronous register preset

.SR Synchronous register reset

.SP Synchronous register preset

.LE Active low latch enable input

.LH Active high latch enable input

.CE Clock enable input

.OE Output enable

.FC Flip-flop mode control

.COM A combinatorial 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

Note that he .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

Syntax:

EQUATIONS

The EQUATIONS statment defines the beginning of the section which describe the logic function of the device.

Description of Combinatorial Logic Design

The combinatorial logic can be described with Boolean logic equations, and truth tables. The description of the functional design it is introduced as was stated before with the keyword EQUATIONS and then follows the logic equations or truth tables:

[WHEN condition THEN] equation;

[ELSE equation];

Example:

EQUATIONS

@ALTERNATE

A = B + C + /D;

ADDR = AB + 5;

```
WHEN X THEN A =B; ELSE A = C;
```

The syntax of truth tables:

```
TRUTH_TABEL ([input_signals] -> [output_signals])

[input_values] -> [output_values];

[input_values] -> [output_values];

...
```

Example:

```
TRUTH_TABELS ([bcd_code] -> [ga, gb, gc, gd])

[0]       [1, 1, 1, 1];

[1]       [1, 1, 1, 0];

[2]       [1, 1, 0, 0];

[3]       [1, 0, 0, 0];

[4]       [1, 0, 0, 0];

[..];
```

In the example the input bcd_code and the output [ga, gb, gc, gd] are sets

Description of Sequential Logic Design

The functional description of sequential logic design can be described with Boolean logic equations, state diagrams, and transitions tables.

Boolean Logic Equations

The syntax of Boolean logic equations at the sequential logic design is the same as the combinatorial logic design, but instead of the `=` operator, we use `:=` operator. In this case the left side of the equation takes the value of the evaluated right side of the equation after the clock the clock impulse.

```
EQUATIONS

        @ALTERNATE

        Q := (A + B) * /RST;

        COUNT := COUNT + 1;
```

Truth tables

The syntax of truth tables is the same as combinatorial logic truth tables, but here we will use the operator `:>`.

Syntax:

```
TRUTH_TABLE ([IN_SIGNALS] :> [REG_SIGNALS] -> [OUTPUTS]

[IN_VALUES] :> [REG_VALUES] -> [OUT_VALUES;]

[IN_VALUES] :> [REG_VALUES] -> [OUT_VALUES;]

[IN_VALUES] :> [REG_VALUES] -> [OUT_VALUES;]

[...];
```

State Diagrams

The State Diagram section contains state descriptions that describe the logic design implemented with programmable logic. 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 which determine the operation of the state machine. An alternative to describing logic with Boolean equations or truth tables is to use a state description.

Symbolic state machines (machines for which the actual state registers and state values are unspecified) require additional declarations for the symbolic state register (state_register) and state names (state) in declarations section.

Syntax:

STATE_DIAGRAM state_reg [-> state_out]

[state state_expression: [equations];

transition_statments; .]

[state state_expression: [equations];

transition_statments; .]

[state state_expression: [equations];

transition_statments; .]

where: state_reg is an identifier or set of identifiers specifying the signals that determine the current state of the machine

state_out is an identifier or set of identifiers that determine the next state of the machine (for designs with external registers)

state_expression is an expression giving the current state equation and is a valid equation that defines the state machine outputs

transition_statment is a condition (IF-THEN-ELSE, CASE or GOTO statement, optionally followed by WITH-ENDWITH transition equations) which force the transition to another statement

Transition statements

Transition statements are: IF-THEN-ELSE, CASE or GOTO, with the well known means from the high level languages. This statements can be followed optionally by the WITH-ENDWITH transition statements.

Syntax:

TRANSITION_STATMENTS next_statment WITH EQUATION;

[EQUATIONS];

ENDWITH

Examples:

EQUATIONS

@ALTERNATE

STATE_DIAGRAM [Q1, Q2]

```
STATE   S0:     O = /Q1 * /Q2;
                IF A THEN S1
                ELSE S0;
```

```
STATE   S1:     O = Q1 * /Q2;

                CASE (A == 0): S0;

        (A == 1): S1;

                ENDCASE

STATE   S2:     O = /Q1 * Q2;

                GOTO S3;

STATE   S3

                IF A==0 THEN S0 WITH O = Q1 * Q2;

                                                ENDWITH
```

Not totally specified functions

In the case of not-totally specified functions we can use the directive @DCSET. The @DCSET directive allows the optimization to use don't cares to optimize partially-specified logic functions. NOTE: This directive overrides attributes 'dc', 'neg' and 'pos'. If we don't use the directive the ABEL program will translate the don't care sets with value 0

Multiple Assignment

If a variable it is assigned more then on time in the left side of the equation descriptions, then the translation procedure will connect the assignments each to other with the OR function.

Example:

```
D = A;

D = B;

D = C;
```

The compilation will translate the module in the following equation:

```
D = A + B + C;
```

The inverted assignment will be translated in the same way:

Example:

```
/D = A;

/D = B;

/D = C;
```

Translated:

```
/D = A + B + C;
```

which is not quite OR function. What signal is taken inverted it depends on the declaration section. The example presented above we presume the declaration of the `true' D signal declaration. From this results that the compiler will translate the equations with errors. So we advise to not use multiple assignments in your function description.

Test

Vectors Section

Test vectors, which are optional, verify that the logic design functions as intended. Test vectors specify the expected operation of a logic device by defining its outputs as a function of its inputs. Design test vectors can be used in conjunction with test vectors generated by PLDtest Plus, which functionally test the programmed device.

The translation procedure will write the test vectors in the .JED JEDEC file, and they will help the simulation of the JEDEC file.

Syntax:

```
TEST_VECTORS [note ] ([inputs] -> [outputs])

[in_values] -> [out_values];

[in_values] -> [out_values];

[in_values] -> [out_values];

.
```

Example:

```
TEST_VECTORS ([A, B] -> [O1, O2]

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

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

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

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

Test vectors can be specified in multiple assignment procedure and the output value can take don't care values (.X)

TRACE

The Trace statement limits which inputs and outputs are displayed in the simulation report. The TRACE statement is used to control the display features. TRACE statements can be placed before a test vector section, or imbedded within a sequence of test vectors.

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

Examples:

```
TRACE                   [A, B] -> [C]);

TEST_VECTORS    ([A, B] -> [CAD])

                                [0] -> [3];

                                [1] -> [2];

TRACE                   ([A, B] -> [D]);

                                [2] -> [1];

                                [3] -> [0];
```

End Statement

The End statement ends the module, and is required.

Syntax:

END module_name

Design considerations

Design of Sequential Circuits

Synchron Versus Asyncron Design: most of PLD Designer suggest and also the Xilinx™ data-book recommend that one should design its sequential circuits as synchron circuits instead of asyncron. Our experience said that designing with asyncron circuits could be problematical as the PLD circuits propagation time could vary by type and family and the ABEL environment do not support asyncron designs. Also could be a problem when translating an ABEL source program into Xilinx™ XNF format, when one should allow asyncron feedback, otherwise you may get translation error. Of course some simple exception can be made when the design is simple and it is recommended to design an asyncron circuit.

The Mealy-Model Versus Moore-Model: Most of PLD designers recommend to use the Moore type sequential circuits, when the outputs are the register's of the PLD circuit. In the case of not registered sequential outputs one have to consider the hazards what could appear in the design.

State Codification: In ABEL HDL state codification is up to the designer, but the code optimization is function of the state codification. So the realized circuits is function of the number of variables used in the state codification. When fitting the design, one get error message because of the number of product terms used, then you should try with a better state codification using the well known methods like `neighbor terms' or to use the strategy proposed by the ABEL Handbook: try to use a better state codification where the variable which has been changed many times in the former codification to change its state as few as possible. The state codification can be made easier if you give symbolic names for each state, and the codes corresponding to this symbolic names are declared separately.

Illegal and Power-Up States: At power-up the output of a flip-flop is undefined, could be L or H. For this reason on power-up the output of the sequential registers can be fixed only by reset signal, which force the register outputs on initial state. Most of PLDs have internal Reset logic, which force the register's output on initial state, so when design you should take this considerations. Can happen that the sequential circuit have illegal states which are not defined in the state specification. This illegal combinations could be source of errors and bugs after power-up, when the output of the sequential circuit appear such an illegal state. So better you should consider this states when designing and to guide this illegal states in legal states after one or two clock cycles.

Architecture Independent versus Detailed Description

The Device keyword is an optional feature in ABEL-HDL, so you do not need to specify a particular PLD architecture in your ABEL-HDL source file. You can also omit pin numbers from signal declarations. When you do not specify a device or pin numbers, you need to specify pin-to-pin attributes about declared signals, since the ABEL-HDL compiler cannot imply signal attributes from pre-determined device attributes. If you do not specify signal attributes or other detailed 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.

The requirement for signal attributes does not mean that a complex design must

always be specified with a particular device in mind. The attributes and dot extensions provided in ABEL-HDL help you to redefine your design to work consistently when moving from one class of device architecture to another.

By using attributes and dot extensions carefully, you can avoid specifying a particular device type, and instead target your design to a more general class of device architectures.

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

Signal Dot Extensions like attributes, are a way you can more precisely describe the behavior of a circuit that may be targeted to different architectures. Dot extensions are applied to signals, and remove the ambiguities in equations.

ABEL HDL assignment operator can be used when writing high-level equations. The = operator specifies a combinatorial assignment where the design is written with only the circuit's inputs and outputs in mind.

The := operator specifies a registered assignment, where you must consider the internal circuit elements (such as output inverters, resets and sets) related to the flip-flops.

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

```
QA := /QA + PRESET;
```

implies that QA 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 QA. 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 := operator, and the .CLK, .OE, .FB, .CLR, .ACLR, .SET, .ASET and .COM dot extensions. These dot extensions help to resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities In the equation above (QA := /QA + PRESET;), there is an ambiguous feedback condition. The signal QA appears on the right side of the equation, but there is no indication of whether that fed-back signal should originate at the register, should come directly from the combinatorial logic that forms the input to the register, or should come from the I/O pin associated with QA. 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 by writing:

```
QA.CLK = CLOCK;         `Register clocked from input'

QA = /QA.FB + PRESET;   `Register feedback'
```

This set of equations describes the circuit completely, and specifies enough information that the circuit will operate identically in virtually any PLD in which it can be fit. 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, as follows:

```
QA.CLK = CLOCK;         `Register clocked from input

QA.D = /QA.Q + PRESET;  `D-type flip-flop 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 the design can be implemented in. Furthermore, the equations only describe 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 (like P16R4 PAL device, for example), versus in a device with non-inverting outputs (such as an EP600).

To maintain the correct pin behavior using detailed equations, one additional language element, a 'buffer' (or its complement, 'invert') attribute, is required. 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 QA. For example, add the following to the declarations section:

```
QA pin istype 'buffer';
```

One way to understand the difference between pin-to-pin and detailed. description methods is to think of detailed descriptions as macrocell

Macrocells Specifications: A macrocell is a block of circuitry normally (but not always) associated with a PLD's I/O pin. Figure 1 illustrates a typical macrocell associated with signal QA:

Figure 1 Detail Macrocell

Detailed descriptions are written for the various input ports (shown in Figure 1 with dot extension labels) of the macrocell. Note that the macrocell shown features a configurable inversion between the Q output of the flip-flop and the output pin labeled QA. If this inverter is used (or if a device is selected that features a fixed inversion), then the behavior seen on the QA output pin will be inverted from the logic applied to or observed on the various macrocell ports, including the feedback port QA.Q.

Pin-to-pin descriptions, on the other hand, allow you to describe your circuit in terms of the behavior expected on an actual output pin, regardless of the architecture of the underlying macrocell. Figure 2 illustrates the pin-to-pin concept:

Figure 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.

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

```
module QA_1

title `Pin-to-pin description'

declarations

        qa pin istype 'reg';

        clock, preset pin;

equations

                qa.clk = clock;

                qa = /qa.fb + preset;


test vectors ([clock, preset] -> QA)
```

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

[. c. , 0 ] -> 0;

[ .c. , 0 ] -> 1;

[ . c. , 0 ] -> 0;

[ .c. , 1 ] -> 1;

[ .c. , 1 ] -> 1;

end QA_1

module QA_2

title `Detailed description'

declarations

        clock, preset pin;

equations

        qa.clk = clock;

        qa.d = /qa.q + preset;

test vectors ([clock, preset]) -> qa)

[ .c. , 1 ] -> 1;

[ . c. , 0 ] -> 0;

[ .c. , 0 ] -> 1;

[ .c. , 0 ] -> 0;

[ .c. , 1 ] -> 1;

[ .c. , 1 ] -> 1;

end QA_2
```

The pin-to-pin description shown if the `generic macrocell' is synthesized from whatever type of macrocell actually exists in the target device.

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

Using Dot Extensions: The source of feedback is normally set by the architecture of the target PLD. If you don't specify a specific feedback path, the design may operate differently in different device types. Specifying feedback paths with the .FB, .Q or .PIN dot extensions, you can eliminate architectural ambiguities. Specifying feedback paths also allows you to use the 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;) will result in register feedback normalized to the pin

value if a register feedback path exists. If no register feedback is available, pin feedback will be used. In this case, the FuseAsm module will check to make sure that the output enable does not conflict with the pin feedback path; an error will be 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 feedback signals that originate at a pin.

- .Q Extension – Signals specified with the .Q extension (for example, count.d = count.q + l;) will originate at the Q output of the associated flip-flop. The value feedback may or may not correspond to the value observed 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 observed 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.

To be architecture-independent, you must think of 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).

The following simple ABEL-HDL design describes a 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 PLD device. Since this design is architecture-independent, it will operate the same (disregarding initial powerup state) regardless of the type of device specified.

```
module pin2pin

title `Pin-to-Pin Architecture independent One Bit Synchronous Circuit'
declarations

        clk, toggle, ena pin 1, 2; 11;

        qout pin 19 istype 'reg';

c = .c.;

z = .z.;

equations

        @alternate

        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 this circuit is implemented in a simple P16R8 PAL device (either by adding a device declaration statement or by specifying the P16R8 in the FuseAsm process), the result would be a circuit like the one illustrated in Figure 3. This circuit is somewhat different from the specified circuit; since the P16R8 features inverted outputs, the design equation has been automatically modified by FuseAsm to fit the P16R8's architecture.

Figure 3. Architecture Independent Implementation of Pin2Pin program in a P16R8

Polarity Control

Automatic polarity control is a powerful feature in ABEL-HDL where ABEL converts a logic function 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 FX are equivalent.

```
F1 = (A & B);

/F2 = /(A & B);

/F3 = /A + /B;
```

In the example above, equation F3 uses two product terms, while equation F1 requires only one. This logic function will use fewer product terms in a non-inverting device such as the P10L8 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. In the following example, the inverted form of FX, equation F3, uses fewer product terms than equation F2.

```
F1 = (A + B) * (C + D);

F2 = A * C + A * D + B * C + B * D;

/F3 = /A * /B + /C * /D;  `equation resulted from F1 by negation
```

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

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

- Using `pos' and `neg' attributes at pin declaration
- Using `invert' and `buffer' attributes at pin declaration

The `pos' and `neg' attributes specify optimization for the polarity specified:

- 'pos' optimize circuit for positive polarity. Unspecified logic in truth tables and state diagrams becomes a 1
- 'neg' optimize circuit for negative polarity. Unspecified logic in truth tables and state diagrams becomes a 0.

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 doesn't 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.

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'.

Excessive number of Product Terms

Sometimes to implement designs in simple and cheap PLDs the excessive number of product term can be an implementation problem. The excessive number of product terms can be minimized with a corresponding state codification and a multi-level product term decomposition. In a simple two level function description the variables of y independent logical function contained in the sets Xi and u, where u is a logical function with the independent variables contained in the set Xj, then:

```
y = F1(Xi, u);

u = F2(Xj);
```

With logical decomposition introducing simpler logical equations with fewer number of product terms the function can be implemented in simpler PLDs. In the above example by declaring the variable y, u as PIN/NODE the number of product terms in each logical function decrease dramatically and the function can be implemented.

The reason why we have to introduce logical decomposition is that the number of AND functions what are OR-ed together in simple PLD structures is 8 and if we do not consider this than in the translation process may occur errors.

The disadvantage of the decomposition is that you loose output pins as the decomposition level increase and with the increasing decomposition the signal time response will increase.

Designing with FPGAs

ABEL-HDL allows you to generate source files with efficient logic for FPGAs, including Logic Cell Arrays (LCAs). With ABEL-HDL, you can describe your logic functions independent of the architecture in which they will be implemented. You can then implement your description into a number of different devices with no changes in many cases.

ABEL-HDL contains language elements that allow you to take advantage of features specific to particular FPGAs. For example, if a device directly supports clock enables, you can specify clock enables in your ABEL-HDL source file equations.

The following design strategies are helpful when designing for FPGAs.

- 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 appropriate for FPGA designs. Information on using dot extensions is provided in the specific FPGA fitter in our case the Xilinx™ 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 Statements Indicate external signals. Pin numbers are not recommended for Xilinx™ 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 later interpret the signal incorrectly and delete it.

- NODE Statements indicate internal signals. Signals declared as nodes are expected to have a source and loads. For example, Figure 1 shows a state machine as a functional block. State bits S1 through S7 are completely internal; all other signals are external.

Figure 4 Hypothetical state machine as a Functional Block

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:

```
declarations

CLOCK, RESET PIN;

I0, l1, I2, I3 PIN;

01, 02 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 5 shows a schematic of combinatorial 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 output to the sub-circuit. The following ABEL sequence shows the ABEL-HDL declarations and equations that would generate the logic shown in Figure 5.

```
module no_intermediate

declarations

        A, B, C, D, E pin;

        X, Y pin;

equations

@alternate
```

```
X = A * B * C + B :+: C;

Y = A * D + A * E) + A * B * C);

end no_intermediate
```

Figure 5 Schematic without intermediate signals

Figure 6 Schematic with Intermediate Signals

Figure 6 shows the same logic using an intermediate signal, M, which is declared as a node and named, but is used only inside the sub-circuit as a component of other, more complex signals. The declarations and equations that would generate the logic are the following:

```
module intermediate

declarations

        A, B, C, D, E pin;

        X, Y pin;

        M node;

equations

        @alternate

        "intermediate signal equations

        M = A * B * C;

        X = M + B :+: C;

        Y = A * D + A * E + M;

end intermediate
```

Both design descriptions are functionally the same. Without the intermediate signal, ABEL 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. 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. . .

An alternative method of creating intermediate equations is to use the @CARRY directive. This directive causes comparators and adders to be generated using intermediate equations for carry logic, resulting in an efficient multi-level implementation.

In general, 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 sometimes 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.

The easyABEL Environment

Translation Programs

Compile (ahdl2pla): Check the source file syntax, compile it, synthesize the macros.

Simulate Equation (plasim): Makes a functional simulation of the logical description conform of the test vectors given in the HDL file.

Optimize (PLAOpt): Makes a logical minimization

PartMap (FuseAsm): The FuseAsm generate the JEDEC file ready to be programmed in the PLD device and also generate a document file. Before this operation you have to define the PLD device, if this is not made, then the Fitter (fit) makes a device for you.

Simulate JEDEC (jedsim): Simulate the JEDEC file, corresponding to the PLD device structure and the given test vectors.

Smart Part Option

This program modules helps the designer to choose the best PLD device (Device Selector program devsel) and to implement the design (based on functional description) in the PLD structure (Device Fitter fit)

Device Selector (devsel)

This module offer from the easyABEL supported device list those devices which correspond the prescriptions given by the designer, such as power consumption, speed, erasability, technology, device manufacturer.

Device Fitter (fit):

The fitter try to fit the design in the selected PLD device, if the fit is successful then makes the pin assignment too.

Running the easyABEL

The program can be started either from DOS™ or Windows™ environment and also you can run the program from the Foundation Project Manager.

Starting from DOS – type:

abel4

You will run in this way the DataIO easyABEL environment. In this way you can edit, test your programs and prepare your homework and then in the laboratory you can implement your project in the Xilinx™ Demonstration Board XC3000 series or XC4000 series depending on your project.

Starting from Foundation Project Manager:

This possibility is accessible only in the departments laboratories, and under Windows™ you have to do the following steps:

- Start the Foundation Project Manager

- Create your own project directory in the directory: C:\XACTUSER; with File\New_Project and also specify the XC family and the part type corresponding to your Demo_Board

- You can start to introduce your design by selecting the HDL Entry icon /with or without HDL wizard/

- Or by selecting the schematic editor icon and then use the Hierarchy\New_Symbol_Wizard to create a symbol with the same name as your ABEL code is, then placing in the schematic you can PUSH in the hierarchy to edit your ABEL code.

Help is available by pressing the ALT+H or clicking with the mouse on the menu HELP command.

Design Translation

Translating the design with easyABEL or Foundation is slightly different but