How to Use AHDL

This section describes how to develop a successful AHDL design. All sample files shown in this section are also available in the \max2work\ahdl directory created during MAX+PLUS II installation. (On a UNIX workstation, the max2work directory is a subdirectory of the /usr directory.)

Design practices are discussed in the following order:

- Introduction ................................................................. 18
- Combinatorial Logic ...................................................... 25
- Sequential Logic ........................................................... 47
- State Machines ............................................................. 54
- Implementing a Hierarchical Project ................................. 69
- Implementing ICELL & SOFT Primitives ............................ 81
- Implementing RAM & ROM ............................................. 83
- Naming a Boolean Operator or Comparator ....................... 84
- Using Iteratively Generated Logic .................................. 86
- Using Conditionally Generated Logic ............................... 87
- Using the Assert Statement .......................................... 89

Go to MAX+PLUS II Help for up-to-date information on how to use AHDL.
AHDL is an easy-to-use text entry language for describing logic designs. You can use the MAX+PLUS II Text Editor or your own text editor to create AHDL Text Design Files (.tdf), which can be incorporated into a project hierarchy together with other design files. You can then compile the project, simulate it, and program Altera devices.

AHDL consists of a variety of elements that are used in behavioral statements to describe logic. This section includes information on how these elements and statements are used; for detailed descriptions and rules, refer to Elements and Design Structure.

The following topics are discussed:

- Using Numbers ................................................................. 18
- Using Constants & Evaluated Functions .............................. 19
- Inserting an AHDL Template........................................... 22
- AHDL Examples ............................................................... 24

Using Numbers

Numbers are used to specify constant values in Boolean expressions and equations, arithmetic expressions, and parameter values. AHDL supports all combinations of decimal, binary, octal, and hexadecimal numbers.

The decode1.tdf file shown in Figure 2-1 is an address decoder that generates an active-high chip enable when the address is 370 Hex.

Figure 2-1. decode1.tdf

```vhdl
SUBDESIGN decode1
(  
  address[15..0] :INPUT;
  chip_enable :OUTPUT;
)
BEGIN
  chip_enable = (address[15] == H'0370');
END;
```
In this sample file, the decimal numbers 15 and 0 are used to specify bits of
the address bus. The hexadecimal number H'0370'' specifies the address
that is decoded.

Figure 2.2 shows a Graphic Design file (.gdf) that is equivalent to
decode1.tdf.

**Figure 2.2. decode1.gdf**

Go to the following topics for more information:

"Numbers in AHDL" on page 102 in *Elements*
"Parameters Statement" on page 142 in *Design Structure*
"Using Constants & Evaluated Functions" on page 19 in this section

**Using Constants & Evaluated Functions**

You can use a constant in an AHDL file to give a descriptive name to a
number or text string. Similarly, you can use an evaluated function to give a
descriptive name to an arithmetic expression. This name, which can be used
throughout a file, can be more informative and readable than the number,
string, or arithmetic expression. For example, the numeric constant
UPPER_LIMIT is more informative than the number 130.
MAX+PLUS II AHDL

Constants and evaluated functions are especially useful if the same number, text string, or arithmetic expression is repeated several times in a file: if it changes, only one statement needs to be changed. In AHDL, constants are implemented with Constant Statements, and evaluated functions are implemented with Define Statements.

The decode2.tdf file shown in Figure 2-3 has the same functionality as decode1.tdf (shown in Figure 2-1 on page 18), but uses the constant IO_ADDRESS instead of the number H"0370".

Figure 2-3. decode2.tdf

```vhdl
CONSTANT IO_ADDRESS := H"0370";

SUBDESIGN decode2

begin
  a[15:0] : INPUT;
  ce      : OUTPUT;
end

BEGIN
  IO_ADDRESS(15:0) == IO_ADDRESS;
END;
```

You can define constants and evaluated functions with arithmetic expressions. Constants and evaluated functions can also be defined with previously defined constants, evaluated functions, or parameters.

In the following example, the constant foo is defined with an arithmetic expression and the constant foo_plus_one is defined with the previously defined constant foo:

```vhdl
CONSTANT foo = 1 + 2 DIV 3 + LOG2(256);
CONSTANT foo_plus_one = foo + 1;
```

In the following example, the evaluated function CEILING_ADD is defined on the basis of the previously defined evaluated function MAX:

```vhdl
DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE CEILING_ADD(a,b) = MAX(a,b) + 1;
```

The Compiler evaluates arithmetic operators in arithmetic expressions and reduces them to numerical values. No logic is generated for these expressions.
The `strcmp.tdf` file shown in Figure 2-4 defines the constant FAMILY and uses it in an Assert Statement to check whether the current device family is FLEX 8000.

**Figure 2-4. `strmp.tdf`**

```vhdl
PARAMETERS
(
  DEVICE_FAMILY  % DEVICE_FAMILY is a predefined Altera parameter %
);

CONSTANT FAMILY = "FLEX8000";

SUBDESIGN strcmp
  a : INPUT;
  b : OUTPUT;
BEGIN
  IF (DEVICE_FAMILY = FAMILY) GENERATE
    ASSERT
      REPORT "Detected compilation for FLEX8000 family"
      SEVERITY INFO;
    b := a;
  ELSE GENERATE
    ASSERT
      REPORT "Detected compilation for % family"
      DEVICE_FAMILY
      SEVERITY ERROR;
    b := a;
  END GENERATE;
END;
```

The `minport.tdf` file shown in Figure 2-5 defines the evaluated function MAX, which ensures a minimum port width in the Subdesign Section:
Figure 2-5. minport.tdf

PARAMETERS (WIDTH);

DEFINE MAX(a,b): (a > b) ? a : b;

SUBDESIGN minport

.dataA(WIDTH, 0);  \* \* INPUT \* \*
.dataB(WIDTH, 0);  \* \* OUTPUT \* \*
BEGIN
  dataB[] = dataA[];
END;

Go to the following topics for more information:

"Constant Statement" on page 147 in Design Structure
"Define Statement" on page 149 in Design Structure
"Quoted & Unquoted Names" on page 97 in Elements
"Using Default Values for Variables" on page 39 in this section

Inserting an AHDL Template

The fastest way to create AHDL designs in MAX+PLUS II is to use the Altera-provided AHDL templates. With the AHDL Template command (Templates menu), available in the MAX+PLUS II Text Editor, you can insert AHDL templates into your TDF to speed design entry.

A single template is available for the overall AHDL file structure. This template, called "Overall Structure," lists all AHDL constructs in separate comment lines in the order in which they appear in a TDF. The syntax of these sections and statements is not included; you must replace the comment line with the correct AHDL syntax for each section you wish to use in your file.

Use the following steps to insert an AHDL template at the current insertion point in a MAX+PLUS II Text Editor file:

1. Save your file with the .tdf extension.

2. Choose AHDL Template (Templates menu). The AHDL Template dialog box is displayed, as shown in Figure 2-6:
Section 2: How to Use AHDL

Figure 2-5. AHDL Template Dialog Box

<table>
<thead>
<tr>
<th>AHDL Template</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Template Section:</strong></td>
</tr>
<tr>
<td><strong>Overall Structure</strong></td>
</tr>
<tr>
<td><strong>Assign Statement</strong></td>
</tr>
<tr>
<td><strong>Boolean Equation</strong></td>
</tr>
<tr>
<td><strong>Case Statement</strong></td>
</tr>
<tr>
<td><strong>Constant Statement</strong></td>
</tr>
<tr>
<td><strong>Default Statement</strong></td>
</tr>
<tr>
<td><strong>For Generate Statement</strong></td>
</tr>
<tr>
<td><strong>Function Prototype Statement (non-parameterized)</strong></td>
</tr>
<tr>
<td><strong>Function Prototype Statement (parameterized)</strong></td>
</tr>
<tr>
<td><strong>If Generate Statement</strong></td>
</tr>
<tr>
<td><strong>If Then Statement</strong></td>
</tr>
<tr>
<td><strong>In-Line Reference (non-parameterized)</strong></td>
</tr>
<tr>
<td><strong>In-Line Reference (parameterized)</strong></td>
</tr>
<tr>
<td><strong>In-Line Reference (named part association)</strong></td>
</tr>
<tr>
<td><strong>Include Statement</strong></td>
</tr>
<tr>
<td><strong>Instance Declaration (non-parameterized)</strong></td>
</tr>
<tr>
<td><strong>Instance Declaration (parameterized)</strong></td>
</tr>
<tr>
<td><strong>Logic Section</strong></td>
</tr>
<tr>
<td><strong>Machine Alias Declaration</strong></td>
</tr>
<tr>
<td><strong>Node Declaration</strong></td>
</tr>
<tr>
<td><strong>Option Statements</strong></td>
</tr>
</tbody>
</table>

3. Select a name in the **Template Section** box.

4. Choose OK.

---

**Figure 2-7. Defaults Statement Template**

```plaintext
DEFAULTS
    _node_name = _constant_value;
END DEFAULTS;
```
MAX+PLUS II AHDL

Each AHDL keyword is capitalized and each variable name starts with two underscores (_) to help you identify them. For example, you would replace the _node_name placeholder in Figure 2-7 with the name of a node. You can also use Syntax Coloring (Options menu) to make keywords and variables easy to see.

MAX+PLUS II provides templates for all AHDL constructs. These templates are listed in alphabetical order, and can be used to replace the comment lines in the "Overall Structure" template.

AHDL Examples

MAX+PLUS II provides AHDL examples to help you enter AHDL designs quickly. The sample AHDL Text Design Files used in this section are available in the \max2work\ahdl directory (a subdirectory of the /usr directory on a UNIX workstation). You can open these sample files with the MAX+PLUS II Text Editor or any standard text editor, save them with a different filename, and edit them as necessary to fit your needs.

MAX+PLUS II AHDL Help also contains examples that you can copy and paste directly into your TDF.

Choose How to Use Help (Help menu) for information on how to copy a help topic.
Combinatorial Logic

Logic is combinatorial if outputs at a specified time are a function only of the inputs at that time. Combinatorial logic is implemented in AHDL with Boolean expressions and equations, truth tables, and a variety of megafunctions and macrofunctions. Examples of combinatorial logic functions include decoders, multiplexers, and adders.

Information on combinatorial logic is available in the following topics:

- Implementing Boolean Expressions & Equations ........................................ 25
- Declaring Nodes ....................................................................................... 27
- Defining Groups ....................................................................................... 28
- Implementing Conditional Logic ............................................................... 31
  - If Then Statement Logic ......................................................................... 31
  - Case Statement Logic ............................................................................ 32
  - If Then Statement vs. Case Statement .................................................. 34
- Creating Decoders .................................................................................... 35
- Using Default Values for Variables .......................................................... 39
- Implementing Active-Low Logic ............................................................... 41
- Implementing Bidirectional Pins .............................................................. 43
- Implementing Tri-State Buses ................................................................. 45

Go to the following topics for more information:

"Using Iteratively Generated Logic" on page 86 in this section
"Using Conditionally Generated Logic" on page 87 in this section

Implementing Boolean Expressions & Equations

Boolean expressions are sets of nodes, numbers, constants, and other Boolean expressions, separated by operators and/or comparators, and optionally grouped with parentheses. A Boolean equation sets a node or group equal to the value of a Boolean expression.
The `bool1.tdf` file shown in Figure 2-8 shows two simple Boolean expressions that represent two logic gates.

**Figure 2-8. bool1.tdf**

```plaintext
SUBDESIGN bool1
(
   a0, a1, b       : INPUT;
   out1, out2      : OUTPUT;
)
BEGIN
   out1 := a1 & !a0;
   out2 := out1 | b;
END;
```

In this sample file, the `out1` output is driven by the logical AND of `a1` and the inverse of `a0`, and the `out2` output is driven by the logical OR of `out1` and `b`. Since these equations are evaluated concurrently, their order in the file is not important.

Figure 2-9 shows a GDF that is equivalent to `bool1.tdf`.

**Figure 2-9. bool1.gdf**

Go to the following topics for more information:

"Boolean Equations" on page 168 in *Design Structure*
"Boolean Expressions" on page 106 in *Elements*
Declaring Nodes

A node, which is declared with a Node Declaration in the Variable Section, can be used to hold the value of an intermediate expression.

Node Declarations are especially useful when a Boolean expression is used repeatedly. The Boolean expression can be replaced with a descriptive node name, which is easier to read.

The boole2.tdf file shown in Figure 2-10 contains the same logic as boole1.tdf, but has only one output.

Figure 2-10. boole2.tdf

SUBDESIGN boole2
(
    a0, a1, b        : INPUT;
    out              : OUTPUT;
)

VARIABLE
    a_equal_2       : NODE;
BEGIN
    a_equal_2 = a1 & !a0;
    out = a_equal_2 & b;
END;

This file declares the node a_equal_2 and assigns the value of the expression a1 & !a0 to it. Using nodes can save device resources when the node is used in several expressions.

Both ordinary nodes (NODE keyword) and tri-state nodes (TRI_STATE_NODE keyword) can be used. NODE and TRI_STATE_NODE differ in that multiple assignments to them yield different results:

- Multiple assignments to nodes of type NODE tie the signals together by wired-AND or wired-OR functions. The default values for variables declared in Defaults Statements determine the behavior: a VCC default produces a wired-AND function; a GND default produces a wired-OR function.

- Multiple assignments to a TRI_STATE_NODE tie the signals to the same node.

- If only one variable is assigned to a TRI_STATE_NODE, it is treated as NODE.
Figure 2-11 shows a GDF that is equivalent to boole2.tdf.

**Figure 2-11. boole2.gdf**

Go to the following topics for more information:

"Defaults Statement" on page 173 in Design Structure
"Implementing Tri-State Buses" on page 45 in this section
"Node Declaration" on page 162 in Design Structure

**Defining Groups**

A group, which can include up to 256 members (or "bits"), is treated as a collection of nodes and acted upon as one unit. A group name can be specified with a single-range group name, dual-range group name, or sequential group name format.

In Boolean equations, a group can be set equal to a Boolean expression, another group, a single node, VCC, GND, 1, or 0. In each case, the value of the group is different. The Options Statement can be used to specify whether the lowest numbered bit of the group will be the MSB, the LSB, or either.

Once a group has been defined, [ ] is a shorthand way of specifying an entire range. For example, a[4 . . 1] can also be denoted by a[1]; similarly, b[5 . . 4] [3 . . 2] can be represented by b[1].
The `group1.tdf` file shown in Figure 2-12 shows simple Boolean expressions that define multiple groups.

**Figure 2-12. group1.tdf**

```vhdl
OPTIONS BIT0 = MSB;
CONSTANT MAX_WIDTH = 1+2-3-3-1; \% MAX_WIDTH = 2
SUBDESIGN group1
(
  a[1:2], use_exp_in[1+2-2..MAX_WIDTH] : INPUT;
  d[1..2], use_exp_out[1+2*2-4..MAX_WIDTH] : OUTPUT;
  dual_range[1..2] : OUTPUT;
)
BEGIN
  use_exp_out[] = use_exp_in[];
  dual_range[[]] = VCC;
END;
```

In this example, the Options Statement is used to specify that the rightmost bit of each group will be the MSB, and a 1 (decimal) is added to group a[]. If 00 is applied to input a[], then the result of this sample program will be d[1] == 1 (decimal). The groups use_exp_in[] and use_exp_out[] show how constants and arithmetic expressions can be used to delimit group ranges.

The following examples illustrate group usage:

- When a group is set equal to another group of the same size, each member on the right is assigned to the member on the left that corresponds in position.

In the following example, each bit in the first group is connected to the corresponding bit in the second group. Bit d2 is connected to bit q3, d1 to q7, and d0 to q6:

```vhdl
d[2..0] = q[8..6]
```

In the following example, each bit in the first group is connected to the corresponding bit in the second group. Bit d1_1 is connected to bit q10, bit d1_0 to q9, bit d0_1 to q8, and bit d0_0 to q7:

```vhdl
d[1..0][1..0] = q[10..7]
```
When a group is set equal to a single node, all bits of the group are connected to the node. In the following example, d2, d1, and d0 are all connected to n:

\[ d[2..0] = n \]

When a group is set equal to VCC or GND, all bits of the group are connected to that value. In the following example, d2, d1, and d0 are all connected to VCC:

\[ d[2..0] = \text{VCC} \]

When a group is set equal to 1 (decimal), only the LSB of the group is connected to the value VCC. All other bits in the group are connected to GND. In the following example, only d0 is connected to VCC; the value 1 (decimal) is sign-extended to \( b'001' \).

\[ d[2..0] = 1 \]

When a group is set equal to another group of a different size, the number of bits in the group on the left side of the equation must be evenly divisible by the number of bits in the group on the right side of the equation. The bits on the left side of the equation are mapped to the right side of the equation, in order. The following equation is legal:

\[ a[4..1] = b[2..1] \]

In this equation, the bits are mapped as follows:

\[ a4 = b2 \]
\[ a3 = b1 \]
\[ a2 = b2 \]
\[ a1 = b1 \]

Go to the following topics for more information:

"Arithmetic Expressions" on page 103 in Elements
"Boolean Equations" on page 168 in Design Structure
"Groups" on page 99 in Elements
"Using Default Values for Variables" on page 39 in this section
Implementing Conditional Logic

Conditional logic chooses among different behaviors depending on the values of the logic inputs. If Then and Case Statements are ideal for implementing conditional logic:

- If Then Statements evaluate one or more Boolean expressions, then describe the behavior for different values of the expressions.
- Case Statements list alternatives that are available for each value of an expression. They evaluate the expression, then select a course of action on the basis of the value of the expression.

Conditional logic implemented with If Then and Case Statements should not be confused with logic that is generated conditionally in an If Generate Statement. Logic that is generated conditionally is not necessarily conditional logic.

If Then Statement Logic

The priority.tdf file shown in Figure 2-13 shows a priority encoder that converts the level of the highest-priority active input into a value. It generates a 2-bit code that indicates the highest-priority input driven by VCC.

Figure 2-13. priority.tdf

```vhdl
SUBDESIGN priority
(
  low, middle, high : INPUT;
  highest_level[1..0] : OUTPUT;
)
BEGIN
IF high THEN
  highest_level[0] = 3;
ELSIF middle THEN
  highest_level[0] = 2;
ELSIF low THEN
  highest_level[0] = 1;
ELSE
  highest_level[0] = 0;
END IF;
END;
```
In this example, the inputs high, middle, and low are evaluated to determine whether they are driven by VCC. The If Then Statement activates the equations that follow the active IF or ELSE clause, e.g., if high is driven by VCC, highest_level[] is 3.

If more than one input is driven by VCC, the If Then Statement evaluates the priority of the inputs, which is determined by the order of the IF and ELSE clauses (the first clause has the highest priority). In priority.tdf, high has the highest priority, middle has the next highest priority, and low has the lowest priority. The If Then Statement activates the equations that follow the highest-priority IF or ELSE clause that is true.

If none of the inputs are driven by VCC, the equations following the ELSE keyword are activated.

Figure 2-14 shows a GDF that is equivalent to priority.tdf.

Figure 2-14. priority.gdf

Go to the following topics for more information:
- "If Then Statement" on page 176 in Design Structure
- "If Then Statement vs. Case Statement" on page 34 in this section

Case Statement Logic

The decoder.tdf file in Figure 2-15 shows a 2-bit-to-4-bit decoder. It converts two binary code inputs into a "one-hot" code.
**Figure 2-15. decoder.tdf**

```
SUBDESIGN decoder
{
    code[1..0] : INPUT;
    out[3..0]  : OUTPUT;
}
BEGIN
    CASE code[] IS
        WHEN 0 => out[] = 'B'0000";
        WHEN 1 => out[] = 'B'0010";
        WHEN 2 => out[] = 'B'0100";
        WHEN 3 => out[] = 'B'1000"
    END CASE;
END;
```

In this example, the input group code [1..0] has the value 0, 1, 2, or 3. The equation following the appropriate \( \Rightarrow \) symbol in the Case Statement is activated. For example, if code [] is 1, out1 is set to 'B"0010". Since the values of the expression are all different, only one WHEN clause can be active at one time.

Figure 2-16 shows a GDF that is equivalent to decoder.tdf.

**Figure 2-16. decoder.gdf**

Go to the following topics for more information:

"Case Statement" on page 172 in Design Structure
"Creating Decoders" on page 35 in this section
"If Then Statement vs. Case Statement," next
If Then Statement vs. Case Statement

If Then and Case Statements are similar. In some cases, you can use either statement to achieve the same results. The following example shows the same operation expressed in both If Then and Case Statement formats:

If Then Statement:

```
IF a[] == 0 THEN
    y = c & d;
ELSIF a[] == 1 THEN
    y = e & f;
ELSIF a[] == 2 THEN
    y = g & h;
ELSIF a[] == 3 THEN
    y = i;
ELSE
    y = GND;
END IF;
```

Case Statement:

```
CASE a[] IS
    WHEN 0 =>
        y = c & d;
    WHEN 1 =>
        y = e & f;
    WHEN 2 =>
        y = g & h;
    WHEN 3 =>
        y = i;
    WHEN OTHERS =>
        y = GND;
END CASE;
```

Important differences exist between If Then and Case Statements:

- Any kind of Boolean expression can be used in an If Then Statement. Each expression following an IF or ELSIF clause may be unrelated to the other expressions in the statement. In a Case Statement, however, a single Boolean expression is compared to a constant in each WHEN clause.

- Using the ELSIF clause can result in logic that is too complex for the MAX+PLUS II Compiler, because each successive ELSIF clause must still test that the preceding IF/ELSIF clauses are false. The following example shows how the Compiler interprets an If Then Statement. If a and b are complex expressions, then the inversion of each expression is likely to be even more complex.

If Then Statement:

```
IF a THEN
    c = d;
END IF;

ELSIF b THEN
    c = e;
END IF;

ELSE
    c = f;
END IF;
```

Compiler Interpretation:

```
IF a THEN
    c = d;
END IF;

IF a & b THEN
    c = e;
END IF;

IF a & !b THEN
    c = f;
END IF;
```
Go to the following topics for more information:

"If Then Statement" on page 176 and "Case Statement" on page 172 in *Design Structure*

**Creating Decoders**

A decoder contains combinatorial logic that interprets input patterns and converts them to output values. In AHDL, you can use a Truth Table Statement or the `lpm_compare` or `lpm_decode` function to create a decoder.

The `7segment.tdf` file shown in Figure 2-17 is a decoder that specifies logic for patterns of light-emitting diodes (LEDs). The LEDs are illuminated in a seven-segment display to show the hexadecimal numbers 0 to 9 and the letters A to F.
Figure 2-17. 7segment.tdf

SUBDESIGN 7segment
{
  i[3..0] : INPUT;
  a, b, c, d, e, f, g : OUTPUT;
}
BEGIN
TABLE
   i[3..0] => a, b, c, d, e, f, g;
   H'0' => 1, 1, 1, 1, 1, 0, 0;
   H'1' => 0, 1, 1, 0, 0, 1, 0;
   H'2' => 1, 1, 0, 1, 1, 0, 0;
   H'3' => 1, 1, 1, 0, 1, 0, 1;
   H'4' => 0, 1, 1, 0, 1, 1, 1;
   H'5' => 1, 0, 1, 1, 1, 1, 1;
   H'6' => 1, 0, 1, 1, 1, 1, 1;
   H'7' => 1, 1, 1, 0, 0, 1, 0;
   H'8' => 1, 1, 1, 1, 1, 1, 1;
   H'9' => 1, 1, 1, 1, 1, 1, 1;
   H'A' => 1, 1, 1, 0, 0, 1, 0;
   H'B' => 0, 1, 1, 1, 1, 1, 1;
   H'C' => 1, 0, 1, 1, 1, 1, 0;
   H'D' => 0, 1, 1, 1, 1, 1, 0;
   H'E' => 1, 0, 0, 0, 0, 1, 1;
   H'F' => 1, 0, 0, 0, 0, 1, 1;
END TABLE;
END;

In this example, the output pattern for all 16 possible input patterns of i[3..0] is described in the Truth Table Statement.

Figure 2-18 shows a CDF that is equivalent to 7segment.tdf.
Section 2: How to Use AHDL

Figure 2-18. 7segment.gdf

The decode3.tdf file shown in Figure 2-19 is an address decoder for a
generalized 16-bit microprocessor system.

Figure 2-19. decode3.tdf

SUSBEGIN decode3
(
    addr[15..0], m/io    : INPUT;
    rom, ram, print, sp[2..1] : OUTPUT;
)
BEGIN

TABLE:

m/io, addr[15..0] => rom, ram, print, sp[1];
1, B'0000000000000000' => 1, 0, 0, B'00';
1, B'0000000000000000' => 0, 1, 0, B'00';
0, B'0000000011111111' => 0, 0, 1, B'01';
0, B'0000000011111111' => 0, 0, 0, B'10';

END TABLE;
END;

37
In this example, thousands of possible input patterns exist, so it is impractical to specify all of them in the Truth Table Statement. Instead, you can use an $x$ (don't care) logic level to indicate that the output does not depend on the input corresponding to the position of the $x$ characters. For example, in the first line of the TABLE statement, rom would be high for all 16,384 input patterns of addr [15..0] that start with 00. Therefore, you only need to specify the common portion of the input pattern (i.e., 00), then use $x$ characters for the rest of the input pattern. With don't care inputs, your project may require fewer device resources.

When you use $x$ (don't care) characters to specify a bit pattern, you must ensure that the pattern cannot assume the value of another bit pattern in the truth table. AHDL assumes that only one condition in a truth table is true at a time; therefore, overlapping bit patterns may cause unpredictable results.

The decoded4.tdf file shown in Figure 2-20 uses the lpm_decode function to achieve the same functionality as decoded1.tdf (described in "Using Numbers" on page 18).

**Figure 2-20. decoded4.tdf**

```
INCLUDE "lpm_decode.inc";

SUBDESIGN decoded4
(
    address[15..0] : INPUT;
    chip_enable : OUTPUT;
)
BEGIN
    chip_enable := lpm_decode(data, address[]);
    with (lpm_width=16, lpm_depth=2, 1, 0)
    BEGIN
        RETURN "exp[11:0] = {}";
    END;
END;
```

Go to the following topics for more information:

"Implementing Conditional Logic" on page 31 in this section
"Truth Table Statement" on page 183 in Design Structure
Using Default Values for Variables

You can define a default value for a node or group that is used when the value of the node or group is not specified elsewhere in the file. AHDL also allows you to assign the value of a node or group more than once in a single file. If these multiple assignments conflict, the default value is used to resolve the conflict. When no defaults are specified, the default value is GND.

You can use the AHDL Defaults Statement to specify default values for variables used in Truth Table, If Then, and Case Statements. For example, since the Logic Synthesizer automatically connects all AHDL truth table outputs to GND when no truth table input conditions are satisfied, you can use one or more Defaults Statements to drive truth table outputs to VCC instead.

You should not confuse default values for variables with default values for ports that are assigned in the Subdesign Section.

The default1.tdf file shown in Figure 2-21 evaluates inputs and chooses one of five ASCII codes based on the inputs.

**Figure 2-21. default1.tdf**

SUBDESIGN default1
{
    i[3..0] : INPUT;
    ascii_code[7..0] : OUTPUT;
}

END

DEFAULTS
    ascii_code[7] = 'B'00111111'; % ASCII Question mark "?" %
END

TABLE
    i[3..0] => ascii_code;

'B'1000" => 'B'01100001"; % 'a' %
'B'0100" => 'B'01100010"; % 'b' %
'B'0010" => 'B'01100011"; % 'c' %
'B'0001" => 'B'01100100"; % 'd' %
END TABLE;

END;
When an input pattern matches one of the patterns shown on the left side of the Truth Table Statement, the table's outputs are set to the corresponding pattern on the right. If the input pattern does not match any pattern on the left side, the outputs default to B'00111111', i.e., nodes ascii_code[5 .. 0] are driven to VCC while nodes ascii_code[7 .. 6] are connected to GND.

The default2.tdf file in Figure 2.22 illustrates how conflicts arise when a single node is assigned more than one value, and how these conflicts are resolved by AHDL.

**Figure 2.22. default2.tdf**

```vhdl
SUBDESIGN default2
{
    a, b, c, select_a, select_b, select_c : INPUT;
    wire_or, wire_and : OUTPUT;
BEGIN
    DEFAULTS
    wire_or = GND;
    wire_and = VCC;
    END_DEFAULTS;

    IF select_a THEN
        wire_or = a;
        wire_and = a;
    END IF;

    IF select_b THEN
        wire_or = b;
        wire_and = b;
    END IF;

    IF select_c THEN
        wire_or = c;
        wire_and = c;
    END IF;
END;
```

In this example, wire_or is set to the values of a, b, or c, depending on the values of the select_a, select_b, and select_c signals. If none of these signals is VCC, then wire_or defaults to GND.
If more than one of the `select_a`, `select_b`, or `select_c` signals are VCC, `wire_or` is set to the logical OR of the corresponding input values. For example, if `select_a` and `select_b` are VCC, `wire_or` is set to the logical OR of `a` and `b`.

The `wire_and` signal works in the same way, except that it defaults to VCC when none of the "select" signals is VCC, and is set to the logical AND of the corresponding input values when more than one of the signals is VCC.

Figure 2-23 shows a GDF that is equivalent to `default2.tdf`.

Go to the following topics for more information:

"Case Statement" on page 172 in Design Structure
"Defaults Statement" on page 173 in Design Structure
"If Then Statement" on page 176 in Design Structure
"Truth Table Statement" on page 183 in Design Structure

**Implementing Active-Low Logic**

An active-low signal becomes active when its value is GND. Active-low signals can be useful for controlling memory, peripheral, and microprocessor chips.
The daisy.tdf file shown in Figure 2-24 is a module of a daisy-chain arbitration circuit. This module makes requests for bus access to the preceding module in the daisy chain. It receives requests for bus access from itself and from the next module in the chain. Bus access is granted to the highest-priority module that requests it.

Figure 2-24. daisy.tdf

SUBDESIGN daisy

/\local_request : INPUT;
/\local_grant : OUTPUT;
/\request_in : INPUT; % from lower priority %
/\request_out : OUTPUT; % to higher priority %
/\grant_in : INPUT; % from higher priority %
/\grant_out : OUTPUT; % to lower priority %

BEGIN

DEFAULTS

/\local_grant = VCC; % active-low output %
/\request_out = VCC; % signals should default %
/\grant_out = VCC; % to VCC %

END DEFAULTS;

IF /\request_in =- GND & /\local_request = GND THEN
 /\request_out = GND;
END IF;

IF /\grant_in =- GND THEN
IF /\local_request = GND THEN
 /\local_grant = GND;
ELSIF /\request_in = GND THEN
 /\grant_out = GND;
END IF;
END IF;

END;

All signals in this file are active low. Altera recommends that you choose a node-naming scheme that clearly indicates active-low signal names—e.g., an initial “n” or a slash (/)—and use it consistently. A slash is not an operator, but is simply part of the signal name.

If Then Statements are used to determine whether modules are active, i.e., whether the signal equals GND. If a signal is active, the equations following the appropriate If Then Statement are activated. The Defaults Statement specifies that a signal is assigned to VCC when it is not active.

Figure 2-25 shows a GDF that is equivalent to daisy.tdf.
Implementing Bidirectional Pins

MAX+PLUS II allows I/O pins in Altera devices to be configured as bidirectional pins. Bidirectional pins can be specified with a BIDIR port that is connected to the output of a TRI primitive. The signal between the pin and the TRI primitive is a bidirectional signal that can be used to drive other logic in the project.

The bus_reg2.tdf file shown in Figure 2-26 implements a register that samples the value found on a tri-state bus. It can also drive the stored value back to the bus.

```
SUBDESIGN bus_reg2
  clk : INPUT;
  oe : INPUT;
  io : BIDIR;
BEGIN
  tri_out  = tri_reg(clk, io);
  io  = tri_out_oe;
END;
```
The bidirectional io signal, driven by TRI, is used as the d input to a D flipflop (DFP). Commas are used as placeholders for the clrn and prn flipflop ports, which default to the inactive state.

Figure 2-27 shows a GDF that is equivalent to bus_reg2.tdf.

**Figure 2-27. bus_reg2.gdf**

You can also connect a bidirectional pin from a lower-level TDF to a top-level pin. The bidirectional output port of the subdesign should be connected to a bidirectional pin at the top level of the hierarchy. The Function Prototype for the lower-level TDF should include the bidirectional pin in the RETURNS clause. The bidir1.tdf file shown in Figure 2-28 includes four instances of the bus_reg2 function shown in Figure 2-27.

**Figure 2-28. bidir1.tdf**

```vhdl
FUNCTION bus_reg2 (clk, oe) RETURNS (io);

SUBDESIGN bidir1
(
    clk, oe : INPUT;
    io[3..0] : BIDIR;
)
BEGIN
    io0 = bus_reg2(clk, oe);
    io1 = bus_reg2(clk, oe);
    io2 = bus_reg2(clk, oe);
    io3 = bus_reg2(clk, oe);
END;
```
Section 2: How to Use AHDL

Go to the following topics for more information:

"Ports" on page 132 in Elements
"Declaring Registers" on page 47 and "Using Default Values for Variables" on page 39 in this section

Implementing Tri-State Buses

TRI primitives that drive OUTPUT or BIDIR ports have an Output Enable input for placing the pin output in a high-impedance state in which it behaves as if it is not connected to the circuit.

You can create a tri-state bus by connecting TRI primitives and BIDIR or OUTPUT ports together with a node of type TRI_STATE_NODE. The control circuitry must ensure that at most one output is enabled (i.e., not in a high-impedance state) at any given time. This enabled output can transmit low (0) and high (1) logic levels onto the bus.

The tri_bus.tdf file shown in Figure 2-29 implements a tri-state bus using a TRI_STATE_NODE-type node created in a Node Declaration:

Figure 2-29. tri_bus.tdf

SUBDESIGN tri_bus

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in[3..1], oe[3..1] : INPUT;</td>
</tr>
<tr>
<td>out1 : OUTPUT;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
VARIABLE

| node : TRI_STATE_NODE; |
|
BEGIN
| node = TRI(in1, oe1); |
| node = TRI(in2, oe2); |
| node = TRI(in3, oe3); |
| out1 = node; |
END;

45
In `tri_bus.tdf`, multiple assignments to `tnode` tie the signals together. The `TRI_STATE_NODE` node type, rather than the ordinary `NODE` node type, is required to implement a tri-state bus: multiple assignments to nodes of type `NODE` tie the signals together by wired-AND or wired-OR functions; whereas multiple assignments to a `TRI_STATE_NODE` node tie the signals to the same node. However, if only one variable is assigned to a `TRI_STATE_NODE`-type node, it is treated as an ordinary `NODE` instead.

Go to the following topics for more information:

"Implementing Boolean Expressions & Equations" on page 25 in this section
"Declaring Nodes" on page 27 in this section
"Implementing Bidirectional Pins" on page 43 in this section
"Node Declaration" on page 162 in `Design Structure`
Sequential Logic

Logic is sequential if outputs at a specified time are a function of the inputs at that time and at some or all preceding times. All sequential circuits must include one or more flip-flops. Sequential logic can be implemented in AHDL with state machines, registers, or latches; LPM functions are also available. State machines are especially useful for implementing sequential logic. Other examples of sequential logic include counters and controllers.

Information on sequential logic is available in the following topics:

- Declaring Registers ................................................................. 47
- Declaring Registered Outputs .................................................... 50
- Creating Counters ..................................................................... 51

Go to the following topics for more information:

“State Machines” on page 54 in this section
“Megafunctions” on page 129 in Elements
“Using Iteratively Generated Logic” on page 86 in this section
“Using Conditionally Generated Logic” on page 87 in this section

Declaring Registers

Registers store data values and synchronize data with a Clock signal. You can declare, i.e., implement, an instance of a register with a Register Declaration in the Variable Section. (You can also implement registers with in-line references in the Logic Section.) AHDL offers several register primitives and also supports registered LPM functions.

Once you have declared a register, you can connect it to other logic in the TDF by using its ports. A port of an instance is used in the following format:

<instance name> . <port name>

The `bur_reg.tdf` file shown in Figure 2-30 uses a Register Declaration to create a byte register that latches values of the 4 inputs onto the 4 outputs on the rising edge of the Clock when the LOAD input is high.
Figure 2-30. bur_reg.tdf

SUBDESIGN bur_reg

  clk, load, d[7..0] : INPUT;
  q[7..0] : OUTPUT;

VARIABLE

  ff[7..0] : DFF;

BEGIN

  ff[1].clk = clk;
  ff[0].ena = load;
  ff[0].d = d[0];
  q[0] = ff[0].q;

END;

The registers are declared as Enable D flipflops (DFFE) in the Variable Section. The first Boolean equation in the Logic Section connects the bur_reg.tdf file's Clock input, clk, to the Clock ports of the ff[7..0] flipflops. The second equation connects the load input to the Clock Enable ports. The third equation connects the file's data inputs, d[7..0], to the data input ports of the ff[7..0] flipflops. The fourth equation connects the file's outputs to the flipflop outputs. All four statements are evaluated concurrently.

You can also declare T, JK, and SR flipflops in the Variable Section, then use them in the Logic Section. For example, for T flipflops (TFF), you would change the Register Declaration to ff[7..0] : TFF; and change ff[0].d to ff[0].t in the third equation. Similarly, for Enable JK flipflops (JKFFE), you would change the Register Declaration to ff[7..0] : JKFFE; and replace the third equation with two equations that connect the ff[0].j and ff[0].k ports to other signals.

If you wish to load a register on a specific rising edge of the global Clock, Altera recommends that you use the Clock Enable input of one of the DFFE, TFFE, JKFFE, or SRFFE Enable-type flipflops to control when the register is loaded.

The lpm_reg.tdf file shown in Figure 2-31 uses an in-line reference to implement an instance of the lpm_dff function that has the same functionality as the bur_reg.tdf file.
Figure 2-31. lpm_reg.tdf

INCLUDE "lpm_dll.inc";
SUBDESIGN lpm_reg

BEGIN
    clk, load, d[7..0] : INPUT;
    q[7..0] : OUTPUT;

    q[1] = lpm_dll (.clock=clk, .enable=load, .data=d[1])
        WITH (.LPM_WIDTH=8)
        RETURNS (.q[1]);
END;

Figure 2-32 shows a GDF that is equivalent to the TDFs in Figures 2-30 and 2-31.

Figure 2-32. reg.gdf

Go to the following topics for more information:

"Declaring Registered Outputs," next
"Ports" on page 132 in Elements
"Register Declaration" on page 163 in Design Structure
Declaring Registered Outputs

You can declare registered outputs of a subdesign by declaring the output ports as flipflops in a Register Declaration in the Variable Section. The `reg_out.tdf` file shown in Figure 2-33 has the same functionality as the `bur_reg.tdf` file shown in Figure 2-30 on page 48, but has registered outputs.

**Figure 2-33. reg_out.tdf**

```plaintext
SUBDESIGN reg_out
{
    clk, load, d[7:0] : INPUT;
    q[7:0] : OUTPUT;
}
VAR
    q[7:0] : DFFE; % also declared as output%1
BEGIN
    q[0].clk = clk;
    q[0].en = load;
    q[1] = d[1];
END;
```

When you assign a value to a registered output in the Logic Section, that value drives the d inputs to the registers. The register's output does not change until the rising edge of the Clock. To define the Clock input to the register, use `<register name>`.clk for the Clock input to the register in the Logic Section. You can implement a global Clock with the GLOBAL primitive or with the Automatic Global Clock option in the Global Project Logic Synthesis dialog box (Assign menu).

In the sample file shown in Figure 2-33, each Enable D flipflop (DFFE) declared in the Variable Section feeds an output with the same name, so you can refer to the q outputs of the declared flipflops without using the q port of the flipflops.

In a top-level TDF, output ports are synonymous with output pins. When you declare the same name for an output port and a register, any logic option assignments on that name are applied to the pin rather than the register. These identical names can prevent you from assigning a register-specific logic option such as I/O Cell Register. Therefore, if you wish to use a register-specific logic option, you must name the registers and ports differently. (However, you may be able to implement the desired functionality in a different way. For example, you can use the Automatic I/O Cell Registers option in the Global Project Logic...
Section 2: How to Use AHDL

Synthesis dialog box to automatically implement registers in I/O cells, regardless of whether you have declared the same name for output ports and registers.)

Go to the following topics for more information:

"Register Declaration" on page 163 in Design Structure
"Implementing a Hierarchical Project" on page 69 in this section

Creating Counters

Counters use sequential logic to count Clock pulses. Some counters can count forward and backward, and can be loaded with data and cleared to zero. Counters can be defined with D flipflops (DFF and DFFE) and If Then Statements or with the lpm_counter function.

The ahdclnt.tdf file shown in Figure 2-34 implements a 16-bit loadable up counter that can be cleared to zero.

Figure 2-34. ahdclnt.tdf

SUBDESIGN ahdclnt
{
    clk, load, ena, clr, d[15..0] : INPUT;
    q[15..0] : OUTPUT;
}

VARIABLE
    count[15..0] : DFF;
BEGIN
    count[1].clk = clk;
    count[1].clr = !clr;

    IF load THEN
        count[1].d = d[1];
    ELSIF ena THEN
        count[1].d = count[1].q + 1;
    ELSE
        count[1].d = count[1].q;
    END IF;
    q[11] = count[1];
END;
In this file, 16 D flipflops are declared in the Variable Section and assigned the names count 0 through count 15. The If Then Statement determines the value that is loaded into the flipflops on the rising Clock edge (e.g., if load is driven by vcc, the flipflops are assigned the value of d []).

The lpm_cnt_tdf file shown in Figure 2-35 uses the lpm_counter function to implement the same functionality as ahdlcnt_tdf:

**Figure 2-35. lpm_cnt_tdf**

```
#include "lpm_counter.inc"
subdesign lpm_cnt
{
    clk, load, ena, clr, d[15..0] : input;
    q[15..0] : output;
}

variable
    my_cnt : lpm_counter with (d[15..0] = 16);

begin
    my_cnt.clock = clk;
    my_cnt.load = load;
    my_cnt.ena = ena;
    my_cnt.clr = clr;
    my_cnt.data[] = d[];
    q[] = my_cnt.q[];
end;
```

Figure 2-36 shows a GDF that is equivalent to ahdlcnt_tdf and lpm_cnt_tdf.
Go to the following topics for more information:

"If Then Statement" on page 176 in Design Structure
"Implementing Conditional Logic" on page 31 in this section
State Machines

State machines, like truth tables and Boolean equations, are easily implemented in AHDL. The language is structured so that you can either assign state bits and state values yourself, or allow the MAX+PLUS II Compiler to do the work for you.

The Compiler uses advanced proprietary heuristic algorithms to make automatic state assignments that minimize the logic resources required to implement the state machine.

You simply need to draw a state diagram and construct a next-state table. The Compiler then performs the following functions automatically:

- Assigns bits, selecting either a T or D flipflop (TFF or DFF) for each bit
- Assigns state values
- Applies sophisticated logic synthesis techniques to derive the excitation equations

To specify a state machine in AHDL, you must include the following items in the TDF:

- State Machine Declaration (Variable Section)
- Boolean control equations (Logic Section)
- State transitions in Truth Table Statements or Case Statements (Logic Section)

You can also import and export AHDL state machines between TDF's and other design files by specifying an input or output signal as a machine port in the Subdesign Section.

The following topics provide information on creating state machines:

- Implementing State Machines ................................................................. 55
- Setting Clock, Reset & Enable Signals .................................................. 57
- Assigning State Machine Bits & Values .............................................. 58
- State Machines with Synchronous Outputs ........................................ 60
- State Machines with Asynchronous Outputs ...................................... 64
- Recovering From Illegal States ......................................................... 66
Section 2: How to Use AHDL

Go to the following topics for more information:

"Importing & Exporting State Machines" on page 77 in this section
"Ports" on page 132 in Elements

Implementing State Machines

You can create a state machine by declaring the name of the state machine, its states, and, optionally, the state machine bits in a State Machine Declaration in the Variable Section.

The simple.tdf file shown in Figure 2-37 has the same functionality as a D flipflop (DFF).

Figure 2-37. simple.tdf

```
SUBDESIGN simple

    clk, reset, d : INPUT;
    q : OUTPUT;

BEGIN
    VARIABLE
        ss : MACHINE WITH STATES [s0, s1];

    begin
        ss.clk = clk;
        ss.reset = reset;

        CASE ss IS
            WHEN ss = s0 =>
                q = '0;
            WHEN ss = s1 =>
                q = '1;
        END CASE;
    END;
```

55
In `simple.tdf`, a state machine with the name `ss` is declared in a State Machine Declaration in the Variable Section. The states of the machine are defined as `s0` and `s1`, and no state bits are declared.

State machine transitions define the conditions under which the state machine changes to a new state. You must conditionally assign the states within a single behavioral construct to specify state machine transitions. Case or Truth Table Statements are recommended for this purpose. For example, in `simple.tdf`, the transitions out of each state are defined in the `WHEN` clauses of the Case Statement.

You can also define an output value for a state with an If Then or Case Statement. In Case Statements, these assignments are made in `WHEN` clauses. For example, in `simple.tdf`, output `q` is assigned to GND when state machine `ss` is in state `s0`, and to VCC when the machine is in state `s1`.

Output values can also be defined in truth tables, as described in “Assigning State Machine Bits & Values” on page 58.

Figure 2-38 shows a GDF that is equivalent to `simple.tdf`.

**Figure 2-38. simple.gdf**

Go to the following topics for more information:

“Implementing Conditional Logic” on page 31 in this section
“Importing & Exporting State Machines” on page 77 in this section
“State Machine Declaration” on page 165 in *Design Structure*
“State Machines with Asynchronous Outputs” on page 64 in this section
“State Machines with Synchronous Outputs” on page 60 in this section
Setting Clock, Reset & Enable Signals

Clock, Reset, and Clock Enable signals control the flipflops of the state register in the state machine. These signals are specified with Boolean control equations in the Logic Section.

In the file `simple1.tdf` shown in Figure 2-39, the state machine Clock is driven by the input clk. The state machine’s asynchronous Reset signal is driven by reset, which is active high. In this design file, the declaration of the ena input in the Subdesign Section and the Boolean equation `ss.ena = ena` in the Logic Section connect the Clock Enable signal.

Figure 2-39. `simple1.tdf`

```
SUBDESIGN simple

    clk, reset, d, ena : INPUT;
    q : OUTPUT;

VARIABLE

    ss : MACHINE WITH STATES (s0, s1);
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    ss.ena = ena;

    CASE ss IS
    WHEN s0 =>
        q = GND;
    IF d THEN
        ss = s1;
    END IF;
    WHEN s1 =>
        q = VCC;
    IF not d THEN
        ss = s0;
    END IF;
    END CASE;
END;
```

Go to the following topics for more information:

"Boolean Control Equations" on page 171 in Design Structure
"Ports" on page 132 in Elements
Assigning State Machine Bits & Values

A state bit is an output of a flipflop used by a state machine to store one bit of the value of the state machine. In most cases, you should allow the MAX+PLUS II Compiler to assign state bits and values to minimize the logic resources required: the Logic Synthesizer automatically minimizes the number of state bits needed, optimizing both device utilization and performance.

However, some state machines may operate faster with state values that use more than the minimum number of state bits. In addition, you may want explicit state bits to be the outputs of a state machine. To control these cases, you can declare state machine bits and values in the State Machine Declaration.

The Global Project Logic Synthesis dialog box (Assign menu) includes a One-Hot State Machine Encoding option that automatically implements one-hot encoding for a project. In addition, the MAX+PLUS II Compiler automatically implements one-hot state machine encoding for FLEX 8000 and FLEX 10K devices, regardless of whether the One-Hot State Machine Encoding option is turned on or off. If you explicitly assign state bits in addition to using automatic one-hot encoding, your project's logic may be implemented inefficiently.

The stepper.tdf file shown in Figure 2-40 implements a stepper motor controller.
Figure 2-40. stepper.tdf

SUBDESIGN stepper
{
  clk, reset : INPUT;
  ccw, cw   : INPUT;
  phase[3:0] : OUTPUT;
}

VARIABLE

ss: MACHINE OF BITS (phase[3:0])
  WTH STATES {
    s0 = B"0000"
    s1 = B"0001"
    s2 = B"0100"
    s3 = B"1000"
  }
BEGIN
  ss.clk = clk;
  ss.reset = reset;

  TABLE
    ss, ccw, cw => ss;
    s0, 1,  x => s3;
    s0, x,  1 => s1;
    s1, 1,  x => s0;
    s1, x,  1 => s2;
    s2, 1,  x => s1;
    s2, x,  1 => s3;
    s3, 1,  x => s2;
    s3, x,  1 => s0;
  END TABLE;
END;

In this example, the phase[3:0] outputs declared in the Subdesign Section are also declared as bits of the state machine ss in the State Machine Declaration. Note that ccw and cw must never both be equal to 1 in the same table. AHDL assumes that only one condition in a truth table is true at a time; therefore, overlapping bit patterns may cause unpredictable results.

Go to “Truth Table Statement” on page 183 in Design Structure for more information.
State Machines with Synchronous Outputs

If the outputs of a state machine depend only on the machine's state, you can specify the state machine outputs in the WITH STATES clause of the State Machine Declaration. These state value assignments make state machine entry less prone to error, and in some cases, the logic may use fewer logic cells.

Figure 2-41 shows a four-state Moore state machine diagram. In Moore state machines, the present state of the state machine depends only on its previous input and previous state, and the present output depends only on the present state.

**Figure 2-41. Moore State Machine Diagram**

![State Machine Diagram]

The `moore1.tdf` file shown in Figure 2-42 implements a four-state Moore state machine.
Figure 2-42. moore1.tdf

SUHDSEIGN moore1
{
    clk : INPUT;
    reset : INPUT;
    y : INPUT;
    z : OUTPUT;
}
VARIABLE
    % current current %
    % state output %
    ss: MACHINE OF BITS (z) WITH STATES
    % 180 = 0,
    s0 = 1,
    s2 = 1,
    s3 = 0;
BEGIN
    ss.clk = clk;
    ss.reset = reset;

    TABLE
    % current current next %
    % state input state %
    ss, y => ss;
    s0, 0 => s0;
    s0, 1 => s2;
    s1, 0 => s0;
    s1, 1 => s2;
    s2, 0 => s2;
    s2, 1 => s3;
    s3, 0 => s3;
    s3, 1 => s1;
END TABLE;
END;

This example defines the states of the state machine with a State Machine Declaration. The state transitions are defined in a next-state table, which is implemented with a Truth Table Statement. In this example, machine ss has four states but only one state bit (z). The MAX+PLUS II Compiler automatically adds another bit and makes appropriate assignments to this synthetic variable to produce a four-state machine. This state machine requires at least two bits.

Figure 2-43 shows a GDF that is equivalent to moore1.tdf.
When state values are used as outputs, as in moore1.tdf, the project may use fewer logic cells, but the logic cells may also require more logic to drive their flipflop inputs. The Compiler's Logic Synthesizer module may not be able to fully minimize the state machine in these cases.

Another way to design state machines with synchronous outputs is to omit state value assignments and to explicitly declare output flipflops. The file moore2.tdf, shown in Figure 2-44, illustrates this alternative method.
Figure 2-44. moore2.tdf

SUEREISIGN moore2
/
clk : INPUT;
reset : INPUT;
y : INPUT;
z : OUTPUT;
)
VARIABLE
ss: MACHINE WITH STATES {s0, s1, s2, s3};
ze: NODE;
BEGIN
ss.clk = clk;
ss.reset = reset;
ss = DFF(ze, clk, VCC, VCC);

<table>
<thead>
<tr>
<th>% state</th>
<th>% input</th>
<th>% next state</th>
<th>% output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0, 0</td>
<td>y</td>
<td>s0, 0</td>
<td>z0</td>
</tr>
<tr>
<td>s0, 1</td>
<td></td>
<td>s2, 1</td>
<td>z1</td>
</tr>
<tr>
<td>s1, 0</td>
<td></td>
<td>s0, 0</td>
<td>z0</td>
</tr>
<tr>
<td>s1, 1</td>
<td></td>
<td>s2, 1</td>
<td>z1</td>
</tr>
<tr>
<td>s2, 0</td>
<td></td>
<td>s2, 1</td>
<td>z1</td>
</tr>
<tr>
<td>s2, 1</td>
<td></td>
<td>s1, 1</td>
<td>z1</td>
</tr>
<tr>
<td>s3, 0</td>
<td></td>
<td>s3, 0</td>
<td>z0</td>
</tr>
<tr>
<td>s3, 1</td>
<td></td>
<td>s1, 1</td>
<td>z1</td>
</tr>
</tbody>
</table>

END TABLE;
END;

Instead of specifying the output with state value assignments in the State Machine Declaration, this example includes a “next output” column after the “next state” column in the Truth Table Statement. This method uses a D flipflop (DFP)—called with an in-line reference—to synchronize the outputs with the Clock.

Go to “Truth Table Statement” on page 183 in *Design Structure* for more information.
Comparators

Two types of comparators are used to compare single nodes or groups: logical and arithmetic. Table 3-6 shows the comparators that can be used in Boolean expressions:

<table>
<thead>
<tr>
<th>Comparator:</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>== (logical)</td>
<td>addr[19..4] == $8'0800'</td>
<td>equal to</td>
</tr>
<tr>
<td>!= (logical)</td>
<td>b2 != b3</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt; (arithmetic)</td>
<td>face[] &lt; power[]</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;= (arithmetic)</td>
<td>money[] &lt;= power[]</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt; (arithmetic)</td>
<td>love[] &gt; money[]</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;= (arithmetic)</td>
<td>delta[] &gt;= 6</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

Logical comparators can compare single nodes, groups of nodes, and numbers without “don’t care” (x) values. If groups of nodes or numbers are compared, the groups must be the same size. The MAX+PLUS II Compiler performs a bitwise comparison on the groups, returning VCC when the comparison is true and GND when the comparison is false.

Arithmetic comparators may only compare groups of nodes and numbers, and the groups must be the same size. The Compiler performs an unsigned magnitude comparison on the groups; that is, each group is interpreted as a positive binary number and compared to the other group.

You can allow the Compiler to replace comparators in Boolean expressions with the lpm_compare function if you use the Use LPM for AHDl Operators logic option, or a logic synthesis style that includes this logic option. Go to “Assigning an Individual Logic Option or Synthesis Style” in MAX+PLUS II Help for more information.
Boolean Operator & Comparator Priorities

Operands separated by logical and arithmetic operators and comparators are evaluated according to the priority rules listed in Table 3-7 (priority 1 is the highest priority). Operations of equal priority are evaluated from left to right. Parentheses () may change the order of evaluation.

Table 3-7. Boolean Operator & Comparator Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator/Comparator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>==</td>
</tr>
<tr>
<td>3</td>
<td>!=</td>
</tr>
<tr>
<td>3</td>
<td>&lt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;=</td>
</tr>
<tr>
<td>3</td>
<td>&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&gt;=</td>
</tr>
<tr>
<td>4</td>
<td>&amp;</td>
</tr>
<tr>
<td>4</td>
<td>!&amp;</td>
</tr>
<tr>
<td>5</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>!$</td>
</tr>
<tr>
<td>6</td>
<td>#</td>
</tr>
<tr>
<td>6</td>
<td>!#</td>
</tr>
</tbody>
</table>

The arithmetic operators that are supported in Boolean expressions are a subset of the arithmetic operators supported in arithmetic expressions.