4.6 The ABEL Hardware Design Language

ABEL is a hardware design language (HDL) that was invented to allow designers to specify logic functions for realization in PLDs. An ABEL program is a text file containing several elements:

- Documentation, including program name and comments.
- Declarations that identify the inputs and outputs of the logic functions to be performed.
- Statements that specify the logic functions to be performed.
- Usually, a declaration of the type of PLD or other targeted device in which the specified logic functions are to be performed.
- Usually, "test vectors" that specify the logic functions' expected outputs for certain inputs.

ABEL language processor

ABEL compiler

ABEL is supported by an *ABEL language processor*, which we'll simply call an *ABEL compiler*. The compiler's job is to translate the ABEL text file into a "fuse pattern" that can be downloaded into a physical PLD. Even though most PLDs can be physically programmed only with patterns corresponding to sum-of-products expressions, ABEL allows PLD functions to be expressed using truth tables or nested "`IF`" statements as well as by any algebraic expression format. The compiler manipulates these formats and minimizes the resulting equations to fit, if possible, into the available PLD structure.

We'll talk about PLD structures, fuse patterns, and related topics later, in \secref{PLDs} and show how to target ABEL programs to specific PLDs. In the meantime, we'll show how ABEL can be used to specify combinational logic functions without necessarily having to declare the targeted device type. Later, in \chapref{seqPLDs}, we'll do the same for sequential logic functions.

4.6.1 ABEL Program Structure

Table 4-10 shows the typical structure of an ABEL program, and Table 4-11 shows an actual program exhibiting the following language features:

identifier

- *Identifiers* must begin with a letter or underscore, may contain up to 31 letters, digits, and underscores, and are case sensitive.

`module`

- A program file begins with a *`module`* statement, which associates an identifier (`Alarm_Circuit`) with the program module. Large programs can have multiple modules, each with its own local title, declarations, and equations. Note that keywords such as "`module`" are not case sensitive.

LEGAL NOTICE ABEL (Advanced Boolean Equation Language) is a trademark of Data I/O Corporation (Redmond, WA 98073).

Table 4-10
Typical structure of an ABEL program.

```
module module name
title string
deviceID device deviceType;
pin declarations
other declarations
equations
equations
test_vectors
test vectors
end module name
```

title

- The *title* statement specifies a title string that will be inserted into the documentation files that are created by the compiler.

string

- A *string* is a series of characters enclosed by single quotes.

device

- The optional *device* declaration includes a device identifier (`ALARMCKT`) and a string that denotes the device type (`'P16V8C'` for a GAL16V8). The compiler uses the device identifier in the names of documentation files that it generates, and it uses the device type to determine whether the device can really perform the logic functions specified in the program.

comment

- *Comments* begin with a double quote and end with another double quote or the end of the line, whichever comes first.

pin declarations

- *Pin declarations* tell the compiler about symbolic names associated with the device's external pins. If the signal name is preceded with the NOT prefix (`!`), then the complement of the named signal will appear on the pin. Pin declarations may or may not include pin numbers; if none are given, the compiler assigns them based on the capabilities of the targeted device.

istype
com

- The *istype* keyword precedes a list of one or properties, separated by commas. This tells the compiler the type of output signal. The "`com`" keyword indicates a combinational output. If no `istype` keyword is given, the compiler generally assumes that the signal is an input unless it appears on the left-hand side of an equation, in which case it tries to figure out the output's properties from the context. For your own protection, it's best just to use the `istype` keyword for all outputs!

other declarations

- *Other declarations* allow the designer to define constants and expressions to improve program readability and to simplify logic design.

equations

- The *equations* statement indicates that logic equations defining output signals as functions of input signals will follow.

equations

- *Equations* are written like assignment statements in a conventional programming language. Each equation is terminated by a semicolon. ABEL uses the following symbols for logical operations:

Table 4-11 An ABEL program for the alarm circuit of Figure 4-11.

```
module Alarm_Circuit
title 'Alarm Circuit Example
J. Wakerly, Micro Systems Engineering'
ALARMCKT device 'P16V8C';

" Input pins
PANIC, ENABLEA, EXITING      pin 1, 2, 3;
WINDOW, DOOR, GARAGE         pin 4, 5, 6;
" Output pins
ALARM                        pin 11 istype 'com';

" Constant definition
X = .X.;

" Intermediate equation
SECURE = WINDOW & DOOR & GARAGE;

equations
ALARM = PANIC # ENABLEA & !EXITING & !(WINDOW & DOOR & GARAGE);

test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,    .X.,    .X.,   .X., .X.,   .X.] -> [    1];
[     0,      0,    .X.,   .X., .X.,   .X.] -> [    0];
[     0,      1,      1,   .X., .X.,   .X.] -> [    0];
[     0,      1,      0,     0, .X.,   .X.] -> [    1];
[     0,      1,      0,   .X.,   0,   .X.] -> [    1];
[     0,      1,      0,   .X., .X.,     0] -> [    1];
[     0,      1,      0,     1,   1,     1] -> [    0];

end Alarm_Circuit
```

&	AND.
#	OR.
!	NOT (used as a prefix).
$	XOR.
!$	XNOR.

& (AND)
# (OR)
! (NOT)
$ (XOR)
!$ (XNOR)

As in conventional programming languages, AND (&) has precedence over OR (#) in expressions. The *@ALTERNATE* directive can be used to make the compiler recognize an alternate set of symbols for these operations: +, ∗, / , :+:, and :∗:, respectively. This book uses the default symbols.

@ALTERNATE

- The optional `test_vectors` statement indicates that test vectors follow.
- *Test vectors* associate input combinations with expected output values; they are used for simulation and testing as explained in Section 4.6.7.

test_vectors
test vectors

.X.

- The compiler recognizes several special constants, including `.X.`, a single bit whose value is "don't-care."

end

- The `end` statement marks the end of the module.

unclocked assignment operator, =

Equations for combinational outputs use the *unclocked assignment operator*, `=`. The left-hand side of an equation normally contains a signal name. The right-hand side is a logic expression, not necessarily in sum-of-products form. The signal name on the left-hand side of an equation may be optionally preceded by the NOT operator `!`; this is equivalent to complementing the right-hand side. The compiler's job is to generate a fuse pattern such that the signal named on the left-hand side realizes the logic expression on the right-hand side.

4.6.2 ABEL Compiler Operation

The program in Table 4-11 realizes the alarm function that we described on page 213. The signal named ENABLE has been coded as `ENABLEA` because `ENABLE` is a reserved word in ABEL.

Notice that not all of the equations appear under the `equations` statement. An equation for an intermediate variable, `SECURE`, appears earlier. This equation is merely a definition that associates an expression with the identifier `SECURE`. The ABEL compiler substitutes this expression for the identifier `SECURE` in every place that `SECURE` appears after its definition.

In Figure 4-19 on page 214 we realized the alarm circuit directly from the SECURE and ALARM expressions, using multiple levels of logic. The ABEL compiler doesn't use expressions to interconnect gates in this way. Rather, it "crunches" the expressions to obtain a minimal two-level sum-of-products result appropriate for realization in a PLD. Thus, when compiled, Table 4-11 should yield a result equivalent to the AND-OR circuit that we showed in Figure 4-20 on page 214, which happens to be minimal.

In fact, it does. Table 4-12 shows the synthesized equations file created by the ABEL compiler. Notice that the compiler creates equations only for the `ALARM` signal, the only output. The `SECURE` signal does not appear anywhere.

The compiler finds a minimal sum-of-products expression for both `ALARM` and its complement, `!ALARM`. As mentioned previously, many PLDs have the ability selectively to invert or not to invert their AND-OR output. The "reverse polarity equation" in Table 4-12 is a sum-of-products realization of `!ALARM`, and would be used if output inversion were selected.

In this example, the reverse-polarity equation has one less product term than the normal-polarity equation for `ALARM`, so the compiler would select this equation if the targeted device has selectable output inversion. A user can also force the compiler to use either normal or reverse polarity for a signal by including the keyword "`buffer`" or "`invert`," respectively, in the signal's `istype` property list. (With some ABEL compilers, keywords "`pos`" and "`neg`" can be used for this purpose, but see Section 4.6.6.)

Table 4-12 Synthesized equations file produced by ABEL for program in Table 4-11.

```
ABEL 6.30

Design alarmckt created Tue Nov 24 1998

Title: Alarm Circuit Example
Title: J. Wakerly, Micro Systems Engineering

 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  -----------------
   4/3        6        1    Pin   ALARM
=========
   4/3          Best P-Term Total: 3
                       Total Pins: 7
                      Total Nodes: 0
            Average P-Term/Output: 3


Equations:

ALARM = (ENABLEA & !EXITING & !DOOR
     # ENABLEA & !EXITING & !WINDOW
     # ENABLEA & !EXITING & !GARAGE
     # PANIC);


Reverse-Polarity Equations:

!ALARM = (!PANIC & WINDOW & DOOR & GARAGE
     # !PANIC & EXITING
     # !PANIC & !ENABLEA);
```

4.6.3 WHEN Statements and Equation Blocks

In addition to equations, ABEL provides the *WHEN statement* as another means to specify combinational logic functions in the *equations* section of an ABEL program. Table 4-13 shows the general structure of a WHEN statement, similar to an IF statement in a conventional programming language. The ELSE clause is optional. Here *LogicExpression* is an expression which results in a value of true (1) or false (0). Either *TrueEquation* or *FalseEquation* is "executed" depending

WHEN statement

```
WHEN LogicExpression THEN
   TrueEquation;
ELSE
   FalseEquation;
```

Table 4-13 Structure of an ABEL WHEN statement.

on the value of *LogicExpression*. But we need to be a little more precise about what we mean by "executed," as discussed below.

In the simplest case, *TrueEquation* and the optional *FalseEquation* are assignment statements, as in the first two `WHEN` statements in Table 4-14 (for `X1` and `X2`). In this case, *LogicExpression* is logically ANDed with the right-hand side of *TrueEquation*, and the complement of *LogicExpression* is ANDed with the right-hand side of *FalseEquation*. Thus, the equations for `X1A` and `X2A` produce the same results as the corresponding `WHEN` statements but do not use `WHEN`.

Notice in the first example that `X1` appears in the *TrueEquation*, but there is no *FalseEquation*. So, what happens to `X1` when *LogicExpression* `(!A#B)` is false? You might think that `X1`'s value should be don't-care for these input combinations, but it's not, as explained below.

Formally, the unclocked assignment operator, `=`, specifies input combinations that should be added to the on-set for the output signal appearing on the left-hand side of the equation. An output's on-set starts out empty, and is augmented each time that the output appears on the left-hand side of an equation. That is, the right-hand sides of all equations for the same (uncomplemented) output are ORed together. (If the output appears complemented on the left-hand side, the right-hand side is complemented before being ORed.) Thus, the value of `X1` is 1 only for the input combinations for which *LogicExpression* `(!A#B)` is true and the right-hand side of *TrueEquation* `(C&!D)` is also true.

In the second example, `X2` appears on the left-hand side of two equations, so the equivalent equation shown for `X2A` is obtained by ORing two right-hand sides after ANDing each with the appropriate condition.

The *TrueEquation* and the optional *FalseEquation* in a `WHEN` statement can be any equation. In addition, `WHEN` statements can be "nested" by using another `WHEN` statement as the *FalseEquation*. When statements are nested, all of the conditions leading to an "executed" statement are ANDed. The equation for `X3` and its `WHEN`-less counterpart for `X3A` in Table 4-14 illustrate the concept.

The *TrueEquation* can be another `WHEN` statement if it's enclosed in braces, as shown in the `X4` example in the table. This is just one instance of the general use of braces described shortly.

Although each of our `WHEN` examples have assigned values to the same output within each part of a given `WHEN` statement, this does not have to be the case. The second-to-last `WHEN` statement in Table 4-14 is such an example.

It's often useful to make more than one assignment in *TrueEquation* or *FalseEquation* or both. For this purpose, ABEL supports equation blocks anywhere that it supports a single equation. An *equation block* is just a sequence of statements enclosed in braces, as shown in the last `WHEN` statement in the table. The individual statements in the sequence may be simple assignment statements, or they may be `WHEN` statements or nested equation blocks. A semicolon is not used after a block's closing brace. Just for fun, Table 4-15 shows the equations that the ABEL compiler produces for the entire example program.

equation block

Table 4-14 Examples of WHEN statements.

```
module WhenEx
title 'WHEN Statement Examples'

" Input pins
A, B, C, D, E, F                 pin;

" Output pins
X1, X1A, X2, X2A, X3, X3A, X4    pin istype 'com';
X5, X6, X7, X8, X9, X10          pin istype 'com';

equations

WHEN (!A # B) THEN X1 = C & !D;

X1A = (!A # B) & (C & !D);

WHEN (A & B) THEN X2 = C # D;
ELSE X2 = E # F;

X2A = (A & B) & (C # D)
    # !(A & B) & (E # F);

WHEN (A) THEN X3 = D;
ELSE WHEN (B) THEN X3 = E;
ELSE WHEN (C) THEN X3 = F;

X3A = (A) & (D)
    # !(A) & (B) & (E)
    # !(A) & !(B) & (C) & (F);

WHEN (A) THEN
  {WHEN (B) THEN X4 = D;}
ELSE X4 = E;

WHEN (A & B) THEN X5 = D;
ELSE WHEN (A # !C) THEN X6 = E;
ELSE WHEN (B # C) THEN X7 = F;

WHEN (A) THEN {
    X8 = D & E & F;
    WHEN (B) THEN X8 = 1; ELSE {X9 = D; X10 = E;}
} ELSE {
    X8 = !D # !E;
    WHEN (D) THEN X9 = 1;
    {X10 = C & D;}
}

end WhenEx
```

Table 4-15 Synthesized equations file produced by ABEL for program in Table 4-14.

```
ABEL 6.30

Design whenex created Wed Dec 2 1998

Title: WHEN Statement Examples

 P-Terms   Fan-in Fan-out Type Name
--------- ------ ------- ---- -----
   2/3       4       1    Pin  X1
   2/3       4       1    Pin  X1A
   6/3       6       1    Pin  X2
   6/3       6       1    Pin  X2A
   3/4       6       1    Pin  X3
   3/4       6       1    Pin  X3A
   2/3       4       1    Pin  X4
   1/3       3       1    Pin  X5
   2/3       4       1    Pin  X6
   1/3       3       1    Pin  X7
   4/4       5       1    Pin  X8
   2/2       3       1    Pin  X9
   2/4       5       1    Pin  X10
=========
  36/42      Best P-Term Total: 30
                    Total Pins: 19
                   Total Nodes: 0
        Average P-Term/Output: 2
```

```
Equations:

X1 = (C & !D & !A
     # C & !D & B);

X1A = (C & !D & !A
     # C & !D & B);

X2 = (D & A & B
     # C & A & B
     # !B & E
     # !A & E
     # !B & F
     # !A & F);

X2A = (D & A & B
     # C & A & B
     # !B & E
     # !A & E
     # !B & F
     # !A & F);

X3 = (C & !A & !B & F
     # !A & B & E
     # D & A);

X3A = (C & !A & !B & F
     # !A & B & E
     # D & A);

X4 = (D & A & B
     # !A & E);

X5 = (D & A & B);


X6 = (A & !B & E
     # !C & !A & E);


X7 = (C & !A & F);



X8 = (D & A & E & F
     # A & B
     # !A & !E
     # !D & !A);

X9 = (D & !A
     # D & !B);

X10 = (C & D & !A
     # A & !B & E);
```

```
Reverse-Polarity Eqns:

!X1 = (A & !B
     # D
     # !C);

!X1A = (A & !B
     # D
     # !C);

!X2 = (!C & !D & A & B
     # !B & !E & !F
     # !A & !E & !F);

!X2A = (!C & !D & A & B
     # !B & !E & !F
     # !A & !E & !F);

!X3 = (!C & !A & !B
     # !A & B & !E
     # !D & A
     # !A & !B & !F);

!X3A = (!C & !A & !B
     # !A & B & !E
     # !D & A
     # !A & !B & !F);

!X4 = (A & !B
     # !D & A
     # !A & !E);

!X5 = (!A
     # !D
     # !B);

!X6 = (A & B
     # C & !A
     # !E);

!X7 = (A
     # !C
     # !F);

!X8 = (A & !B & !F
     # D & !A & E
     # A & !B & !E
     # !D & A & !B);

!X9 = (!D
     # A & B);

!X10 = (A & B
     # !D & !A
     # !C & !A
     # A & !E);
```

```
truth_table  (input-list -> output-list)
             input-value -> output-value;
             . . .
             input-value -> output-value;
```

Table 4-16 Structure of an ABEL truth table.

4.6.4 Truth Tables

ABEL provides one more way to specify combinational logic functions—the *truth table*, with the general format shown in Table 4-16. The keyword *truth_table* introduces a truth table. The *input-list* and *output-list* give the names of the input signals and the outputs that they affect. Each of these lists is either a single signal name or a *set*; sets are described fully in Section 4.6.5. Following the truth-table introduction are a series of statements, each of which specifies an input value and a required output value using the “->” operator. For example, the truth table for an inverter is shown below:

truth table
truth_table
input-list
output-list

unclocked truth-table operator, ->

```
truth_table (X -> NOTX)
             0 -> 1;
             1 -> 0;
```

The list of input values does not need to be complete; only the on-set of the function needs to be specified unless don’t-care processing is enabled (see Section 4.6.6). Table 4-17 shows how the prime-number detector function described on page 213 can be specified using an ABEL program. For convenience, the identifier NUM is defined as a synonym for the set of four input bits [N3,N2,N1,N0], allowing a 4-bit input value to be written as a decimal integer.

Table 4-17 An ABEL program for the prime number detector.

```
module PrimeDet
title '4-Bit Prime Number Detector'

" Input and output pins
N0, N1, N2, N3                   pin;
F                                pin istype 'com';

" Definition
NUM = [N3,N2,N1,N0];

truth_table (NUM -> F)
               1 -> 1;
               2 -> 1;
               3 -> 1;
               5 -> 1;
               7 -> 1;
              11 -> 1;
              13 -> 1;
end PrimeDet
```

Both truth tables and equations can be used within the same ABEL program. The `equations` keyword introduces a sequence of equations, while the `truth_table` keyword introduces a single truth table.

4.6.5 Ranges, Sets, and Relations

Most digital systems include buses, registers, and other circuits that handle a group of two or more signals in an identical fashion. ABEL provides several shortcuts for conveniently defining and using such signals.

range

The first shortcut is for naming similar, numbered signals. As shown in the pin definitions in Table 4-18, a *range* of signal names can be defined by stating the first and last names in the range, separated by "`..`". For example, writing "`N3..N0`" is the same as writing "`N3,N2,N1,N0`." Notice in the table that the range can be ascending or descending.

set

Next, we need a facility for writing equations more compactly when a group of signals are all handled identically, in order to reduce the chance of errors and inconsistencies. An ABEL *set* is simply a defined collection of signals that is handled as a unit. When a logical operation such as AND, OR, or assignment is applied to a set, it is applied to each element of the set.

Each set is defined at the beginning of the program by associating a set name with a bracketed list of the set elements (e.g., `N=[N3,N2,N1,N0]` in Table 4-18). The set element list may use shortcut notation (`YOUT=[Y1..Y4]`), but the element names need not be similar or have any correspondence with the set name (`COMP=[EQ,GE]`). Set elements can also be constants (`GT=[0,1]`). In any case, the number and order of elements in a set are significant, as we'll see.

Most of ABEL's operators, can be applied to sets. When an operation is applied to two or more sets, all of the sets must have the same number of elements, and the operation is applied individually to set elements in like positions, regardless of their names or numbers. Thus, the equation "`YOUT = N & M`" is equivalent to four equations:

```
Y1 = N3 & M3;
Y2 = N2 & M2;
Y3 = N1 & M1;
Y4 = N0 & M0;
```

When an operation includes both set and nonset variables, the nonset variables are combined individually with set elements in each position. Thus, the equation "`ZOUT = (SEL & N) # (!SEL & M)`" is equivalent to four equations of the form "`Zi = (SEL & Ni) # (!SEL & Mi)`" for `i` equal `0` to `3`.

relation
relational operator

Another important feature is ABEL's ability to convert "relations" into logic expressions. A *relation* is a pair of operands combined with one of the *relational operators* listed in Table 4-19. The compiler converts a relation into a logic expression that is 1 if and only if the relation is true.

The operands in a relation are treated as unsigned integers, and either operand may be an integer or a set. If the operand is a set, it is treated as an unsigned

Table 4-18 Examples of ABEL ranges, sets, and relations.

```
module SetOps
title 'Set Operation Examples'

" Input and output pins
N3..N0, M3..M0, SEL                       pin;
Y1..Y4, Z0..Z3, EQ, GE, GTR, LTH, UNLUCKY     pin istype 'com';

" Definitions
N    = [N3,N2,N1,N0];
M    = [M3,M2,M1,M0];
YOUT = [Y1..Y4];
ZOUT = [Z3..Z0];

COMP = [EQ,GE];
GT   = [ 0, 1];
LT   = [ 0, 0];

equations

YOUT = N & M;
ZOUT = (SEL & N) # (!SEL & M);
EQ = (N == M);
GE = (N >= M);
GTR = (COMP == GT);
LTH = (COMP == LT);
UNLUCKY = (N == 13) # (M == ^hD) # ((N + M) == ^b1101);

end SetOps
```

binary integer with the leftmost variable representing the most significant bit. By default, numbers in ABEL programs are assumed to be base-10. Hexadecimal and binary numbers are denoted by a prefix of "`^h`" or "`^b`," respectively, as shown in the last equation in Table 4-18.

^h hexadecimal prefix
^b binary prefix

ABEL sets and relations allow a lot of functionality to be expressed in very few lines of code. For example, the equations in Table 4-18 generate minimized equations with 69 product terms, as shown in the summary in Table 4-20.

Table 4-19 Relational operators in ABEL.

Symbol	*Relation*
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Table 4-20 Synthesized equations summary produced by ABEL for program in Table 4-18.

```
 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  -----------------
   1/2        2        1    Pin   Y1
   1/2        2        1    Pin   Y2
   1/2        2        1    Pin   Y3
   1/2        2        1    Pin   Y4
   2/2        3        1    Pin   Z0
   2/2        3        1    Pin   Z1
   2/2        3        1    Pin   Z2
   2/2        3        1    Pin   Z3
  16/8        8        1    Pin   EQ
  23/15       8        1    Pin   GE
   1/2        2        1    Pin   GTR
   1/2        2        1    Pin   LTH
  16/19       8        1    Pin   UNLUCKY
=========
  69/62         Best P-Term Total: 53
                       Total Pins: 22
                      Total Nodes: 0
            Average P-Term/Output: 4
```

*4.6.6 Don’t-Care Inputs

Some versions of the ABEL compiler have a limited ability to handle don’t-care inputs. As mentioned previously, ABEL equations specify input combinations that belong to the on-set of a logic function; the remaining combinations are assumed to belong to the off-set. If some input combinations can instead be assigned to the d-set, then the program may be able to use these don’t-care inputs to do a better job of minimization.

@DCSET

dc

?= don’t-care unclocked assignment operator

The ABEL language defines two mechanisms for assigning input combinations to the d-set. In order to use either mechanism, you must include the compiler directive `@DCSET` in your program, or include “`dc`” in the `istype` property list of the outputs for which you want don’t-cares to be considered.

The first mechanism is the *don’t-care unclocked assignment operator*, `?=`. This operator is used instead of `=` in equations to indicate that input combinations matching the right-hand side should be put into the d-set instead of the on-set. Although this operator is documented in the ABEL compiler that I use, unfortunately it is broken, so I’m not going to talk about it anymore.

The second mechanism is the truth table. When don’t-care processing is enabled, any input combinations that are not explicitly listed in the truth table are put into the d-set. Thus, the prime BCD-digit detector described on page 230 can be specified in ABEL as shown in Table 4-21. A don’t-care value is implied for input combinations 10–15 because these combinations do not appear in the truth table and the `@DCSET` directive is in effect.

```
module DontCare
title 'Dont Care Examples'
@DCSET

" Input and output pins
N3..N0, A, B                  pin;
F, Y                         pin istype 'com';

NUM = [N3..N0];
X = .X.;

truth_table (NUM->F)
              0->0;
              1->1;
              2->1;
              3->1;
              4->0;
              5->1;
              6->0;
              7->1;
              8->0;
              9->0;

truth_table ([A,B]->Y)
             [0,0]->0;
             [0,1]->X;
             [1,0]->X;
             [1,1]->1;

end DontCare
```

Table 4-21 ABEL program using don't-cares.

It's also possible to specify don't-care combinations explicitly, as shown in the second truth table. As introduced at the very beginning of this section, ABEL recognizes `.X.` as a special one-bit constant whose value is "don't-care." In Table 4-21, the identifier "`X`" has been equated to this constant just to make it easier to type don't-cares in the truth table. The minimized equations resulting from Table 4-21 are shown in Table 4-22. Notice that the two equations for `F` are not equal; the compiler has selected different values for the don't-cares.

```
Equations:
F = (!N2 & N1
     # !N3 & N0);
Y = (B);

Reverse-Polarity Equations:
!F = (N2 & !N0
     # N3
     # !N1 & !N0);
!Y = (!B);
```

Table 4-22 Minimized equations derived from Table 4-21.

```
test_vectors  (input-list -> output-list)
              input-value -> output-value;
              . . .
              input-value -> output-value;
```

Table 4-23 Structure of ABEL test vectors.

4.6.7 Test Vectors

test_vectors

input-list
output-list

ABEL programs may contain optional test vectors, as we showed in Table 4-11 on page 249. The general format of test vectors is very similar to a truth table and is shown in Table 4-23. The keyword `test_vectors` introduces a truth table. The *input-list* and *output-list* give the names of the input signals and the outputs that they affect. Each of these lists is either a single signal name or a set. Following the test-vector introduction are a series of statements, each of which specifies an input value and an expected output value using the "`->`" operator.

ABEL test vectors have two main uses and purposes:

1. After the ABEL compiler translates the program into "fuse pattern" for a particular device, it simulates the operation of the final programmed device by applying the test-vector inputs to a software model of the device and comparing its outputs with the corresponding test-vector outputs. The designer may specify a series of test vectors in order to double-check that device will behave as expected for some or all input combinations.
2. After a PLD is physically programmed, the programming unit applies the test-vector inputs to the physical device and compares the device outputs with the corresponding test-vector outputs. This is done to check for correct device programming and operation.

Unfortunately, ABEL test vectors seldom do a very good job at either one of these tasks, as we'll explain.

The test vectors from Table 4-11 are repeated in Table 4-24, except that for readability we've assumed that the identifier `X` has been equated to the don't-care constant `.X.`, and we've added comments to number the test vectors.

Table 4-24 actually appears to be a pretty good set of test vectors. From the designer's point of view, these vectors fully cover the expected operation of the alarm circuit, as itemized vector-by-vector below:

1. If `PANIC` is 1, then the alarm output (F) should be on regardless of the other input values. All of the remaining vectors cover cases where `PANIC` is 0.
2. If the alarm is not enabled, then the output should be off.
3. If the alarm is enabled but we're exiting, then the output should be off.

4-6. If the alarm is enabled and we're not exiting, then the output should be on if any of the sensor signals `WINDOW`, `DOOR`, or `GARAGE` is 0.

7. If the alarm is enabled, we're not exiting, and all of the sensor signals are 1, then the output should be off.

```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[    1,      X,      X,      X,    X,      X] -> [    1];   1
[    0,      0,      X,      X,    X,      X] -> [    0];   2
[    0,      1,      1,      X,    X,      X] -> [    0];   3
[    0,      1,      0,      0,    X,      X] -> [    1];   4
[    0,      1,      0,      X,    0,      X] -> [    1];   5
[    0,      1,      0,      X,    X,      0] -> [    1];   6
[    0,      1,      0,      1,    1,      1] -> [    0];   7
```

Table 4-24
Test vectors for the alarm circuit program in Table 4-11.

The problem is that ABEL doesn't handle don't-cares in test-vector inputs the way that it should. For example, by all rights, test vector 1 should test 32 distinct input combinations corresponding to all 32 possible combinations of don't-care inputs `ENABLEA`, `EXITING`, `WINDOW`, `DOOR`, and `GARAGE`. But it doesn't. In this situation, the ABEL compiler interprets "don't care" as "the user doesn't care what input value I use," and it just assigns 0 to all don't-care inputs in a test vector. In this example, you could have erroneously written the output equation as "`F = PANIC & !ENABLEA # ENABLEA & ...`"; the test vectors would still pass even though the panic button would work only when the system is disabled.

The second use of test vectors is in physical device testing. Most physical defects in logic devices can be detected using the *single stuck-at fault model*, which assumes that any physical defect is equivalent to having a single gate input or output stuck at a logic 0 or 1 value. Just putting together a set of test vectors that seems to exercise a circuit's functional specifications, as we did in Table 4-24, doesn't guarantee that all single stuck-at faults can be detected. The test vectors have to be chosen so that every possible stuck-at fault causes an incorrect value at the circuit output for some test-vector input combination.

single stuck-at fault model

Table 4-25 shows a complete set of test vectors for the alarm circuit when it is realized as a two-level sum-of-products circuit. The first four vectors check for stuck-at-1 faults on the OR gate, and the last three check for stuck-at-0 faults on the AND gates; it turns out that this is sufficient to detect all single stuck-at faults. If you know something about fault testing you can generate test vectors for small circuits by hand (as I did in this example), but most designers use automated third-party tools to create high-quality test vectors for their PLD designs.

```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[    1,      0,      1,      1,    1,      1] -> [    1];   1
[    0,      1,      0,      0,    1,      1] -> [    1];   2
[    0,      1,      0,      1,    0,      1] -> [    1];   3
[    0,      1,      0,      1,    1,      0] -> [    1];   4
[    0,      0,      0,      0,    0,      0] -> [    0];   5
[    0,      1,      1,      0,    0,      0] -> [    0];   6
[    0,      1,      0,      1,    1,      1] -> [    0];   7
```

Table 4-25
Single-stuck-at-fault test vectors for the minimal sum-of-products realization of the alarm circuit.