# Using ABEL with Xilinx CPLDs

XAPP075 January, 1997 (Version 1.0) Application Note

**Summary**

This application note provides a basic overview of the ABEL language and gives examples showing how to use ABEL to fully utilize the specific features of Xilinx CPLDs.

**Xilinx Families**

XC9500, XC7300



## Introduction

ABEL (Advanced Boolean Expression Language), combined with the Xilinx fitter software, provides a complete behavioral development environment for entering, simulating, and implementing designs for Xilinx CPLDs. And, because ABEL was developed specifically for programmable logic devices, it provides several important features that support the Xilinx CPLD architecture.

## ABEL Language Structure

ABEL designs are organized into modules. Each module contains at least one set of declarations, logic descriptions, and an optional set of test vectors. Most designs are completely specified in a single module. However, using the hierarchal feature found in ABEL 6, multi-module designs can also be specified. Figure 1 shows the distinct sections of code that are necessary to completely specify a design.

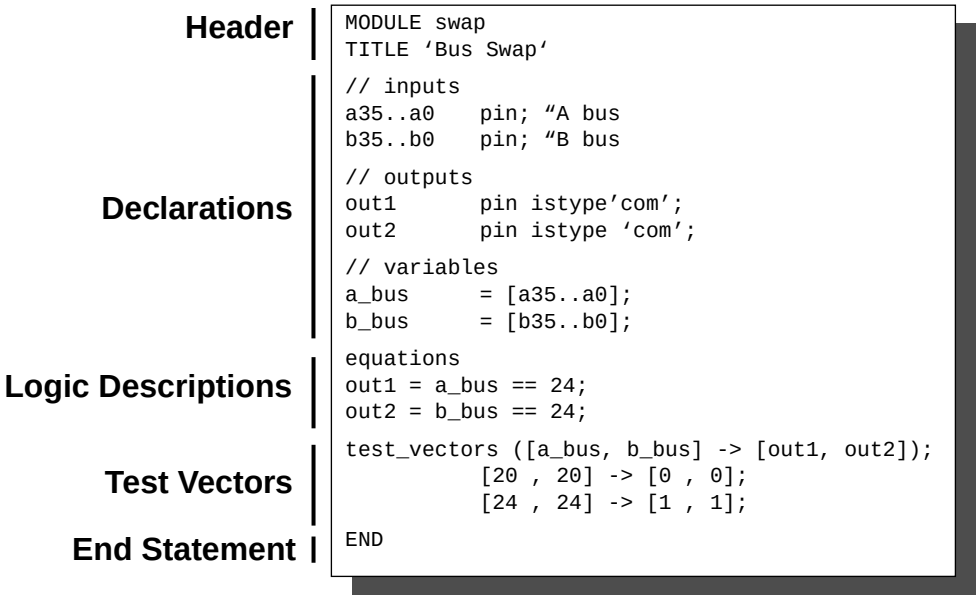



**Figure 1: ABEL Module Structure**

## The Header Section

The module name is specified by the keyword **Module.** An optional **Title** may also be used after the module name to further describe the design. The double quote or the double slash can also be used to add comments. At the end of each module definition the keyword **End** is used to specify the end of the design.

For example:

```
MODULE mydesign

TITLE  'version 1 of mydesign'
"Additionally, comments can be added using
"the double quote
// or the double slash

END
```

## The Declarations Section

Declarations are used to define constants, signals, and sets, and to pass property statements to the fitter for controlling device specific features which are not directly supported by ABEL. The various elements contained in the Declaration section are:

- Constants — Constants are declared by assigning a value to a constant name.
- Input pins — specified with the **pin** keyword without types.
- Output pins — specified by the **pin** keyword and either registered or combinatorial by using the **istype** keyword.
- Nodes or buried logic — specified by the **node** keyword and can be either registered or combinatorial by using the **istype** keyword.
- Arrays of signals — declared by ending the signal name with a number and using the double period (**..**) syntax.
- Sets — declared to make the ABEL code easier to read and write by replacing long redundant signals with a single reference. In addition, higher level operations performed on sets allow very powerful and complex designs to be specified very quickly.

For example:

```
"Declaring constants
On = 1;
Off = 0;

"Declaring input pins
my_input, my_clk pin;

"Declaring output pins
my_combinatorial_output pin istype 'com';
my_registered_ouput     pin istype 'reg';

"Declaring nodes
my_combinatorial_node  node isypte 'com';
my_registered_node     node istype 'reg';

"Declaring a 5-bit array of nodes
Count4..Count0  node istype 'reg';

"Declaring a set to reference all 5-bits
Counter = [Count4..Count0];

"Property statements
xepld PROPERTY 'fast on';"Sets all outputs
                         "to fast slew.
```

Note that in the declarations section, the order in which constants and sets are defined is important. If constant X is defined with constant Y, for example, then constant Y must be defined first.

## The Logic Description Section

The logic description section specifies the functions of the design. This can be done in three ways: Equations, Truth Tables, and State Diagrams. Equations are useful for designs with regular patterns such as counters or multiplexors. Truth Tables are a good entry method for designs that do not have regular patterns, such as a 7-segment LED decoder. State Diagrams are useful for specifying designs with complex state machines.

Following the Declarations section, the design section is delimited by using the keyword **Equations**, **Truth_Table**, or **State_Diagram.** State_diagrams and truth tables using sequential logic will need an accompanying equations section to define clock signals as well. These keywords must be used when switching between the three design methods.

### *Equations*

Equation design entry primarily consists of assignment statements. These can be combinatorial assignments (**=**), or registered assignments using the delay operator (**:=**). All registered equations using the delay operator (:=) will behave as being implemented as an edge triggered flip-flop. Therefore, they must also have a clock associated with the signal name. This is done by using the **.clk** dot extension.

For example:

```
my_registered_node := my_input;
my_registered_node.clk = my_clk;
```

This is logically equivalent to describing the same logic with detailed dot extensions as follows.

```
my_registered_node.d = my_input;
my_registered_node.clk = my_clk;
```

Using equations to specify a design is similar to programming in other languages, except the context of the equations are evaluated in parallel rather than sequentially. In

the following example, the order in which the equations are written is only important in programming languages.

In ABEL, all of the equations are evaluated concurrently, thus, the order in which they are presented is *not* important.

For example, in a normal programming language such as C, the code:

```
x = x + 1;
total = total +x;
```

is not the same as

```
total = total + x;
x = x+1;
```

However, in ABEL

```
x := x+1;
total :=  total + x;
```

is the same as

```
total := total + x;
x := x + 1;
```

In order to process information sequentially in digital logic, registers are used. In the previous example, note that the assignments are made with a "**:=**" . This means that in order for **x** or **total** to actually change values, a rising edge clock signal must be received by the register. The implementation for this design would look something like the following (line numbers are added to accompany the following explanation):

```
1  MODULE example1
2
3  my_clock pin;
4  x7..x0 node istype 'reg';
5  total7..total0 pin istype 'reg';
6
7  x = [x7..x0];
8  total = [total7..total0];
9
10 @carry 4;"Limit the carry chain to
           "4-bits
11
12 EQUATIONS"Signals the beginning of
           "an equation section
13 [x, total].clk = my_clock;
           " Set the clock signals for
           " registers to my_clock
14
15 x := x+1;  "This will implement an
              "counter counting by 1
16 total := total + x;
             "Note this is an adder that
             "uses the @carry directive
             "to implement the 8 bit
            "adder with two 4-bit adders
17
18 TEST_VECTORS ([my_clock] -> [x, total])
19
20 [.C.] -> [ 1 , 0];
21 [.C.] -> [ 2 , 1];
22 [.C.] -> [ 3 , 3];
23 [.C.] -> [ 4 , 6];
24 [.C.] -> [ 5 , 10];
25 [.C.] -> [ 6 , 15];
26 [.C.] -> [ 7 , 21];
27
28 END
```

The beginning of this design (as for all ABEL designs) defines the module name. Following the module name are the declarations of the module. On line 3, the clock input is defined. On line 4, the nodes that will contain the information for **x** are declared as an 8-bit register. Line 5 declares the output pins for **total**, an 8-bit register. On line 7 and 8, the 8 bits are renamed to a single variable name, allowing the equations to be written quickly and clearly. Line 10 is an ABEL compiler directive which limits the lookahead carry chain of the adder to 4 bits. On line 12, the **EQUATIONS** keyword signifies the end of the Declarations section and the beginning of the logic descriptions.

Every register must have a clock associated with it and in this design, line 13 specifies that the **x** and **total** registers are clocked by the input **myclock**. The actual logic assignments on lines 15 and 16 determine how the registers are affected when **myclock** goes high. At the rising clock edge, **x** (after clock) will get the value of **x** (before clock) **+ 1**, and **total** (after clock) will get the value of **total** (before clock) **+ x** (before clock). Figure 2 shows a block diagram of the circuit described by this code.

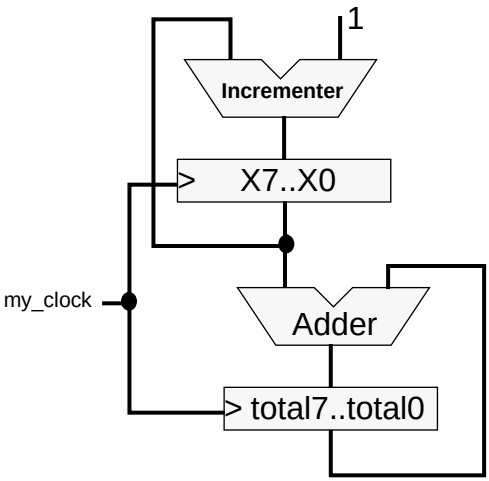



**Figure 2: Block Diagram**

### Test Vectors

Test vectors are also included at the end of this design. These test vectors show the expected values of the output for each rising clock signal, and are extremely useful for verifying the design. Test vectors are also included in the JEDEC programming file which can be used in the XC9500 family to perform an **INTEST operation** through the **JTAG** port.

### Dot Extensions

Dot extensions, illustrated in Figure 3, give more control over the implementation of the design. Dot extensions such as **.AP** or **.AR** are used to specify asynchronous preset and asynchronous reset for flip-flops. Other common dot extensions are **.OE** used to specify the output enable, and **.PIN** used with bi-directional signals. For a complete set of supported dot extensions, please refer to the documentation for the particular version of ABEL being used.

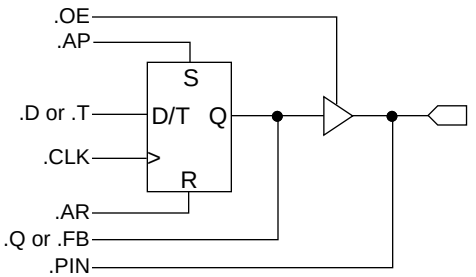



**Figure 3: Directly Supported ABEL Dot Extensions**

### Logical Operators

Logical operators allow the user to manipulate signals logically and can be applied to both signals and sets. When applied to sets, they are applied bitwise, and the sets must be the same size.

The operators are shown in Table 1.

### Arithmetic Operators

Arithmetic and relational operators can be used on sets to quickly generate adders, counters, and comparators. Large arithmetic functions, such as adders and magnitude comparators, will generate very wide equations to implement the carry look ahead signals. In order to control the width of these lookahead carry equations, the compiler directive, **@carry**, can be used. For example, **@carry 4**, would limit the carry chain to 4-bits. An 8-bit adder therefore, would be implemented as two 4 bit adders. Each adder would perform the carry lookahead in parallel for its own four bits. However, a carry signal will be generated by the lower four bits that will be cascaded to the higher order adder. Typically, carry chain lengths of 3 or 4 is a good trade-off between speed and density.

### Relational Operators

Relational operators are also used for generating the boolean result for conditional equations. These are useful for making the comparisons used in address decoding. Relational operators are used in conjunction with the conditional equation, **when...then...else** statement. Using conditional equations simplifies the task of building components with control signals.

For example, using operators:

```
"Using the logical OR operator.
my_reg_output := my_input # my_reg_node;
my_reg_output.clk = my_clock;

"Using the arithmetic operator
sum := a + b;
sum.clk = my_clock;

"Using the relational operators with
"conditional equations
WHEN (a != b) THEN
     c := my_input;
ELSE
     c := my_reg_node;
c.clk = my_clock;
```

Conditional equations allow the designer to make certain assignments depending upon the result of a relational operator. In the previous example, **c** is assigned **my_input** if **a** did not equal **b**. Otherwise, **c** will be assigned to the value stored in **my_registered_node**. Multiple assignments are made by containing them within the **{}** symbols.

The **else** statement does not need to be included in every conditional equation. When there are multiple conditions, they can be specified one at a time, however, make sure all possibilities are covered, otherwise unexpected behavior may occur in the design. For example:

```
"Using multiple WHEN without ELSE
"statements, the description of a mux
"is done by conditionally setting Output
"to Data_A or Data_B. Note the {} are used
"to make multiple assignments for each
"conditional.
Output2.clk = myclock;
WHEN (select == 0) THEN
{Output = Data_A;
Output2 := ValueA;}
WHEN (select == 1) THEN
{Output = Data_B;
 Output2 := ValueB;}
WHEN (select == 2) THEN
{Output = Data_C;
 Output2 := ValueC;}
"Note, select may also equal 3 and not
"defining the behavior for 3 will make
"the condition a don't care.
```

**Table 1: ABEL Operators**

| Logical | | Arithmetic | | Relational | |
|---|---|---|---|---|---|
| & | AND | - | Twos complement | == | Equal |
| # | OR | A-B | Subtraction | != | Not equal |
| ! | NOT | A+B | Addition | < | Less than |
| $ | XOR | << | Shift left | <= | Less than or equal |
| | | >> | Shift right | > | Greater than |
| | | | | >= | Greater than or equal |

## Using Truth Tables

Truth table design entry consists of specifying the input signals and the responding output signals, which can be registered or combinatorial. Truth tables are handy when specifying designs with irregular patterns, such as a 7-segment LED decoder.

A truth table starts with a header specifying all of the input and output signals. A transition specified with the **->** syntax is a combinatorial transition, while a **:>** specifies a registered transition. Registered truth tables must also include a clock equation to specify the clock input to the register. For example:

```
"Syntax for defining a truth table
TRUTH_TABLE ([ my_input ] ->
[my_combinatorial output] :>
[my_registered_output])

"A mux described with a truth table

EQUATIONS  "Note that we define the clock
           "with an equation first.
Output.clk = my_clock;

"Then, we specify the truth table input
"and outputs.
TRUTH_TABLE
([select]:> [ Output ])
  [ 0 ]  :> [ Data_A ];
"If select is 0,then Output gets Data_A
  [ 1 ]  :> [ Data_B ];
"If select is 1, then Output gets Data_B
```

### Entering Test Vectors

Test vectors can optionally be added to perform a functional test of the logic descriptions. For the XC9500 family, they are included in the JEDEC file which can also be used with the JTAG INTEST feature to test the functionality of the physical device in the system.

The **TEST_VECTORS** keyword signifies the end of the previous section and the start of the test vectors that are used to functionally simulate the design. Test vectors are entered in the same format as truth tables, and special signals are defined to help make the process easier. The three most common are **.C.** , **.Z.**, and **.X.**. The **.C.** signal represents a clock that starts from a logical low, goes to a logical high, and returns to a logical low. This is more convenient than entering three test vectors for each clock pulse. The **.Z.** is used to represent signals that are 3-stated, and the **.X.** signal represents a don't care signal. Don't care signals can be used both for inputs and outputs.

## Using State Diagrams

State diagrams tend to produce very legible and easy to maintain code. The following steps are used in creating a design using state diagrams.

1. Declare the state bits.
2. Declare a name for the set of state bits.
3. Assign a value for each state.
4. Define the state machine clock signals with equations.
5. Define the state transitions using the name defined in step 2.

Declaring state bits, and defining how each state is represented by those state bits is done by using the sets and constant declarations presented earlier. After defining the state bits and states, an equations section is needed to specify clock signals and other control logic. Using the STATE_DIAGRAM keyword, the designer can now specify assignments and state transitions for each of those states. To specify an unconditional state transition, use the **GOTO** statement. For conditional transitions, use the **IF...THEN... ELSE** statement. Note that this is different from the **WHEN...THEN...ELSE** used in the equations section.

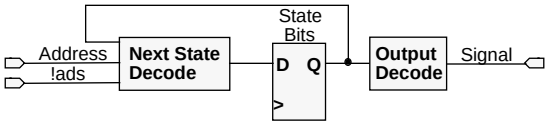
For example:

```
module traf

title 'Traffic Light Controler'

" A controller is needed to control the
" timing of a traffic light.
```

```
" The green light should be lit for thirty
" seconds.  Then a yellow light for two
" seconds, and a red light for
" thirty seconds We then repeat the entire
" cycle. This design must run on a clock
" which has a period of 1 sec.

"clock in with a period of 1 second.
clk             pin;

"reset to determine initial state.
reset           pin;

"Declare some nodes for a counter.
Count4..Count0  node istype 'reg';
Counter = [Count4..Count0];

"Step 1.  Declare the state bits.
"These bits are also our outputs
"in this example...
red             pin istype 'reg';
yellow          pin istype 'reg';
green           pin istype 'reg';

"Step 2.  Declare a name for the set of
"state bits.
"Step 3.  Assign a unique value for the
"for each state.
Light =   [green, yellow, red];
GO =      [ 1,      0,     0 ];
CAUTION = [ 0,      1,     0 ];
STOP =    [ 0,      0,     1 ];

Equations
"Step 4. Set up the clock and reset lines
"for the state machine.
green.ap    = reset;
red.ar      = reset;
yellow.ar   = reset;
Counter.ar  = reset;
Counter.clk = clk;
[green, yellow, red].clk = clk;

"Step 5.  Define the state transitions
"using the name defined in step 2.

State_Diagram Light

state GO:
IF (Counter < 30) then GO with
     Counter := Counter + 1;
ELSE goto CAUTION with
     Counter := Counter + 1;

state CAUTION:
IF (Counter != 0) then CAUTION with
     Counter := Counter + 1;
ELSE goto STOP with
     Counter := Counter + 1;

state STOP:
IF (Counter < 30) then STOP with
    Counter := Counter + 1;
ELSE goto GO with
    Counter := 1;

test_vectors
([clk,reset] -> [red, yellow, green])

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

  [.c., 0]   ->   [0,0,1];
@repeat 29
 {[.c., 0]   ->   [0,0,1];}

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

  [.c., 0]   ->   [1,0,0];
@repeat 29
 {[.c., 0]   ->   [1,0,0];}

  [.c., 0]   ->   [0,0,1];
@repeat 29
 {[.c., 0]   ->   [0,0,1];}

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

  [.c., 0]   ->   [1,0,0];
@repeat 29
 {[.c., 0]   ->   [1,0,0];}

end
```

Assignments can be made for state diagrams in two ways. If the assignment is made combinatorially in the state, then the outputs will be decoded from the state bits. This will improve the density of the final design, however it will add some delay. As shown in Figure 4, decoding from the present state adds a level of logic to achieve the desired output. For example:

```
zero_state:
output1 = 0;
GOTO one_state;

one_state:
output1 = 1;
GOTO zero_state;
```



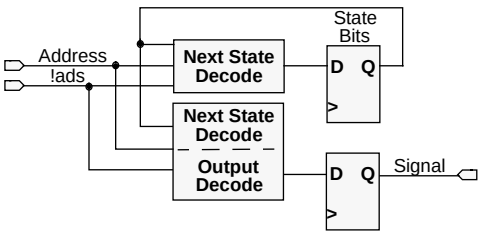

Output decoder may minimize product term requirement, but $t_{CO}$ is slower.

**Figure 4: Decoding Present State**

If the signal is declared as a register type, then while the next state is determined, the output is decoded at the same time, and will transition at the next clock edge. This implementation is shown in Figure 5. To implement this in ABEL, we use the **with** statement in our state diagram.

For example:

```
zero_state :
GOTO one_state
with output1 := 1;

one_state :
GOTO zero_state
with output1 := 0;
```




Duplication of the next state decoder may increase product term count, but $t_{CO}$ is faster.

**Figure 5: Decoding Next State**

## Using Property Statements

Although ABEL was initially developed for PLDs, there are still device-specific features that ABEL does not support directly. Instead, ABEL provides a **property** statement allowing device specific commands to be passed to the fitter software. Property statements must be placed in the declarations section. These property statements allow the user to control the following:

- Slew rates
- Logic optimizations
- Logic placement
- Power settings
- Preload values

### fast

The **fast** property controls the output slew rate, and there can only be one **fast** property used in each design. If there are only a few signals that require a fast slew rate, they can be listed individually after the property, and the remaining signals will be slew rate limited. Or, if there are only a few signals that need to be slew rate limited, then those signals can be listed.

```
xepld property ‘fast on’;
“all pins have fast slew rate

xepld property ‘fast on x1 x2’;
“ only x1 and x2 are fast
“ the remaining pins are slew limited

xepld property ‘fast off x1 x2’;
“only x1 and x2 are slew limited
“the remaining pins are fast
```

### logic_opt

The **logic_opt** property allows the user to control the logic optimization done by the fitter. This should be used on selective nodes, where collapsing those nodes would cause the design to become very large.

```
xepld property ‘logic_opt off’;
“Preserves all combinatorial nodes

xepld property ‘logic_opt off x1’;
“preserve x1 and collapse other nodes to
fitter limits
```

### minimize

The **minimize** property is used to prevent boolean minimization on equations, and is primarily used to prevent removal of redundant product terms in combinatorial logic.

```
xepld property ‘minimize off x1 x2’;
“keep redundant product terms for x1 & x2
```

### partition

The **partition** property is used when specific placement of logic is desired.

```
xepld property ‘partition fb1 x1 x2’;
“place the functions of x1 and x2 in
“function block 1

xepld property ‘partition fb1_2 x1’;
“place the function x1 in
“function block 1, macrocell 2
```

### pwr

The **pwr** property controls the power settings for individual macrocells.

```
xepld property ‘pwr low’;
“places all macrocells in low power mode

xepld property ‘pwr low x1 x2’
“places x1 and x2 in low power mode
“the remaining in STD power mode

xepld property ‘pwr std x1 x2’
“places x1 and x2 in STD power mode
“the remaining in low power mode
```

### .prld

The **.prld** property controls the initial state of the registers at power up. Note that because this property is only passed on and used by the fitter, and is not used by ABEL, the pre-load value will not be reflected in test vectors. The default preload value is 0 for all XC9500 registers. Therefore, only registers that require a value of 1 need to be specified.

For example:

```
xepld property ‘equation x1.prld = VCC’;
“preload register x1 to a 1
```

## DESIGN EXAMPLES

The following examples demonstrate some basic, specific design principles.

### Bi-Directional pins

This example shows how to implement a bi-directional signal in ABEL. Bi-directional signals are commonly found whenever a bus is being used by several different devices. This usually involves some kind of control signal to allow only one device to drive the bus at a given time. In this example, the input pin **write** is used to control if data is being driven on to the data pins **D7..D0** from an outside source, such as a microprocessor, or if data is being driven from the CPLD to be read from the data pins by an external source

```
module bidi;

“This design will take a value from the
“pins D7..D0 and store it in a Register
“when the signal, write, is high. When
“write goes low, it will output the
“saved value at pins D7..D0

“inputs
write           pin;
myclock         pin;

“Bi-directional signal also has a register
“associated with it.
D7..D0          pin istype ‘reg’;

“Define my sets
Data = [D7..D0];

Equations;
Data.oe = !write;  “3-State the data lines
                   “when writing to register
Data.clk = myclock;

WHEN (write==1) THEN Data := Data.pin;
“When we are writing to the part, read the
“data pins and save in data register.
ELSE
Data := Data;
“Else, drive the data pins with the value
“saved in the register so we can read it
“back.
end;
```

### Latches

Latches can be implemented in two ways. In the first example, latch_output utilizes the asynchronous set and reset of a flip-flop to implement a latch.

```
module ltest1

input,le        pin;
latch_output    pin istype ‘reg’;

equations

latch_output.ap = input & le;
latch_output.ar = !input & le;
latch_output.clk = 0; “Clock must be
                      “grounded
latch_output.d = 0; “D-input must be
                    “grounded

test_vectors    ([input, le] ->
[latch_output])
                [0,0] -> [0];
                [1,0] -> [0];
                [0,1] -> [0];
                [1,1] -> [1];
                [1,0] -> [1];
                [0,1] -> [0];

end;
```

Latches can also be implemented combinatorially by using a feedback path and providing a redundant product term to cover glitches. This will require the following code:

```
MODULE comlatch;

le          pin;
input       pin;
latch_out   pin  istype ‘com,retain’;
```

```
// The ABEL compiler will retain redundant
// logic for the latch_out output
// because they have the RETAIN attribute.
// However, the MINIMIZE OFF property
//statement is required to instruct the
// Xilinx fitter to also retain the
// redundant logic.

xepld property 'minimize off latch_out';
" The fitter will retain redundant logic
" for these nodes

EQUATIONS;

    latch_out = input & le
   " latch is transparent high
# latch_out & !le
" latch data on falling edge of le
# latch_out & input;
" Redundant product term

TEST_VECTORS
([ le, input] -> [latch_out]);
  [1 , 0  ]   -> [   0 ];  " transparent
  [1 , 1  ]   -> [   1 ];  " transparent
  [0 , 1  ]   -> [   1 ];  " latch a 1
  [0 , 0  ]   -> [   1 ];  " change input
  [1 , 0  ]   -> [   0 ];  " transparent
  [0 , 0  ]   -> [   0 ];  " latch a 0
  [0 , 1  ]   -> [   0 ];  " change data

END; " All modules must have an END state-
ment
```

## Counters

Counters are useful in a variety of applications, such as memory interfaces, generating delay states, or simple state machines. This example shows how to build a loadable up/down counter with a count enable.

```
module counter

"32-bit Up/Down counter with parallel load
"and enable

"Outputs
Q31..Q0      pin istype 'reg';

"Inputs
D31..D0      pin;
Load         pin;            " Load Cmd
Count_Enable pin;            " Count Cmd
UpDown       pin;           " Up/Down Cmd
myclk        pin;            " Clock

Counter = [Q31..Q0];
Input   = [D31..D0];

Equations
@carry 4;
Counter.clk = myclk;
WHEN (!Load & Count_Enable & UpDown)
     THEN Counter := Counter + 1
else
WHEN (!Load & Count_Enable & !UpDown)
     THEN Counter := Counter - 1
else
WHEN (Load)
     THEN Counter := Input;
else
Counter := Counter
end;
```

## Multiplexers

Multiplexers can be used to control the data flow of a design. The following example demonstrates how to implement a 16 bit 4-1 registered multiplexer:

```
module mux2
A15..A0         pin; " Inputs
B15..B0         pin; " Inputs
C15..C0         pin; " Inputs
D15..D0         pin; " Inputs
Q15..Q0        pin istype 'reg'; " Output

Sel1..Sel0      pin;
clk             pin;

Output  = [Q20..Q0];
DataA = [A20..A0];
DataB = [B20..B0];
DataC = [C20..C0];
DataD = [D20..D0];
Select  = [Sel1..Sel0];

Equations

Output.clk = clk;

WHEN Select == 0 THEN Output := DataA
else WHEN Select == 1 THEN Output := DataB
else WHEN Select == 2 THEN Output := DataC
else Output := DataD;

end
```

# Conclusion

ABEL allows complex behavioral designs to be easily implemented and simulated. In addition, the special features and capabilities of the device are easily accessed through ABEL property statements. ABEL is a simple yet powerful software tool that provides the designer with an efficient language for developing XC7300 and XC9500 designs.