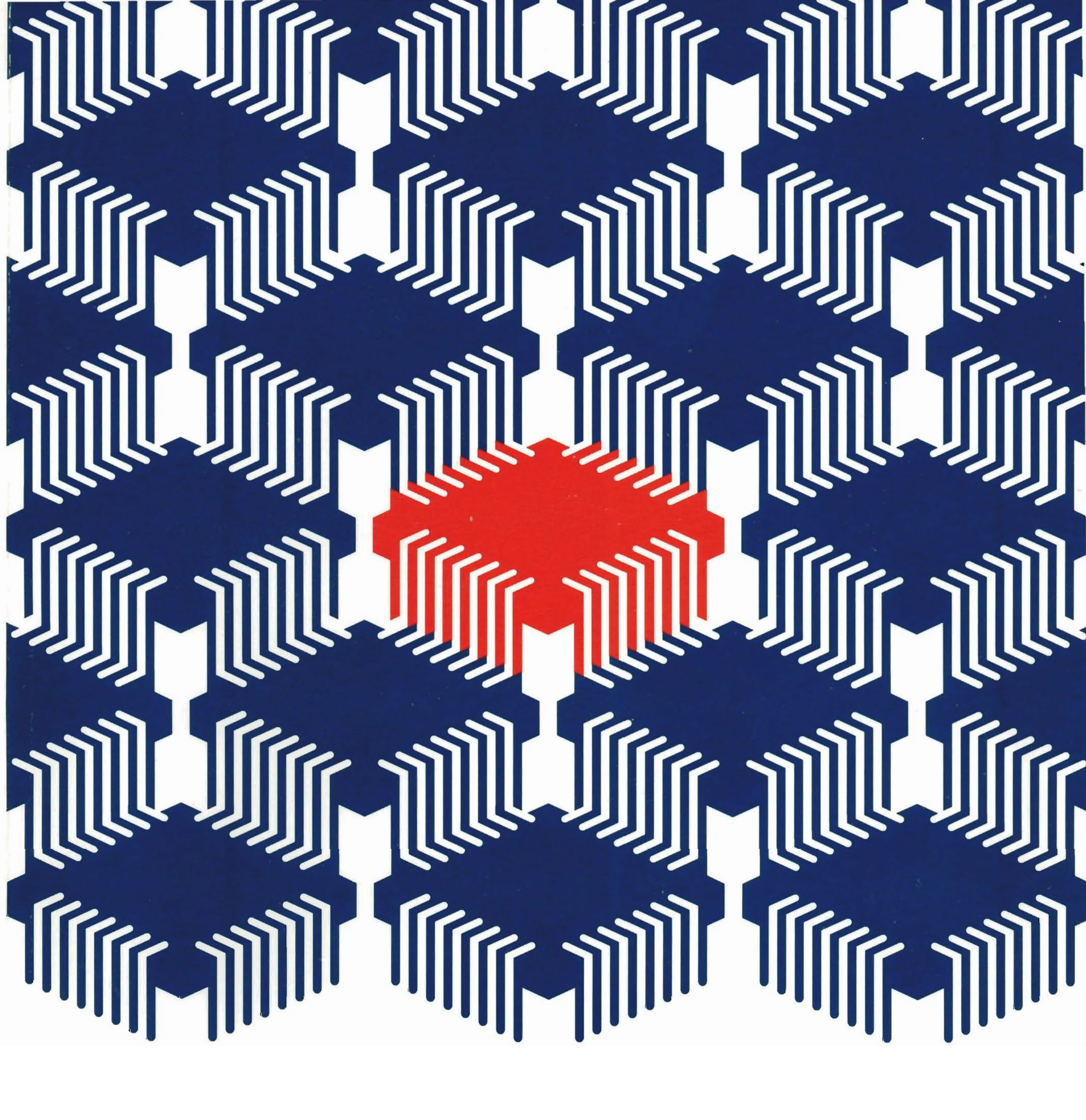


ALTERA®

MAX+PLUS® II AHDL

ALTERA®

MAX+PLUS® II AHDL

MAX+PLUS® II

Programmable Logic Development System

AHDL

ALTERA

Altera Corporation
101 Innovation Drive
San Jose, CA 95134
(408) 544-7000

MAX+PLUS II AHDL
Version 6.0
November 1995 P25-04802-02

Contents

Preface

MAX+PLUS II Documentation xii
  MAX+PLUS II Documents xii
  MAX+PLUS II Help xiii
  How to Use MAX+PLUS II Documentation xiii
Documentation Conventions xiv
  Terminology xiv
  Typographic Conventions xv
  Key Combinations xii
  Backus-Naur Form xii
MAX+PLUS II Help Updates xiii
Sample Files xix
About MAX+PLUS II AHDL xx

Section 1 Introduction

AHDL Design Entry 2
  How Does AHDL Work? 2
Text Design File Structure 4
  Text Design File Sections 4
  Files in a Project Hierarchy 7
  Include Files 7
MAX+PLUS II Text Editor 9
  AHDL Templates & Examples 9
  AHDL Context-Sensitive Help 10
  Syntax Coloring 10
  Resource & Device Assignments 11

Error Location 12
Compiling AHDL Text Design Files 13
Golden Rules 14
General Design Entry Golden Rules 15
General MAX+PLUS II Golden Rules 16

Section 2 How to Use AHDL

Introduction 18
Using Numbers 18
Using Constants & Evaluated Functions 19
Inserting an AHDL Template 22
AHDL Examples 24
Combinatorial Logic 25
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
Sequential Logic 47
Declaring Registers 47
Declaring Registered Outputs 50
Creating Counters 51
State Machines 54
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
Implementing a Hierarchical Project 69
Using Altera-Provided Unparameterized Functions 69
Using Altera-Provided Parameterized Functions 73
Using Custom Megafunctions & Macrofunctions 76
Importing & Exporting State Machines 77
Implementing LCELL & 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

Section 3 Elements

Reserved Keywords & Identifiers 92
  Reserved Keywords 92
  Reserved Identifiers 93
Symbols 94
Quoted & Unquoted Names 97
Groups 99
  Group Notations 99
  Group Ranges & Subranges 100
Numbers in AHDL 102
Arithmetic Expressions 103
Boolean Expressions 106
  Logical Operators 107
  Boolean Expressions Using NOT 108
  Boolean Expressions Using AND, NAND, OR, NOR, XOR, & XNOR 108
  Arithmetic Operators in Boolean Expressions 109
  Comparators 111
  Boolean Operator & Comparator Priorities 112
Primitives 113
  Buffer Primitives 114
    CARRY Primitive 114
    CASCADE Primitive 116
    EXP Primitive 117
    GLOBAL Primitive 118
    LCELL Primitive 120
    OPNDRN Primitive 122
    SOFT Primitive 123
    TRI Primitive 125
  Flipflop & Latch Primitives 125
  Primitive/Port Interconnections 127
Megafunctions 129
Old-Style Macrofunctions 131
Ports 132
  Ports of the Current File 132
  Ports of Instances 133
Parameters 136

Section 4 Design Structure

Overview 140
Title Statement 141
Parameters Statement 142
Include Statement 145
Constant Statement 147
Define Statement 149
Function Prototype Statement 151
Options Statement 154
Assert Statement 155
Subdesign Section 157
Variable Section 159
    Instance Declaration 160
    Node Declaration 162
    Register Declaration 163
    State Machine Declaration 165
    Machine Alias Declaration 166
Logic Section 168
    Boolean Equations 168
    Boolean Control Equations 171
    Case Statement 172
    Defaults Statement 173
    If Then Statement 176
    If Generate Statement 178
    For Generate Statement 179
    In-Line Logic Function Reference 180
    Truth Table Statement 183

Section 5 Style Guide

General Style Guidelines 188
White Space 190
Comments & Documentation 191
Naming Conventions 192
Indentation Guidelines 193

Glossary 197

Index 223

Illustrations

Figure		Page
1-1	MAX+PLUS II & AHDL Design Entry	3
1-2	AHDL Text Design File Structure	6
2-1	**decode1.tdf**	18
2-2	**decode1.gdf**	19
2-3	**decode2.tdf**	20
2-4	**strcmp.tdf**	21
2-5	**minport.tdf**	22
2-6	**AHDL Template** Dialog Box	23
2-7	Defaults Statement Template	23
2-8	**boole1.tdf**	26
2-9	**boole1.gdf**	26
2-10	**boole2.tdf**	27
2-11	**boole2.gdf**	28
2-12	**group1.tdf**	29
2-13	**priority.tdf**	31
2-14	**priority.gdf**	32
2-15	**decoder.tdf**	33
2-16	**decoder.gdf**	33
2-17	**7segment.tdf**	36
2-18	**7segment.gdf**	37
2-19	**decode3.tdf**	37
2-20	**decode4.tdf**	38
2-21	**default1.tdf**	39
2-22	**default2.tdf**	40
2-23	**default2.gdf**	41

Figure		Page
2-24	**daisy.tdf**	42
2-25	**daisy.gdf**	43
2-26	**bus_reg2.tdf**	43
2-27	**bus_reg2.gdf**	44
2-28	**bidir1.tdf**	44
2-29	**tri_bus.tdf**	45
2-30	**bur_reg.tdf**	48
2-31	**lpm_reg.tdf**	49
2-32	**reg.gdf**	49
2-33	**reg_out.tdf**	50
2-34	**ahdlcnt.tdf**	51
2-35	**lpm_cnt.tdf**	52
2-36	**count.gdf**	53
2-37	**simple.tdf**	55
2-38	**simple.gdf**	56
2-39	**simple1.tdf**	57
2-40	**stepper.tdf**	59
2-41	Moore State Machine Diagram	60
2-42	**moore1.tdf**	61
2-43	**moore1.gdf**	62
2-44	**moore2.tdf**	63
2-45	Mealy State Machine Diagram	64
2-46	**mealy.tdf**	65
2-47	**mealy.gdf**	66
2-48	**recover.tdf**	68
2-49	**macro1.tdf**	70
2-50	**macro2.tdf**	71
2-51	**macro.gdf**	72
2-52	**lpm_add1.tdf**	74
2-53	**lpm_add2.tdf**	75
2-54	**ss_def.tdf**	78
2-55	**ss_use.tdf**	78
2-56	**top1.tdf**	79
2-57	**top1.gdf**	79
2-58	**top2.tdf**	80
2-59	NODE Variables & SOFT Primitives	82
2-60	**boole3.tdf**	84
2-61	**boole3.rpt** & **boole1.rpt** Excerpts	85
2-62	**iter_add.tdf**	86
2-63	**condlog1.tdf**	87
2-64	**condlog2.tdf**	90

Tables

Table		Page
3-1	AHDL Symbols	94
3-2	Quoted & Unquoted Names	98
3-3	Arithmetic Operators and Comparators Used in Arithmetic Expressions	103
3-4	Logical Operators Used in Boolean Expressions	107
3-5	Arithmetic Operators Used in Boolean Expressions	109
3-6	Comparators Used in Boolean Expressions	111
3-7	Boolean Operator & Comparator Priorities	112
3-8	Global Signal Availability	118
3-9	MAX+PLUS II Flipflops & Latches	126
3-10	Primitive/Port Interconnections	127
3-11	Primitive/Port to Register Connections	128
3-12	MAX+PLUS II Megafunctions	129
3-13	Commonly Used Ports	135

MAX+PLUS II Fundamentals

This section describes the MAX+PLUS II manual and on-line help documentation and conventions. You should be familiar with this information before using MAX+PLUS II documentation.

- MAX+PLUS II Documentation xii
- Documentation Conventions xiv
- MAX+PLUS II Help Updates xviii
- Sample Files xix
- About ***MAX+PLUS II AHDL*** xx

MAX+PLUS II Documentation

MAX+PLUS II documentation is designed for the novice as well as for the experienced user. It includes manuals and extensive, illustrated Help.

MAX+PLUS II Documents

MAX+PLUS II printed documents contain the following information:

MAX+PLUS II Getting Started	Contains step-by-step instructions on how to install MAX+PLUS II hardware, software, and licenses on PCs and workstations. It also provides an overview of the entire MAX+PLUS II system, and a tutorial that takes you from design entry to device programming. In addition, it contains information about MAX+PLUS II command-line operation and Altera's support services.
MAX+PLUS II AHDL	Contains complete information on the Altera Hardware Description Language (AHDL), including a detailed *How to Use AHDL* section with many examples.
MAX+PLUS II VHDL	Provides information on how to use the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) with MAX+PLUS II, including a *How to Use MAX+PLUS II VHDL* section with many examples. (Available if you purchase PLSM-VHDL or PLSM-VHDLWS.)
MAX+PLUS II Help Poster	Provides handy and colorful descriptions of how to use on-line help in MAX+PLUS II.

MAX+PLUS II for workstations also includes the following Software Interface Guides:

- ***Cadence & MAX+PLUS II Software Interface Guide***
- ***Mentor Graphics & MAX+PLUS II Software Interface Guide***
- ***Synopsys & MAX+PLUS II Software Interface Guide***
- ***Viewlogic Powerview & MAX+PLUS II Software Interface Guide***

MAX+PLUS II Help

Your primary source of information on MAX+PLUS II is the complete on-line help. All of the information necessary to enter, compile, and verify a design and to program an Altera device is available in MAX+PLUS II Help.

Help also provides introductions to all MAX+PLUS II applications, guidelines for designing circuits with MAX+PLUS II, pin and logic cell numbers for each Altera device package, and summaries of other Altera documents, such as application notes, that can assist you with logic design.

How to Use MAX+PLUS II Documentation

How you use MAX+PLUS II documentation depends on your level of expertise and your approach to learning how to use a new tool.

If you are a novice user, you should take time to read the ***MAX+PLUS II Getting Started*** manual and complete the *MAX+PLUS II Tutorial.* Once you begin using MAX+PLUS II applications, you will find that the easy-to-use, extensive on-line help can quickly turn you into an expert MAX+PLUS II user. For basic information on using on-line help, refer to the *MAX+PLUS II Help Poster.* More detailed information on using Help is available in *MAX+PLUS II — A Perspective* in ***MAX+PLUS II Getting Started***.

If you are an experienced circuit designer or one who prefers to learn by experimenting, you will find the on-line help invaluable. Context-sensitive and menu-driven help give instant access to all MAX+PLUS II information.

Regardless of your level of expertise, you must follow the installation instructions provided in ***MAX+PLUS II Getting Started***. Before you install the MAX+PLUS II hardware and software, you should also read the **read.me** file provided on the first **Install** diskette or on the CD-ROM. If you are using the CD-ROM on a PC, the **read.me** file is located in the **\pc\maxplus2** directory; on a workstation, it is located in the **/cdrom** directory. Once you have installed MAX+PLUS II, you can open the **read.me** file through the Help menu in MAX+PLUS II.

Altera Applications Engineers are also available to answer your questions. For more information about Altera's technical support services, see *Appendix B: Altera Support Services* in ***MAX+PLUS II Getting Started***.

Documentation Conventions

MAX+PLUS II manuals and MAX+PLUS II Help use the following conventions to make it easy for you to find and interpret information.

Terminology

The following terminology is used throughout MAX+PLUS II Help and manuals:

Term:	**Meaning:**
Button 1	Left mouse button.
Button 2	Right button on a two-button mouse, or middle and right buttons on a three-button mouse.
"point to"	Indicates that you should move the mouse so that the pointer is over the specified item.
"press"	Indicates that you must hold down a mouse button or key.
"click"	Indicates a quick press and release of a mouse button.
"double-click"	Indicates two clicks in rapid succession.
"choose"	Indicates that you need to use a mouse or key combination to start an action. For example, when you use the mouse to choose a button, you point to the button and click Button 1. When you use the keyboard to choose a command, you press Alt and then type letters that are underlined in the menu bar and menu.
"select"	Indicates that you need to highlight text and/or objects or an option in a dialog box with a key combination or the mouse. A selection does not start an action. For example: Select the AND2 primitive, then choose **Delete** from the Edit menu.
"turn on"/"turn off"	Indicates that you must click Button 1 on a checkbox or choose a menu command to turn a function on or off.

Typographic Conventions

MAX+PLUS II documentation uses the following typographic conventions:

Visual Cue:	**Meaning:**
Bold Initial Capitals	Command names, dialog box titles, button names, and diskette names are shown in bold, with initial capital letters. For example: **Find Text** command, **Save As** dialog box, **Start** button, and **Install** diskette.
bold	Directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold. Examples: **\maxplus2** directory, **d:** drive, **chiptrip.gdf** file. These items are not case-sensitive in the Windows environment; however, they are case-sensitive in the workstation environment. MAX+PLUS II Help shows these items in the case appropriate to the workstation environment.
Initial Capitals	Keyboard keys, user-editable application window fields, and menu names are shown with initial capital letters. For example: Delete key, the Start Time field, the Options menu.
"Subheading Title"	Subheadings within a manual section are enclosed in quotation marks. In manuals, titles of help topics are also shown in quotation marks.
Italic Initial Capitals	Help categories, section titles in manuals, application note and brief names, checkbox options, and options in dialog boxes are shown in italics with initial capital letters. For example: *Text Editor Procedures*, the *Check Outputs* option, the *Directories* box in the **Open** dialog box.
italics	Variables are enclosed in angle brackets (< >) and shown in italics. For example: *<filename>*, *<project name>***.acf** file.
Bold Italics	Manual titles are shown in bold italics with initial capital letters. For example: ***MAX+PLUS II Getting Started***.

Visual Cue:	**Meaning:**
`Courier font`	Anything that must be typed exactly as it appears is shown in Courier. For example: `c:\max2work\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), and primitive and macrofunction names (e.g., `DFF` and `16CUDSLR`) are shown in Courier.
`Bold Courier font`	In syntax descriptions, bold Courier may be used to help distinguish literal text from variables.
1., 2., 3.,..., a., b., c.,..., *and* i., ii., iii.,...	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
👣	In MAX+PLUS II manuals, the feet show you where to go for more information on a particular topic.
☝	In MAX+PLUS II Help, the upward-pointing hand indicates that you can click Button 1 (the left mouse button) on any portion of the illustration that follows it to get help on that item. The mouse pointer changes to an upward-pointing hand when it is over a picture or word for which help is available.
Special symbols	Special symbols are used for these items: ↵ Enter key (manuals only) ⨼ Low-to-high transition ⨽ High-to-low transition

Key Combinations

Key combinations and sequences appear in the following format:

Format Cue:	Meaning:
Key1+Key2	A plus (+) symbol indicates that you must hold down the first key when you press the second key. For example: Ctrl+L means that you must hold down Ctrl while pressing L, then release both keys.
Key1,Key2	A comma (,) indicates that you must press the keys sequentially. For example: Alt,F1 means that you must press the Alt key and release it, then press the F1 key and release it.

Backus-Naur Form

The Backus-Naur Form (BNF) defines the syntax of the text file formats and message variables. BNF uses the following notation:

Characters:	Meaning:
::=	"is defined as"
<...>	Identifiers (i.e., variables)
[...]	Optional items
{ ... }	Repeated items (zero or more times)
... | ...	Indicates a choice between items
:n:n	Suffix indicates a range (e.g., *<name char>:1:8* means "from 1 to 8 name characters")
italics	Variables in syntax descriptions
`Courier font`	Literal text in syntax descriptions. Bold Courier font is sometimes used to help distinguish literal text from *italic variables* in syntax descriptions.

MAX+PLUS II Help Updates

MAX+PLUS II Help is updated whenever the MAX+PLUS II software is updated; therefore, the on-line information is always current.

☞ If you find a discrepancy between a MAX+PLUS II manual and the MAX+PLUS II on-line Help, you should rely on the MAX+PLUS II Help information.

You can get information on changes to MAX+PLUS II software and Help by choosing **New Features in this Release** (Help Menu) in MAX+PLUS II. Late-breaking news on Help and software is also available with the **READ.ME** command (Help menu).

Sample Files

A number of sample design files are copied to your hard disk when you install MAX+PLUS II. The installation procedure automatically creates subdirectories for these files.

The pathnames below are shown using the PC pathname convention of backslash (\) characters, but UNIX pathnames use forward slash (/) characters. On a UNIX workstation, the **/max2work** directory is a subdirectory of the **/usr** directory. Otherwise, the file and directory organization is identical.

- The **\max2work\chiptrip** directory contains all files for the **chiptrip** tutorial project that is described in ***MAX+PLUS II Getting Started***.
- The **\max2work\ahdl** directory contains all sample files used to illustrate AHDL features in MAX+PLUS II Help and in the ***MAX+PLUS II AHDL*** manual.
- The **\max2work\vhdl** directory contains all sample files used to illustrate VHDL features in MAX+PLUS II Help and in the ***MAX+PLUS II VHDL*** manual.
- The **\max2work\edif** directory contains all sample files used to illustrate EDIF features in MAX+PLUS II Help.

Go to "MAX+PLUS II File Organization" in the *MAX+PLUS II Installation* section of the ***MAX+PLUS II Getting Started*** manual for more information about MAX+PLUS II directory structure.

Go to the Altera-provided Software Interface Guide for your third-party environment for information on the directory structure and sample files installed for third-party interfaces to MAX+PLUS II.

About *MAX+PLUS II AHDL*

MAX+PLUS II AHDL contains the following sections:

Section 1: Introduction discusses basic AHDL features and the order in which AHDL statements appear in a Text Design File (**.tdf**). This section also summarizes essential rules and guidelines that will help you use AHDL effectively.

Section 2: How to Use AHDL describes how to develop a successful AHDL design. Altera recommends that you read the topics in this section sequentially.

Section 3: Elements describes the basic elements of a TDF. These elements are used in the behavioral statements described in *Section 4: Design Structure.*

Section 4: Design Structure describes all behavioral statements and sections used in AHDL.

Section 5: Style Guide provides guidelines for formatting TDFs to improve readability and avoid errors.

Glossary

Index

Section

1

Introduction

This section provides an overview of the Altera Hardware Description Language (AHDL). Some of the characteristics that distinguish AHDL as a hardware description language are discussed, and AHDL file structure is briefly described. This section also describes how to enter and process an AHDL file, and provides essential rules and guidelines for using AHDL effectively.

■ AHDL Design Entry 2
■ Text Design File Structure 4
■ MAX+PLUS II Text Editor 9
■ Compiling AHDL Text Design Files 13
■ Golden Rules 14

Go to MAX+PLUS II Help for complete and up-to-date information on AHDL.

AHDL Design Entry

The Altera Hardware Description Language (AHDL) is a high-level, modular language that is completely integrated into the MAX+PLUS II system. It is especially well suited for designing complex combinatorial logic, group operations, state machines, truth tables, and parameterized logic. You can use the MAX+PLUS II Text Editor or another text editor to create AHDL Text Design Files (**.tdf**). You can then compile TDFs to create output files for simulation, timing analysis, and device programming. In addition, the MAX+PLUS II Compiler can generate AHDL Text Design Export File (**.tdx**) and Text Design Output File (**.tdo**) that can be saved as TDFs and re-used as design files.

How Does AHDL Work?

AHDL statements and elements are powerful, versatile, and easy to use. You can create entire hierarchical projects with AHDL, or mix AHDL TDFs with other types of design files in a hierarchical design (called a "project" in MAX+PLUS II). In addition, AHDL TDFs can be parameterized.

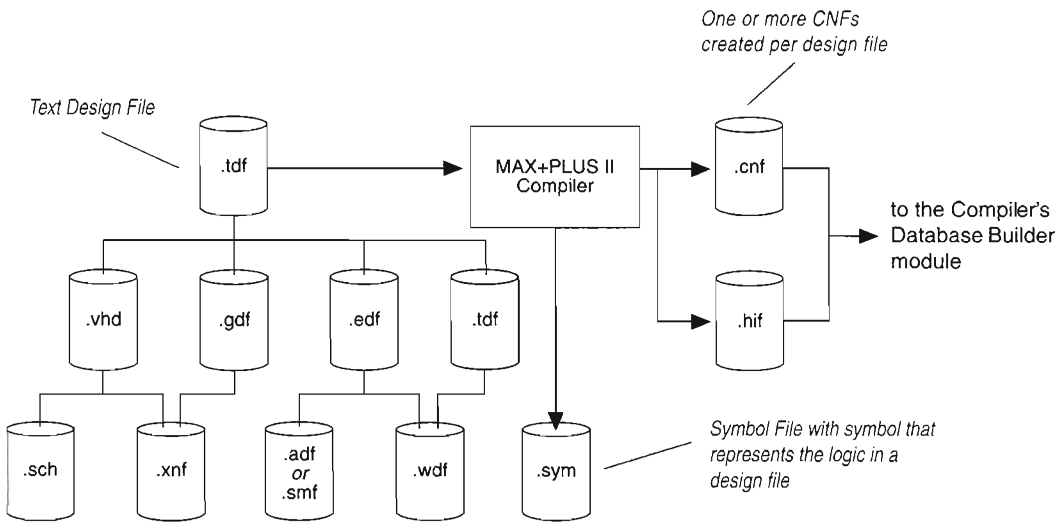
Although you can use any ASCII text editor to create AHDL designs, the MAX+PLUS II Text Editor allows you to take advantage of features available only in MAX+PLUS II while you enter, compile, and debug an AHDL design.

AHDL designs are easily incorporated into a design hierarchy. In the Text Editor, you can automatically create a symbol that represents a TDF and incorporate it into a Graphic Design File (**.gdf**). Similarly, you can incorporate custom functions, and over 300 Altera-provided megafunctions and macrofunctions—including Library of Parameterized Modules (LPM) functions—into any TDF by automatically creating an Include File (**.inc**) in the Text Editor. Altera provides Include Files for all mega- and macrofunctions shipped with MAX+PLUS II.

You can use Assign menu commands or an Assignment & Configuration File (**.acf**) to make resource and device assignments to allocate device resources for AHDL TDFs. You can also check AHDL syntax or perform a full compilation to debug and process your project. Any errors can be automatically located by the Message Processor and highlighted in the Text Editor window.

Figure 1-1 shows how TDFs can be integrated into the MAX+PLUS II system. A hierarchical project can contain TDFs, GDFs, EDIF Input Files (**.edf**), OrCAD Schematic Files (**.sch**), and VHDL Design Files (**.vhd**) at any level of the project hierarchy. In contrast, Waveform Design Files (**.wdf**), Altera Design Files (**.adf**), State Machine Files (**.smf**), and Xilinx Netlist Format Files (**.xnf**) can be used only at the lowest level of a project hierarchy, unless the entire project consists of a single WDF, ADF, SMF, or XNF File.

Figure 1-1. MAX+PLUS II & AHDL Design Entry



One or more CNFs created per design file
Text Design File
.tdf
MAX+PLUS II Compiler
.cnf
to the Compiler's Database Builder module
.vhd
.gdf
.edf
.tdf
.hif
.sch
.xnf
.adf or .smf
.wdf
.sym
Symbol File with symbol that represents the logic in a design file

Text Design File Structure

A Text Design File (**.tdf**) is an ASCII text file, written in AHDL, that can be entered with the MAX+PLUS II Text Editor or any standard text editor.

Text Design File Sections

The following AHDL sections and statements are listed in the order in which they appear in a TDF. Figure 1-2 also shows a TDF and the AHDL sections and statements that it can contain, and how Include Files and files in a project hierarchy can be used with AHDL.

- (Optional) Title Statement — provides comments for the Report File (**.rpt**) generated by the MAX+PLUS II Compiler.
- (Optional) Include Statement — specifies an Include File that replaces the Include Statement in the TDF.
- (Optional) Constant Statement — specifies a symbolic name that can be substituted for a constant.
- (Optional) Define Statement — defines an evaluated function, which is a mathematical function that returns a value that is based on optional arguments.
- (Optional) Parameters Statement — declares one or more parameters that control the implementation of a parameterized megafunction or macrofunction. A default value can be specified for each parameter.
- (Optional) Function Prototype Statement — declares the ports of a logic function and the order in which those ports must be declared in an in-line reference. In parameterized functions, it also declares the parameters used by the function.
- (Optional) Options Statement — sets the default bit-ordering for the file, or for the project if the file is a top-level TDF.
- (Optional) Assert Statement — allows you to test the validity of an arbitrary expression and report the results.
- (Required) Subdesign Section — declares the input, output, and bidirectional ports of an AHDL TDF.

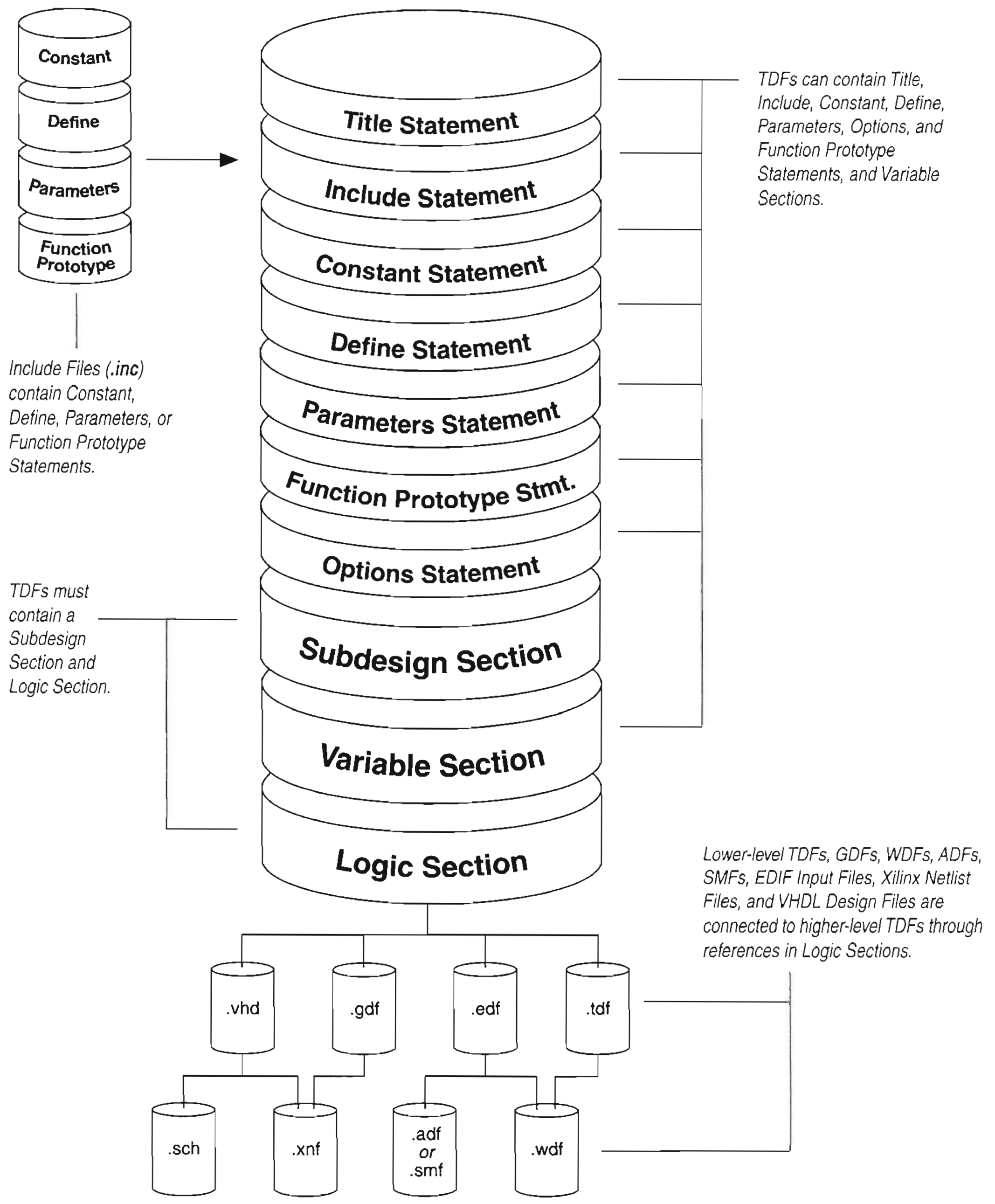
- (Optional) Variable Section — declares variables that represent and hold internal information. Variables can be declared for ordinary or tri-state nodes, primitives, megafunctions, macrofunctions, and state machines. Variables can also be generated conditionally with an If Generate Statement. The Variable Section can include any of the following constructs:

  - Instance Declaration
  - Node Declaration
  - Register Declaration
  - State Machine Declaration
  - Machine Alias Declaration
  - If Generate Statement

- (Required) Logic Section — defines the logical operations of the file. The Logic Section can define logic with Boolean equations, conditional logic, and truth tables. It also supports conditional and iterative logic generation, and the capability to test the validity of an arbitrary expression and report the results. The Logic Section can include any of the following constructs:

  - Defaults Statement
  - Assert Statement
  - Boolean Equations
  - Boolean Control Equations
  - Case Statement
  - For Generate Statement
  - If Generate Statement
  - If Then Statement
  - In-Line Logic Function Reference
  - Truth Table Statement

Figure 1-2. AHDL Text Design File Structure



Constant
Define
Parameters
Function Prototype
Include Files (**.inc**) contain Constant, Define, Parameters, or Function Prototype Statements.
Title Statement
Include Statement
Constant Statement
Define Statement
Parameters Statement
Function Prototype Stmt.
Options Statement
Subdesign Section
Variable Section
Logic Section
TDFs can contain Title, Include, Constant, Define, Parameters, Options, and Function Prototype Statements, and Variable Sections.
TDFs must contain a Subdesign Section and Logic Section.
Lower-level TDFs, GDFs, WDFs, ADFs, SMFs, EDIF Input Files, Xilinx Netlist Files, and VHDL Design Files are connected to higher-level TDFs through references in Logic Sections.
.vhd
.gdf
.edf
.tdf
.sch
.xnf
.adf or .smf
.wdf

AHDL is a concurrent language. All behavior specified in the Logic Section of a TDF is evaluated at the same time rather than sequentially. Equations that assign multiple values to the same AHDL node or variable are logically connected (ORed if the node or variable is active high, ANDed if it is active low). See "Defaults Statement" on page 173 in *Design Structure* and "Using Default Values for Variables" on page 39 in *How to Use AHDL* for more information.

A TDF must contain a Subdesign Section and a Logic Section. It can optionally contain a single Variable Section, Options Statement, Title Statement, and Defaults Statement, and one or more Include, Constant, Define, and Function Prototype Statements.

The last entries in a TDF are the Subdesign Section, Variable Section (optional), and Logic Section, which together contain the behavioral description of the TDF.

See *Design Structure* on page 139 for more information about AHDL statements. Go to the Backus-Naur Form (BNF) syntax descriptions of each AHDL section in MAX+PLUS II Help using **Search for Help on** (Help menu).

Files in a Project Hierarchy

Files in a project hierarchy can be TDFs, GDFs, WDFs, ADFs, SMFs, EDIF Input Files, OrCAD Schematic Files, AHDL Design Files, or Xilinx Netlist Format Files. Each logic function is connected through its input and output ports to the design file at the next higher level. For more information, see "Implementing a Hierarchical Project" on page 69 in *How to Use AHDL*.

Include Files

An Include File is an ASCII text file (with the extension **.inc**) that can be imported into a TDF with an AHDL Include Statement. The contents of the Include File replace the Include Statement that calls the file. Include Files can contain Function Prototype, Constant, Define, and Parameters Statements.

Each Altera-provided megafunction and macrofunction has an Include File that contains its Function Prototype:

- The Include Files for megafunctions, including LPM functions, are located in the **maxplus2\max2lib\mega_lpm** directory created during installation.
- The Include Files for macrofunctions are located in the **\maxplus2\max2inc** directory created during installation.

  On UNIX workstations, the **maxplus2** directory is a subdirectory of the **/usr** directory.

When you have a design file open in a Graphic, Text, or Waveform Editor window, you can choose **Create Default Include File** (File menu) to automatically generate an Include File that contains a default Function Prototype for the design file. You can also manually create an Include File with the MAX+PLUS II Text Editor or another standard text editor.

Go to "Creating a Default Include File" in MAX+PLUS II Help for more information.

MAX+PLUS II Text Editor

AHDL Text Design Files (with the extension **.tdf**) can be entered with the MAX+PLUS II Text Editor or any other text editor that follows standard ASCII character conventions. If your text editor has both document and non-document modes, you must use non-document mode, i.e., save the file as text only.

The MAX+PLUS II Text Editor allows you to take advantage of the following unique MAX+PLUS II features while you enter, compile, and debug an AHDL TDF:

- AHDL templates and examples
- AHDL context-sensitive help
- Syntax coloring
- Resource and device assignments
- Error location

Go to MAX+PLUS II Text Editor Help for more information on using the MAX+PLUS II Text Editor.

AHDL Templates & Examples

MAX+PLUS II provides both AHDL templates and AHDL examples to make design entry easier for you.

- *AHDL Templates*—You can insert AHDL templates into your TDF, then replace placeholder variables in the templates with your own identifiers and expressions.

- *AHDL Examples*—MAX+PLUS II provides a number of AHDL examples that are used to illustrate AHDL features in the *How to Use AHDL* section of this manual. These examples are available in the **\max2work\ahdl** directory (a subdirectory of the **/usr** directory on a UNIX workstation), and in MAX+PLUS II AHDL Help. You can customize these examples to fit your needs.

Go to "Inserting an AHDL Template" on page 22 and "AHDL Examples" on page 24 in *How to Use AHDL* for more information on using AHDL templates and examples.

AHDL Context-Sensitive Help

If the current file has the extension **.tdf**, the MAX+PLUS II Text Editor provides context-sensitive help on all AHDL keywords, operators, comparators, and punctuation, as well as on all MAX+PLUS II-provided primitives, megafunctions, and macrofunctions.

When you choose the context-sensitive Help button ( ) from the toolbar or press Shift+F1, the pointer turns into a question mark pointer. You can then click Button 1 on a word or character in an AHDL. If context-sensitive help is available for that item, the relevant information is displayed. Otherwise, Help shows a list of all items for which context-sensitive help is available.

Syntax Coloring

The MAX+PLUS II Text Editor allows you to view various elements of a TDF in different colors. Syntax coloring can help you improve file readability and accuracy. For instance, it can help you identify misspelled keywords and sections of files that have been commented out by mistake.

To turn the syntax coloring feature on or off:

✓ Choose **Syntax Coloring** from the Options menu.

You can also use the **Color Palette** command (Options menu) to customize the assigned colors for comments, illegal characters, megafunctions and macrofunctions, reserved identifiers and keywords, strings, and text.

Go to "Using Syntax Coloring in Text Files" and "Changing Colors in MAX+PLUS II" in MAX+PLUS II Help for more information.

Resource & Device Assignments

You can specify resource assignments—i.e., pin, logic cell, I/O cell, embedded cell, Logic Array Block (LAB), Embedded Array Block (EAB), row, column, chip, clique, logic option, connected pin, and timing assignments—as well as device assignments for a TDF to guide logic synthesis and fitting for your project. You can choose to have the Compiler automatically fit your project into the best combination of devices from a target device family and assign the resources within them. You can also select a node or pin name in the Text Editor and enter a specific assignment for it with the **Pin/Location/Chip**, **Clique**, **Logic Options**, **Timing Requirements**, **Connected Pins**, and other commands on the Assign menu. (Assign menu commands are also available in all other MAX+PLUS II applications.) You can also enter assignments with the Floorplan Editor or by editing the Assignment & Configuration File (**.acf**) in the Text Editor.

For example, you can assign a logic synthesis style that tailors logic synthesis to your needs, and specify precisely how to divide a large project into multiple devices, and make timing assignments to achieve speed performance on individual logic functions.

MAX+PLUS II provides the *Use LPM for AHDL Operators* logic option, which allows the Compiler to substitute `lpm_add_sub` and `lpm_compare` functions automatically for the following operators and comparators:

Operator/ Comparator:	**Description:**
+	addition
-	subtraction
==	numeric equality
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Go to MAX+PLUS II Help for more information on making resource and device assignments.

Error Location

MAX+PLUS II interactively reports and locates errors that occur as you process your project. As you compile a project, a Message Processor window opens and lists error, information, and warning messages for the project. You can locate the source(s) of a message by double-clicking Button 1 on the message text. MAX+PLUS II then automatically opens the design file that contains the source of the message, regardless of its location in the project hierarchy or the type of application that created it. If the error occurs in a TDF, MAX+PLUS II opens a Text Editor window and highlights the text that caused the error. You can also locate errors in the assignment floorplan for the project in the MAX+PLUS II Floorplan Editor.

Go to "Locating the Source of a Message" using **Search for Help on** (Help menu) for more information on error location.

Compiling AHDL Text Design Files

MAX+PLUS II automatically compiles AHDL TDFs. When you have finished entering a TDF, you can check its syntax with the **Project Save & Check** command (File menu), or compile all files in a project with the **Project Save & Compile** command (File menu). If you wish to generate an AHDL Text Design Output File (**.tdo**) for a compiled project, you can turn on the Compiler's **Generate AHDL TDO File** command (Processing menu) before compiling the project. After the project has compiled successfully, you can perform optional simulation and timing analysis, and then program one or more devices.

For complete on-line information on the MAX+PLUS II Text Editor, go to MAX+PLUS II Help. For detailed instructions and suggestions on how to use AHDL sections and statements to develop a project, go to *How to Use AHDL* on page 17. For a step-by-step tutorial on how to enter, compile, simulate, and program a project that includes AHDL TDFs with MAX+PLUS II, go to *MAX+PLUS II Tutorial* in the ***MAX+PLUS II Getting Started*** manual.

Golden Rules

The following golden rules will help you use AHDL effectively:

- Use the Text Editor's **Syntax Coloring** command (Options menu) to help you identify typographical errors and different sections of AHDL code.
- Follow the formatting and naming guidelines described in the *Style Guide* on page 187 to improve readability and avoid errors.
- Although AHDL is not case-sensitive, Altera recommends that you follow the capitalization rules in the *Style Guide* to improve readability.
- Use constants and evaluated functions, which are created in Constant and Define Statements, to improve readability and to avoid errors.
- You do not need to create AHDL Function Prototypes for primitives. However, you can redefine primitives with a Function Prototype Statement to change the calling order of inputs in your TDF.
- Do not use nested If Then Statements when a Case Statement can be used instead.
- When you use the MAX+PLUS II Text Editor to create a TDF, each line can be up to 255 characters long. However, the ideal line length is the number of characters your screen can accommodate. Press Enter to end a line.
- You can start new lines wherever white space (i.e., blank lines, tabs, and spaces) is allowed, without any effect on meaning. White space is allowed between major AHDL constructs.
- Keywords, names, and numbers must be separated by the appropriate symbols or operators, and one or more spaces.
- Comments must be enclosed in percent symbols (`%`). A comment can include any character except `%`, since the MAX+PLUS II Compiler ignores everything between the percent symbols. Comments enclosed in percent symbols cannot be nested .

VHDL-style comments (`--`) can be nested within `%`-style comments. If you use VHDL-style comments for documentation-type comments, you can use the `%`-style comments to exclude sections of code from compilation (i.e., "comment out" sections of code).

- When connecting a primitive to another primitive, you must use only "legal" interconnections; not all primitives may connect to all other primitives. For a list of legal interconnections for primitives, see "Primitive/Port Interconnections" on page 127 in *Elements*.
- Do not create your own cross-coupled structures; use only the `expdff`, `explatch`, `inpltch`, `nandltch`, and `norltch` macrofunctions provided with MAX+PLUS II. (These macrofunctions are not optimized for FLEX 8000 and FLEX 10K architectures.) Avoid tying multiple instances of `expdff`, `explatch`, `inpltch`, `nandltch`, and `norltch` macrofunctions together. Multiple instances of these macrofunctions should always be separated by `LCELL` primitives.

General Design Entry Golden Rules

- You should use Altera-provided primitives and AHDL logical operators rather than the equivalent LPM functions in most cases: they are much more convenient to instantiate. For example, 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.
- Use LPM megafunctions rather than equivalent old-style macrofunctions in most cases: the former are more convenient to instantiate and easier to modify if your design changes.
- Use the Design Doctor to check the reliability of your project logic during compilation. Go to "Project Reliability Guidelines" in MAX+PLUS II Help for information on how to create reliable projects.
- Do not attempt to create your own logic functions to implement RAM or ROM: use Altera-provided megafunctions instead.
- When you start a new design file, specify the target device family with **Device** (Assign menu) right away, so that you can take advantage of

family-specific macrofunctions. If you do not specify a device family, the family for the current project is assumed.

General MAX+PLUS II Golden Rules

- When you start work on a new design file, name it as the current project with **Project Set Project to Current File** or **Project Name** (File menu) right away so that you can compile it easily. You can always change the project name later.
- Use the built-in hierarchy traversal features in MAX+PLUS II to move between design files for the current hierarchy tree. To open a lower-level file in a hierarchy, open the top-level file and then use the Hierarchy Display window or **Hierarchy Down** (File menu) to open the lower-level files. If you choose **Open** or **Retrieve** (File menu) to open a lower-level file, that file is considered to be the top of a different hierarchy tree, and resource, device, and probe assignments that you enter are saved only for that hierarchy, not for the project.
- When you create an editable ancillary file for a project, the icon for the file will appear in the Hierarchy Display if you use the same filename as the project.
- Don't edit any MAX+PLUS II system files, including HIFs, TOK files, **maxplus2.idx** files, or the **maxplus2.ini** file.
- Use the **Save As** command (File menu) if you wish to rename a design file or an ancillary file. Do not rename design files from outside of the MAX+PLUS II system (e.g., from DOS or with the Windows File Manager).
- When you have completed a project, use **Project Archive** (File menu) to save a complete backup copy of all project files that will not be affected by future edits or deletions.

Section

2

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 LCELL & 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.

Introduction

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.

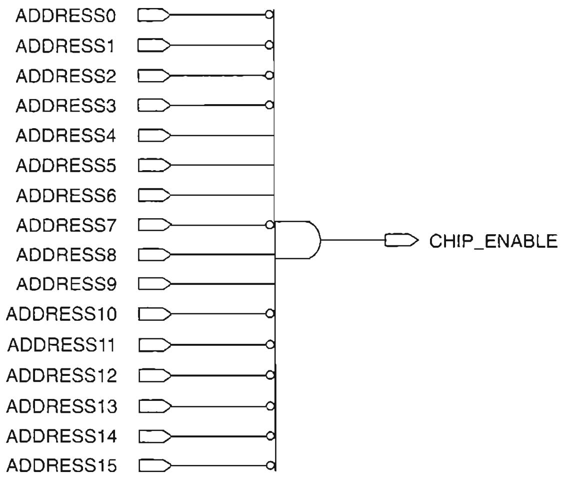
Figure 2-1. decode1.tdf

```
SUBDESIGN decode1
(
   address[15..0]         :INPUT;
   chip_enable            :OUTPUT;
)
BEGIN
   chip_enable = (address[15..0] == 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`.

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

```
CONSTANT IO_ADDRESS = H"0370";

SUBDESIGN decode2
(
   a[15..0]       : INPUT;
   ce             : OUTPUT;
)
BEGIN
   ce = (a[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`:

```
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`:

```
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. strcmp.tdf

```
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[MAX(WIDTH,0)..0] : INPUT;
   dataB[MAX(WIDTH,0)..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.

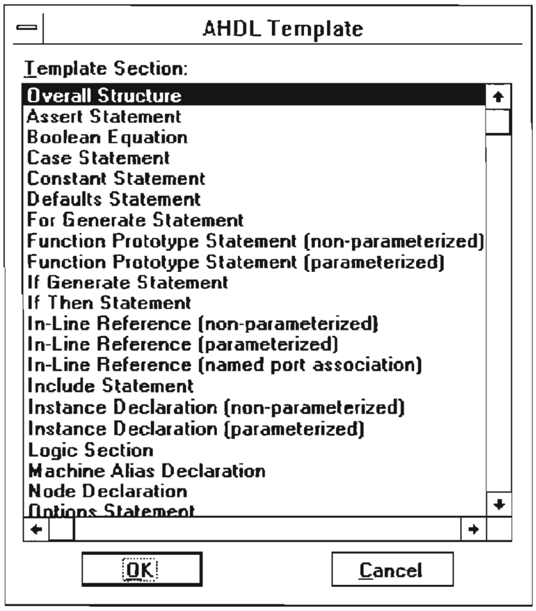
A single template is available for the overall AHDL file structure. This template, called "Overall Structure," lists all AHDL constructs in separate comment lines in the order in which they appear in a TDF. The syntax of these sections and statements is not included; you must replace the comment line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

Figure 2-6. AHDL Template Dialog Box



Overall Structure
Assert Statement
Boolean Equation
Case Statement
Constant Statement
Defaults Statement
For Generate Statement
Function Prototype Statement (non-parameterized)
Function Prototype Statement (parameterized)
If Generate Statement
If Then Statement
In-Line Reference (non-parameterized)
In-Line Reference (parameterized)
In-Line Reference (named port association)
Include Statement
Instance Declaration (non-parameterized)
Instance Declaration (parameterized)
Logic Section
Machine Alias Declaration
Node Declaration
Options Statement

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

4. Choose **OK**.

1. Shortcuts are available for this command. Go to "AHDL Template command" in MAX+PLUS II Help for details.

2. All AHDL templates are also available in the ASCII **ahdl.tpl** file, which is automatically installed in the **\maxplus2** directory (a subdirectory of the **/usr** directory on a UNIX workstation).

Once you have inserted a template into your TDF, you must replace all variables in the template with your own logic. Figure 2-7 shows the Defaults Statement template.

Figure 2-7. Defaults Statement Template

```
DEFAULTS
   __node_name = __constant_value;
END DEFAULTS;
```

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 **boole1.tdf** file shown in Figure 2-8 shows two simple Boolean expressions that represent two logic gates.

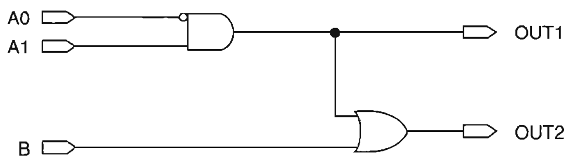
Figure 2-8. boole1.tdf

```
SUBDESIGN boole1
(
   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 **boole1.tdf**.

Figure 2-9. boole1.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_equals_2    : NODE;
BEGIN
   a_equals_2 = a1 & !a0;
   out = a_equals_2 # b;
END;
```

This file declares the node `a_equals_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:

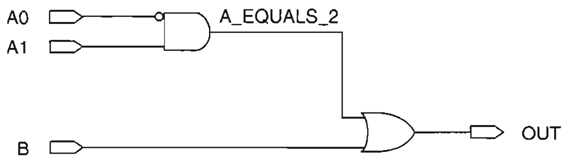
- Multiple assignments to nodes of type `NODE` tie the signals together by wired-AND or wired-OR functions. The default values for variables declared in Defaults Statements determine the behavior: a `VCC` default produces a wired-AND function; a `GND` default produces a wired-OR function.

- 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[]`; similarly, `b[5..4][3..2]` can be represented by `b[][]`.

The **group1.tdf** file shown in Figure 2-12 shows simple Boolean expressions that define multiple groups.

Figure 2-12. group1.tdf

```
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[5..4][3..2]                    : OUTPUT;
)
BEGIN
   d[] = a[] + B"10";
   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 (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 q8, d1 to q7, and d0 to q6:

  ```
  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:

  ```
  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] = 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

```
SUBDESIGN priority
(
   low, middle, high          : INPUT;
   highest_level[1..0]        : OUTPUT;
)
BEGIN
   IF high THEN
      highest_level[] = 3;
   ELSIF middle THEN
      highest_level[] = 2;
   ELSIF low THEN
      highest_level[] = 1;
   ELSE
      highest_level[] = 0;
   END IF;
END;
```

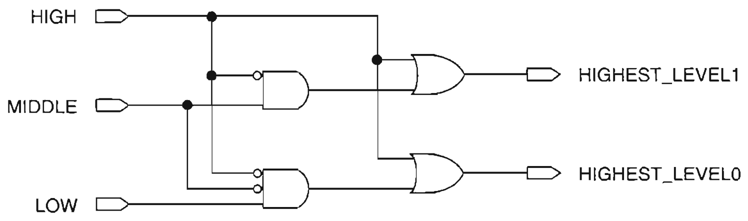
In this example, the inputs `high`, `middle`, and `low` are evaluated to determine whether they are driven by `VCC`. The If Then Statement activates the equations that follow the active `IF` or `ELSE` clause, e.g., if `high` is driven by `VCC`, `highest_level[]` is 3.

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 `ELSIF` 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.

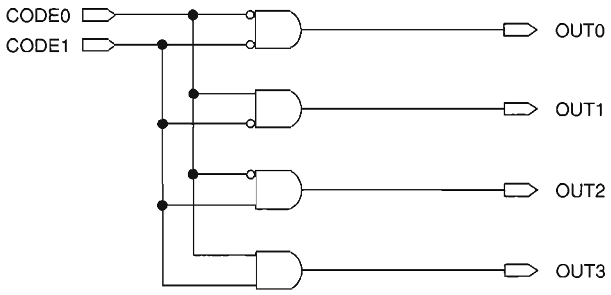
Figure 2-15. decoder.tdf

```
SUBDESIGN decoder
(
   code[1..0]    : INPUT;
   out[3..0]     : OUTPUT;
)
BEGIN
   CASE code[] IS
      WHEN 0 => out[] = B"0001";
      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 => 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;

ELSIF b THEN
   c = e;

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.

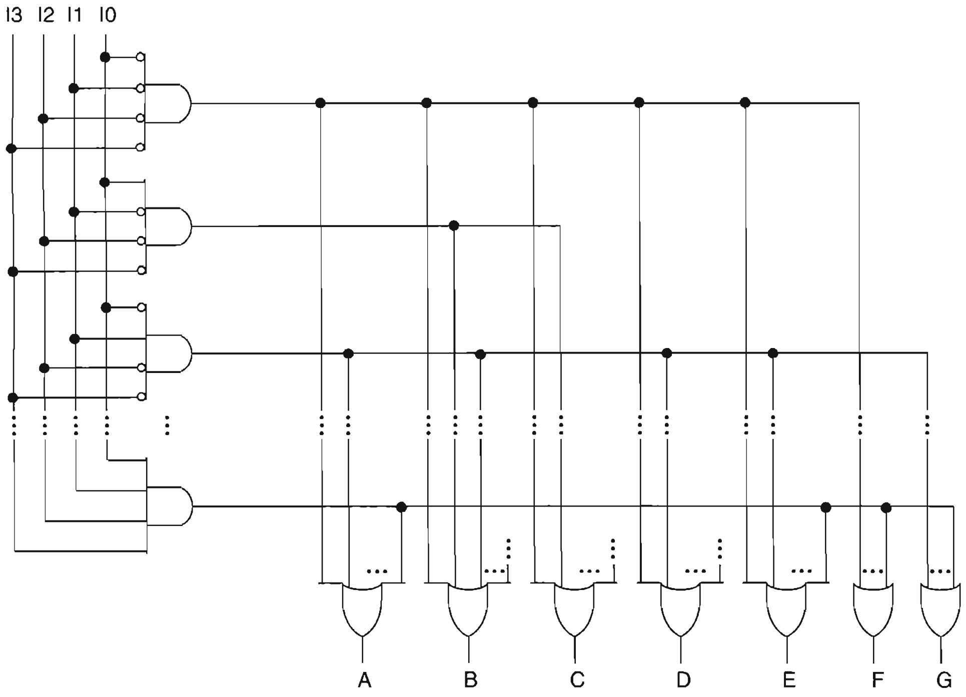
Figure 2-17. 7segment.tdf

```
%    -a-                                %
% f|    |b                              %
%    -g-                                %
% e|    |c                              %
%    -d-                                %
%                                       %
% 0 1 2 3 4 5 6 7 8 9 A b C d E F       %
%                                       %

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, 1, 0;
      H"1"       =>   0, 1, 1, 0, 0, 0, 0;
      H"2"       =>   1, 1, 0, 1, 1, 0, 1;
      H"3"       =>   1, 1, 1, 1, 0, 0, 1;
      H"4"       =>   0, 1, 1, 0, 0, 1, 1;
      H"5"       =>   1, 0, 1, 1, 0, 1, 1;
      H"6"       =>   1, 0, 1, 1, 1, 1, 1;
      H"7"       =>   1, 1, 1, 0, 0, 0, 0;
      H"8"       =>   1, 1, 1, 1, 1, 1, 1;
      H"9"       =>   1, 1, 1, 1, 0, 1, 1;
      H"A"       =>   1, 1, 1, 0, 1, 1, 1;
      H"B"       =>   0, 0, 1, 1, 1, 1, 1;
      H"C"       =>   1, 0, 0, 1, 1, 1, 0;
      H"D"       =>   0, 1, 1, 1, 1, 0, 1;
      H"E"       =>   1, 0, 0, 1, 1, 1, 1;
      H"F"       =>   1, 0, 0, 0, 1, 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 GDF that is equivalent to **7segment.tdf**.

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

```
SUBDESIGN 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,       B"00XXXXXXXXXXXXXX"      => 1,   0,     0,       B"00";
      1,       B"100XXXXXXXXXXXXX"      => 0,   1,     0,       B"00";
      0,       B"0000001010101110"      => 0,   0,     1,       B"00";
      0,       B"0000001011011110"      => 0,   0,     0,       B"01";
      0,       B"0000001101110000"      => 0,   0,     0,       B"10";
   END TABLE;
END;
```

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 **decode4.tdf** file shown in Figure 2-20 uses the lpm_decode function to achieve the same functionality as **decode1.tdf** (described in "Using Numbers" on page 18).

Figure 2-20. decode4.tdf

```
INCLUDE "lpm_decode.inc";

SUBDESIGN decode4
(
   address[15..0]    : INPUT;
   chip_enable       : OUTPUT;
)
BEGIN
   chip_enable = lpm_decode(.data[]=address[])
      WITH (LPM_WIDTH=16, LPM_DECODES=2^10)
      RETURNS (.eq[H"0370"]);
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;
)
BEGIN
   DEFAULTS
      ascii_code[] = B"00111111"; % ASCII question mark "?" %
   END DEFAULTS;

   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

```
SUBDESIGN default2
(
   a, b, c                          : INPUT;
   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;
```

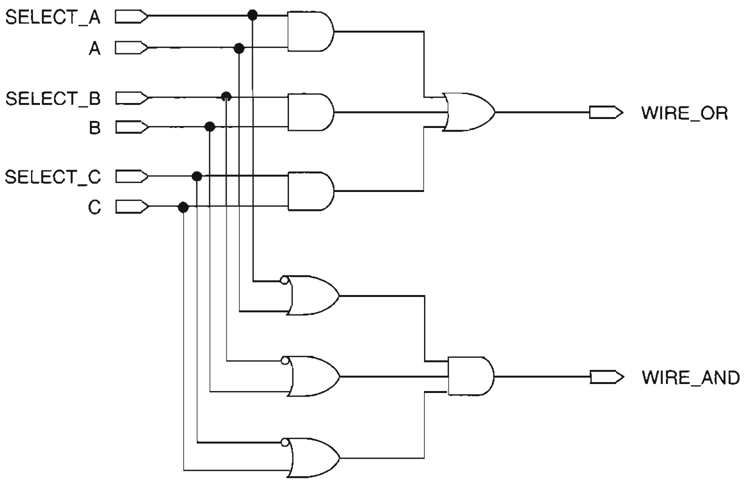
In this example, `wire_or` is set to the values of `a`, `b`, or `c`, depending on the values of the `select_a`, `select_b`, and `select_c` signals. If none of these signals is VCC, then `wire_or` defaults to GND.

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**.

Figure 2-23. default2.gdf



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

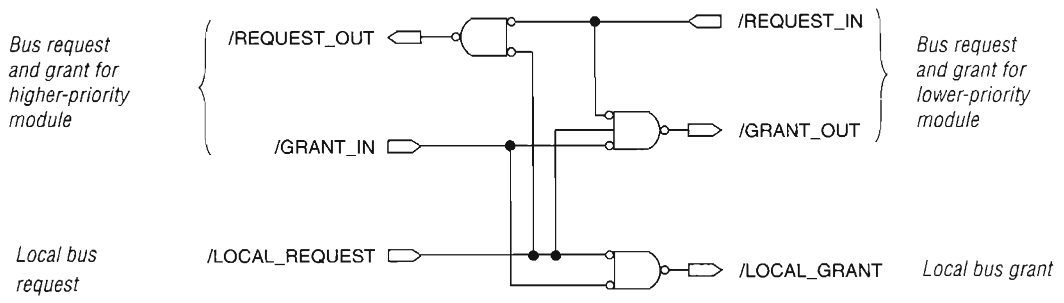
```
SUBDESIGN daisy
(
   /local_request   : INPUT;
   /local_grant     : OUTPUT;
   /request_in      : INPUT;       % from lower priority  %
   /request_out     : OUTPUT;      % to higher priority   %
   /grant_in        : INPUT;       % from higher priority %
   /grant_out       : OUTPUT;      % to 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**.

Figure 2-25. daisy.gdf



Go to the following topics for more information:

"Naming a Boolean Operator or Comparator" on page 84 in this section
"Using Default Values for Variables" on page 39 in this section

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.

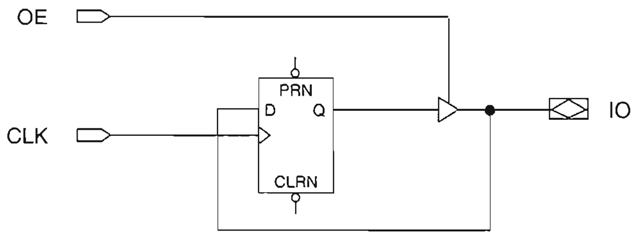
Figure 2-26. bus_reg2.tdf

```
SUBDESIGN bus_reg2
(
   clk   : INPUT;
   oe    : INPUT;
   io    : BIDIR;
)
BEGIN
   dff_out = DFF(io, clk, ,);
   io = TRI(dff_out, oe);
END;
```

The bidirectional `io` signal, driven by `TRI`, is used as the `d` input to a D flipflop (`DFF`). 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

```
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;
```

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
(
   in[3..1], oe[3..1] : INPUT;
   out1               : OUTPUT;
)

VARIABLE
   tnode : TRI_STATE_NODE;
BEGIN
   tnode = TRI(in1, oe1);
   tnode = TRI(in2, oe2);
   tnode = TRI(in3, oe3);
   out1 = tnode;
END;
```

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` 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 flipflops. 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 `d` inputs onto the `q` 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]           : DFFE;
BEGIN
   ff[].clk = clk;
   ff[].ena = load;
   ff[].d = d[];
   q[] = ff[].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[].d` to `ff[].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[].j` and `ff[].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.

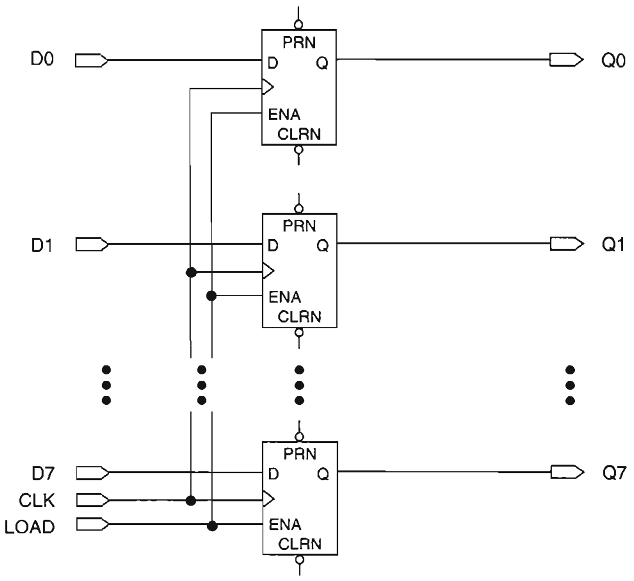
The **lpm_reg.tdf** file shown in Figure 2-31 uses an in-line reference to implement an instance of the `lpm_dff` function that has the same functionality as the **bur_reg.tdf** file.

Figure 2-31. lpm_reg.tdf

```
INCLUDE "lpm_dff.inc";
SUBDESIGN lpm_reg
(
   clk, load, d[7..0] : INPUT;
   q[7..0]            : OUTPUT;
)
BEGIN
   q[] = lpm_dff (.clock=clk, .enable=load, .data[]=d[])
      WITH (LPM_WIDTH=8)
   RETURNS (.q[]);
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

```
SUBDESIGN reg_out
(
   clk, load, d[7..0]     : INPUT;
   q[7..0]                : OUTPUT;
)
VARIABLE
   q[7..0]                : DFFE; % also declared as outputs %
BEGIN
   q[].clk = clk;
   q[].ena = load;
   q[] = d[];
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**

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

Figure 2-34. ahdlcnt.tdf

```
SUBDESIGN ahdlcnt
(
   clk, load, ena, clr, d[15..0]    : INPUT;
   q[15..0]                         : OUTPUT;
)
VARIABLE
   count[15..0]                     : DFF;
BEGIN
   count[].clk = clk;
   count[].clrn = !clr;

   IF load THEN
      count[].d = d[];
   ELSIF ena THEN
      count[].d = count[].q + 1;
   ELSE
      count[].d = count[].q;
   END IF;

   q[] = count[];
END;
```

In this file, 16 D flipflops are declared in the Variable Section and assigned the names count0 through count15. 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[]).

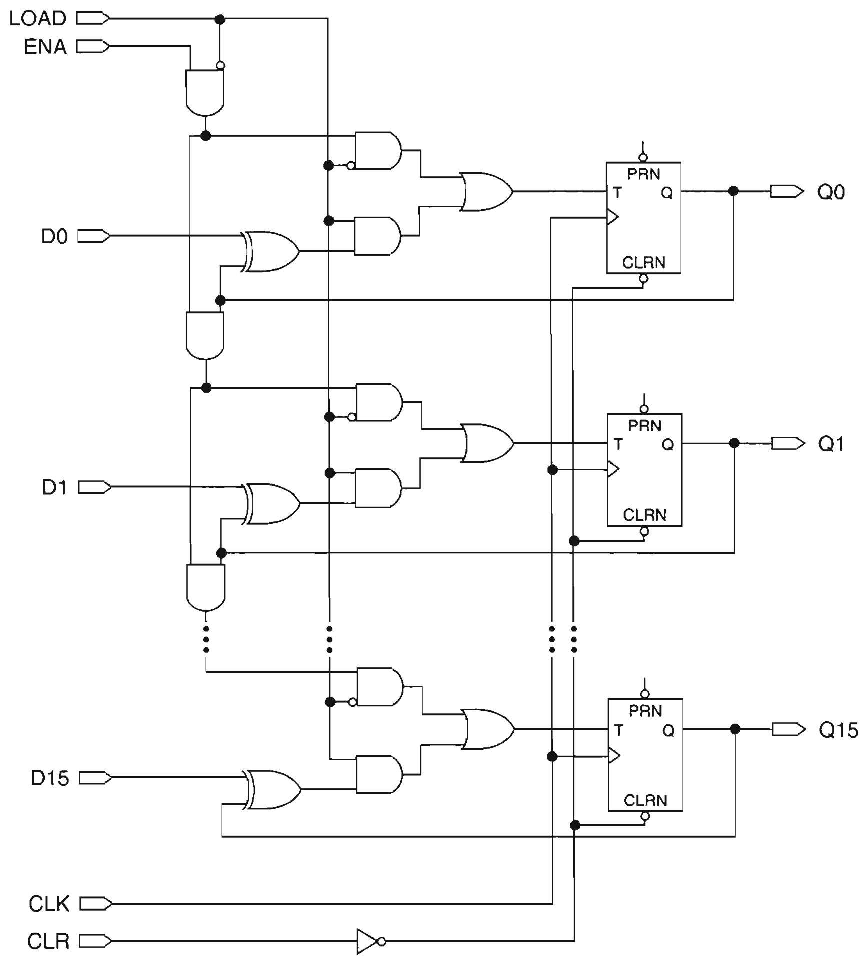
The **lpm_cnt.tdf** file shown in Figure 2-35 uses the lpm_counter function to implement the same functionality as **ahdlcnt.tdf**:

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_cntr: lpm_counter WITH (LPM_WIDTH=16);

BEGIN
   my_cntr.clock  = clk;
   my_cntr.aload  = load;
   my_cntr.cnt_en = ena;
   my_cntr.aclr   = clr;
   my_cntr.data[] = d[];
   q[] = my_cntr.q[];
END;
```

Figure 2-36 shows a GDF that is equivalent to **ahdlcnt.tdf** and **lpm_cnt.tdf**.

Figure 2-36. count.gdf



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 TDFs 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

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

   CASE ss IS
      WHEN s0 =>
         q = GND;

         IF d THEN
            ss = s1;
         END IF;
      WHEN s1 =>
         q = VCC;

         IF !d THEN
            ss = s0;
         END IF;
   END CASE;
END;
```

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.

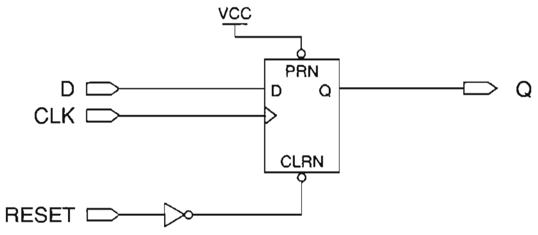
State machine transitions define the conditions under which the state machine changes to a new state. You must conditionally assign the states within a single behavioral construct to specify state machine transitions. Case or Truth Table Statements are recommended for this purpose. For example, in **simple.tdf**, 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 !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])
      WITH STATES  (
         s0 = B"0001",
         s1 = B"0010",
         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.

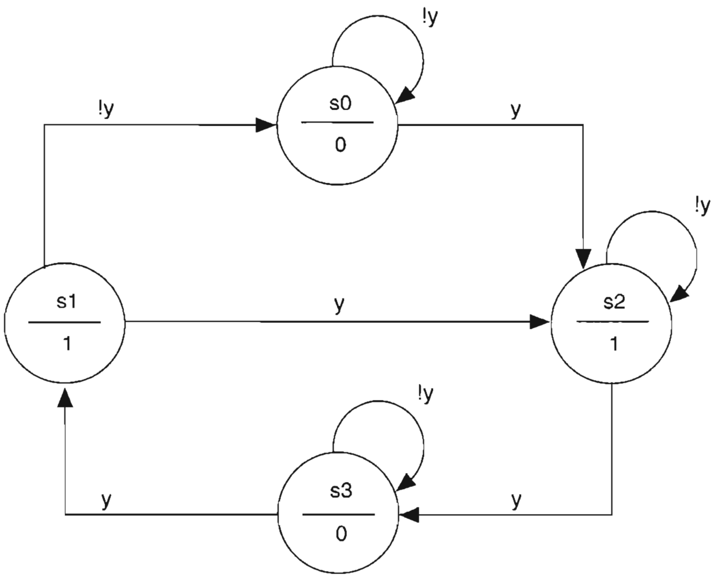
Go to "Truth Table Statement" on page 183 in *Design Structure* for more information.

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



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

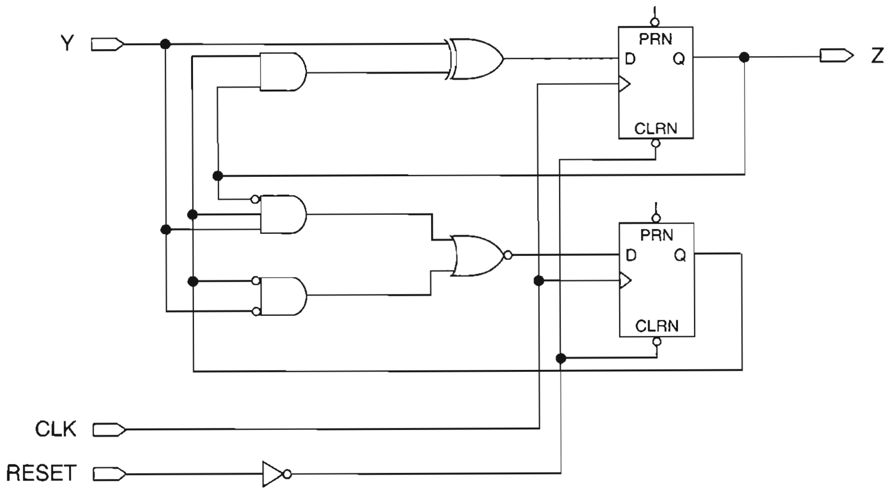
Figure 2-42. moore1.tdf

```
SUBDESIGN moore1
(
   clk      : INPUT;
   reset    : INPUT;
   y        : INPUT;
   z        : OUTPUT;
)
VARIABLE
                    %   current  current %
                    %   state    output  %
   ss: MACHINE OF BITS (z)
         WITH STATES   (s0 =      0,
                         s1 =      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**.

Figure 2-43. moore1.gdf



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

```
SUBDESIGN moore2
(
   clk   : INPUT;
   reset : INPUT;
   y     : INPUT;
   z     : OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);
   zd: NODE;
BEGIN
   ss.clk   = clk;
   ss.reset = reset;

   z = DFF(zd, clk, VCC, VCC);

   TABLE
   %  current  current      next     next   %
   %  state    input        state    output %
      ss,      y      =>    ss,      zd;

      s0,      0      =>    s0,      0;
      s0,      1      =>    s2,      1;
      s1,      0      =>    s0,      0;
      s1,      1      =>    s2,      1;
      s2,      0      =>    s2,      1;
      s2,      1      =>    s3,      0;
      s3,      0      =>    s3,      0;
      s3,      1      =>    s1,      1;
   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 (DFF)—called with an in-line reference—to synchronize the outputs with the Clock.

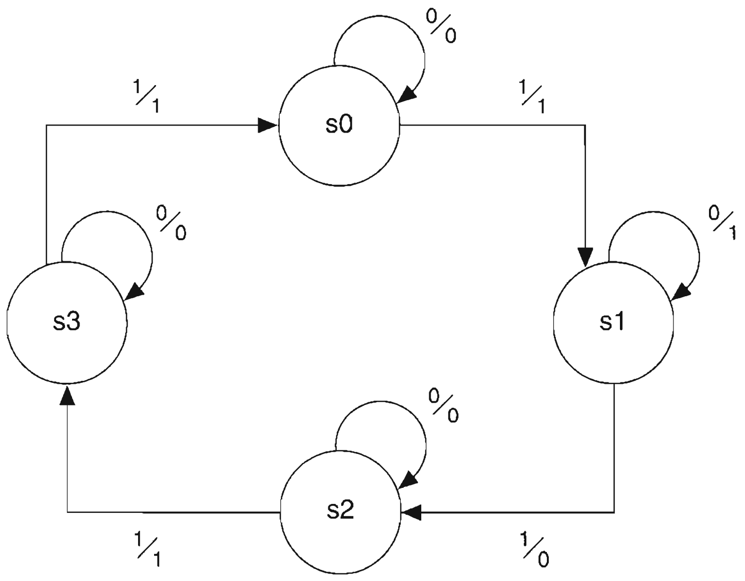
Go to "Truth Table Statement" on page 183 in *Design Structure* for more information.

State Machines with Asynchronous Outputs

AHDL supports the implementation of state machines with asynchronous outputs. The outputs of these types of state machines can change whenever the inputs change, regardless of Clock transitions.

Figure 2-45 shows a four-state Mealy state machine diagram. In Mealy state machines, the outputs are a function of the inputs and the current state.

Figure 2-45. Mealy State Machine Diagram



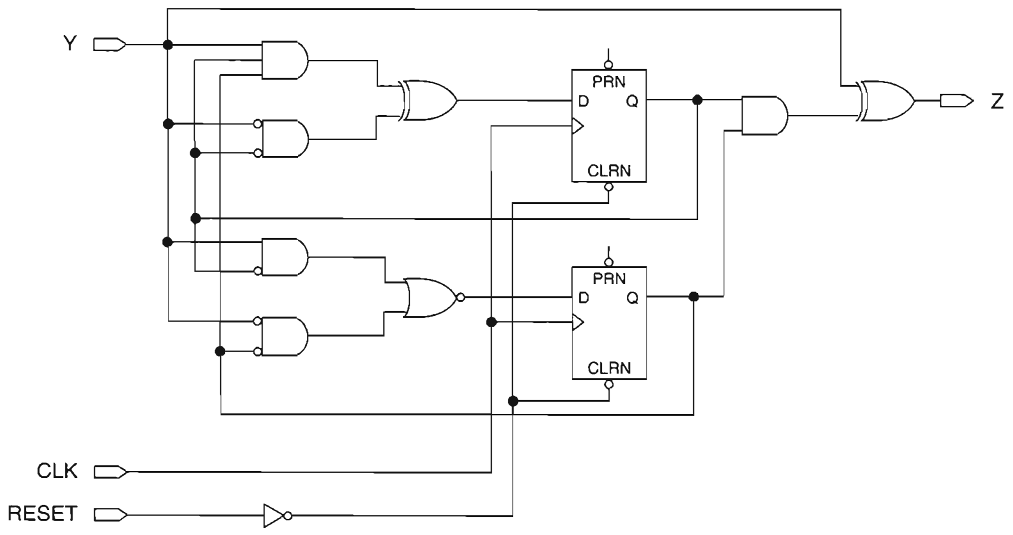
The **mealy.tdf** file shown in Figure 2-46 implements a four-state Mealy state machine with asynchronous outputs.

Figure 2-46. mealy.tdf

```
SUBDESIGN mealy
(
   clk      : INPUT;
   reset    : INPUT;
   y        : INPUT;
   z        : OUTPUT;
)

VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);
BEGIN
   ss.clk = clk;
   ss.reset = reset;

   TABLE
  % current  current   current     next  %
  %  state    input     output      state %
     ss,      y         => z,       ss;

     s0,      0         => 0,       s0;
     s0,      1         => 1,       s1;
     s1,      0         => 1,       s1;
     s1,      1         => 0,       s2;
     s2,      0         => 0,       s2;
     s2,      1         => 1,       s3;
     s3,      0         => 0,       s3;
     s3,      1         => 1,       s0;
   END TABLE;
END;
```

Figure 2-47 shows a GDF that is equivalent to **mealy.tdf**.

Figure 2-47. mealy.gdf



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

Recovering From Illegal States

Logic generated for a state machine by the MAX+PLUS II Compiler will behave as you specified in the TDF. However, state machine designs that explicitly declare state bits and which also do not use one-hot encoding often allow state bit values that are not assigned to valid states. These unassigned state bit values are called illegal states. A design that enters an illegal state—for example, as a result of setup or hold time violations—can cause erroneous outputs. Although Altera recommends that state machine inputs meet all setup and hold time requirements, you can make a state machine recover from an illegal state by forcing the illegal state to a known state with a Case Statement.

The **Global Project Logic Synthesis** dialog box (Assign menu) includes a *One-Hot State Machine Encoding* option that automatically implements one-hot encoding—which automatically assigns all state bits to valid states—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.

To recover from illegal states, you must name all illegal states in a state machine. The `WHEN OTHERS` clause in the Case Statement, which forces each transition from an illegal state to a known state, applies only to states that have been declared but are not mentioned in a `WHEN` clause. The `WHEN OTHERS` clause can force the required transitions only if all illegal states are defined in the State Machine Declaration.

For an *n*-bit state machine, 2^*n* possible states exist. If you declare *n* bits in a state machine, you should continue to add dummy state names until the number of states reaches a power of 2. The **recover.tdf** file shown in Figure 2-48 contains a state machine that can recover from illegal states.

Figure 2-48. recover.tdf

```
SUBDESIGN recover
(
   clk : INPUT;
   go  : INPUT;
   ok  : OUTPUT;
)
VARIABLE
   sequence : MACHINE
            OF BITS (q[2..0])
            WITH STATES (
               idle,
               one,
               two,
               three,
               four,
               illegal1,
               illegal2,
               illegal3);
BEGIN
   sequence.clk = clk;

   CASE sequence IS
      WHEN idle =>
         IF go THEN
            sequence = one;
         END IF;
      WHEN one =>
         sequence = two;
      WHEN two =>
         sequence = three;
      WHEN three =>
         sequence = four;
      WHEN OTHERS =>
         sequence = idle;
   END CASE;

   ok = (sequence == four);
END;
```

This example contains 3 bits: `q2`, `q1`, and `q0`. Therefore, 2^3 states, i.e., 8 states, exist. Since only 5 of the states are declared, 3 dummy state names were added, creating a total of 8 states.

Implementing a Hierarchical Project

AHDL TDFs can be mixed with other design files in a project hierarchy. Lower-level files in a project hierarchy can either be Altera-provided mega- or macrofunctions or user-defined functions.

Information on implementing a hierarchical project is available in the following topics:

- Using Altera-Provided Unparameterized Functions 69
- Using Altera-Provided Parameterized Functions 73
- Using Custom Megafunctions & Macrofunctions 76
- Importing & Exporting State Machines 77

Go tothe following sources for more information:

"Renaming a Megafunction or Macrofunction in the Current Project" in MAX+PLUS II Help
"Megafunctions" on page 129 and "Old-Style Macrofunctions" on page 131 in *Elements*

Using Altera-Provided Unparameterized Functions

MAX+PLUS II includes libraries of primitives and old-style macrofunctions that are not inherently parameterized; in addition, some megafunctions are not inherently parameterized. All MAX+PLUS II logic functions can be used to create hierarchical logic designs. Mega- and macrofunctions are automatically installed in subdirectories of the **\maxplus2\max2lib** directory created during installation; primitive logic is built into AHDL. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

There are two ways to use (i.e., insert an instance of) an unparameterized function in AHDL:

- Declare a variable for the function, i.e., an instance name, in an Instance Declaration in the Variable Section, and use ports of the instance of the function in the Logic Section.

- Use an in-line logic function reference in the Logic Section of the TDF.

The inputs and outputs of mega- and macrofunctions must be declared with a Function Prototype Statement. (Function Prototypes are not required for primitives.) MAX+PLUS II provides Include Files (**.inc**) that contain Function Prototypes for all MAX+PLUS II mega- and macrofunctions in the **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories, respectively. With an Include Statement, you can import the contents of an Include File into a TDF to declare the Function Prototype of a MAX+PLUS II mega- or macrofunction.

The **macro1.tdf** file shown in Figure 2-49 shows a 4-bit counter connected to a 4-bit-binary-to-16-line decoder. These macrofunctions are called with Instance Declarations in the Variable Section.

Figure 2-49. macro1.tdf

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN macro1
(
   clk          : INPUT;
   out[15..0]   : OUTPUT;
)
VARIABLE
   counter      : 4count;
   decoder      : 16dmux;
BEGIN
   counter.clk = clk;
   counter.dnup = GND;
   decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
   out[15..0] = decoder.q[15..0];
END;
```

This file uses Include Statements to import Function Prototypes for two Altera-provided macrofunctions: `4count` and `16dmux`. In the Variable Section, the variable `counter` is declared as an instance of the `4count` function, and the variable `decoder` is declared as an instance of the `16dmux` function. The input ports of both functions, which are in the format *<instance name>*.*<port name>*, are defined on the left side of the Boolean equations in the Logic Section; the output ports are defined on the right.

The **macro2.tdf** file shown in Figure 2-50 has the same functionality as **macro1.tdf**, but creates instances of the two functions with in-line references and the nodes `q[3..0]`:

Figure 2-50. macro2.tdf

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN macro2
(
   clk          : INPUT;
   out[15..0]   : OUTPUT;
)
VARIABLE
   q[3..0]      : NODE;
BEGIN
   (q[3..0], ) = 4count (clk, , , , , GND, , , , );

% equivalent in-line ref. with named port association  %
%  (q[3..0], ) = 4count (.clk=clk, .dnup=GND);         %

% equivalent in-line ref. with named port association  %
% and RETURNS clause specifying which outputs are used %
%  q[3..0] = 4count (.clk=clk, .dnup=GND)              %
%                 RETURNS (qd, qc, qb, qa);            %

   out[15..0]  = 16dmux (.(d, c, b, a)=q[3..0]);
% equivalent in-line ref. with positional port association %
% out[15..0]  = 16dmux (q[3..0]);                          %
END;
```

The Function Prototypes for the two macrofunctions, which are stored in the Include Files **4count.inc** and **16dmux.inc**, are shown below:

```
FUNCTION 4count (clk, clrn, setn, ldn, cin, dnup, d, c, b, a)
   RETURNS (qd, qc, qb, qa, cout);

FUNCTION 16dmux (d, c, b, a)
   RETURNS (q[15..0]);
```

The in-line references for `4count` and `16dmux` appear in the first and second Boolean equations in the Logic Section, respectively. The in-line reference for `4count` uses positional port association, whereas the in-line reference for `16dmux` uses named port association. The input ports of both macrofunctions are defined on the right side of the in-line references; the output ports are defined on the left.

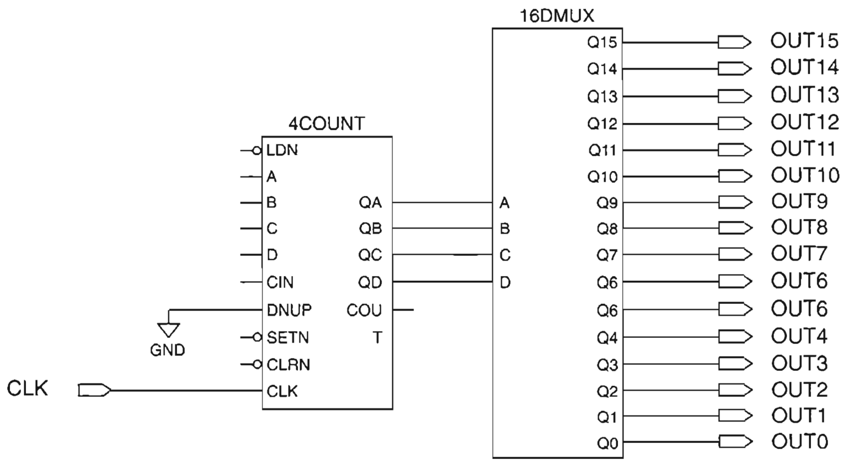
Comments show the equivalent in-line references for different styles of port association. In an in-line reference, ports on the right-hand side of the equals symbol (=) can be listed with either positional or named port association; ports on the left-hand side of the equals symbol always use positional port association. When positional port association is used, the order of ports is important because there is a one-to-one correspondence between the order of the ports in the Function Prototype and the ports defined in the Logic Section. In the in-line reference for `4count`, commas are used as placeholders for ports that are not explicitly connected.

A `RETURNS` clause, which is based on the `RETURNS` clause in the Function Prototype, is optional in an in-line reference. The `RETURNS` clause can be used to list the subset of the function's outputs that is used in the instance. In **macro2.tdf**, the second comment that shows an alternative in-line reference for `4count` omits the `cout` output of `4count` from the `RETURNS` clause; therefore, only the `q[3..0]` outputs are listed in the in-line reference and a comma placeholder is not required for `cout`.

Primitives and old-style macrofunctions always have default values for unconnected inputs. In contrast, megafunctions do not necessarily have default values for unconnected inputs.

Figure 2-51 shows a GDF that is equivalent to **macro1.tdf** and **macro2.tdf**.

Figure 2-51. macro.gdf



Go to the following topics for more information:

"Function Prototype Statement" on page 151 in *Design Structure*
"In-Line Logic Function Reference" on page 180 in *Design Structure*
"Include Statement" on page 145 in *Design Structure*
"Instance Declaration" on page 160 in *Design Structure*
"Logic Section" on page 168 in *Design Structure*
"Ports" on page 132 in *Elements*

Using Altera-Provided Parameterized Functions

MAX+PLUS II includes inherently parameterized megafunctions, including LPM functions. For example, parameters are used to specify the width of a port or whether a block of RAM should be implemented as synchronous or asynchronous memory. Parameterized functions can contain other subdesigns, which may be parameterized or unparameterized. Parameters can also be used on some old-style macrofunctions that are not inherently parameterized. (Primitives cannot be parameterized.) All MAX+PLUS II logic functions can be used to create hierarchical logic designs. Mega- and macrofunctions are automatically installed in subdirectories of the **\maxplus2\max2lib** directory created during installation; primitive logic is built into AHDL.

Parameterized functions are instantiated with an in-line logic function reference or an Instance Declaration in the same way as unparameterized functions, as described in "Using Altera-Provided Unparameterized Functions," with a few additional steps:

- The logic function instance must include a `WITH` clause, which is based on the `WITH` clause in the Function Prototype, that lists the parameters used by the instance. You can use the `WITH` clause to optionally assign parameter values on an instance; however, for all required parameters in a function, a parameter value must be supplied somewhere within the project. If the instance itself does not include some or all of the values for required parameters, the Compiler searches for them in the parameter value search order described on page 136 in *Elements*.

- Since parameterized megafunctions do not necessarily have default values for unconnected inputs, you must ensure that all required ports are connected. In contrast, primitives and old-style macrofunctions always have default values for unconnected inputs.

The inputs, outputs, and parameters of the function are declared with a Function Prototype Statement. MAX+PLUS II provides Include Files that contain Function Prototypes for all MAX+PLUS II mega- and macrofunctions in the **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories, respectively. With an Include Statement, you can import the contents of an Include File into a TDF to declare the Function Prototype of a MAX+PLUS II mega- or macrofunction.

The **lpm_add1.tdf** file shown in Figure 2-52 implements an 8-bit adder with an in-line logic function reference to the parameterized `lpm_add_sub` megafunction:

Figure 2-52. lpm_add1.tdf

```
INCLUDE "lpm_add_sub.inc";

SUBDESIGN lpm_add1
(
   a[8..1], b[8..1]  : INPUT;
   c[8..1]           : OUTPUT;
   carry_out         : OUTPUT;
)

BEGIN
% Megafunction instance with positional port association %
-- (c[], carry_out, ) = lpm_add_sub(GND, a[], b[], GND)
--                         WITH (LPM_WIDTH=8,
--                               LPM_REPRESENTATION="unsigned");
% Equivalent instance with named port association %
-- (c[], carry_out, ) = lpm_add_sub(.dataa[]=a[], .datab[]=b[],
--                            .cin=GND, .add_sub=GND)
--                         WITH (LPM_WIDTH=8,
--                               LPM_REPRESENTATION="unsigned");
END;
```

The Function Prototype for `lpm_add_sub`, which is stored in the Include File **lpm_add_sub.inc**, is shown below:

```
FUNCTION lpm_add_sub(cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-
   1..0], add_sub)
   WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDERTYPE,
      ONE_INPUT_IS_CONSTANT)
   RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

Only the `LPM_WIDTH` parameter is required, and the instance of the `lpm_add_sub` function in **lpm_add1.tdf** specifies parameter values only for the `LPM_WIDTH` and `LPM_REPRESENTATION` parameters.

The **lpm_add2.tdf** file shown in Figure 2-53 is identical to **lpm_add1.tdf**, but implements the 8-bit adder with an Instance Declaration:

Figure 2-53. lpm_add2.tdf

```
INCLUDE "lpm_add_sub.inc";

SUBDESIGN lpm_add2
(
   a[8..1], b[8..1] : INPUT;
   c[8..1]          : OUTPUT;
   carry_out        : OUTPUT;
)

VARIABLE
   8bitadder : lpm_add_sub WITH (LPM_WIDTH=8,
                  LPM_REPRESENTATION="unsigned");
BEGIN
   8bitadder.cin = GND
   8bitadder.dataa[] = a[]
   8bitadder.datab[] = b[]
   8bitadder.add_sub = GND
   c[] = 8bitadder.result[]
   carry_out = 8bitadder.cout
END;
```

Go to the following topics for more information:

"Function Prototype Statement" on page 151 in *Design Structure*
"In-Line Logic Function Reference" on page 180 in *Design Structure*
"Include Statement" on page 145 in *Design Structure*
"Instance Declaration" on page 160 in *Design Structure*
"Logic Section" on page 168 in *Design Structure*
"Parameters" on page 136 in *Elements*
"Ports" on page 132 in *Elements*

Using Custom Megafunctions & Macrofunctions

You can easily create and use custom megafunctions or macrofunctions in AHDL TDFs.

Once you have defined the logic for a custom function in a design file, a few steps are required to use the function in other TDFs or other types of design files.

To prepare a custom AHDL-based mega- or macrofunction for use in other design files:

1. Compile and optionally simulate the design file to ensure that it functions correctly.

2. If you plan to use the function in multiple projects, you should designate the directory that contains the design file as a user library with **User Libraries** (Options menu) or save a copy of the file to an existing user library directory. Otherwise, save a copy of the file to the directory containing the project that will use the custom function.

3. With the file open in a Text Editor window, create an Include File and a symbol that represent the current file:

   a. Choose **Create Default Include File** (File menu) to create an Include File that can be used in a higher-level TDF. With an Include Statement, you can import the contents of an Include File into a TDF to declare the Function Prototype of a custom mega- or macrofunction.

   b. Choose **Create Default Symbol** (File menu) to create a symbol that can be used in a GDF.

Once you have prepared a function for use in other design files, you can create a new TDF and insert an instance of the function with an Instance Declaration or an in-line reference. You can use custom functions in exactly the same way as Altera-provided functions. See "Using Altera-Provided Unparameterized Functions" on page 69 and "Using Altera-Provided Parameterized Functions" on page 73 for more information.

Importing & Exporting State Machines

You can import and export state machines between TDFs and other design files by specifying an input or output port as `MACHINE INPUT` or `MACHINE OUTPUT` in the Subdesign Section. The Function Prototype that represents the file containing the state machine must indicate which inputs and outputs are state machines by prefixing the signal names with the keyword `MACHINE`.

☞ `MACHINE INPUT` and `MACHINE OUTPUT` port types cannot be used in a top-level design file. Although top-level files with these port types do not compile fully, you can use **Project Save & Check** (File menu) to check their syntax and **Create Default Include File** (File menu) to create an Include File that represents the current file.

You can rename a state machine with a temporary name by entering a Machine Alias Declaration in the Variable Section. You can use a machine alias in the file where the state machine is created or in a file that uses a `MACHINE INPUT` port to import a state machine. You can then use this name instead of the original state machine name.

The **ss_def.tdf** file shown in Figure 2-54 defines and exports the state machine `ss` with the `MACHINE OUTPUT` port `ss_out`.

Figure 2-54. ss_def.tdf

```
SUBDESIGN ss_def
(
   clk, reset, count    : INPUT;
   ss_out               : MACHINE OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s1, s2, s3, s4, s5);
BEGIN
   ss_out = ss;

   CASE ss IS
      WHEN s1=>
           IF count THEN ss = s2; ELSE ss = s1; END IF;
      WHEN s2=>
           IF count THEN ss = s3; ELSE ss = s2; END IF;
      WHEN s3=>
           IF count THEN ss = s4; ELSE ss = s3; END IF;
      WHEN s4=>
           IF count THEN ss = s5; ELSE ss = s4; END IF;
      WHEN s5=>
           IF count THEN ss = s1; ELSE ss = s5; END IF;
   END CASE;

   ss.(clk, reset) = (clk, reset);
END;
```

The **ss_use.tdf** file shown in Figure 2-55 imports a state machine with the MACHINE INPUT port ss_in.

Figure 2-55. ss_use.tdf

```
SUBDESIGN ss_use
(
   ss_in    : MACHINE INPUT;
   out      : OUTPUT;
)
BEGIN
   out = (ss_in == s2) OR (ss_in == s4);
END;
```

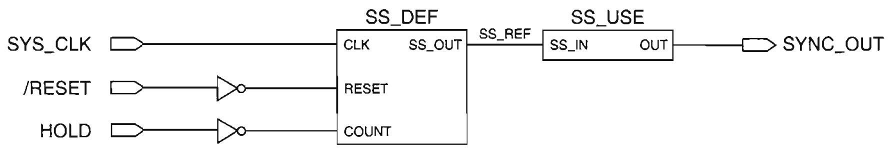
The **top1.tdf** file shown in Figure 2-56 uses in-line references to insert instances of the functions `ss_def` and `ss_use`. The Function Prototypes for `ss_def` and `ss_use` include `MACHINE` keywords that indicate which inputs and outputs are state machines.

Figure 2-56. top1.tdf

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);

SUBDESIGN top1
(
   sys_clk, /reset, hold      : INPUT;
   sync_out                   : OUTPUT;
)
VARIABLE
   ss_ref: MACHINE;   % Machine Alias Declaration %
BEGIN
   ss_ref = ss_def(sys_clk, !/reset, !hold);
   sync_out = ss_use(ss_ref);
END;
```

Figure 2-57 shows a GDF that is equivalent to **top1.tdf**.

Figure 2-57. top1.gdf



An external state machine can also be implemented in a top-level TDF with an Instance Declaration in the Variable Section. The **top2.tdf** file shown in Figure 2-58 has the same functionality as **top1.tdf**, but uses Instance Declarations instead of in-line references to instantiate the functions.

Figure 2-58. top2.tdf

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);

SUBDESIGN top2
(
   sys_clk, /reset, hold      : INPUT;
   sync_out                   : OUTPUT;
)
VARIABLE
   sm_macro                   : ss_def;
   sync                       : ss_use;
BEGIN
   sm_macro.(clk, reset, count) = (sys_clk, !/reset, !hold);
   sync.ss_in = sm_macro.ss_out;
   sync_out = sync.out;
END;
```

Go to the following topics for more information:

"Implementing a Hierarchical Project" on page 69 in this section
"Machine Alias Declaration" on page 166 in *Design Structure*
"State Machines" on page 54 in this section
"Using Altera-Provided Parameterized Functions" on page 73 in this section
"Using Altera-Provided Unparameterized Functions" on page 69 in this section
"Using Custom Megafunctions & Macrofunctions" on page 76 in this section

Implementing LCELL & SOFT Primitives

You can limit the extent of logic synthesis by changing NODE variables into SOFT and LCELL primitives. NODE variables and LCELL primitives provide the greatest control over logic synthesis. SOFT primitives provide less control over logic synthesis.

NODE variables, which are declared with a Node Declaration in the Variable Section, place very few restrictions on logic synthesis. During synthesis, the Logic Synthesizer replaces each instance of a NODE variable with the logic that the variable represents. It then minimizes the logic to fit into a single logic cell. This method usually yields the greatest speed, but may result in logic that is too complex or hard to fit.

SOFT buffers provide more control over resource usage than NODE variables. The Logic Synthesizer chooses when to replace instances of SOFT primitives with LCELL primitives. SOFT buffers may help eliminate logic that is too complex and make the project easier to fit, but may increase logic cell utilization and reduce speed performance.

LCELL primitives provide the most control. The Logic Synthesizer minimizes all logic that drives an LCELL primitive so that the logic fits into a single logic cell. LCELL primitives are always implemented in a logic cell, and they are never removed from the project even if they are fed by a single input. If the project is minimized so that an LCELL primitive is fed by a single input, you can use a SOFT primitive instead of an LCELL primitive so that the SOFT primitive is removed during logic synthesis.

MAX+PLUS II provides several logic options that automatically insert or remove SOFT and LCELL buffers at appropriate locations in the project. See "Assigning a Logic Option" in MAX+PLUS II Help for more information.

Figure 2-59 shows two versions of a TDF: one is implemented with NODE variables and one with SOFT primitives. In nodevar, the variable odd_parity is declared as a NODE, then assigned the value of the Boolean expression `d0 $ d1 $ ... $ d8`. In `softbuf`, the Compiler will replace some of the `SOFT` primitives with `LCELL` primitives during processing to improve device utilization.

Figure 2-59. NODE Variables & SOFT Primitives

TDF with NODE Variables:

```
SUBDESIGN nodevar
(
   .
   .
   .
)
VARIABLE
   odd_parity : NODE;
BEGIN
   odd_parity = d0 $ d1 $ d2
              $ d3 $ d4 $ d5
              $ d6 $ d7 $ d8;
END;
```

TDF with SOFT Primitives:

```
SUBDESIGN softbuf
(
   .
   .
   .
)
VARIABLE
   odd_parity : NODE;
BEGIN
   odd_parity = SOFT(d0 $ d1 $ d2)
            $ SOFT(d3 $ d4 $ d5)
            $ SOFT(d6 $ d7 $ d8);
END;
```

Go to the following sources for more information:

"Declaring Nodes" on page 27 in this section
"Guiding the Partitioner" in MAX+PLUS II Help
"LCELL Primitive" on page 120 and "SOFT Primitive" on page 123 in
    Elements

Implementing RAM & ROM

MAX+PLUS II (and AHDL) provide several LPM functions and other megafunctions that allow you to implement RAM and ROM in MAX+PLUS II devices. The generic, scalable nature of each of these functions ensures that you can use them to implement any supported type of RAM or ROM in MAX+PLUS II.

Altera does not recommend creating custom logic functions to implement memory. You should use Altera-provided functions in all cases where you wish to implement RAM or ROM.

The following megafunctions can be used to implement RAM and ROM in MAX+PLUS II:

Name:	**Description:**
`lpm_ram_dq`	Synchronous or asynchronous memory with separate input and output ports
`lpm_ram_io`	Synchronous or asynchronous memory with a single I/O port
`lpm_rom`	Synchronous or asynchronous read-only memory
`csdpram`	Cycle-shared dual port-memory
`csfifo`	Cycle-shared first-in first-out (FIFO) buffer

In these LPM functions, parameters are used to determine the input and output data widths; the number of data words stored in memory; whether data inputs, address/control inputs, and outputs are registered or unregistered; whether an initial memory content file is to be included for a RAM block; and so on.

Choose **Megafunctions/LPM** (Help menu) for detailed information on memory megafunctions.

Naming a Boolean Operator or Comparator

You can name Boolean operators and comparators in AHDL files to make it easy to enter resource assignments and to interpret the Equations Section of the project's Report File (**.rpt**).

The **boole3.tdf** file shown in Figure 2-60 is identical to the **boole1.tdf** file (shown inFigure 2-8), but uses named operators. The operator name is separated from the operator by a colon (:); the name can contain up to 32 name chararacters.

Figure 2-60. boole3.tdf

```
SUBDESIGN boole3
(
   a0, a1, b  : INPUT;
   out1, out2 : OUTPUT;
)
BEGIN
   out1 = a1 tiger:& !a0;
   out2 = out1 panther:# b;
END;
```

The following Report File excerpts show the difference between **boole3.rpt** and **boole1.rpt** for the first of the two equations:

Figure 2-61. boole3.rpt & boole1.rpt Excerpts

```
-- boole3.rpt equations:
-- Node name is 'out1' from file "boole3.tdf" line 7, column 2
-- Equation name is 'out1', location is LC3_A1, type is output
out1 = tiger~0;


-- Node name is 'tiger~0' from file "boole3.tdf" line 7, column 18
-- Equation name is 'tiger~0', location is LC2_A1, type is buried
tiger~0 = LCELL( _EQ002);
  _EQ002 = !a0 &  a1;

-- boole1.rpt equations:
-- Node name is 'out1' from file "boole1.tdf" line 7, column 2
-- Equation name is 'out1', location is LC3_A1, type is output
out1 = _LC2_A1;

-- Node name is ':33' from file "boole1.tdf" line 7, column 12
-- Equation name is '_LC2_A1', type is buried
LC2_A1 = LCELL( _EQ001);
  _EQ001 = !a0 &  a1;
```

Depending on the logic of the equation, a named operator can produce multiple node names; however, all names are based on the operator name and are thus easily recognizable in the Report File. In **boole3.rpt**, a single node, `tiger~0`, is generated for the first equation. In **boole1.tdf**, the Compiler assigns the net ID `:33` to the same node.

After you have compiled a project, you can use the named operator-based node names shown in the Report File to enter resource assignments for future compilations, even if the project logic changes. The names of logic cells created from named operators remain constant if you change unrelated logic (e.g., other equations) in the file. For example, you can enter an assignment on the node `tiger~0`. In contrast, if operators are unnamed, only net ID numbers are available, and these numbers are randomly reassigned with each compilation.

Go to the following topics for more information:

"Implementing Boolean Expressions & Equations" on page 25 in this section
"Logical Operators" on page 107 in *Elements*
"Quoted & Unquoted Names" on page 97 in *Elements*

Using Iteratively Generated Logic

When you wish to use multiple blocks of logic that are very similar, you can use the For Generate Statement to iteratively generate logic based on a numeric range delimited by arithmetic expressions.

The **iter_add.tdf** file shown in Figure 2-62 shows an example of iterative logic generation:

Figure 2-62. iter_add.tdf

```
CONSTANT NUM_OF_ADDERS = 8;

SUBDESIGN iter_add
(
   a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1], cin : INPUT;
   c[NUM_OF_ADDERS..1], cout                     : OUTPUT;
)

VARIABLE
   sum[NUM_OF_ADDERS..1], carryout[(NUM_OF_ADDERS+1)..1] :

BEGIN
   carryout[1] = cin;
   FOR i IN 1 TO NUM_OF_ADDERS GENERATE
     sum[i] = a[i] $ b[i] $ carryout[i];   % Full Adder %
     carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
   END GENERATE;
   cout = carryout[NUM_OF_ADDERS+1];
   c[] = sum[];
END;
```

In **iter_add.tdf**, the For Generate Statement is used to instantiate full adders that each perform one bit of the `NUM_OF_ADDERS`-bit (i.e., 8-bit) addition. The `carryout` of each bit is generated along with each full adder.

The If Generate Statement is especially useful with For Generate Statements that handle special cases differently, for example, the first and last stages of a multi-stage multiplier. See "Using Conditionally Generated Logic," next, for more information.

Go to "For Generate Statement" on page 179 in *Design Structure* for more information.

Using Conditionally Generated Logic

You can generate logic conditionally with If Generate Statements, if, for example, you wish to implement different behavior based on the value of an arithmetic expression. An If Generate Statement lists a series of behavioral statements that are activated after the positive evaluation of one or more arithmetic expressions.

The **condlog1.tdf** file shown in Figure 2-63 uses an If Generate Statement to implement different behavior for the `output_b` output on the basis of the current device family:

Figure 2-63. condlog1.tdf

```
PARAMETERS (DEVICE_FAMILY);

SUBDESIGN condlog1
(
   input_a : INPUT;
   output_b : OUTPUT;
)
BEGIN
   IF DEVICE_FAMILY == "FLEX8K" GENERATE
      output_b = input_a;
   ELSE GENERATE
      output_b = LCELL(input_a);
   END GENERATE;
END;
```

The If Generate Statement is especially useful with For Generate Statements that handle special cases differently, for example, the first and last stages of a multi-stage multiplier. See "Using Iteratively Generated Logic" on page 86 for more information on For Generate Statements.

MAX+PLUS II includes the predefined parameter `DEVICE_FAMILY`, as shown in the example above, and the predefined evaluated function `USED`, which can be used in arithmetic expressions. The `DEVICE_FAMILY` parameter can be used to test the current device family for the project, which is specified with **Device** (Assign menu). The `USED` evaluated function can be used to test whether an optional port has been used in the current instance. `USED` takes the port name as input and returns a value of `FALSE` if the port is not used.

You can find numerous additional examples of If Generate Statements in the TDFs that implement LPM functions in MAX+PLUS II. These TDFs are located in the **mega_lpm** subdirectory of the **\maxplus2\max2lib** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

Go to "If Generate Statement" on page 178 in *Design Structure* for more information.

Using the Assert Statement

You can use the Assert Statement to test the validity of any arbitrary expression that uses parameters, numbers, evaluated functions, or the used or unused status of a port. You might, for example, use the Assert Statement to determine whether the value of an optional parameter falls within a range determined by the value of a second parameter.

When you use an Assert Statement with conditions, you list the acceptable values for the assertion conditions. If a value is unacceptable, the assertion is activated and a message is issued. If you use an Assert Statement without conditions, the assertion is always activated.

The Compiler evaluates each assertion condition only once, after the Compiler Netlist Extractor module has resolved all parameter values. An assertion cannot depend on the value of a signal that is implemented in the device. For example, if an Assert Statement is placed after an If Then Statement of the form `IF a = VCC THEN c = d`, the assertion condition cannot depend on the value of `a`.

The **condlog2.tdf** file shown in Figure 2-64 has the same functionality as **condlog1.tdf** (shown in Figure 2-63 on page 87), but uses Assert Statements in the Logic Section to report which logic was generated by the If Generate Statement:

Figure 2-64. condlog2.tdf

```
PARAMETERS (DEVICE_FAMILY);

SUBDESIGN condlog2
(
   input_a : INPUT;
   output_b : OUTPUT;
)
BEGIN
   IF DEVICE_FAMILY == "FLEX8000" GENERATE
      output_b = input_a;
      -- Assertion is always activated if there is no condition
      ASSERT
         REPORT "Compiling for FLEX8000 family"
         SEVERITY INFO;
   ELSE GENERATE
      output_b = LCELL(input_a);
      -- Assertion is activated if current family is not FLEX10K
      -- or FLEX 8000. Severity defaults to ERROR
      ASSERT (DEVICE_FAMILY == "FLEX10K")
         REPORT "Compiling for % family", DEVICE_FAMILY;
   END GENERATE;
END;
```

You can find numerous additional examples of Assert Statements in the TDFs that implement LPM functions in MAX+PLUS II. These TDFs are located in the **mega_lpm** subdirectory of the **\maxplus2\max2lib** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

Section

3

Elements

This section describes the basic format of an AHDL Text Design File (**.tdf**) and its elements. These elements are used in the behavioral statements described in *Design Structure* on page 139.

- Reserved Keywords & Identifiers 92
- Symbols 94
- Quoted & Unquoted Names 97
- Groups 99
- Numbers in AHDL 102
- Arithmetic Expressions 103
- Boolean Expressions 106
- Primitives 113
- Megafunctions 129
- Old-Style Macrofunctions 131
- Ports 132
- Parameters 136

Go to MAX+PLUS II Help for complete and up-to-date information on AHDL elements. For information on element syntax, refer to MAX+PLUS II Help.

Reserved Keywords & Identifiers

Reserved keywords are used for beginnings, endings, and transitions of AHDL statements and for the predefined constant values GND and VCC.

Reserved keywords differ from reserved identifiers in that keywords can be used as symbolic names when they are enclosed in single quotation marks ('), whereas reserved identifiers cannot. Both reserved keywords and reserved identifiers can be used freely in comments.

Altera recommends that you enter all keywords and reserved identifiers in capital letters for easy readability. See *Style Guide* on page 187 for more information.

Reserved Keywords

The following list shows all AHDL reserved keywords:

AND	FUNCTION	OUTPUT
ASSERT	GENERATE	PARAMETERS
BEGIN	GND	REPORT
BIDIR	HELP_ID	RETURNS
BITS	IF	SEGMENTS
BURIED	INCLUDE	SEVERITY
CASE	INPUT	STATES
CLIQUE	IS	SUBDESIGN
CONNECTED_PINS	LOG2	TABLE
CONSTANT	MACHINE	THEN
DEFAULTS	MOD	TITLE
DEFINE	NAND	TO
DESIGN	NODE	TRI_STATE_NODE
DEVICE	NOR	VARIABLE
DIV	NOT	VCC
ELSE	OF	WHEN
ELSIF	OPTIONS	WITH
END	OR	XNOR
FOR	OTHERS	XOR

Reserved Identifiers

The following list shows all AHDL reserved identifiers:

`CARRY`	`JKFFE`	`SRFFE`
`CASCADE`	`JKFF`	`SRFF`
`CEIL`	`LATCH`	`TFFE`
`DFFE`	`LCELL`	`TFF`
`DFF`	`MCELL`	`TRI`
`EXP`	`MEMORY`	`WIRE`
`FLOOR`	`OPNDRN`	`X`
`GLOBAL`	`SOFT`	

Symbols

Table 3-1 lists the symbols that have predefined meanings in AHDL. This table includes symbols that are used as operators and comparators in Boolean expressions and as operators in arithmetic expressions.

Table 3-1. AHDL Symbols (Part 1 of 3)

Symbol		Function
_ - /	(underscore) (dash) (forward slash)	User-defined identifiers used as legal characters in symbolic names.
--	(two dashes)	Starts a VHDL-style comment, which extends to the end of the line. (Go to "Comments & Documentation" on page 191 in *Style Guide* for more information.)
%	(percent)	Encloses AHDL-style comments. (Go to "Comments & Documentation" on page 191 in *Style Guide* for more information.)
( )	(left & right parentheses)	Enclose and define sequential group names. Enclose pin names in Subdesign Sections and Function Prototype Statements. Optionally enclose inputs and outputs of truth tables in Truth Table Statements. Enclose bits and states of State Machine Declarations. Enclose highest priority operations in Boolean and arithmetic expressions. Enclose parameter definitions in Parameters Statements and parameter names in Function Prototype Statements, Instance Declarations, and in-line references. Optionally enclose the condition in an Assert Statement. Enclose the arguments of evaluated functions in Define Statements.
[ ]	(left & right brackets)	Enclose the range of a single- or dual-range group name.
'...'	(single quotation marks)	Enclose quoted symbolic names.

Table 3-1. AHDL Symbols (Part 2 of 3)

Symbol		Function
`"..."`	(double quotation marks)	Enclose strings in Title Statements, Parameters Statements, and Assert Statements. Enclose a filename in Include Statements. Enclose digits in non-decimal numbers.
`.`	(period)	Separates symbolic names of logic function variables from port names. Separates extensions from filenames.
`..`	(ellipsis)	Separates most significant bit from least significant bit in ranges.
`;`	(semicolon)	Ends AHDL statements and sections.
`,`	(comma)	Separates members of sequential groups and lists.
`:`	(colon)	Separates symbolic names from types in declarations.
`=`	(equals)	Assigns default GND and VCC values to inputs in a Subdesign Section. Assigns settings to options in an Options Statement. Assigns a default value to a parameter in a Parameters Statement or an in-line reference. Assigns values to state machine states. Assigns values in Boolean equations. Connects a signal to a port in an in-line reference that uses named port association.
`=>`	(arrow)	Separates inputs from outputs in Truth Table Statements. Separates WHEN clauses from Boolean expressions in Case Statements.
`+`	(plus)	Addition operator
`-`	(minus)	Subtraction operator
`==`	(two equal signs)	Numeric or string equality operator
`!`	(exclamation point)	NOT operator
`!=`	(exclamation equals)	Not equal to operator
`>`	(greater than)	Greater than comparator
`>=`	(greater than equals)	Greater than or equal to comparator

Table 3-1. AHDL Symbols (Part 3 of 3)

Symbol		Function
<	(less than)	Less than comparator
<=	(less than equals)	Less than or equal to comparator
&	(ampersand)	AND operator
!&	(exclamation ampersand)	NAND operator
$	(dollar sign)	XOR operator
!$	(exclamation dollar)	XNOR operator
#	(pound sign	OR operator
!#	(exclamation pound)	NOR operator
?	(question mark)	Ternary operator

Go to the following topics for more information:

"Arithmetic Expressions" on page 103 in this section
"Arithmetic Operators in Boolean Expressions" on page 109 in this section
"Boolean Equations" on page 168 in *Design Structure*
"Comparators" on page 111 in this section
"Logical Operators" on page 107 in this section

Quoted & Unquoted Names

Three types of names exist in AHDL:

- Symbolic names are user-defined identifiers in AHDL. They are used to name the following parts of a TDF:
  - Internal and external nodes and groups
  - Constants
  - State machine variables, state bits, and state names
  - Instances
  - Parameters
  - Memory segments
  - Evaluated functions
  - Named operators
- Subdesign names are user-defined names for lower-level design files. The subdesign name must be the same as the TDF filename.
- Port names are symbolic names that identify the input or output of a logic function.

☞ Compiler-generated pin names that contain the tilde (~) character may appear in the Fit File (**.fit**) for a project. If you back-annotate the Fit File assignments, these names will then appear in the project's Assignment & Configuration File (**.acf**). The tilde character is reserved for Compiler-generated names only; you cannot use it in your own pin, node, and group (bus) names.

Two notations are available for subdesign, symbolic, and port names: quoted and unquoted. Quoted names are enclosed in single quotation marks ('); unquoted names are not.

☞ When you create a default symbol for a TDF that includes quoted port names, the quotes are not included in the pinstub names shown in the symbol.

Table 3-2 summarizes the characteristics of subdesign, symbolic, and port names:

Table 3-2. Quoted & Unquoted Names

Legal Name Characters *Note (1)*	Unquoted Subdesign Name	Quoted Subdesign Name	Unquoted Symbolic Name	Quoted Symbolic Name	Unquoted Port Name	Quoted Port Name
`A-Z`	✓	✓	✓	✓	✓	✓
`a-z`	✓	✓	✓	✓	✓	✓
`0-9`	✓	✓	✓	✓	✓	✓
Underscore (_)	✓	✓	✓	✓	✓	✓
Slash (/)	No	No	✓	✓	✓	✓
Dash (-)	No	✓	No	✓	No	✓
Digits only (`0-9`)	✓	✓	No	✓	✓	✓
Keyword	No	✓	No	✓	No	✓
Identifier	No	✓	No	No	No	✓
Max. Characters	32	32	32	32	32	32

Note:
(1) The delimiters of ranges in single-range and dual-range group names can also include the operators described in "Arithmetic Expressions" on page 103.

For example, legal unquoted and quoted symbolic names include:

```
a     /a1     '-bar'     'table'     '1221'
```

Illegal unquoted and quoted symbolic names include:

```
-foo     node     55     'bowling4$'

'a_name_with_more_than_32_characters'     'has a space'
```

Go to "Reserved Keywords & Identifiers" on page 92 for more information.

Groups

Symbolic names and ports of the same type may be declared and used as groups in Boolean expressions and equations.

A group, which can include up to 256 members (or "bits"), is treated as a collection of nodes and acted upon as a single unit.

Groups in the Logic Section or Variable Section of a TDF can consist of nodes. Single nodes and the constants `GND` and `VCC` may be duplicated to form groups in Boolean expressions and equations.

Group Notations

Groups can be declared with the following three notations:

1. A single-range group name consists of a symbolic name or port name followed by a single range of numbers enclosed in brackets, e.g., `a[4..1]`. The symbolic or port name, together with the longest number in the range, can contain up to 32 name characters.

   Once a group has been defined, `[]` is a shorthand way of specifying the entire range. For example, `a[4..1]` can also be denoted by `a[]`.

   A single number can be used in place of a range, e.g., `a[5]`. However, this notation signifies a single symbolic name, not a group, and is equivalent to the name `a5`.

2. A dual-range group name consists of a symbolic name or port name followed by two ranges enclosed in brackets, e.g., `d[6..0][2..0]`. The symbolic or port name, together with the longest number in each range, can contain up to 32 name characters.

   The dual-range group notation is useful for specifying groups of buses and for designs with two-dimensional topologies. Once a group has been defined, `[][]` is a shorthand way of specifying both ranges. For example, `b[6..0][3..2]` can also be denoted by `b[][]`.

   An individual node within the group can be referenced as `name[y][z]` or `namey_z`, where `y` and `z` are numbers in the group ranges.

3. A sequential group name consists of a list of symbolic names, ports, or numbers, separated by commas and enclosed in parentheses, e.g., `(a, b, c)`. Single- and dual-range group names can be listed within the parentheses. For example, `(a, b, c[5..1])` is a legal group.

   This notation is useful for specifying port names. For example, the input ports of variable `reg` of type `DFF` can be written as `reg.(d, clk, clrn, prn)`.

   The following two sets of examples show two groups specified with different notations:

```
b[5..0]
(b5, b4, b3, b2, b1, b0)
b[]

b[log2(256)..1+2-1]
b[2^8..3 mod 1]
b[2*8..8 div 2]
```

☞ Compiler-generated pin names that contain the tilde (~) character may appear in the Fit File (**.fit**) for a project. If you back-annotate the Fit File assignments, these names will then appear in the project's Assignment & Configuration File (**.acf**). The tilde character is reserved for Compiler-generated names only; you cannot use it in your own pin, node, and group (bus) names.

Group Ranges & Subranges

Ranges of single- or dual-range group names can consist of numbers or arithmetic expressions that are separated by two periods (`..`) and enclosed in brackets `[]`. For example:

`a[4..1]`	is a group with members `a4`, `a3`, `a2`, and `a1`.
`d[B"10"..B"00"]`	is a group with members `d2`, `d1`, and `d0`.
`b[2*2..2-1]`	is a group with members `b4`, `b3`, `b2`, and `b1`. The group range delimiters are defined with arithmetic expressions.

`q[MAX..0]`	is a legal group if the constant `MAX` has been previously defined in a Constant Statement.
`c[MIN(a,b)..0]`	is a legal group if the evaluated function `MIN` has been previously defined in a Define Statement.

Regardless of whether a range delimiter is a number or an arithmetic expression, the Compiler resolves and interprets the delimiters as decimal values (integers).

Subranges include a subset of the nodes specified in a declared group, and can be specified in a number of ways. Commas can be used as placeholders only in groups on the left side of a Boolean equation or in an in-line reference.

For example, if you declare the group `c[5..1]`, you can use the following subranges of this group:

```
c[3..1]
c[4..2]
c4
c[5]
(c2, , c4)
```

In the subrange `(c2, , c4)`, a comma is used to hold the place of an unassigned group member.

Ranges are normally listed in descending order. To list ranges in ascending order or in both ascending and descending order, you must specify the `BITO` option with the Options Statement to prevent the Compiler from issuing warning messages. In dual-range group names, the `BITO` option affects both of the ranges.

Go to the following topics for more information:

"Arithmetic Expressions" on page 103 in this section
"Defining Groups" on page 28 in *How to Use AHDL*
"Numbers in AHDL" on page 102 in this section
"Quoted & Unquoted Names" on page 97 in this section
"Using Constants & Evaluated Functions" on page 19 in *How to Use AHDL*

Numbers in AHDL

You can use decimal, binary, octal, and hexadecimal numbers in any combination in AHDL. The syntax for each radix (numbering system) is shown below.

Radix:	**Values:**
Decimal	*<series of digits 0 to 9>*
Binary	`B"`*<series of 0's, 1's, X's>*`"`(where `X` = "don't care")
Octal	`O"`*<series of digits 0 to 7>*`"` or
	`Q"`*<series of digits 0 to 7>*`"`
Hexadecimal	`X"`*<series from 0 to 9, A to F>*`"` or
	`H"`*<series from 0 to 9, A to F>*`"`

The following examples show valid AHDL numbers:

```
B"0110X1X10"
Q"4671223"
H"123AECF"
```

The following rules apply to AHDL numbers:

- The MAX+PLUS II Compiler always interprets numbers in Boolean expressions as groups of binary digits; numbers in group ranges are interpreted as decimal values.

- Numbers cannot be assigned to single nodes in Boolean equations. Use `VCC` and `GND` instead.

Go to the following topics for more information:

"Arithmetic Expressions" on page 103 in this section
"Boolean Equations" on page 168 in *Design Structure*
"Boolean Expressions" on page 106 in this section
"Using Numbers" on page 18 in *How to Use AHDL*

Arithmetic Expressions

Arithmetic expressions can be used to define evaluated functions in Define Statements, constants in Constant Statements, and as the delimiters of group ranges.

In the following example, a range is defined with an arithmetic expression:

```
SUBDESIGN foo
(
   a[4..2+1-3+8] : INPUT;
)
```

In the following examples, a constant and an evaulated function are defined with arithmetic expressions:

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

DEFINE MIN(a,b) = ((a < b) ? a : b);
```

The arithmetic operators and comparators used in these expressions perform basic arithmetic and comparison operations on the numbers used in the expression. Table 3-3 shows the arithmetic operators and comparators used in AHDL arithmetic expressions:

Table 3-3. Arithmetic Operators and Comparators Used in Arithmetic Expressions (Part 1 of 2)

Operator/ Comparator	Example	Description	Priority
+ (unary)	`+1`	positive	1
- (unary)	`-1`	negative	1
`!`	`!a`	NOT	1
`^`	`a ^ 2`	exponent	1
`MOD`	`4 MOD 2`	modulus	2
`DIV`	`4 DIV 2`	division	2
`*`	`a * 2`	multiplication	2
`LOG2`	`LOG2(4-3)`	logarithm base2	2
`+`	`1+1`	addition	3

Table 3-3. Arithmetic Operators and Comparators Used in Arithmetic Expressions (Part 2 of 2)

Operator/ Comparator	Example	Description	Priority
`-`	`1-1`	subtraction	3
== (numeric)	`5 == 5`	numeric equality	4
== (string)	`"a" = "b"`	string equality	4
`!=`	`5 != 4`	not equal to	4
`>`	`5 > 4`	greater than	4
`>=`	`5 >= 5`	greater than or equal to	4
`<`	`a < b+2`	less than	4
`<=`	`a <= b+2`	less than or equal to	4

`&`
`AND`
	`a & b`
	`a AND b`
		AND	5

`!&`
`NAND`
	`1 !& 0`
	`1 NAND 0`
		NAND (AND inverter)	5

`$`
`XOR`
	`1 $ 1`
	`1 XOR 1`
		XOR (exclusive OR)	6

`!$`
`XNOR`
	`1 !$ 1`
	`1 XNOR 1`
		XNOR (exclusive NOR)	6

`#`
`OR`
	`a # b`
	`a OR b`
		OR	7

`!#`
`NOR`
	`a !# b`
	`a NOR b`
		NOR (OR inverter)	7

`?`	`(5<4) ? 3:4`	ternary	8

The unary plus (+) and minus (–) are prefix operators. The + operator does not affect its operand, but you may use it for documentation purposes (i.e., to indicate a positive number).

☞ The predefined evaluated functions CEIL and FLOOR can also be used in arithmetic expressions. The ceiling (CEIL) of a real number is the smallest integer that is at least that real number; the floor (FLOOR) is the largest integer that is at most that real number. Although these operation apply to all arithmetic expressions, they are meaningful only for LOG2 and DIV operations in which the result can be a real number. The ceiling or floor is obtained by enclosing the expression in parentheses and prepending CEIL or FLOOR to it.

The following examples show the ceiling and floor of a real number:

```
CEIL(LOG2(255)) = 8
FLOOR(LOG2(255)) = 7
```

The following rules apply to all arithmetic expressions:

- Arithmetic expressions must resolve to non-negative numbers.
- When the result of LOG2 is not an integer, the result is automatically rounded up to the next integer. For example, `LOG2(257) = 9`.

☞ The arithmetic operators that are supported in arithmetic expressions are a superset of the arithmetic operators supported in Boolean expressions, which are described in "Arithmetic Operators in Boolean Expressions" on page 109.

Go to "Numbers in AHDL" on page 102 for more information.

Boolean Expressions

Boolean expressions consist of operands that are separated by logical and arithmetic operators and comparators, and are optionally grouped within parentheses. Expressions are used in Boolean equations as well as in other statements such as Case and If Then Statements.

A Boolean expression may be one of the following:

- An operand
  For example: `a, b[5..1], 7, VCC`
- An in-line logic function reference
  For example: `out[15..0] = 16dmux (q[3..0]);`
- A prefix operator (`!` or `-`) applied to a Boolean expression
  For example: `!c`
- Two Boolean expressions separated by a binary (non-prefix) operator
  For example: `d1 $ d3`
- A Boolean expression enclosed in parentheses
  For example: `(!foo & bar)`

The result of every Boolean expression has the same width as its operands.

You can name Boolean operators and comparators in AHDL files to make it easy to enter resource assignments and to interpret the Equations Section of the Report File (**.rpt**). For more information, go to "Naming a Boolean Operator or Comparator" on page 84 in *How to Use AHDL*.

Go to the following topics for more information:

"Arithmetic Operators in Boolean Expressions" on page 109 in this section
"Boolean Operator & Comparator Priorities" on page 112 in this section
"Boolean Equations" on page 168 in *Design Structure*
"Comparators" on page 111
"Implementing Boolean Expressions & Equations" on page 25 in *How to Use AHDL*
"In-Line Logic Function Reference" on page 180 in *Design Structure*
"Logical Operators" on page 107 in this section
"Numbers in AHDL" on page 102 in this section

Logical Operators

Table 3-4 shows logical operators that can be used in Boolean expressions:

Table 3-4. Logical Operators Used in Boolean Expressions

Operator	Example	Description

`!`
`NOT`
	`!tob`
	`NOT tob`
		one's complement (prefix inverter)

`&`
`AND`
	`bread & butter`
	`bread AND butter`
		AND

`!&`
`NAND`
	`a[3..1] !& b[5..3]`
	`a[3..1] NAND b[5..3]`
		AND inverter

`#`
`OR`
	`trick # treat`
	`trick OR treat`
		OR

`!#`
`NOR`
	`c[8..5] !# d[7..4]`
	`c[8..5] NOR d[7..4]`
		OR inverter

`$`
`XOR`
	`foo $ bar`
	`foo XOR bar`
		exclusive OR

`!$`
`XNOR`
	`x2 !$ x4`
	`x2 XNOR x4`
		exclusive NOR

Each operator represents a two-input logic gate, except the `NOT` (`!`) operator, which is a prefix inverter on a single node. You can use either the name or the symbol to represent a logical operator.

Expressions that use these operators are interpreted differently, depending on whether the operands are single nodes, groups, or numbers. Also, expressions with the `NOT` operator are interpreted differently from those with other logical operators.

You can name Boolean operators and comparators in AHDL files to make it easy to enter resource assignments and to interpret the Equations Section of the Report File (**.rpt**). For more information, go to "Naming a Boolean Operator or Comparator" on page 84 in *How to Use AHDL*.

☞ You can allow the Compiler to replace AND operators in Boolean expressions with the `lpm_add_sub` 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 Expressions Using NOT

The NOT operator (`!`) is a prefix inverter. The behavior of the NOT operator depends on the operand that it affects.

Three operand types can be used with the NOT operator:

- If the operand is a single node, GND, or VCC, a single inversion operation is performed. For example, `!a` means that the signal `a` passes through an inverter.
- If the operand is a group of nodes, every member of the group passes through an inverter. For example, the group `!a[4..1]` is interpreted as `(!a4, !a3, !a2, !a1)`.
- If the operand is a number, it is treated as a binary number with as many bits as the group context in which it is used, and every bit is inverted. For example, `!9` in a four-member group context is interpreted as `!B"1001"`, which is the same as `B"0110"`.

Boolean Expressions Using AND, NAND, OR, NOR, XOR, & XNOR

Four operand combinations exist with the binary (non-prefix) operators, and each of these combinations is interpreted differently.

- If both operands are single nodes or the constants GND or VCC, the operator performs the logical operation on the two elements, e.g., `(a & b)`.
- If both operands are groups of nodes, the operator acts upon the corresponding nodes of each group, producing a bitwise set of operations between the groups. The groups must be the same size. For example, `(a, b, c) # (d, e, f)` is interpreted as `(a # d, b # e, c # f)`.
- If one operand is a single node, GND, or VCC, and the other operand is a group of nodes, the single node or constant is duplicated to form a group of the same size as the other operand. The expression is then

treated as a group operation. For example, `a & b[4..1]` is interpreted as `(a & b4, a & b3, a & b2, a & b1)`.

- If both operands are numbers, the shorter number is sign-extended to match the size of the other number. The expression is then treated as a group operation. For example, in the expression `(3 # 8)`, the `3` and `8` are converted to the binary numbers `B"0011"` and `B"1000"`, respectively. The expression then becomes `B"1011"`.

- If one operand is a number and the other is a node or group of nodes, the number is truncated or sign-extended to match the size of the group. If any significant bits are truncated, an error message is generated. The expression is then treated as a group operation. For example, in the expression `(a, b, c) & 1`, the `1` is converted to `B"001"` and the expression becomes `(a, b, c) & (0, 0, 1)`. The expression is then interpreted as `(a & 0, b & 0, c & 1)`.

An expression that uses `VCC` as an operand is interpreted differently from an expression that uses `1` as an operand. In the first equation shown below, the number `1` is sign-extended to match the size of the group. In the second equation, the node `VCC` is duplicated to form a group of the same size. Each equation is then treated as a group operation.

```
(a, b, c) & 1   = (0, 0, c)
(a, b, c) & VCC = (a, b, c)
```

Arithmetic Operators in Boolean Expressions

Arithmetic operators are used to perform arithmetic addition and subtraction operations on groups and numbers in Boolean expressions. Table 3-4 shows the available operators.

Table 3-5. Arithmetic Operators Used in Boolean Expressions

Operator	Example	Description
+ (unary)	`+1`	positive
- (unary)	`-a[4..1]`	negative
+	`count[7..0] + delta[7..0]`	addition
-	`rightmost_x[] - leftmost_x[]`	subtraction

The unary plus (+) and minus (-) are prefix operators. The + operator does not affect its operand, but you may use it for documentation purposes (i.e., to indicate a positive number). The - operator interprets its operand as a binary representation of a number if it is not already a number. It then performs a two's complement unary-minus operation on the operand.

The following rules apply to the other arithmetic operators:

- Operations are performed between two operands, which must be groups of nodes or numbers.
- If both operands are groups of nodes, the groups must be the same size.
- If both operands are numbers, the shorter number is sign-extended to match the size of the other operand.
- If one operand is a number and the other is a group of nodes, the number is truncated or sign-extended to match the size of the group. If any significant bits are truncated, the MAX+PLUS II Compiler generates an error message.

1. When you add two groups together on the right side of a Boolean equation with the + operator, you can place a `0` on the left of each group to sign-extend the width of the group. This method provides an extra bit of information to the group on the left side of the equation that can be used as a carry-out signal.

   In the following example, the groups `count[7..0]` and `delta[7..0]` are sign-extended with zeros to provide information to the `cout` carry-out signal:

   ```
   (cout, answer[7..0]) = (0, count[7..0]) + (0,
      delta[7..0])
   ```

2. You can name Boolean operators and comparators in AHDL files to make it easy to enter resource assignments and to interpret the Equations Section of the Report File (**.rpt**). For more information, go to "Naming a Boolean Operator or Comparator" on page 84 in *How to Use AHDL*.

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

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 3-6. Comparators Used in Boolean Expressions

Comparator:	Example	Description
`==` (logical)	`addr[19..4] == B"B800"`	equal to
`!=` (logical)	`b1 != b3`	not equal to
`<` (arithmetic)	`fame[] < power[]`	less than
`<=` (arithmetic)	`money[] <= power[]`	less than or equal to
`>` (arithmetic)	`love[] > money[]`	greater than
`>=` (arithmetic)	`delta[] >= 0`	greater than or equal to

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

Priority	Operator/Comparator	
1	-	(negative)
1	!	(NOT)
2	+	(addition)
2	-	(subtraction)
3	==	(equal to)
3	!=	(not equal to)
3	<	(less than)
3	<=	(less than or equal to)
3	>	(greater than)
3	>=	(greater than or equal to)
4	&	(AND)
4	!&	(NAND)
5	$	(XOR)
5	!$	(XNOR)
6	#	(OR)
6	!#	(NOR)

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

Primitives

MAX+PLUS II provides a variety of primitive functions for designing circuits for Altera devices.

AHDL TDFs use statements, operators, and keywords in place of certain Graphic Design File (**.gdf**) primitives:

- The `INPUT`, `OUTPUT`, and `BIDIR` ports in AHDL replace the `INPUT`, `OUTPUT`, and `BIDIR` primitives used in GDFs.
- The `AND`, `NAND`, `BAND`, `BNAND`, `OR`, `NOR`, `BOR`, `BNOR`, `XNOR`, `XOR`, and `NOT` logic primitives in GDFs are replaced by logical operators in AHDL.
- The `VCC` and `GND` primitives in GDFs are replaced by `VCC` and `GND` keywords in AHDL.
- The GDF Title Block primitive is replaced by an AHDL Title Statement.
- The GDF `PARAM` and `CONSTANT` primitives are replaced by AHDL Parameters and Constant Statements.

This section provides information about the available primitives, possible interconnections between primitives and ports, descriptions of each primitive, and their AHDL Function Prototypes. The Function Prototypes are not required in TDFs. However, you may redefine the calling order of the primitive inputs with a Function Prototype Statement.

This section discusses the following topics:

- Buffer Primitives 114
- Flipflop & Latch Primitives 125
- Primitive/Port Interconnections 127

Go to the following topics for more information:

"Constant Statement" on page 147 in *Design Structure*
"Function Prototype Statement" on page 151 in *Design Structure*
"Logical Operators" on page 107 in this section
"Parameters Statement" on page 142 in *Design Structure*
"Ports" on page 132 in this section
"Subdesign Section" on page 157 in *Design Structure*
"Title Statement" on page 141 in *Design Structure*

Buffer Primitives

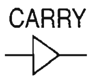
The following buffer primitives are provided:

`CARRY`	=	Carry Buffer
`CASCADE`	=	Cascade Buffer
`EXP`	=	Expander Buffer
`GLOBAL`	=	Global Buffer (`SCLK` is also available for backward compatibility)
`LCELL`	=	Logic Cell Buffer (`MCELL` is also available for backward compatibility)
`OPNDRN`	=	Open Drain Buffer
`SOFT`	=	Soft Buffer
`TRI`	=	Tri-State Buffer

All buffer primitives except `TRI` and `OPNDRN` allow you to control the logic synthesis process. In most circumstances, you do not need to use these buffers; however, if the Compiler indicates that your project is too complex and cannot be processed, you can insert them in parts of the project that cause logic expansion, thus guiding the Logic Synthesizer to produce special results.

For help with using these primitives in your projects, contact Altera Applications. Go to "Contacting Altera" in MAX+PLUS II Help for information on contacting Altera.

CARRY Primitive



Function Prototype:

```
FUNCTION CARRY (in)
   RETURNS (out);
```

The `CARRY` primitive designates the carry-out logic for a function, and acts as the carry-in to another function. The carry function implements fast carry-chain logic for functions such as adders and counters.

The `CARRY` primitive is supported only for the FLEX 8000 and FLEX 10K device families; it is ignored for other devices.

When you use a `CARRY` primitive, you must observe the following rules:

- A `CARRY` primitive can feed one or two cones of logic. If the `CARRY` primitive feeds two cones of logic, then one and only one of the cones of logic must be buffered by another CARRY primitive. In this case, both cones of logic are implemented in the same logic cell. You must follow this rule to tie down the sum and carry-out functions for the first stage of an adder or counter.

- A cone of logic fed by a CARRY primitive can have up to two inputs. A third input is allowed only if it is a CARRY input.
- A cone of logic that feeds a CARRY primitive can have up to two inputs. A third input is allowed only if it is a CARRY input.
- The CARRY primitive cannot feed an OUTPUT or OUTPUTC pin.
- The CARRY primitive cannot be fed by an INPUT or INPUTC pin or a register.
- Two CARRY primitives cannot feed the same gate.

If you use the CARRY primitive incorrectly, it is ignored and the Compiler issues a warning.

You can allow the Compiler to automatically insert or remove CARRY primitives during logic synthesis with the *Carry Chain* logic option or a logic synthesis style that includes the *Carry Chain* option.

Multi-level synthesis may implement a FLEX 8000 or FLEX 10K register in the counter mode even if it is not fed by a CARRY buffer. To prevent a register that is not explicitly fed by a CARRY buffer from using the counter mode, set the *Carry Chain* logic option to IGNORE for that register. If the register is explicitly fed by a CARRY buffer, you must also set the *Carry Chain* logic option to IGNORE on the CARRY buffer.

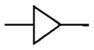
The following example shows a register implemented in counter mode without a carry chain input:

```
q   = dff ( (eqn & load # data & !load ) & clear,
      ... ); ↵
eqn = q & !ena # !q & ena; ↵
```

Go to "Assigning an Individual Logic Option or Synthesis Style" in MAX+PLUS II Help using **Search for Help on** for more information.

CASCADE Primitive

CASCADE



Function Prototype:

```
FUNCTION CASCADE (in)
    RETURNS (out);
```

The CASCADE buffer designates the cascade-out function from an AND or OR gate, and acts as a cascade-in to another AND or OR gate. The cascade-in function allows a cascade, which is a fast output located on each combinatorial logic cell, to be ORed or ANDed with the output of an adjacent combinatorial logic cell in the device. With the CASCADE primitive, the AND or OR gate that feeds the CASCADE primitive and the AND or OR gate that is fed by the CASCADE primitive are placed in the device, with the first symbol logically ORed or ANDed into the second.

The CASCADE primitive is supported only for the FLEX 8000 and FLEX 10K device families; it is ignored for other devices.

When you use a CASCADE primitive, you must observe the following rules:

- A CASCADE primitive can only feed or be fed by a single gate, which must be an AND or an OR gate.

  An inverted OR gate is treated as an AND gate and vice-versa. Logical equivalents of AND gates are BAND, BNAND, and NOR. Logical equivalents of OR gates are BOR, BNOR, and NAND.

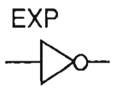
- Two CASCADE primitives cannot feed the same gate.
- A CASCADE primitive cannot feed an XOR gate.
- A CASCADE primitive cannot feed an OUTPUT or OUTPUTC pin primitive or a register.
- The De Morgan's inversion theorem implementation of cascaded AND and OR gates requires all primitives in a cascaded chain to be of the same type. A cascaded AND gate cannot feed a cascaded OR gate, and vice-versa.

If you use the CASCADE primitive incorrectly, it is ignored and the Compiler issues a warning.

You can allow the Compiler to automatically insert or remove CASCADE primitives during logic synthesis with the *Cascade Chain* logic option or a logic synthesis style that includes the *Cascade Chain* logic option.

Go to "Assigning an Individual Logic Option or Synthesis Style" in MAX+PLUS II Help using **Search for Help on** for more information.

EXP Primitive



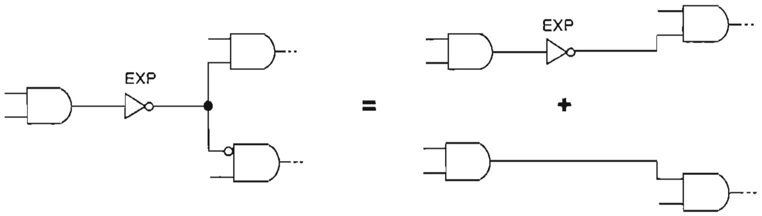
Function Prototype:

```
FUNCTION EXP (in)
   RETURNS (out);
```

The EXP expander buffer specifies that an expander product term is desired in the project. The expander product is inverted in the device.

☞ The EXP primitive is supported only for the MAX 5000, MAX 7000, and MAX 9000 device families; it is treated as a NOT gate in other device families. Refer to individual device data sheets for information on how specific devices use logic cells and expander product terms.

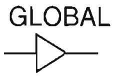
Whether or not an expander product term is used depends on the logic polarity required by the destination functions. For example, if an EXP buffer feeds two AND gates (i.e., product terms) and the second AND gate has a inverted input, the EXP feeding the inverted input is removed during logic synthesis, thereby creating positive logic. The EXP feeding the non-inverted input is not removed and the expander product term is used to implement logic, as shown in the illustration below. (Normally, the Logic Synthesizer determines where to insert or remove EXP buffers. Altera recommends that only experienced MAX+PLUS II designers should use the EXP primitive in their projects.)



In devices that contain multiple Logic Array Blocks (LABs), the EXP buffer output can only feed logic within a single LAB. The EXP is duplicated for each LAB that requires it. If a project contains a large number of expanders, the Logic Synthesizer may convert them into LCELL buffers to balance expander product term and logic cell usage.

☞ Do not use EXP primitives to create an intentional delay or asynchronous pulse. The delay of these elements varies with temperature, power supply voltage, and device fabrication process, so race conditions can occur and create an unreliable circuit.

GLOBAL Primitive



Function Prototype:

```
FUNCTION GLOBAL (in)
    RETURNS (out);
```

The `GLOBAL` buffer indicates that a signal must use a global (synchronous) Clock, Clear, Preset, or Output Enable signal, instead of signals generated with internal logic or driven by ordinary I/O pins. Table 3-8 shows how global signals are used in different device families.

1. The `SCLK` buffer is available for backward compatibility and can be used in place of a `GLOBAL` primitive only to specify a global Clock in Classic and MAX 5000 devices. `SCLK` may be used to request global clocking of a register when the Clock is driven by a pin. A direct connection must exist from the input pin to `SCLK` to the register.

2. The `GLOBAL` primitive is ignored for devices that do not support it. In addition, global and array clocking are not allowed within the same Logic Array Block (LAB) in MAX 5000 devices.

Table 3-8. Global Signal Availability

Device Family	Global Clock	Global Clear	Global Preset	Global Output Enable
Classic	✓			
MAX 5000	✓	✓ *(1)*	✓ *(1)*	✓ *(1)*
MAX 7000	✓	✓		✓
FLEX 8000	✓	✓	✓	✓
MAX 9000	✓	✓		✓
FLEX 10K	✓	✓	✓	✓

Note:
(1) Available for EPS464 devices only.

If an input pin feeds directly to the input of GLOBAL, the output of GLOBAL can be used to feed a Clock, Clear, Preset, or Output Enable input to a primitive. A direct connection must exist from the output of GLOBAL to the input of the register or the TRI buffer. A NOT gate may be required between the input pin and GLOBAL when the GLOBAL buffer feeds the Output Enable input of a TRI buffer.

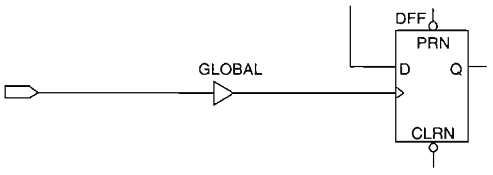
A single input may pass through GLOBAL before feeding the Clock, Clear, or Preset input of a register, or the Output Enable input of a TRI buffer.

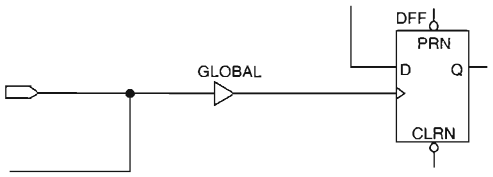
Global signals propagate more quickly than array signals and may free up device resources for other logic. GLOBAL should be used to implement global clocking in a portion or all of the project. To verify that registers are globally clocked, you can refer to the Report File for the processed project.

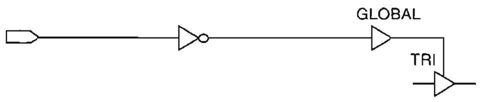
If your MAX 5000 device project contains global and array (asynchronous) clocking and the Fitter module of the Compiler cannot find a fit, removing the GLOBAL buffer may make a fit possible. If you encounter a similar problem with a MAX 7000 project, replace array clocking with global clocking.

As an alternative to using the GLOBAL primitive, you can direct the Compiler to automatically select an existing signal in a project to be a global Clock, Clear, Preset, or Output Enable signal with the **Global Project Logic Synthesis** command (Assign menu).

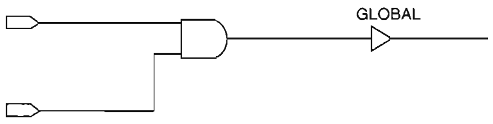
The following illustration shows some legal uses of the GLOBAL buffer:





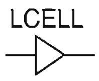


The following illustration shows an illegal use of the GLOBAL buffer:



Go to "Clock Configuration Guidelines," "Preset & Clear Configuration Guidelines," and "Master Reset Guidelines" using **Search for Help on** (Help menu).

LCELL Primitive



Function Prototype:

```
FUNCTION LCELL (in)
   RETURNS (out);
```

The LCELL buffer allocates a logic cell for the project. The LCELL buffer produces the true and the complement of a logic function and makes both available to all logic in the device. (The output of the LCELL buffer must feed through a NOT gate to use the complement of the logic function.)

The MCELL buffer, which has the same functionality as the LCELL buffer, is available for backward compatibility with earlier versions of MAX+PLUS II. New projects should use LCELL exclusively.

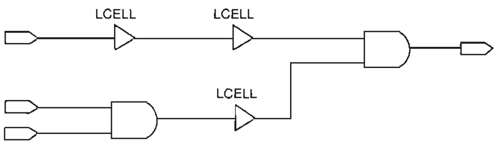
An LCELL buffer always consumes one logic cell. It is not removed from a project during logic synthesis.

Do not use LCELL primitives to create an intentional delay or asynchronous pulse. The delay of these elements varies with temperature, power supply voltage, and device fabrication process, so race conditions can occur and create an unreliable circuit.

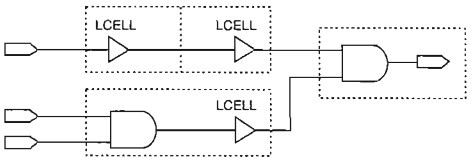
The following illustrations show the effect of logic synthesis on a project when LCELL or SOFT buffers are used. The first project (1) requires four logic cells after synthesis (in dotted boxes) and three input pins. If the LCELL buffers are replaced with SOFT buffers, as shown in (2), the SOFT buffers are removed by logic synthesis and the project requires one logic cell and three inputs.

(1)

Before Logic Synthesis



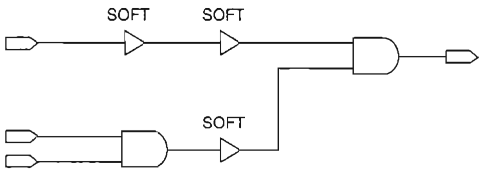
After Logic Synthesis



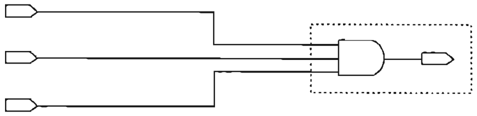
LCELL forces a buffer

(2)

Before Logic Synthesis



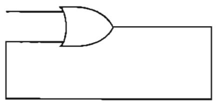
After Logic Synthesis



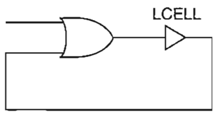
If the *Delay Chains* option is turned on in the **Design Doctor Settings** dialog box (Processing menu), the Compiler issues a warning message for any series of LCELL or EXP primitives used to create an intentional delay or asynchronous pulse.

In standard logic synthesis, combinatorial feedback from a logic cell to itself is illegal unless an LCELL buffer is used. The Logic Synthesizer detects illegal combinatorial feedback and issues an error message when a project is compiled with standard synthesis. If you use multi-level synthesis, the Logic Synthesizer automatically inserts an LCELL buffer.

Illegal



Legal

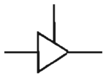


MAX+PLUS II includes several logic options that automatically insert LCELL and SOFT buffers during project compilation. In addition, you can use the Compiler's **Fitter Settings** command (Processing menu) to direct the Compiler to automatically insert LCELL buffers into a project if they are needed to achieve a fit when user-defined resource and device assignments would otherwise prevent a project from fitting.

Go to "Assigning an Individual Logic Option or Synthesis Style" and "Delay Chain Guidelines" using **Search for Help on** (Help menu).

OPNDRN Primitive

OPNDRN



Function Prototype:
```
FUNCTION OPNDRN (in)
   RETURNS (out);
```

The OPNDRN primitive is similar to a TRI primitive, with a single input and a single output. The OPNDRN primitive is equivalent to a TRI primitive whose Output Enable input is fed by any signal, but whose primary input is fed by a GND primitive.

If the input to the OPNDRN primitive is low, the output will be low. If the input is high, the output will be a high-impedance logic level.

☞ The OPNDRN primitive is supported only for the FLEX 10K device family; it is converted to a TRI primitive for other devices.

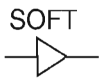
If you turn on the *Automatic Open-Drain Pins* option in the **Global Project Logic Synthesis** dialog box (Assign menu) for a FLEX 10K-based project, the Compiler converts the following structures to the OPNDRN primitive:

- A TRI primitive whose Output Enable input is fed by any signal, but whose primary input is fed by a GND primitive
- A TRI primitive whose Output Enable input is fed by the complement of its primary input.

When you use an OPNDRN buffer, you must observe the following rules:

- An OPNDRN buffer may drive only one BIDIR or BIDIRC pin.
- If an OPNDRN buffer feeds logic, it must also feed a BIDIR or BIDIRC pin. If it feeds a BIDIR or BIDIRC pin, it may not feed any other outputs.

SOFT Primitive



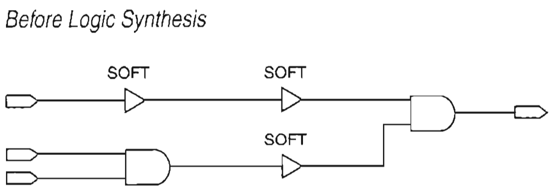
Function Prototype:

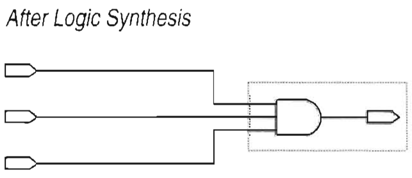
```
FUNCTION SOFT (in)
   RETURNS (out);
```

The SOFT buffer specifies that a logic cell may be needed in the project. During project processing, the Logic Synthesizer examines the logic feeding the primitive and determines whether a logic cell is needed. If it is needed, the SOFT buffer is converted into an LCELL; if not, the SOFT buffer is removed.

The following illustrations show the effect of logic synthesis on a project that uses SOFT buffers. In the first project (1), the Logic Synthesizer removes the SOFT buffers, and the project uses one logic cell. In the second project (2), it removes one SOFT buffer and converts the other into an LCELL buffer. This LCELL buffer reduces the complexity of the project, i.e., the number of product terms. The second project also uses one logic cell.

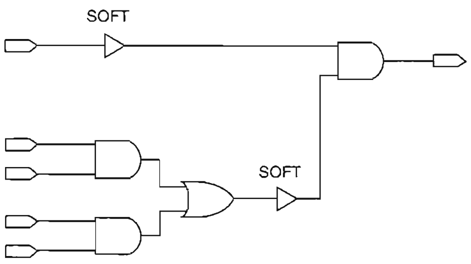
(1)



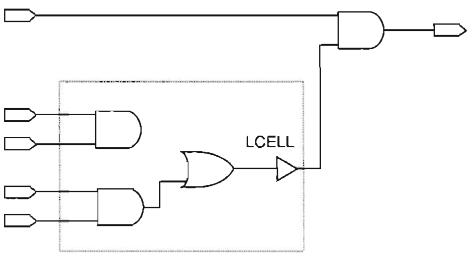


(2)

Before Logic Synthesis

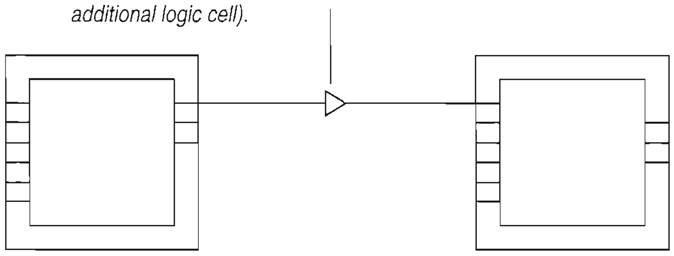


After Logic Synthesis



If the Compiler indicates that the project is too complex, you can edit the project by inserting `SOFT` buffers to prevent logic expansion. For example, you can add a `SOFT` buffer at the combinatorial output of a logic function to decouple two combinatorial circuits:

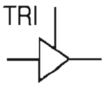
A SOFT buffer inserted between two combinatorial circuits prevents logic expansion (although it may consume one additional logic cell).



MAX+PLUS II includes logic options that automatically insert or ignore `SOFT` and `LCELL` buffers during project compilation.

Go to "Assigning an Individual Logic Option or Synthesis Style" and "Delay Chain Guidelines" using **Search for Help on** (Help menu) in MAX+PLUS II Help.

TRI Primitive



Function Prototype:
```
FUNCTION TRI (in, oe)
    RETURNS (out);
```

The TRI primitive is a tri-state buffer with an input, output, and Output Enable signal. If the Output Enable signal is high, the output will be driven by the input.

The Output Enable defaults to VCC.

If the Output Enable of a TRI buffer is connected to VCC or a logic function that will minimize to true, a TRI buffer may be converted into a SOFT buffer during logic synthesis.

When you use a TRI buffer, you must observe the following rules:

- A TRI buffer may drive only one BIDIR or BIDIRC pin. You must use a BIDIR or BIDIRC pin if feedback is included after the TRI buffer.
- If a TRI buffer feeds logic, it must also feed a BIDIR or BIDIRC pin. If it feeds a BIDIR or BIDIRC pin, it may not feed any other outputs.

Flipflop & Latch Primitives

Table 3-9 lists the MAX+PLUS II flipflop and latch primitives and their Function Prototypes. All flipflops are positive-edge-triggered; latches are level-sensitive.

☞ When the Latch or Clock Enable (ena) input is high, the flipflop or latch passes the signal from the data input(s) to q. When the ena input is low, the state of q is maintained, regardless of the data input(s).

For devices that do not support Latch and/or Clock Enable, logic synthesis generates logic equations containing flipflops with Clock Enables and latches with Latch Enables. These logic equations correctly emulate the logic specified in your project.

Table 3-9. MAX+PLUS II Flipflops & Latches

Primitive	AHDL Function Prototype
`LATCH`	`FUNCTION LATCH (d, ena) RETURNS (q);`
`DFF`	`FUNCTION DFF (d, clk, clrn, prn) RETURNS (q);`
`DFFE`	`FUNCTION DFFE (d, clk, clrn, prn, ena) RETURNS (q);`
`JKFF`	`FUNCTION JKFF (j, k, clk, clrn, prn) RETURNS (q);`
`JKFFE`	`FUNCTION JKFFE (j, k, clk, clrn, prn, ena) RETURNS (q);`
`SRFF`	`FUNCTION SRFF (s, r, clk, clrn, prn) RETURNS (q);`
`SRFFE`	`FUNCTION SRFFE (s, r, clk, clrn, prn, ena) RETURNS (q);`
`TFF`	`FUNCTION TFF (t, clk, clrn, prn) RETURNS (q);`
`TFFE`	`FUNCTION TFFE (t, clk, clrn, prn, ena) RETURNS (q);`

Notes:

`clk`	=	Register Clock Input
`clrn`	=	Clear Input
`d, j, k, r, s, t`	=	Data input from Logic Array
`ena`	=	Latch Enable or Clock Enable Input
`prn`	=	Preset Input
`q`	=	Output

Primitive/Port Interconnections

Not all primitives and ports may connect to all other primitives and ports in a design file. Table 3-10 shows the possible interconnections for all AHDL primitives and ports.

Table 3-10. Primitive/Port Interconnections

Source	Destination											
	`OUTPUT`	`BIDIR`	`TRI` *(1)*	`GLOBAL`	`LCELL`	`EXP`	`SOFT`	Logic	`INPUT` *(2)*	`CARRY`	`CASCADE`	`OPNDRN`
`INPUT`	Y	N	Y	Y	na	na	na	Y	Y	na	na	Y
`OUTPUT` *(3)*	N	N	N	N	N	N	N	N	N	na	na	N
`BIDIR`	N	N	Y	N	na	na	na	Y	Y	na	na	Y
`TRI`	Y	Y	N *(4)*	na	N *(4)*	N *(4)*	N *(4)*	N *(4)*	N *(4)*	na	na	N *(4)*
`GLOBAL` *(5)*	na	N	*(6)*	Y	na	na	na	na	na	Y	na	
`LCELL` *(7)*	Y	N	Y	na	na	na	na	Y	na	na	na	Y
`EXP`	na	N	na	N	na	na	na	Y	na	na	na	na
`SOFT`	Y	N	na	N	na	na	na	Y	na	na	na	na
`VCC`	Y	N	Y	N	na	na	na	Y	Y	na	na	Y
`GND`	Y	N	Y	N	na	na	na	Y	Y	na	na	Y
Logic	Y	N	Y	N	Y	Y	Y	Y	Y	Y	Y	Y
Register Output	Y	N	Y	N	na	na	Y	Y	Y	na	na	Y
`CARRY`	na	na	na	na	Y	na	na	Y	na	na	na	na
`CASCADE`	na	na	na	na	na	na	na	Y	na	na	na	na
`OPNDRN`	Y	Y	N *(4)*	na	N *(4)*	N *(4)*	N *(4)*	N *(4)*	N *(4)*	na	na	N *(4)*

Notes:

(1) Includes both data and Output Enable inputs to `TRI`.
(2) The `INPUT` (or `IN`) port can only be fed by device pins or higher levels in the hierarchy.
(3) The `OUTPUT` (or `OUT`), `OUTPUTC`, `BIDIR` (or `INOUT`), and `BIDIRC` primitives/ports can only drive out to device pins or higher levels in the hierarchy.
(4) These connections change to legal (Y) or not advisable (na) only if the output of the `TRI` or `OPNDRN` is also connected to a `BIDIR` (or `INOUT`) or `BIDIRC` primitive/port.
(5) `SCLK` in MAX+PLUS (DOS) designs is interpreted as `GLOBAL`. New projects should only use `GLOBAL`.
(6) Connecting a `GLOBAL` output to the `TRI` Output Enable input is legal in FLEX 8000 and FLEX 10K devices. Connections to Output Enable in other devices are ignored.
(7) `MCELL` in pre-version 3.0 MAX+PLUS II design files is interpreted as `LCELL`. New projects should only use `LCELL`.

Y	Interconnection is legal.
N	Interconnection is illegal.
na	Interconnection is legal but not advisable or may implement logic inefficiently.

Table 3-11 shows the possible connections to registers for primitives and ports.

Table 3-11. Primitive/Port to Register Connections

Source	Register			
	INPUT *(1)*	CLK	PRN	CLRN
INPUT	Y	Y	Y	Y
OUTPUT *(2)*	N	N	N	N
BIDIR *(2)*	Y	Y	Y	Y
TRI	N *(3)*	N *(3)*	N *(3)*	N *(3)*
GLOBAL *(4)*	na	Y	Y	Y
LCELL *(5)*	na	Y	Y	Y
EXP	na	na	na	na
SOFT	na	Y	Y	Y
VCC	Y	N	Y	Y
GND	Y	N	Y	Y
Logic	Y	Y	Y	Y
Register Output	Y	Y	Y	Y
CARRY	na	na	na	na
CASCADE	na	na	na	na
OPNDRN	N *(3)*	N *(3)*	N *(3)*	N *(3)*

Notes:

(1) The INPUT (or IN) port can only be fed by device pins or higher levels in the hierarchy.
(2) The OUTPUT (or OUT), OUTPUTC, BIDIR (or INOUT), and BIDIRC primitives/ports can only drive out to device pins or higher levels in the hierarchy.
(3) These connections change to legal (Y) or not advisable (na) only if the output of the TRI or OPNDRN is also connected to a BIDIR (or INOUT) or BIDIRC primitive/port.
(4) SCLK in MAX+PLUS (DOS) designs is interpreted as GLOBAL. New projects should only use GLOBAL.
(5) MCELL in pre-version 3.0 MAX+PLUS II design files is interpreted as LCELL. New projects should only use LCELL.
Y Interconnection is legal.
N Interconnection is illegal.
na Interconnection is legal but not advisable or may implement logic inefficiently.

Megafunctions

MAX+PLUS II megafunctions are a collection of complex logic functions, including Library of Parameterized Modules (LPM) functions, that can be used in logic designs. These megafunctions are particularly efficient in optimizing logic functions for Altera devices. The MAX+PLUS II installation procedure automatically installs these megafunctions in the **\maxplus2\max2lib\mega_lpm** directory. This directory also contains an Include File (**.inc**) with a Function Prototype for each megafunction. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

You may use these megafunctions freely in all MAX+PLUS II logic designs. When the MAX+PLUS II Compiler analyzes a logic circuit, it automatically removes all unused gates and flipflops, thereby ensuring that design efficiency is not reduced.

Table 3-12 describes all MAX+PLUS II megafunctions. Names of LPM functions that are currently included in the LPM 2.0.1/2.1.0 standard start with "`lpm_`." Detailed information on these megafunctions is available in MAX+PLUS II Help.

Table 3-12. MAX+PLUS II Megafunctions (Part 1 of 2)

Type	Name	Description
Gates	`lpm_and`	Parameterized AND Gate
	`lpm_bustri`	Parameterized Tri-State Buffer
	`lpm_clshift`	Parameterized Combinatorial Shifter Module
	`lpm_constant`	Parameterized Constant Generator Module
	`lpm_decode`	Parameterized Decoder Module
	`lpm_inv`	Parameterized Inverter Module
	`lpm_mux`	Parameterized Multiplexer Module
	`lpm_or`	Parameterized OR Gate
	`lpm_xor`	Parameterized XOR Gate

Table 3-12. MAX+PLUS II Megafunctions (Part 2 of 2)

Type	Name	Description
Arithmetic Components	`lpm_abs`	Parameterized Absolute Value
	`lpm_add_sub`	Parameterized Adder/Subtractor Module
	`lpm_decode`	Parameterized Comparator Module
	`lpm_counter`	Parameterized Counter Module
	`lpm_mult`	Parameterized Multiplier Module
Storage Components	`lpm_dff`	Parameterized D-Type Flipflop and Shift Register Module
	`lpm_latch`	Parameterized Latch Module
	`lpm_ram_dq`	Random Access Memory with Separate Input and Output Ports
	`lpm_ram_io`	Random Access Memory with a Single I/O Port
	`lpm_rom`	Read-Only Memory
	`lpm_tff`	Parameterized T-Type Flipflop Module
	`csdpram`	Cycle-Shared Dual-Port Random Access Memory
	`csfifo`	Cycle-Shared FIFO
Other Functions	`a6502`	6502 Microprocessor
	`ntsc`	NTSC Video Control Signal Generator
	`pll`	Rising- and Falling-Edge Detector

Choose **Megafunctions/LPM** (Help menu) in MAX+PLUS II for detailed information about megafunctions. Go to "Implementing a Hierarchical Project" on page 69 in *How to Use AHDL* for information on using megafunctions.

Old-Style Macrofunctions

MAX+PLUS II old-style macrofunctions are a collection of high-level building blocks that can be used in logic designs. The MAX+PLUS II installation procedure automatically installs these macrofunctions in the **\maxplus2\max2lib** directory and its subdirectories. The **\maxplus2\max2inc** directory, which is also installed automatically, contains an Include File (**.inc**) with a Function Prototype for each macrofunction. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

You may use these macrofunctions freely in all MAX+PLUS II logic designs. When the MAX+PLUS II Compiler analyzes a logic circuit, it automatically removes all unused gates and flipflops, thereby ensuring that design efficiency is not reduced. All input ports also have default signal values, so that unused inputs can simply be left unconnected.

Choose **Old-Style Macrofunctions** (Help menu) in MAX+PLUS II for detailed information about old-style macrofunctions. Go to "Implementing a Hierarchical Project" on page 69 in *How to Use AHDL* for information on using old-style macrofunctions.

Ports

A port is an input or output of a logic function. A port can appear in two locations:

- A port that is an input or output of the current file is declared in the Subdesign Section.
- A port that is an input or output of an instance of a primitive or lower-level design file is used in the Logic Section.

Ports of the Current File

A port that is an input or output of the current file is declared in the following format within the Subdesign Section:

<port name> : *<port type>* [= *<default port value>*]

The following port types are available:

```
INPUT          MACHINE INPUT
OUTPUT         MACHINE OUTPUT
BIDIR
```

When a TDF is the top-level file in a hierarchy, the port name is synonymous with a pin name. The optional default port value, which is either `VCC` or `GND`, can be specified for `INPUT` and `BIDIR` port types. This default value is used only if the port is left unconnected when an instance of the TDF is used in a higher-level design file.

In the following example, the input, output, and bidirectional ports of the file are declared in the Subdesign Section:

```
SUBDESIGN top
(
   foo, bar, clk1, clk2, c[4..0][6..0]    : INPUT = VCC;
   % VCC is default port value %
   a0, a1, a2, a3, a4                     : OUTPUT;
   b[7..0]                                : BIDIR;
)
```

You can import and export state machines between TDFs and other design files by specifying an input or output port as MACHINE INPUT or MACHINE OUTPUT in the Subdesign Section. The Function Prototype that represents the file must indicate which ports are state machines. MACHINE INPUT and MACHINE OUTPUT ports can only be used in lower-level files in a project hierarchy.

Ports of Instances

A port that is an input or output of an instance of a logic function is connected in the Logic Section. To connect a logic function to other portions of a TDF, you insert an instance of the function with an in-line reference or Instance Declaration or declare a state machine with a State Machine Declaration, and then use ports of the function in the Logic Section.

If you use an in-line reference with positional port association to create an instance of a logic function, the order of the ports, not the names, is important. The order of ports is defined in the Function Prototype for the function.

If you use an Instance Declaration or an in-line reference with named port association to create an instance of a logic function, the names of the ports, not their order, are important.

In the following example, an instance of a D flipflop is declared as the variable `reg` in the Variable Section, then used in the Logic Section:

```
VARIABLE
    reg : DFF;
BEGIN
    reg.clk = clock
    reg.d   = data_input
    output  = reg.q
END;
```

Port names can thus be connected to other nodes in the Logic Section in the following format:

<instance name>.*<port name>* = *<node name>*

The *<instance name>* is a user-defined name for a function. The *<port name>* is identical to the port name that is declared as an input or output of the file in the Subdesign Section of a lower-level TDF, or to a pin name in another type of design file. This *<port name>* is synonymous with the pinstub name for the symbol that represents an instance of the design file in a Graphic Design File (**.gdf**).

As illustrated by the example above, if you use an Instance Declaration to create an instance of a logic function, the names of the ports in the design file that defines the logic function is important. The same is true of the right-hand sides of in-line logic function references that use named port association. (The left-hand side of all in-line references use positional port association.) The following example shows the Function Prototype for the `21mux` macrofunction and an in-line reference that uses named port association:

```
FUNCTION 21MUX (s, a, b)
   RETURNS (y);
.
.
.
BEGIN
   output = 21MUX (.s = select, .b = dataB, .a = dataA);
END;
```

The nodes `output`, `select`, `dataA`, and `dataB` are connected to the `y`, `s`, `a`, and `b` ports of the `21mux` macrofunction. Thus, within an in-line reference that uses named port association, the ports on the right-hand side of the equals symbol (=) are listed in the following format:

.*<port name>* = *<node name>*

In contrast, if you use an in-line reference with positional port association to create an instance of a logic function, the order of the nodes listed in the in-line reference, not the port names of the instantiated logic function, is important. The order of ports is defined in the Function Prototype for the function. The following example shows an in-line reference for the same `21mux` macrofunction that uses positional port association:

```
BEGIN
   output = 21MUX (select, dataA, dataB);
END;
```

All Altera-provided logic functions have predefined port (pinstub) names, which are shown in the Function Prototype. Commonly used primitive port names are shown in Figure 3-13.

Table 3-13. Commonly Used Ports

Port Name:	Definition:
`.q`	Output of a flipflop or latch
`.d`	Data input to a D flipflop or latch
`.t`	Toggle input to a T flipflop
`.j`	J input to a JK flipflop
`.k`	K input to a JK flipflop
`.s`	Set input to an SR flipflop
`.r`	Reset input to an SR flipflop
`.clk`	Clock input to a flipflop
`.ena`	Clock Enable input to a flipflop, Latch Enable input to a latch, or Enable input to a state machine
`.prn`	Active-low Preset input to a flipflop
`.clrn`	Active-low Clear input to a flipflop
`.reset`	Active-high Reset input to a state machine
`.oe`	Output Enable input to a `TRI` primitive
`.in`	Primary input to `CARRY`, `CASCADE`, `EXP`, `TRI`, `OPNDRN`, `SOFT`, `GLOBAL`, and `LCELL` primitives
`.out`	Output of `CARRY`, `CASCADE`, `EXP`, `TRI`, `OPNDRN`, `SOFT`, `GLOBAL`, and `LCELL` primitives

Go to the following topics for more information:

"Machine Alias Declaration" on page 166 in *Design Structure*
"Port Syntax" in MAX+PLUS II Help
"Quoted & Unquoted Names" on page 97 in this section
"Subdesign Section" on page 157 in *Design Structure*

Parameters

Attributes of a megafunction or macrofunction that determine the logic created or used to implement the function, i.e., characteristics that determine the size, behavior, or silicon implementation of a function. For example, parameters are often used to define the width of a bus.

A parameterized function is a function whose behavior is controlled by one or more parameters. Some logic functions, such as the megafunctions in the Library of Parameterized Modules (LPM), are inherently parameterized and require parameter values to be assigned. Parameters can also optionally be assigned to some functions that are not inherently parameterized, such as old-style macrofunctions, to determine their style of implementation.

When you use an existing parameterized function, such as an LPM function, you can customize the parameters used and assign parameter values on an instance-by-instance basis. In a GDF, you can customize an instance (i.e., symbol) with the Graphic Editor's **Edit Ports/Parameters** command (Symbol menu). In an AHDL TDF, you can declare parameters and assign values when you create an instance with an Instance Declaration or an in-line reference.

Parameter values are not necessarily specified on an instance-by-instance basis. Because parameter values can be inherited from higher hierarchical levels a hierarchical project, the Compiler searches for parameter values in the following parameter value search order:

1. As part of the instance of the logic function. For example, in a TDF, in an instance that is created in an Instance Declaration or an in-line reference, you can declare which parameters are used and optionally assign their values. In a GDF, you can select a symbol and use the **Edit Ports/Parameters** dialog box (Symbol menu) to assign parameter values for that instance.

2. As part of the instance of the logic function at the next higher hierarchy level. The parameter values for an instance of a logic function apply to the subdesigns of that logic function if the subdesign instances do not have assigned parameter values.

3. In the global project default parameter values specified with the **Global Project Parameters** dialog box (Assign menu). These values are stored in the Assignment & Configuration file (**.acf**) for the project.

4. In the optional default value listed in the Parameters Statement(s) of the TDF or the `PARAM` primitives of the GDF that defines the logic function. These default values apply only to the file in which they are listed, they are not applied the file's subdesigns.

When you create a parameterized design file, you can specify the parameters used within that file and optional default parameter values (which are used only if no parameter values are specified elsewhere). In a GDF, you specify the parameters used within the current file with `PARAM` primitives; in a TDF, the parameters used within the current file are specified in a Parameters Statement. Once you create a parameterized design file, you can use the **Create Default Include File** and **Create Default Symbol** commands (File menu) to create default AHDL Function Prototypes (in Include Files) and symbols (in Symbol Files), respectively, that include the names (but not the values) of parameters used within the file. You can edit the parameters and parameter values for a Symbol File with the Symbol Editor's **Enter Parameters** command (Element menu). These parameter names and values then appear as the defaults for each instance of the symbol when it is first entered in a GDF. Once you enter the symbol in a GDF, these default parameters and values can be customized with **Edit Ports/Parameters** on an instance-by-instance basis.

MAX+PLUS II allows you to assign global, project-wide default values for parameters with the **Global Project Parameters** command (Assign menu). As an alternative to using **Global Project Parameters**, you can specify default parameter settings in the Global Project Parameters Section of the ACF.

The following guidelines apply to parameters:

- All logic options can be assigned as parameters for individual instances of mega- or macrofunctions. A logic option that is assigned to a logic function instance as a parameter overrides the global project default synthesis style—which is specified with **Global Project Logic Synthesis** (Assign menu)—for that instance. However, if an instance has the same logic option assigned both as a parameter and as an individual logic option, the logic option setting overrides the parameter setting. In addition, logic options cannot be assigned as global, project-wide default parameter values with **Global Project Parameters**.

- You cannot assign a value to the predefined Altera parameter `DEVICE_FAMILY`, which represents the device family assigned for the project. However, you can use the parameter value in comparisons.

The legal values are `FLEX10K`, `FLEX8000`, `MAX9000`, `MAX7000E`, `MAX7000`, `MAX5000`, `CLASSIC`, and `EP330/EP320`.

- The predefined Altera `LATENCY` parameter can be assigned to an instance of a mega- or macrofunction. However, the parameter applies only to that instance, and is not inherited by the subdesigns of that instance.
- Parameters appear on the top right corner of a symbol in the Graphic or Symbol Editor if **Show Parameters** (Options menu) is turned on. (**Show All** also displays or hides all parameters in the current GDF.)

Double-clicking Button 1 on a parameter opens the **Edit Ports/Parameters** or **Enter Parameters** dialog box in the Graphic and Symbol Editors, respectively. **Show Parameters** or **Show All** displays or hides all parameters in the current GDF. You can print a Graphic or Symbol Editor file that shows parameters by turning on **Show Parameters** (or **Show All**) before printing the file.

Go to the following sources for more information:

"Customizing a Mega- or Macrofunction's Ports & Parameters" in MAX+PLUS II Help
"Entering a Parameter" in MAX+PLUS II Help
"Parameters Statement" on page 142 in *Design Structure*
"Showing Parameters and Probe & Resource Assignments" in MAX+PLUS II Help
"Specifying Global Project Parameters" in MAX+PLUS II Help

Section

4

Design Structure

This section describes basic AHDL design structure. AHDL sections and statements are described in the order in which they appear in a Text Design File (**.tdf**).

- Overview 140
- Title Statement 141
- Parameters Statement 142
- Include Statement 145
- Constant Statement 147
- Define Statement 149
- Function Prototype Statement 151
- Options Statement 154
- Assert Statement 155
- Subdesign Section 157
- Variable Section 159
- Logic Section 168

Go to MAX+PLUS II Help for complete and up-to-date information on AHDL design structure.

Overview

An AHDL TDF must contain, at a minimum, a Subdesign Section and a Logic Section. All other sections and statements are optional. In this section, information is provided in the order in which the statements and sections appear in the TDF.

For information on how to create an AHDL design, go to *How to Use AHDL* on page 17.

For information on recommended file structure, go to "Text Design File Structure" on page 4 in *Introduction.*

For information on the syntax of AHDL sections and statements, choose **AHDL** (Help menu), then click Button 1 on "Syntax."

For information on AHDL syntax that is no longer supported, go to "Obsolete AHDL Statements" in MAX+PLUS II Help using **Search for Help on**.

Title Statement

The Title Statement allows you to provide documentary comments for the Report File (**.rpt**) generated by the Compiler. The following example shows a Title Statement:

```
TITLE "Display Controller";
```

The Title Statement has the following characteristics:

- A Title Statement begins with the keyword `TITLE`, followed by a text string enclosed in double quotation marks (`"`). The statement ends with a semicolon (`;`).

- If a Title Statement is included in a TDF, the title appears at the top of the Report File. In the example shown above, the title `Display Controller` appears in the Report File.

Title Statements must conform to the following rules:

- The string can contain a maximum of 255 characters and may not contain end-of-line or end-of-file characters. To include quotation marks in the title, use two quotation marks. For example:
  `TITLE """EPM5130"" Display Controller";`

- The Title Statement can only be used once in a TDF.

- The Title Statement must be placed outside of all other AHDL sections.

Go to "Title Statement Syntax" in MAX+PLUS II Help for more information.

Parameters Statement

The Parameters Statement allows you to declare one or more parameters that control the implementation of a parameterized megafunction or macrofunction. The following example shows a Parameters Statement:

```
PARAMETERS
(
   FILENAME = "myfile.mif", -- optional default value follows "=" sign
   WIDTH,
   AD_WIDTH = 8,
   NUMWORDS = 2^AD_WIDTH
);
```

The Parameters Statement has the following characteristics:

- A Parameters Statement begins with the keyword `PARAMETERS`, followed by a list of one or more parameters and optional default values, enclosed in parentheses `()`.
- Parameters in the parameter list are separated by commas (`,`); parameter names are separated from optional default values by an equals symbol (=). In the example shown above, only the `WIDTH` parameter does not have a default value.
- Parameter names can be user-defined symbolic names or predefined Altera parameters.
- Parameter values can consist of text strings enclosed in double quotation marks (`"`), which are evaluated as strings. When parameter values are unquoted, the Compiler attempts to treat them as arithmetic expressions; failing that, they are treated as strings.
- The statement ends with a semicolon (`;`).
- Once a parameter has been defined, you can use it throughout the TDF.

Parameter Statements must conform to the following rules:

- A parameter can only be used after it is declared.
- Each parameter name must be unique.
- The parameter name cannot contain spaces. Use underscores to separate words and improve readability.
- The Parameters Statement can be used any number of times in a TDF.
- The Parameters Statement must be placed outside of all other AHDL sections.
- Parameters used in the definition of other parameters must already be defined.
- Circular references are not allowed. The following example shows a circular reference:

```
PARAMETERS
(
   FOO = BAR;
   BAR = FOO;
);
```

When a project is compiled, the Compiler searches for parameter values in the the following order:

1. As part of the instance of the logic function. For example, in a TDF, in an instance that is created in an Instance Declaration or an in-line reference, you can declare which parameters are used and optionally assign their values. In a GDF, you can select a symbol and use the **Edit Ports/Parameters** command (Symbol menu) to assign parameter values for that instance.

2. As part of the instance of the logic function at the next higher hierarchy level. The parameter values for an instance of a logic function apply to the subdesigns of that logic function if the subdesign instances do not have assigned parameter values.

3. In the global project default parameter values specified with the **Global Project Parameters** command (Assign menu). These values are stored in the Assignment & Configuration file (**.acf**) for the project.

4. In the optional default value listed in the Parameters Statement(s) of the TDF or the `PARAM` primitives of the GDF that defines the logic function. These default values apply only to the file in which they are listed, they are not applied the file's subdesigns.

Go to the following sources for more information:

"Parameters Statement Syntax" in MAX+PLUS II Help
"Using Altera-Provided Parameterized Functions" on page 73 in *How to Use AHDL*

Include Statement

The Include Statement allows you to import text from an Include File (**.inc**) into the current file. The following example shows an Include Statement:

```
INCLUDE "const.inc";
```

The Include Statement has the following characteristics:

- The Include Statement begins with the keyword `INCLUDE`, followed by the name of the file to be included, enclosed in double quotation marks (`"`).
- If you do not specify a filename extension, the Compiler assumes the extension **.inc**.
- The statement ends with a semicolon (`;`).
- When the Compiler processes the project, the text from the Include File is substituted for the Include Statement. In the example shown above, the file **const.inc** is substituted for the text `INCLUDE "const.inc";`.

Include Statements are often used to include Function Prototypes for a lower-level design file in a TDF. To use a megafunction or macrofunction, you must first define its logic in a design file. You must then use a Function Prototype Statement to specify the ports of the function. Alternatively, you can use Include Statements to include Function Prototypes that are saved in Include Files. You can then insert an instance of the logic function with an Instance Declaration or an in-line reference.

You can automatically generate an Include File that contains a Function Prototype for a design file with **Create Default Include File** (File menu).

When you compile a file, the Compiler searches your computer's directories for Include Files in the following order:

1. The project directory.
2. Any user libraries specified with **User Libraries** (Options menu).
3. The **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories created during installation.

If you change a TDF that includes an Include File, you can use **Project Save & Check** (File menu) or fully recompile the project to update the view of the project's hierarchy tree that is displayed in the Hierarchy Display window.

Include Statements must conform to the following rules:

- The filename specified in the Include Statement cannot contain a path name.
- In the workstation environment, filenames are case-sensitive. In MAX+PLUS II documentation, filenames may be listed in upper- or lowercase letters. However, the case of the filename in an Include Statement must match the case of the Include File name. Altera-provided macrofunction and megafunction design files all have lowercase filenames; therefore, their corresponding Include Files list function names with lowercase letters.
- An Include Statement must be placed outside of all other AHDL sections.
- An Include Statement can appear any number of times in a TDF.

Include Files must conform to the following rules:

- Names of Include Files must have the extension **.inc**.
- Include Files should contain only Function Prototype, Define, Parameters, or Constant Statements.
- Include Files cannot contain Subdesign Sections.
- Include Files cannot be nested.

Go to the following sources for more information:

"Creating a Default Include File" in MAX+PLUS II Help
"Implementing a Hierarchical Project" on page 69 in *How to Use AHDL*
"Include Statement Syntax" in MAX+PLUS II Help

Constant Statement

The Constant Statement allows you to substitute a meaningful symbolic name for a number or an arithmetic expression. The symbolic name simply represents that number. The following examples show Constant Statements:

```
CONSTANT UPPER_LIMIT = 130;

CONSTANT BAR = 1 + 2 DIV 3 + LOG2(256);

CONSTANT FOO = 1;
CONSTANT FOO_PLUS_ONE = FOO + 1;
```

The Constant Statement has the following characteristics:

- The Constant Statement begins with the keyword `CONSTANT`, followed by a symbolic name, an equals symbol (=), and a number (including a radix, if necessary) or an arithmetic expression.
- The statement ends with a semicolon (`;`).
- Once a constant is declared, you can use it to represent the number throughout the TDF. In the example shown above, you can use `UPPER_LIMIT` in the Logic Section to represent the decimal number `130`.
- Constants can be declared as arithmetic expressions. These arithmetic expressions can include previously defined constants.

  ☞ The Compiler evaluates arithmetic expressions in Constant Statements and reduces them to numerical values. No logic is generated for these expressions.

Constant Statements must conform to the following rules:

- A constant can only be used after it is declared.
- Each constant name must be unique.
- The constant name cannot contain spaces. Use underscores to separate words and improve readability.
- The Constant Statement can be used any number of times in a TDF.

- The Constant Statement must be placed outside of all other AHDL sections.
- Constants used in the definition of other constants must already be defined.
- Circular references are not allowed. The following example shows a circular reference:

```
CONSTANT FOO = BAR;
CONSTANT BAR = FOO;
```

Go to the following sources for more information:

"Constant Statement Syntax" in MAX+PLUS II Help
"Define Statement" on page 149 in this section
"Numbers in AHDL" on page 102 in *Elements*
"Using Constants & Evaluated Functions" on page 19 in *How to Use AHDL*

Define Statement

The Define Statement allows you to define an evaluated function, which is a mathematical function that returns a value that is based on optional arguments.

The following example defines the evaluated function MAX, which ensures that the Subdesign Section declares at least one port.

```
DEFINE MAX(a,b) = (a > b) ? a : b;
SUBDESIGN
(
   dataa[MAX(WIDTH,0)..0]: INPUT;
   datab[MAX(WIDTH,0)..0]: OUTPUT;
)
BEGIN
   datab[] = dataa[];
END;
```

The Define Statement has the following characteristics:

- The Define Statement begins with the keyword DEFINE, followed by a symbolic name and a list of one or more arguments enclosed in parentheses ().
- Arguments in the argument list are separated by commas (,). An equals symbol (=) separates the argument list from an arithmetic expression.

  ☞ 1. If no arguments are listed, an evaluated function behaves as a constant.

  2. The Compiler evaluates arithmetic expressions in Define Statements and reduces them to numerical values. No logic is generated for these expressions.
- The statement ends with a semicolon (;).
- Once an evaluated function has been defined, you can use it throughout the TDF.

- Evaluated functions can be defined in terms of previously defined evaluated functions. For example, the following `MIN_ARRAY_BOUND` function is based on the `MAX` function defined above:

  ```
  DEFINE MIN_ARRAY_BOUND(x) = MAX(0,x) + 1;
  ```

Define Statements must conform to the following rules:

- An evaluated function can only be used after it has been defined.
- Each evaluated function must be unique.
- The evaluated function name cannot contain spaces. Use underscores to separate "words" and improve readability.
- The Define Statement can be used any number of times in a TDF.
- The Define Statement must be placed outside of all other AHDL sections.

Go to the following sources for more information:

"Constant Statement" on page 147 in this section
"Define Statement Syntax" in MAX+PLUS II Help
"Using Constants & Evaluated Functions" on page 19 in *How to Use AHDL*
"Using Numbers" on page 18 in *How to Use AHDL*

Function Prototype Statement

Function Prototype Statements have the same function as symbols in schematic design files. Both provide a shorthand description of a logic function, listing its name and its input, output, and bidirectional ports. Machine ports can also be used for functions that import or export state machines.

However, megafunction and macrofunction input port default values are not automatically assigned as they are in MAX+PLUS II Graphic Editor files; you must assign them explicitly in the Subdesign Section of a TDF. You can also assign a default value for bidirectional ports in the Subdesign Section. However, output ports cannot be assigned a default value.

When you wish to implement an instance of a mega- or macrofunction, you must ensure that its logic is defined in its own design file. You then use a Function Prototype Statement to specify the ports of the function, and implement an instance of the function with an in-line reference or an Instance Declaration.

The following examples show Function Prototype Statements. The first is for a parameterized function; the second is for an unparameterized function:

```
FUNCTION lpm_add_sub (cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-
   1..0], add_sub)
   WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDERTYPE,
      ONE_INPUT_IS_CONSTANT)
   RETURNS (result[LPM_WIDTH-1..0], cout, overflow);

FUNCTION compare (a[3..0], b[3..0])
    RETURNS (less, equal, greater);
```

The Function Prototype Statement has the following characteristics:

- The keyword `FUNCTION` is followed by the name of the function. In the examples shown above, the function names are `lpm_add_sub` and `compare`.

- A list of input ports to the function follows the name. In the first example shown above, the input ports are `cin`, `dataa[LPM_WIDTH-1..0]`, and `datab[LPM_WIDTH-1..0]`; in the second, they are `a3`, `a2`, `a1`, `a0`, `b3`, `b2`, `b1`, and `b0`.

- In a parameterized function, the keyword WITH and a parameter name list follow the input port list. The list is enclosed in parentheses (); the individual parameter names are separated by commas (,).

- The keyword RETURNS is followed by a list of output and bidirectional ports of the function. In the first example shown above, the output ports are result,[LPM_WIDTH-1..0], cout, and overflow; in the second, they are less, equal, and greater.

- Both the input and output lists are enclosed in parentheses; the individual port names are separated by commas.

- When you import or export a state machine, the Function Prototype for the file must use a machine port (identified by the MACHINE keyword) to indicate which inputs and outputs are state machines. For example:

```
FUNCTION ss_def (clock, reset, count)
   RETURNS (MACHINE ss_out);
```

- The Function Prototype Statement ends with a semicolon (;).

- A Function Prototype Statement must be placed outside of the Subdesign Section in a TDF, and it must be placed before the logic function is instantiated in an in-line reference or Instance Declaration.

To implement an instance of a primitive, you also use an in-line reference or an Instance Declaration. However, in contrast to mega- and macrofunctions, primitive logic is predefined, so you do not need to define the primitive logic in a separate design file. In addition, you do not need to use a Function Prototype Statement unless you wish to change the order of the primitive inputs.

The following example shows the default Function Prototype for a JKFF primitive:

```
FUNCTION JKFF (j, k, clk, clrn, prn)
   RETURNS (q);
```

The following example shows a modified Function Prototype for a JKFF primitive:

```
FUNCTION JKFF (k, j, clk, clrn, prn)
   RETURNS (q);
```

As an alternative to using a Function Prototype Statement in a file, you can use an Include Statement to call an Include File (**.inc**) that contains a Function Prototype Statement. MAX+PLUS II also provides the **Create Default Include File** command (File menu), which automatically creates an Include File containing a Function Prototype for any design file.

Function Prototypes for all MAX+PLUS II megafunctions and macrofunctions are stored in Include Files in the **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories, respectively. On-line help for all megafunctions, macrofunctions, and primitives shows the Function Prototype for each Altera-provided function. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

Go to the following sources for more information:

"Creating a Default Include File" in MAX+PLUS II Help
"Function Prototype Statement Syntax" in MAX+PLUS II Help
"Implementing a Hierarchical Project" on page 69 in *How to Use AHDL*
"Ports" on page 132 in *Elements*

Options Statement

The Options Statement sets the BITO option to specify whether the lowest numbered bit of a group will be the most significant bit (MSB), the least significant bit (LSB) or either, depending on its location.

The Options Statement begins with the keyword OPTIONS, followed by the BITO option and setting. The Options Statement ends with a semicolon (;).

The following example shows an Options Statement:

```
OPTIONS BIT0 = MSB;
```

In this example, the lowest numbered bit of a group is specified as the MSB. The other settings available are LSB and ANY.

An Options Statement at the beginning of a TDF sets the default bit-ordering for the entire file. If the file is a top-level TDF, the Options Statement applies to the entire project. If the file is lower in the project hierarchy, the Options Statement specifies the bit-ordering only for that file.

Go to the following sources for more information:

"Defining Groups" on page 28 in *How to Use AHDL*
"Options Statement Syntax" in MAX+PLUS II Help

Assert Statement

The Assert Statement allows you to test the validity of any arbitrary expression that uses parameters, numbers, evaluated functions, or the used or unused status of a port.

The following example shows an Assert Statement:

```
ASSERT (WIDTH > 0)
REPORT   Width (%) must be a positive integer" WIDTH
   SEVERITY    ERROR
   HELP_ID     INTVALUE; -- for internal Altera use only
```

The Assert Statement has the following characteristics:

- The keyword `ASSERT` is followed by an arithmetic expression that is optionally enclosed in parentheses `()`. When the expression is false, the assertion is activated and the message string following the `REPORT` keyword is displayed in the Message Processor. If you do not specify a condition, the assertion is always activated.

- The `REPORT` keyword is followed by a message string and optional message variables. The message string is enclosed in double quotation marks (`"`), and can include `%` characters that are substituted with the values of optional message variables. If no `REPORT` keyword is used, an assertion that is activated displays a generic message of the following format in the Message Processor:

  <severity>: `Line` *<line number>*, `File` *<filename>*: `Assertion failed`

- Optional message variables consist of one or more parameters, evaluated functions, or arithmetic expressions. Multiple message variables are separated by commas (`,`). The values of the message variables are substituted, in order, for the `%` characters in the quoted message string. In the example shown above, the value of `WIDTH` is substituted for the `%` in the quoted message string.

- The optional `SEVERITY` keyword is followed by a severity level of `ERROR`, `WARNING`, or `INFO`. If no severity is specified, it defaults to `ERROR`.

- The `HELP_ID` keyword and help string are used in some Altera-provided logic functions and are reserved for internal Altera use.

- The statement ends with a semicolon (;).
- The Assert Statement can be used within the Logic Section or outside of any other AHDL section.

Go to the following sources for more information:

"Assert Statement Syntax" in MAX+PLUS II Help
"Naming a Boolean Operator or Comparator" on page 84 in *How to Use AHDL*

Subdesign Section

The Subdesign Section declares the input, output, and bidirectional ports of the TDF.

The following example shows a Subdesign Section:

```
SUBDESIGN top
(
   foo, bar, clk1, clk2  : INPUT = VCC;
   a0, a1, a2, a3, a4    : OUTPUT;
   b[7..0]               : BIDIR;
)
```

The Subdesign Section has the following characteristics:

- The keyword `SUBDESIGN` is followed by the subdesign name. The subdesign name must be the same as the TDF filename. In this example, the subdesign name is `top`.
- The list of signals is enclosed in parentheses `()`.
- Signal names are represented by symbolic names such as `foo`, and are assigned a port type such as `INPUT`.
- Signal names are separated by commas (`,`), are followed by a colon (`:`) and a port type, and end with a semicolon (`;`).
- The port type may be `INPUT`, `OUTPUT`, `BIDIR`, `MACHINE INPUT`, or `MACHINE OUTPUT`. In the example shown above, the `foo`, `bar`, `clk1`, and `clk2` signals are inputs and `a0`, `a1`, `a2`, `a3`, and `a4` are outputs. The bus `b[7..0]` is bidirectional.
- The `MACHINE INPUT` and `MACHINE OUTPUT` keywords are used to import and export state machines between TDFs and other design files. However, `MACHINE INPUT` and `MACHINE OUTPUT` port types cannot be used in a top-level TDF.
- You can optionally assign a default value of `GND` or `VCC` after the port type (otherwise, no default value is assumed). In the example shown above, `VCC` is the default value for the input signals unless they are assigned in a higher-level file (assignments in a higher-level file take precedence).

In a top-level design file, `INPUT`, `OUTPUT`, and `BIDIR` port types represent actual device pins. In a lower-level design file, all port types are the inputs and outputs of the file, but not of the project itself.

Go to the following sources for more information:

"Importing & Exporting State Machines" on page 77 in *How to Use AHDL*
"Ports" on page 132 in *Elements*
"Subdesign Section Syntax" in MAX+PLUS II Help

Variable Section

The optional Variable Section is used to declare and/or generate any variables used in the Logic Section. AHDL variables are similar to variables in a high-level programming language; they are used to define buried (internal) logic.

The following example shows a Variable Section:

```
VARIABLE
   a, b, c      : NODE;
   temp         : halfadd;
   tsnode       : TRI_STATE_NODE;
   IF DEVICE_FAMILY == "FLEX8000" GENERATE
      8kadder      : flex_adder;
      d,e          : NODE;
   ELSE GENERATE
      7kadder      : pterm_adder;
      f,g          : NODE;
   END GENERATE;
```

The Variable Section can include one or more of the following statements or constructs:

- Instance Declaration 160
- Node Declaration 162
- Register Declaration 163
- State Machine Declaration 165
- Machine Alias Declaration 166

The Variable Section can also contain If Generate Statements, which can be used to generate Instance, Node, Register, State Machine, and Machine Alias Declarations.

The Variable Section has the following characteristics:

- The keyword `VARIABLE` begins the Variable Section.

- User-defined, symbolic variable names are separated from each other by commas (`,`) and from the variable type by a colon (`:`). The variable type can be `NODE`, `TRI_STATE_NODE`, <*primitive*>, <*megafunction*>, <*macrofunction*>, or <*state machine declaration*>. In the example shown above, the internal variables are `a`, `b`, and `c` of type `NODE`; `temp`, an instance of the macrofunction `halfadd`; and `tsnode`, an instance of type `TRI_STATE_NODE`.

- Each entry in the list of variables ends with a semicolon (;).

Compiler-generated names that contain the tilde (~) character may appear in the Fit File (**.fit**) for a project. If you back-annotate the Fit File assignments, these names will then appear in the project's Assignment & Configuration File (**.acf**). The tilde character is reserved for Compiler-generated names only; you cannot use it in your own pin, node, and group (bus) names.

Go to the following sources for more information:

"If Generate Statement" on page 178 in this section
"Variable Section Syntax" in MAX+PLUS II Help

Instance Declaration

Each individual usage, or instance, of a particular logic function can be declared as a variable with an Instance Declaration in the Variable Section. After it is declared, you can use the input and output ports of each logic function as ports in the Logic Section.

When you wish to implement an instance of a megafunction or macrofunction, you must ensure that its logic is defined in its own design file. You then use a Function Prototype Statement to specify the ports and parameters of the function, and implement an instance of the function with an in-line reference or an Instance Declaration.

To implement an instance of a primitive, you also use an in-line reference or an Instance Declaration. However, in contrast to mega- and macrofunctions, primitive logic is predefined, so you do not need to define the primitive logic in a separate design file. In most cases, a Function Prototype Statement is not needed. See "Function Prototype Statement" on page 151 for more information.

To use an Instance Declaration, you declare a variable of type *<primitive>*, *<megafunction>*, or *<macrofunction>* in the Variable Section. For a parameterized mega- or macrofunction, the declaration includes a list of the parameters used by the instance and optional parameter values. Once you declare the variable, you can use ports of the instance of the function in the following format:

<instance name> . *<port name>*

For example, if you wish to incorporate the `compare` and `adder` functions (taken from the example in "Function Prototype Statement" on page 151) into your current TDF, make the following Instance Declarations in the Variable Section:

```
VARIABLE
   comp  : compare;
   adder : lpm_add_sub WITH (LPM_WIDTH = 8)
```

The variables `comp` and `adder` are instances of the functions `compare` and `lpm_add_sub`, which have the following inputs and outputs:

```
a[3..0], b[3..0]        : INPUT;       -- inputs to compare
less, equal, greater    : OUTPUT;      -- outputs of compare

a[8..1], b[8..1]        : INPUT;       -- inputs of adder
sum[8..1]               : OUTPUT;      -- outputs of adder
```

You can therefore use the following ports of `comp` and `adder` in the current Logic Section:

```
comp.a[], comp.b[], comp.less, comp.equal, comp.greater

adder.dataa[], adder.datab[], adder.result[]
```

These ports can be used in any behavioral statement in the same way as nodes.

Since all primitives have only one output, you can use the name of a primitive without a port name (e.g., without `.q` or `.out`) on the right side of an equation if you want to use its output. Similarly, for all primitives that have a single primary input (i.e., all primitives except `JKFF`, `JKFFE`, `SRFF`, and `SRFFE`), you can use the name of a primitive without a port name (e.g., without `.d`, `.t`, or `.in`) on the left side of an equation to connect the primitive to its primary input. See "Register Declaration" on page 163 for more information.

When MAX+PLUS II compiles a project, the Compiler searches for parameter values for each instance of a mega- or macrofunction in the parameter value search order described on page 142.

Go to the following sources for more information:

"Creating a Default Include File" in MAX+PLUS II Help
"If Generate Statement" on page 178 in this section
"Implementing a Hierarchical Project" on page 69 in *How to Use AHDL*

"Primitives," "Megafunctions," "Old-Style Macrofunctions," and "Ports" beginning on page 113 in *Elements*
"Variable Section Syntax" in MAX+PLUS II Help

Node Declaration

AHDL supports two types of nodes: NODE and TRI_STATE_NODE.

Both types are all-purpose variable types used to store signals that have not been declared in the Subdesign Section or elsewhere in the Variable Section. Therefore, a variable of either type can be used on the left or right side of an equation.

Both NODE and TRI_STATE_NODE are similar to the INPUT, OUTPUT, and BIDIR port types of the Subdesign Section, in that they represent a single wire that propagates signals.

☞ Compiler-generated names that contain the tilde (~) character may appear in the Fit File (**.fit**) for a project. If you back-annotate the Fit File assignments, these names will then appear in the project's Assignment & Configuration File (**.acf**). The tilde character is reserved for Compiler-generated names only; you cannot use it in your own pin, node, and group (bus) names.

The following example shows a Node Declaration:

```
SUBDESIGN node_ex
(
   a, oe  : INPUT;
   b      : OUTPUT;
   c      : BIDIR;
)

VARIABLE
   b      : NODE;
   t      : TRI_STATE_NODE;
BEGIN
   b = a;
   out = b   % therefore out = a %
   t = TRI(a, oe);
   t = c;   % t is bus of c and tri_stated a %
END;
```

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.

The following primitives and signals can feed TRI_STATE_NODE nodes:

- TRI primitives
- INPUT ports from a design file at a higher hierarchical level
- OUTPUT and BIDIR ports from a design file at a lower hierarchical level
- BIDIR ports of the current file
- Other nodes declared as TRI_STATE_NODE types in the current file

Go to the following sources for more information:

"Declaring Nodes" on page 27 in *How to Use AHDL*
"If Generate Statement" on page 178 in this section
"Implementing Tri-State Buses" in *How to Use AHDL*
"Variable Section Syntax" in MAX+PLUS II Help

Register Declaration

A Register Declaration is used to declare registers, including D, T, JK, and SR flipflops (DFF, DFFE, TFF, TFFE, JKFF, JKFFE, SRFF, and SRFFE) and latches (LATCH). The following example shows a Register Declaration:

```
VARIABLE
   ff : TFF;
```

The name of this instance of a T flipflop is ff. After making this declaration, you can use the input and output ports of the instance of ff in the format *<instance name>*.*<port name>*:

```
ff.t
ff.clk
ff.clrn
ff.prn
ff.q
```

Since all primitives have only one output, you can use the name of an instance of a primitive without appending a port name (e.g., without `.q` or `.out`) on the right side of an equation if you want to use its output. Similarly, for all primitives that have a single primary input, i.e., all primitives except `JKFF`, `JKFFE`, `SRFF`, and `SRFFE`, you can use the name of an instance of a primitive without a port name (e.g., without `.d`, `.t`, or `.in`) on the left side of an equation to connect the primitive to its primary input.

For example, the `DFF` Function Prototype is `FUNCTION DFF(d, clk, clrn, prn) RETURNS (q);`. In the following TDF excerpt, `a = b` is equivalent to `a.d = b.q`:

```
VARIABLE
   a, b : DFF;
BEGIN
   a = b;
END;
```

Go to the following sources for more information:

"Declaring Registered Outputs" on page 50 in *How to Use AHDL*
"Declaring Registers" on page 47 in *How to Use AHDL*
"If Generate Statement" on page 178 in this section
"Ports" on page 132 and "Primitives" on page 113 in *Elements*
"Variable Section Syntax" in MAX+PLUS II Help

State Machine Declaration

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

The following example shows a State Machine Declaration:

```
VARIABLE
   ss :  MACHINE
         OF BITS (q1, q2, q3)
         WITH STATES (
            s1 = B"000",
            s2 = B"010",
            s3 = B"111");
```

The state machine name is `ss`. The state bits `q1`, `q2`, and `q3` are outputs of registers for this machine. The states of this state machine are `s1`, `s2`, and `s3`, each of which is assigned a numerical state value for the state bits `q1`, `q2`, and `q3`.

A State Machine Declaration has the following characteristics:

- The state machine name is a symbolic name. In the example shown above, the state machine name is `ss`.
- The state machine name is followed by a colon (`:`) and the keyword `MACHINE`.
- The State Machine Declaration must include a list of states, and can include a list of state bit names.
- Optional state bits are specified with the keywords `OF BITS`, followed by a comma-separated list of symbolic names; the list must be enclosed in parentheses `()`. The example shown above specifies the state bits `q1`, `q2`, and `q3`.
- States are specified by the keywords `WITH STATES`, followed by a comma-separated list of symbolic names; the list must also be enclosed in parentheses. The example shown above specifies the states `s1`, `s2`, and `s3`.
- The first state listed in the `WITH STATES` clause is the Reset state for the state machine.

- The state names may be optionally assigned to a value with an equals symbol (=) followed by a numerical value. In the example shown above, `s1` is assigned to `B"000"`, `s2` is assigned to `B"010"`, and `s3` is assigned to `B"111"`.

- You can use a Machine Alias Declaration, as described below, to assign an alternate name to a state machine that is declared in the current file or imported from another file.

- A semicolon (`;`) ends a State Machine Declaration.

Each state of a state machine is represented by a unique pattern of high and low flipflop output signals. The state bits are the flipflops required by the machine to store the states. The number of states has the following relationship to the number of state bits in a state machine:

<number of states> <= 2^*<number of state bits>*

Go to the following sources for more information:

"Assigning State Machine Bits & Values" on page 58 in *How to Use AHDL*
"If Generate Statement" on page 178 in this section
"Importing & Exporting State Machines" on page 77 in *How to Use AHDL*
"Numbers in AHDL" on page 102 in *Elements*
"Recovering From Illegal States" on page 66 in *How to Use AHDL*
"Instance Declaration Syntax" and "Variable Section Syntax" in MAX+PLUS II Help
"State Machines" on page 54 in *How to Use AHDL*
"Variable Statement Syntax" in MAX+PLUS II Help

Machine Alias Declaration

You can rename a state machine with a temporary name using a Machine Alias Declaration in the Variable Section. You can use a machine alias in the file where the state machine is created, or in a file that uses a `MACHINE INPUT` port to import a state machine. You can then use this name instead of the original state machine name. For example:

```
FUNCTION ss_def (clock, reset, count)
   RETURNS (MACHINE ss_out);
•
•
•
VARIABLE
   ss : MACHINE;
BEGIN
   ss = ss_def (sys_clk, reset, !hold);

   IF ss == s0 THEN
         •
         •
         •
   ELSIF ss == s1 THEN
         •
         •
         •
END;
```

A Machine Alias Declaration has the following characteristics:

- The machine alias is a symbolic name. It is followed by a colon (`:`) and the keyword `MACHINE`. In the example shown above, `ss` is the machine alias.
- You can import and export state machines between TDFs and other design files by specifying an input or output port as `MACHINE INPUT` or `MACHINE OUTPUT` in the Subdesign Section.
- When you import or export a state machine, the Function Prototype that represents the file must indicate which inputs and outputs are state machines. In the example shown above, `ss_out` is the state machine name.
- A semicolon (`;`) ends a Machine Alias Declaration.

`MACHINE INPUT` and `MACHINE OUTPUT` port types cannot be used in a top-level TDF.

Go to the following sources for more information:

"Importing & Exporting State Machines" on page 77 in *How to Use AHDL*
"Variable Section Syntax" in MAX+PLUS II Help

Logic Section

The Logic Section specifies the logical operations of the TDF and is the body of a TDF. This section is required. One or more of the following statements or constructs may be used in this section:

- Boolean Equations....168
- Boolean Control Equations....171
- Case Statement....172
- Defaults Statement....173
- If Then Statement....176
- If Generate Statement....178
- For Generate Statement....179
- Truth Table Statement....183

☞ The Logic Section can also include Assert Statements. Go to "Assert Statement" on page 155 for more information.

The `BEGIN` and `END` keywords enclose the Logic Section. A semicolon (`;`) follows the `END` keyword and terminates this section. The Defaults Statement must be the first statement in the section.

AHDL is a concurrent language. The Compiler evaluates all behavior specified in the Logic Section of a TDF at the same time rather than sequentially. Equations that assign multiple values to the same AHDL node of type `NODE` or variable are logically ORed. See "Defaults Statement" on page 173 for more information.

Boolean Equations

Boolean equations are used in the Logic Section of your AHDL TDF to represent the connection of nodes, the flow of inputs into and the flow of outputs from input and output pins, primitives, megafunctions, macrofunctions, and state machines.

The following example shows a complex Boolean equation:

```
a[] = ((c[] & -B"001101") + e[6..1]) # (p, q, r, s, t, v);
```

The left side of the equation can be a symbolic, port, or group name. You can use the `NOT (!)` operator to invert any item on the left. The right side of the equation consists of a Boolean expression, which is evaluated as described in "Boolean Operator & Comparator Priorities" on page 112 in *Elements.*

The equals symbol (=) is used in Boolean equations to indicate that the result of the Boolean expression on the right side is the source of the symbolic node or group on the left side. The single equals symbol differs from the double equals symbol (==), which is used as a comparator.

In the example shown above, the Boolean expression on the right is evaluated according to the Boolean equation priority rules:

1. The binary number `B"001101"` is negated and becomes `B"110011"`. The unary minus (-) has first priority.
2. `B"110011"` is anded (`&`) with the group `c[]`. This expression has second priority because it is enclosed in parentheses.
3. The result of the group expression in step 2 is added to the group `e[6..1]`.
4. The result of the expression in step 3 is ORed (#) with the group `(p, q, r, s, t, v)`. This expression has last priority.

The final result is assigned to the group `a[]`.

For the sample equation shown above to be legal, 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 rules apply to Boolean equations:

- Multiple assignments to a variable are logically ORed (#), except when the default for the variable is VCC.

- If the number of nodes on the left side of the Boolean equation equals the number of nodes on the right, a one-to-one correspondence exists.

- If a single node, GND, or VCC on the right side of an equation is assigned to a group, the node or constant is duplicated to match the size of the group. For example: `(a, b) = e;` is the same as `a = e; b = e;`

- If both the left and right sides of the equation are groups of the same size, each member on the right is assigned to the member on the left that corresponds in position. For example: `(a, b) = (c, d);` is the same as `a = c; b = d;`

☞ When you add two groups together on the right side of a Boolean equation with the + operator, you can place a `0` on the left of each group to sign-extend the width of the group. This method provides an extra bit of information to the group on the left side of the equation that can be used as a carry-out signal. In the following example, the groups `count[7..0]` and `delta[7..0]` are sign-extended with zeros to provide information to the `cout` carry-out signal:

```
cout, answer[7..0]) = (0, count[7..0]) + (0, delta[7..0])
```

- If the left and right sides of an equation have groups of different sizes, the number of bits in the group on the left must be evenly divisible by the number of bits in the group on the right. 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
```

- A group of nodes or numbers cannot be assigned to a single node.

- If a number on the right side of an equation is assigned to a group, the number is truncated or sign-extended to match the size of the group. If any significant bits are truncated, the Compiler issues an error message. Each member on the right is assigned to the member on the left with the corresponding position. For example, `(a, b) = 1;` is the same as `a = 0; b = 1;`

- Commas can be used to hold the places of unassigned group members in a Boolean equation. The following example shows commas that hold the places of two members of the group `(a, b, c, d)`:

```
(a, , c, ) = B"1011";
```

  In this example, both `a` and `c` are assigned the value 1.

- A semicolon (`;`) ends each equation.

Go to the following sources for more information:

"Boolean Equation Syntax" in MAX+PLUS II Help
"Boolean Expressions" on page 106 in *Elements*
"Boolean Operator & Comparator Priorities" on page 112 in *Elements*
"Defaults Statement" on page 173 in this section
"Implementing Boolean Expressions & Equations" on page 25 in *How to Use AHDL*

Boolean Control Equations

Control equations are Boolean equations used in the Logic Section to set up the state machine Clock, Reset, and Clock Enable signals.

The following examples show Boolean control equations:

```
ss.clk = clk1;
ss.reset = a & b;
ss.ena = clk1ena;
```

Boolean control equations have the following characteristics:

- You can define the Clock, Reset, and Clock Enable inputs of each state machine in the format *<state machine name>*`.`*<port name>*. In the example above, these inputs are defined for the state machine `ss`.
- You can use the state machine name declared in the State Machine Declaration as the state machine name in the control equations.
- The Clock signal *<state machine name>*`.clk` must always be assigned a value.
- If the start state of the state machine has been assigned a non-zero value, then the Reset signal *<state machine name>*`.reset` assignment is required; otherwise, it is optional.
- Assigning the Clock Enable signal *<state machine name>*`.ena` to a value is always optional.
- A semicolon (`;`) ends each equation.

Go to the following sources for more information:

"Boolean Control Equation Syntax" in MAX+PLUS II Help
"Setting Clock, Reset & Enable Signals" on page 57 in *How to Use AHDL*
"State Machines" on page 54 in *How to Use AHDL*

Case Statement

The Case Statement lists the alternatives that may be activated depending on the value of the variable, group, or expression following the `CASE` keyword.

The following example shows a Case Statement:

```
CASE f[].q IS
   WHEN H"00" =>
      addr[] = 0;
      s = a & b;
   WHEN H"01" =>
      count[].d = count[].q + 1;
   WHEN H"02", H"03", H"04" =>
      f[3..0].d = addr[4..1];
   WHEN OTHERS =>
      f[].d = f[].q;
END CASE;
```

The Case Statement has the following characteristics:

- The keywords `CASE` and `IS` enclose a Boolean expression, group, or state machine (in the example shown above, `f[].q`).
- The Case Statement is terminated by the keywords `END CASE` and a semicolon (`;`).
- One or more unique alternatives are listed in the `WHEN` clauses in the body of the Case Statement. Each `WHEN` clause begins with the keyword `WHEN`.
- In each alternative `WHEN` clause, one or more comma-separated constant values are followed by an arrow symbol (=>). In this example, the `H"02"`, `H"03"`, and `H"04"` constant values are listed in a single `WHEN` clause; the `H"00"` and `H"01"` constant values are listed in separate `WHEN` clauses.

- If the Boolean expression following the `CASE` keyword evaluates to a specific alternative, all the behavioral statements following the arrow are activated. In the example shown above, if `f[].q` evaluates to `H"01"`, the Boolean equation `count[].d = count[].q + 1` is activated.

- When no other alternative is true, the optional keywords `WHEN OTHERS` define the default alternative. In the example shown above, if `f[].q` does not equal `H"00"`, `H"01"`, or `H"CF"`, the Boolean equation `f[].d = f[].q` is activated.

- The Defaults Statement defines the default behavior if the `WHEN OTHERS` clause is not used.

- If the Case Statement is used to define the transitions of a state machine, the keywords `WHEN OTHERS` cannot be used to recover from illegal states of an *n*-bit state machine unless the state machine contains exactly 2^*n* states.

- Each behavioral statement ends with a semicolon (`;`).

Go to the following sources for more information:

"Case Statement Syntax" in MAX+PLUS II Help
"Implementing Conditional Logic" on page 31 in *How to Use AHDL*
"Recovering From Illegal States" on page 66 in *How to Use AHDL*

The following topics in *How to Use AHDL* show additional examples of Case Statements:

"Implementing State Machines" on page 55
"Setting Clock, Reset & Enable Signals" on page 57
"Importing & Exporting State Machines" on page 77

Defaults Statement

The Defaults Statement allows you to specify default values for variables used in Truth Table, If Then, and Case Statements. Since active-high signals automatically default to `GND`, Defaults Statements are required only for active-low signals.

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

The following example shows a Defaults Statement:

```
BEGIN
   DEFAULTS
      a = VCC;
   END DEFAULTS;
   IF y & z THEN
      a = GND;   % a is active low %
   END IF;
END;
```

The Defaults Statement has the following characteristics:

- It is enclosed by the keywords DEFAULTS and END DEFAULTS and ends with a semicolon (;).
- The body of the Defaults Statement consists of one or more Boolean equations that assign constant values to variables. In the example shown above, the Defaults Statement assigns the default value VCC to the variable a.
- Each equation ends with a semicolon (;).
- The Defaults Statement is activated if a variable that follows it is undefined for certain conditions. In the example shown above, the variable a is undefined when y or z is a logical low, so the equation (a = VCC) in the Defaults Statement is activated.

The following rules apply to Defaults Statements:

- Only one Defaults Statement is allowed in the Logic Section, and it must be the first statement after the BEGIN keyword.
- If a single variable is assigned a value more than once in a Defaults Statement, all assignments but the last are ignored.
- A Defaults Statement cannot be used to set a default value of X (don't care) to a variable.

- Multiple assignments to a node of the type NODE variable outside of a Defaults Statement are logically ORed, except when the default for the variable is VCC. The following TDF excerpt illustrates the default values for two variables: a with default value GND, and bn with default value VCC:

```
BEGIN
   DEFAULTS
      a = GND;
      bn = VCC;
   END DEFAULTS;
   IF c1 THEN
      a = a1;
      bn = b1n;
   END IF;
   IF c2 THEN
      a = a2;
      bn = b2n;
   END IF;
END;
```

  This example is equivalent to the following equations:

```
a = c1 & a1 # c2 & a2;
bn = (!c1 # b1n) & (!c2 # b2n);
```

- Active-low variables that are assigned more than once should be given a default value of VCC. In the following example, reg[].clrn is given a default value of VCC:

```
SUBDESIGN 5bcount
(
   d[5..1]    : INPUT;
   clk        : INPUT;
   clr        : INPUT;
   sys_reset  : INPUT;
   enable     : INPUT;
   load       : INPUT;
   q[5..1]    : OUTPUT;
)
VARIABLE
   reg[5..1]   : DFF;

BEGIN
   DEFAULTS
      reg[].clrn = VCC;
   END DEFAULTS;
```

```
      reg[].clk = clk;
      q[]       = reg[];
      IF sys_reset # clr THEN
         reg[].clrn = GND;
      END IF;

      !reg[].prn  = (load & d[]) & !clr;
      !reg[].clrn = load & !d[];

      reg[] = reg[] + (0, enable);
   END;
```

Go to the following sources for more information:

"Defaults Statement Syntax" in MAX+PLUS II Help
"Using Default Values for Variables" on page 39 in *How to Use AHDL*

If Then Statement

The If Then Statement lists a series of behavioral statements to be activated after the positive evaluation of one or more Boolean expressions.

The following example shows an If Then Statement:

```
IF a[] == b[] THEN
   c[8..1] = H "77";
   addr[3..1] = f[3..1].q;
   f[].d = addr[] + 1;
ELSIF g3 $ g4 THEN
   f[].d = addr[];
ELSE
   d = VCC;
END IF;
```

The If Then Statement has the following characteristics:

- The keywords `IF` and `THEN` enclose the Boolean expression to be evaluated and are followed by one or more behavioral statements, each of which ends with a semicolon (`;`).

- The keywords `ELSIF` and `THEN` enclose any additional Boolean expressions to be evaluated, and are also followed by one or more behavioral statements. These optional statements can be repeated.

- The behavioral statement(s) following the keyword `THEN` are activated for the first expression that evaluates to true.

- The keyword `ELSE` followed by one or more behavioral statements is similar to the `WHEN OTHERS` default alternative of the Case Statement. If none of the previously evaluated Boolean equations is true, then the behavioral statement(s) following `ELSE` are activated. In the example shown above, if neither expression evaluates to true, the equation `d = VCC` is activated. The `ELSE` clause is also optional.

- Expressions following `IF` and `ELSIF` keywords (in the example shown above, `a[] == b[]` and `g3 $ g4`) are evaluated concurrently.

- The keywords `END IF` and a semicolon (`;`) end the If Then Statement.

- An If Then Statement may generate logic that is too complex for the MAX+PLUS II Compiler. If an If Then Statement contains complex expressions, then the inversion of each expression is likely to be even more complex. In the following example, if `a` and `b` are complex expressions, then the inversion of each expression is likely to be even more complex.

If Then Statement: **Compiler Interpretation:**

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

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

ELSE                       IF !a & !b THEN
c = f;                         c = f;

END IF;                    END IF;
```

Unlike If Then Statements, which can evaluate only Boolean expressions, If Generate Statements can evaluate the superset of arithmetic expressions. The essential difference between an If Then Statement and an If Generate Statement is that the former is evaluated in hardware (silicon), whereas the latter is evaluated when the design is compiled.

Go to the following sources for more information:

"Boolean Equations" on page 168 in this section
"If Then Statement Syntax" in MAX+PLUS II Help
"If Then Statement vs. Case Statement" on page 34 in *How to Use AHDL*
"Implementing Conditional Logic" on page 31 in *How to Use AHDL*

The following topics in *How to Use AHDL* show additional examples of If Then Statements:

"Implementing Active-Low Logic" on page 41
"Creating Counters" on page 51

If Generate Statement

The If Generate Statement lists a series of behavioral statements that are activated after the positive evaluation of an arithmetic expression.

The following example shows an If Generate Statement:

```
IF DEVICE_FAMILY == "FLEX8K" GENERATE
   c[] = 8kadder(a[], b[], cin);
ELSE GENERATE
   c[] = otheradder(a[], b[], cin);
END GENERATE;
```

The If Generate Statement has the following characteristics:

- The keywords `IF` and `GENERATE` enclose the arithmetic expression to be evaluated and are followed by one or more behavioral statements, each of which ends with a semicolon (`;`). These statements are activated if the expression is true.
- The keywords `ELSE GENERATE` are followed by one or more behavioral statements, each of which ends with a semicolon. These statements are activated if the arithmetic expression is false.
- The keywords `END GENERATE` and a semicolon (`;`) end the If Generate Statement.
- The If Generate Statement can be used in the Logic Section or in the Variable Section.

1. Unlike If Then Statements, which can evaluate only Boolean expressions, If Generate Statements can evaluate the superset of arithmetic expressions. The essential difference between an If Then Statement and an If Generate Statement is that the former is evaluated in hardware (silicon), whereas the latter is evaluated when the design is compiled.

2. The If Generate Statement is especially useful with For Generate Statements that handle special cases differently, for example, the least significant bit of a multi-stage multiplier. It can also be used to test parameter values, as shown in the example above.

Go to the following sources for more information:

"If Generate Statement Syntax" in MAX+PLUS II Help
"Using Conditionally Generated Logic" on page 87 in *How to Use AHDL*
"Naming a Boolean Operator or Comparator" on page 84 in *How to Use AHDL*

For Generate Statement

The following example shows an iterative For Generate Statement:

```
CONSTANT NUM_OF_ADDERS = 8;
SUBDESIGN 4gentst
(
   a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1], cin : INPUT;
   c[NUM_OF_ADDERS..1], cout                     : OUTPUT;
)
VARIABLE
   carryout[(NUM_OF_ADDERS+1)..1] : NODE;
BEGIN
   carryout[1] = cin;
   FOR i IN 1 TO NUM_OF_ADDERS GENERATE
      c[i] = a[i] $ b[i] $ carryout[i]   ;% Full Adder %
      carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
   END GENERATE;
   cout = carryout[NUM_OF_ADDERS+1];
END;
```

The For Generate Statement has the following characteristics:

- The keywords `FOR` and `GENERATE` enclose the following items:

    1. A temporary variable name, which consists of a symbolic name that is used only within the context of the For Generate Statement, i.e., the variable ceases to exist after the Compiler processes the statement. In the example shown above, the variable is `i`. This variable name cannot be a constant, parameter, or node name that is used elsewhere in the project.

    2. The word `IN`, which is followed by a range delimited by two arithmetic expressions. The arithmetic expressions are separated by the `TO` keyword. In the example shown above, the arithmetic expressions are `1` and `NUM_OF_ADDERS`. The range endpoints can consist of expressions containing only constants and parameters; variables are not required.

- The `GENERATE` keyword is followed by one or more logic statements, each of which ends with a semicolon (`;`).

- The keywords `END GENERATE` and a semicolon (`;`) end the For Generate Statement.

Go to the following sources for more infomation:

"Arithmetic Expressions" on page 103 in *Elements*
"For Generate Statement Syntax" in MAX+PLUS II Help
"Using Iteratively Generated Logic" on page 86 in *How to Use AHDL*

In-Line Logic Function Reference

An in-line logic function reference is a Boolean equation that implements a logic function. It is a shorthand method for implementing a logic function that uses only one line of the Logic Section and does not require a Variable Declaration.

When you wish to implement an instance of a megafunction or macrofunction, you must ensure that its logic is defined in its own design file. You then use a Function Prototype Statement of the function, and implement an instance of the function with an in-line reference or an Instance Declaration.

To implement an instance of a primitive, you also use an in-line reference or an Instance Declaration. However, in contrast to mega- and macrofunctions, primitive logic is predefined, so you do not need to define the primitive logic in a separate design file. In most cases, a Function Prototype Statement is not needed. See "Function Prototype Statement" on page 151 for more information.

The following examples show the Function Prototypes for the `compare` and `lpm_add_sub` functions. The `compare` function has input ports `a[3..0]` and `b[3..0]` and output ports `less`, `equal`, and `greater`; the `lpm_add_sub` function has the input ports `dataa[LPM_WIDTH-1..0]`, `dataa[LPM_WIDTH-1..0]`, `cin`, and `add_sub`, and output ports `result[LPM_WIDTH-1..0]`, `cout`, and `overflow`.

```
FUNCTION compare (a[3..0], b[3..0])
   RETURNS (less, equal, greater);
FUNCTION lpm_add_sub (cin, dataa[LPM_WIDTH-1..0],
   datab[LPM_WIDTH-1..0], add_sub)
   WITH (LPM_WIDTH, LPM_REPRESENTATION)
   RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

The in-line logic function references for the `compare` and `lpm_add_sub` functions appear on the right side of the equations below:

```
(clockwise, , counterclockwise) = compare(position[], target[]);
sum[] = lpm_add_sub (.datab[] = b[], .dataa[] = a[])
   WITH (LPM_WIDTH = 8)
   RETURNS (.result[]);
```

The in-line reference for a logic function has the following characteristics:

- The function name on the right side of the equals symbol (=) is followed by a signal list enclosed in parentheses `()`, containing symbolic names, decimal numbers, or groups, separated by commas (`,`). These items correspond to the input ports of the function.

- In the signal list, port names can be given through positional port association or named port association:

  - In the `compare` example shown above, the `a[3..0]` and `b[3..0]` inputs of `compare` are connected to the variables named `position[]` and `target[]`, respectively, through positional port association. When you use positional port association, you can use commas as placeholders for outputs that are not connected to a variable. In `compare`, the `equal` output is not connected to any variable, so an extra comma is

4
Design Structure

needed to hold its place in the group on the left side of the equation.

– In the lpm_add_sub example shown above, the .datab[] and .dataa[] inputs of lpm_add_sub are connected to the variables b[] and a[], respectively, through named port association. Port names are connected to variables with an equals symbol (=).

1. Port names must have the format .*<port name>* on both the left and right sides of in-line references that use named port association.

2. Named port association is supported only on the right side of an in-line reference. The left side of an in-line reference is always connected to variables by positional port association.

■ In a parameterized function, the keyword WITH and parameter name list follows the input port list. The list is enclosed in parentheses; parameter names are separated by commas. Only the parameters used by the instance are declared; optional parameter values are separated from parameter names by an equals symbol. In the lpm_add_sub example shown above, the LPM_WIDTH parameter is assigned a value of 8. If no parameter values are assigned in the in-line reference, the Compiler searches for them in the parameter value search order described on page 142.

■ On the left side of the in-line reference, the outputs of the function are connected to variables. In the compare example shown above, the function's less and greater outputs are connected to the variables clockwise and counterclockwise, respectively, through positional port association. Similarly, in the lpm_add_sub example, the function's sum[] outputs are connected through positional port association.

■ The values of the variables, which are determined elsewhere in the Logic Section, feed the associated inputs and outputs. In the compare example shown above, the values of position[] and target[] feed the inputs of compare. The values of output ports less and greater feed clockwise and counterclockwise, respectively. These variables may be used in other operations in the Logic Section.

Go to the following sources for more information:

"Boolean Equations" on page 168 in this section
"Function Prototype Statement" on page 151 in this section
"Implementing a Hierarchical Project" on page 69 in *How to Use AHDL*
"In-Line Logic Function Reference Syntax" in MAX+PLUS II Help

The following topics in *How to Use AHDL* show additional examples of in-line references:

"Implementing Bidirectional Pins" on page 43
"Using Altera-Provided Unparameterized Functions" on page 69
"Using Altera-Provided Parameterized Functions" on page 73
"Implementing LCELL & SOFT Primitives" on page 81

Truth Table Statement

The Truth Table Statement is used to specify combinatorial logic or state machine behavior. In an AHDL truth table, each entry in the table contains a combination of input values that will produce specified output values. These output values can be used as feedback to specify state transitions and outputs of state machines.

The following example shows a Truth Table Statement:

```
TABLE
   a0,     f[4..1].q    =>    f[4..1].d,    control;

   0,      B"0000"      =>    B"0001",      1;
   0,      B"0100"      =>    B"0010",      0;
   1,      B"0XXX"      =>    B"0100",      0;
   X,      B"1111"      =>    B"0101",      1;
END TABLE;
```

The Truth Table Statement has the following characteristics:

- The truth table heading consists of the keyword `TABLE`, followed by a comma-separated list of table inputs, an arrow symbol (=>), and a comma-separated list of table outputs. The heading ends with a semicolon (`;`).

- Truth table inputs are Boolean expressions; truth table outputs are variables. In the example shown above, the input signals are `a0` and `f[4..1].q`; the output signals are `f[4..1].d` and `control`.
- The body of the table consists of one or more entries, each spanning one or more lines and ending with a semicolon.
- An entry consists of a comma-separated list of inputs and a comma-separated list of numerical outputs. The inputs and outputs are separated by `=>`.
- Each signal has a one-to-one correspondence with the values in each entry. Thus, the first entry in the example shown above signifies that when `a0` has the value `0` and `f[4..1].q` has the value `B"0000"`, then `f[4..1].d` will have the value `B"0001"`, and `control` will have the value `1`.
- Input and output values can be numbers, predefined constants `VCC` or `GND`, symbolic constants (i.e., symbolic names used as constants), or groups of numbers or constants. Input values can also be `X` (don't care).
- Input and output values correspond to the inputs and outputs of the table heading.
- The keywords `END TABLE`, followed by a semicolon (`;`), end the truth table.

The following rules apply to the Truth Table Statement:

- The names in the table heading can be either single nodes or groups.
- Every conceivable combination of input values need not be listed. You can use an `X` (don't care) to indicate that the output does not depend on the input corresponding to the position of the `X`. The following example specifies that if `a0` is high and `f4` is low, the value of the other inputs is not important. Therefore, you can specify the common portion of the input pattern (in this example, `0`), then use `X` characters for the rest of the input pattern (in this example, `XXX`).

```
TABLE
   a0,   f[4..1].q   =>   f[4..1].d,   control;
   0,    B"0000"     =>   B"0001",     1;
   0,    B"0100"     =>   B"0010",     0;
   1,    B"0XXX"     =>   B"0100",     0;
   X,    B"1111"     =>   B"0101",     1;
END TABLE;
```

- The number of comma-separated items in a truth table row must equal the number of comma-separated items in the truth table heading.

- The Defaults Statement assigns output values in cases when the actual inputs do not match the input values of the table.

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.

Go to the following sources for more information:

"Using Default Values for Variables" on page 39 in *How to Use AHDL* for information on how to specify default values for truth table outputs.
"Truth Table Statement Syntax" in MAX+PLUS II Help

The following topics in *How to Use AHDL* show additional examples of Truth Table Statements:

"Creating Decoders" on page 35
"Using Default Values for Variables" on page 39
"Assigning State Machine Bits & Values" on page 58
"State Machines with Synchronous Outputs" on page 60
"State Machines with Asynchronous Outputs" on page 64

Section

5

Style Guide

This style guide provides suggestions for formatting Text Design Files (**.tdf**) to improve readability and thus avoid errors. These are recommendations only and are not required for your TDFs to compile successfully. Examples that illustrate guidelines are provided.

☞ You can use the Text Editor's **Syntax Coloring** command (Options menu) to identify typographical errors and different sections of AHDL code. Go to "Syntax Coloring" on page 10 in *Introduction* for more information.

Style guidelines are discussed in the following order:

- General Style Guidelines 188
- White Space 190
- Comments & Documentation 191
- Naming Conventions 192
- Indentation Guidelines 193

Go to MAX+PLUS II Help for complete and up-to-date information on style guidelines.

General Style Guidelines

- All keywords, device names, constants, and primitives should be entered in capital letters; all other text should be lowercase, including filenames, megafunctions, and macrofunctions.

Unformatted:

```
case tap is
   when test_logic_reset =>
      if !tms then
         tap = run_test/idle;
      end if;

   when run_test/idle =>
      if tms then
         tap = select_dr_scan;
      end if;

   when select_dr_scan =>
      if tms then
        tap = select_ir_scan;
      else
         tap = capture_dr;
      end if;
   ...
end case;
```

Formatted:

```
CASE tap IS
   WHEN test_logic_reset =>
      IF !tms THEN
         tap = run_test/idle;
      END IF;

   WHEN run_test/idle =>
      IF tms THEN
         tap = select_dr_scan;
      END IF;

   WHEN select_dr_scan =>
      IF tms THEN
         tap = select_ir_scan;
      ELSE
         tap = capture_dr;
      END IF;
   ...
 END CASE;
```

- Either list all input and output ports on the same line, or enter `: INPUT;` after each line of inputs and `: OUTPUT;` after each line of outputs. With this formatting style, all ports are clearly labeled.

- Lines should not be longer than the width of the screen. If necessary, move part of a line to the next line and indent it. The Text Editor provides the **Auto-Indent** command (Options menu), and the **Increase Indent** and **Decrease Indent** commands (Edit menu), to help you indent text easily.

- Place opening and closing parentheses of the Subdesign Section and of Parameters Statements on a separate line to easily distinguish inputs and outputs. This formatting style also allows you to add and edit signal and parameter names easily.

Unformatted:

```
SUBDESIGN s (i1, i2, i3: INPUT;
   o1, o2, o3: OUTPUT;)
BEGIN
   ...
END;
```

Formatted:

```
SUBESIGN s
(
   i1, i2, i3 : INPUT;
   o1, o2, o3 : OUTPUT;
)
BEGIN
   ...
END;
```

- Do not use quoted symbolic names if you can use unquoted names.

Unformatted:

```
VARIABLE
   tap: MACHINE WITH STATES (
         'Test-Logic-Reset',
         'Run-Test/Idle',
         'Select-DR-Scan',
         'Capture-DR',
         'Shift-DR',
         'Exit1-DR',
         'Pause-DR',
         'Exit2-DR',
         'Update-DR',
         'Select-IR-Scan',
         'Capture-IR',
         'Shift-IR',
         'Exit1-IR',
         'Pause-IR',
         'Exit2-IR',
         'Update-IR');
```

Formatted:

```
VARIABLE
   tap: MACHINE WITH STATES (
         test_logic_reset,
         run_test_idle,
         select_dr_scan,
         capture_dr,
         shift_dr,
         exit1_dr,
         pause_dr,
         exit2_dr,
         update_dr,
         select_ir_scan,
         capture_ir,
         shift_ir,
         exit1_ir,
         pause_ir,
         exit2_ir,
         update_ir);
```

- Follow the indentation guidelines under “Indentation Guidelines” on page 193.

White Space

- Use white space (blank lines, spaces, and tabs) around logical groups.
- Do not place extra spaces before semicolons (;) commas (,), closing double quotation marks ("), or closing parentheses ()), or after opening double quotation marks (") or opening parentheses (().

Unformatted:

```
DFF ( d , clk , clrn , pren );
```

Formatted:

```
DFF(d, clk, clrn, prn)
```

- Use tabs and spaces to align colons, truth table entries, etc.

Unformatted:

```
in1, clk : INPUT;
out1, out2, out3 : OUTPUT:
bus[8..1] : BIDIR
```

Formatted:

```
in1, clk            : INPUT;
out1, out2, out3    : OUTPUT;
bus[8..1]           : BIDIR
```

- Leave a blank space before an opening parenthesis to separate it from a keyword.

Unformatted:

```
OF BITS(q[3..0])
```

Formatted:

```
OF BITS (q[3..0])
```

- Place one blank space before and after operators and comparators (unless you are aligning signal names).

Unformatted:

```
enable = !a3&a2&a1&a0;

enable = (a[]==B"0111");
```

Formatted:

```
enable = !a3 & a2 & a1 & a0;

enable = (a[] == B"0111");
```

Comments & Documentation

- Describe the design at the beginning of the TDF with a substantial comment in natural language. Specifically, describe the ports and function of the design.

- Use comments where appropriate to document the file. These comments should provide relevant information about the corresponding statement, and should be updated along with the file.

- Do not duplicate a statement as a comment.

With Duplication:

```
IF clear THEN
   % load q[] with 0 %
   q[] = 0;
END IF;
```

Without Duplication:

```
IF clear THEN
   q[] = 0;
END IF;
```

- Place a comment either directly above the section it describes at the same indentation level, or aligned with other comments to the right of the lines of code.

- Leave one blank space between documentation text and the percent symbol (%) for AHDL-style comments or two dashes (--) for VHDL-style comments. Align the opening and closing symbols for easy readability.

Unformatted:

```
%Leave one blank space between%
%the percent symbol and the%
%documenting text. Line up%
%opening and closing percent%
%symbols for easy readability.%
```

Formatted:

```
% Leave one blank space between %
% the percent symbol and the    %
% documenting text. Line up     %
% opening and closing percent   %
% symbols for easy readability. %
```

- VHDL-style comments can be nested within %-style comments. If you use VHDL-style comments (--) for documentation-type comments, you can then use the %-style comments to exclude sections of code from compilation (i.e., "comment out" sections of code).

Naming Conventions

- All symbolic names and identifiers should be meaningful and completely understandable, and should reflect the purpose or action of the function.

Ambiguous Name:	**Unambiguous Name:**
`direction`	`up`
`access_mode`	`access_memory`

- Active-low signals should be specified with a clear and consistent notation. The notations shown below are supported in MAX+PLUS II design files. You should choose one notation and use it throughout a project.

```
/write
nchip_enable
resetn
```

- Use underscores to separate "words" in symbolic and simple names.

Unformatted:	**Formatted:**
`regload`	`reg_load`
`idleio`	`idle_io`
`goidle`	`go_idle`
`CSYNCPREEQJ`	`CSYNC_PRE_EQ_J`

- Do not use abbreviations unless they are obvious.

Ambiguous Name:	**Unambiguous Name:**
`c`	`clk`
`clrg`	`clear_reg`
`sb`	`sync_bit`
`sm`	`select_mem`
`rdrm`	`refresh_dram`
`f`	`forward`
`r`	`reverse`

- The Title Statement should include a short, descriptive name for the design.

```
TITLE "NTSC Waveform Generator";
```

- Replace numbers with constants to provide meaningful names and a visual reference for all numbers. Only use 0 and 1 in the code.

```
CONSTANT TERMINAL_COUNT = 103;
```

Indentation Guidelines

This section illustrates recommended indentation of AHDL sections and statements.

☞ You can use the Text Editor's **Auto-Indent** command (Options menu), and **Increase Indent** and **Decrease Indent** commands (Edit menu), to help you indent text easily.

Parameters Statement, Subdesign Section, and Logic Section Indentation:

```
PARAMETERS
(
   % parameter %
   % parameter %
);

SUBDESIGN
(
   % inputs  %
   % outputs %
   % bidirs  %
)
BEGIN
   % statement %
   % statement %
END;
```

If Then Statement Indentation:

```
IF expression1 THEN
   % statement %
   % statement %
ELSIF expression2 THEN
   % statement %
   % statement %
ELSE
   % statement %
   % statement %
END IF;
```

☞ Use a similar style for If Generate and For Generate statements.

Case Statement Indentation:

```
CASE expression IS
   WHEN constant1 =>
      % statement %
      % statement %

   WHEN constant2 =>
      % statement %
      % statement %

   WHEN constant3 =>
      % statement %
      % statement %

   WHEN OTHERS =>
      % statement %
      % statement %
END CASE;
```

or:

```
CASE expression IS
   WHEN constant1 => % statement %
   WHEN constant2 => % statement %
   WHEN constant3 => % statement %
   WHEN OTHERS    => % statement %
END CASE;
```

Truth Table Statement Indentation:

```
TABLE
   ss,    inputs[]   => outputs[],ss;

   s0,    B"xxxxx0"  => B"000001",s1;
   s1,    B"xxxx01"  => B"000011",s2;
   s2,    B"xxx011"  => B"000111",s3;
   s3,    B"xx0111"  => B"001111",s4;
   s4,    B"x01111"  => B"011111",s5;
   s5,    B"011111"  => B"111111",s0;
END TABLE;
```

Variable Section & State Machine Declaration Indentation:

```
VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3);

   tt: MACHINE
       OF BITS (q[3..0])
       WITH STATES (
          t0 = B"0001",
          t1 = B"0010",
          t2 = B"0100",
          t3 = B"1000");
```

Assert Statement Indentation:

```
ASSERT condition
   REPORT "message"
      % message variables %
   SEVERITY ERROR;
```

Glossary

This glossary defines selected terms used in MAX+PLUS II documentation.

Choose **Glossary** (Help menu) to view the full MAX+PLUS II glossary on-line.

A

ACF *see* Assignment & Configuration File.

active-high node A node that is activated when it is assigned a value of one (`1` in AHDL or `'1'` in VHDL) or `VCC` (e.g., `ena`, `clk`).

active-low node A node that is activated when it is assigned a value of zero (`0` in AHDL or `'0'` in VHDL) or `GND` (e.g., `clrn`, `prn`, `oen`). In AHDL design files, an active-low node should be assigned a default value of `VCC` with the Defaults Statement.

ADF *see* Altera Design File.

Altera Design File (.adf) An ASCII-format file (with the extension **.adf**) for Boolean equation entry, used with Altera's A+PLUS software. ADFs use a netlist format and Boolean equations to describe a design. The MAX+PLUS II Compiler automatically translates an ADF into a Compiler Netlist File (**.cnf**) during project compilation.

An ADF is also generated when a State Machine File (**.smf**) is compiled.

ancillary file A file that is associated with a MAX+PLUS II project, but is not a design file in the project hierarchy tree. Most ancillary files also do not contain design logic. User-editable ancillary files with the same filename as the project appear in the Hierarchy Display window. See the following list:

Editable Ancillary Files:
Assignment & Configuration File (**.acf**)
Assignment & Configuration Output File (**.aco**)
Command File (**.cmd**)
EDIF Command File (**.edc**)
Fit File (**.fit**)
Hexadecimal (Intel-format) File (**.hex**)
History File (**.hst**)
Include File (**.inc**)
JTAG Chain File (**.jcf**)
Library Mapping File (**.lmf**)
Log File (**.log**)
Memory Initialization File (**.mif**)
Memory Initialization Output File (**.mio**)
Message Text File (**.mtf**)
Programmer Log File (**.plf**)
Report File (**.rpt**)
Simulator Channel File (**.scf**)
Standard Delay Format (SDF) Output File (**.sdo**)
Symbol File (**.sym**)
Table File (**.tbl**)
Tabular Text File (**.ttf**)
Text Design Export File (**.tdx**)
Text Design Output File (**.tdo**)
Timing Analyzer Output File (**.tao**)
Vector File (**.vec**)
VHDL Memory Model Output File (**.vmo**)

Non-Editable Ancillary Files:
Compiler Netlist File (**.cnf**)
Hierarchy Interconnect File (**.hif**)
JEDEC File (**.jed**)
Node Database File (**.ndb**)
Programmer Object File (**.pof**)
Raw Binary File (**.rbf**)
Serial Bitstream File (**.sbf**)
Simulator Initialization File (**.sif**)
Simulator Netlist File (**.snf**)
SRAM Object File (**.sof**)

ASCII American Standard Code for Information Interchange. Text editing software used for any MAX+PLUS II text file, e.g., Text Design File (**.tdf**), Library Mapping File (**.lmf**), or Vector File (**.vec**), must conform to this textual data coding system.

assignment In AHDL and VHDL, assignment refers to the transfer of a value to a symbolic name or group, usually through a Boolean equation. The value on the right side of the equation is assigned to the symbolic name or group on the left.

assignment (resource) *see* resource assignment.

Assignment & Configuration File (.acf) An ASCII file (with the extension **.acf**) that stores information about probe, pin, location, chip, clique, logic option, timing, connected pin and device assignments, as well as configuration settings for the Compiler, Simulator, and Timing Analyzer for an entire project.

The ACF stores information entered with menu commands in all MAX+PLUS II applications, as well as pin, location, and chip assignments entered in the Floorplan Editor window. You can also edit an ACF manually in a Text Editor window.

B

binary The base 2 number system (radix). Binary digits are 0 and 1.

In AHDL, binary numbers are indicated with the following notation:

`B"`*<series of 0, 1, and X characters>*`"`

where `X`="don't care." Example: `B"0110X1X10"`

BIT0 option An option that prevents certain warning messages if you use the lowest-numbered bit of a group as anything other than the least significant bit (LSB).

When you declare a group with a range of numbers, the first number listed is always the most significant bit (MSB), the last is always the LSB. If you specify a range in ascending order, a warning message is issued unless you have used the `BIT0` option to specify that the lowest numbered bit is the MSB. If you set the `BIT0` option to MSB, a warning message is generated if you specify a range in descending order. If you set `BIT0` to `ANY`, you may specify ranges in either ascending or descending order without receiving a warning.

Boolean logic Logic that obeys the theorems of Boolean algebra (George Boole, "The Laws of Thought," 1854). The Boolean portion of a design is the portion which can be implemented in the `AND-OR` matrix of a device.

buried node A combinatorial or registered signal that does not drive an output pin.

buried register A register in an Altera device that does not drive its output to a pin. A buried register can be located on an I/O pin or on a logic cell that has no output to a pin. A buried register can be used to implement internal logic.

bus (or group) name The name of a bus (or group) of up to 256 nodes.

A single-range or dual-range name consists of up to 32 name characters, followed by one or two ranges of numbers or arithmetic expressions in brackets. (Dual-range names are not supported in Waveform Editor files.) The start and end of the number range are separated by two periods. Each number in the sequence represents an individual node (or bit).

Example: bus `a[4..1]` consists of the nodes `a4`, `a3`, `a2`, and `a1`.

Example: bus `b[2..1][1..0]` consists of the nodes `b2_1`, `b2_0`, `b1_1`, and `b1_0`.

A sequential name, consisting of a comma-separated list of names, can be entered in AHDL Text Design Files (**.tdf**) and Graphic Design Files (**.gdf**). In TDFs and ACFs, this list of names must be enclosed in parentheses. Sequential bus names can include single- and dual-range bus names.

Example: `a[3..0],dout[6..4],z3`

The first name in the series of names in a single-range, dual-range, or sequential name is the most significant bit (MSB) of the bus; the last name is the least significant bit (LSB).

An arbitrary bus name, consisting of up to 32 name characters, can be entered in a Waveform Design File (**.wdf**), Simulator Channel File (**.scf**), or Vector File (**.vec**). An arbitrary bus name does not indicate how many members are included in the bus.

bus pinstub The location on the boundary of a mega- or macrofunction symbol, represented by an "x" in the Symbol File (**.sym**), that represents multiple inputs or outputs to the function. A bus (thick line) drawn in a Graphic Editor file must connect to a bus pinstub with the same number of bits to be recognized as a connection to the function.

C

chip A group of logic functions defined as a single, named unit. A chip is assigned to an actual device by either the user or the Compiler.

You can make chip assignments on logic functions in design files. Items that are assigned to the same chip are placed in the same device during compilation. The term device always refers to an actual programmable logic device, whereas the term chip always refers to a group of logic functions.

When the Compiler processes a project, each chip name is assigned to a corresponding programming file for a particular device.

Classic An Altera device family based on Altera's original EPROM-based EPLD architecture. MAX+PLUS II provides support for the following Classic devices: EP220, EP320I, EP330, EP600I, EP610, EP610I, EP900I, EP910, EP910I, EP1800I, and EP1810 devices.

Clear An input signal that asynchronously resets a register, regardless of the Clock signal.

clique A group of logic functions defined as a single, named unit. The Compiler attempts to keep clique members together when it fits the project. A clique assignment allows you to group all logic on a speed-critical path, thus improving performance.

If possible, all clique members are assigned to the same LAB. If the clique members will not fit into a single LAB, they are placed in the same row (in FLEX 10K, FLEX 8000, and MAX 9000 devices only) or the same device.

Clock A signal that triggers registers.

In a flipflop or state machine, the Clock is an edge-sensitive signal. The output of the flipflop can change only on the Clock edge. For example, in a D flipflop, the input value is stored and placed on the output at the Clock edge.

In some cases, MAX+PLUS II lists the Latch Enable input to a latch as a Clock, e.g., in a Delay Matrix timing analysis.

Clock Enable The level-sensitive signal on an enabled flipflop, i.e., a flipflop with an "E" suffix, including `DFFE`, `TFFE`, `SRFFE`, and `JKFFE`. When the Clock Enable is low, Clock transitions on the Clock input to the flipflop are ignored.

combinatorial feedback Feedback from a logic cell that goes back into the device's logic array. It is the direct function of the inputs to a logic cell, and does not retain values from earlier inputs.

combinatorial output Output from a logic cell that is a direct function of the inputs, without regard to the Clock; i.e., it does not retain values resulting from earlier inputs.

comment In the Graphic and Symbol Editors, a comment is a free-floating block of text used to document the design. It is not associated with any object. A comment stands alone anywhere within Graphic Editor files. A comment also stands alone within the symbol border of a Symbol Editor file. Comments are ignored by the Compiler, and can be used to document various sections of a file.

In the Waveform Editor, a comment is a line of text used to annotate the waveforms in the waveform drawing area. It is not associated with any waveform. A comment is anchored to the time on the time scale where the first character is entered. A label appears in the Name field to indicate a comment line; when a comment is added between two existing nodes, it appears in a blank space, which is inserted between the waveforms. Comments are ignored by the Compiler.

In all MAX+PLUS II text files except VHDL Design Files (**.vhd**) and Assignment & Configuration Files (**.acf**), e.g., in Report Files (**.rpt**), Vector Files (**.vec**), and Text Design Files (**.tdf**), a comment is any string of characters enclosed in percent symbols (`%`). You can insert comments wherever white space is allowed in text files.

In VHDL Design Files and ACFs, comments begin with two dashes (`--`) and continue to the End-of-Line. AHDL TDFs also support VHDL-style comments. If you use a VHDL-style comment in a TDF, you must separate the two dashes from any preceding symbolic name with at least one space.

ACFs also support comments consisting of any string of characters enclosed between `/*` and `*/` characters. You can insert comments at any location in the file.

comparator A comparator is an operator used to compare nodes, groups, and numbers. AHDL provides the following comparators:

Comparator:	Definition:
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`<=`	less than or equal to
`>=`	greater than or equal to

Compiler Netlist File (.cnf) A binary file (with the extension **.cnf**) that contains the data from a design file. The CNF is created by the Compiler Netlist Extractor module of the MAX+PLUS II Compiler.

cone of logic A group of logic functions whose outputs eventually feed into a single gate.

Configuration EPROM Altera's family of serial EPROMs, which are designed to configure FLEX 8000 and FLEX 10K devices. This device family includes the EPC1, EPC1213, and EPC1064 devices.

construct A unit in a text design language such as AHDL, VHDL, or EDIF.

D

database A flattened representation of all design files in a MAX+PLUS II project hierarchy. The database is used internally by Compiler modules during compilation.

De Morgan's Inversion Theorem A theorem developed by Augustus De Morgan that is used in Boolean algebra. This theorem states that the complement of the product of the factors equals the sum of the complements of the addends; or that the complement of the sum of the addends equals the products of the complement of each factor.

Example: `!(A & B) = !A # !B`

decimal The base 10 number system (radix). Decimal digits are 0 through 9.

In AHDL, no special notation is needed to indicate decimal digits.

delimiter A text string, character, or keyword used to define the beginning or the end of a statement or construct in a text file.

For example, `[` and `]` are delimiters of AHDL group ranges and `%` is a comment delimiter in many MAX+PLUS II text files.

design file A file that contains logic for a MAX+PLUS II project and is compiled by the Compiler. The following files are design files:

- Altera Design File (**.adf**)
- EDIF Input File (**.edf**) *
- Graphic Design File (**.gdf**) *
- OrCAD Schematic File (**.sch**) *
- State Machine File (**.smf**)
- Text Design File (**.tdf**) *
- VHDL Design File (**.vhd**) *
- Waveform Design File (**.wdf**)
- Xilinx Netlist Format File (**.xnf**)

An asterisk (*) indicates the design files that can exist as top-level files in hierarchical projects. Other design files must be the only design file in a project or must exist at the bottom level of a hierarchical project.

device A device refers to an Altera programmable logic device, including Classic, MAX 5000, MAX 7000, MAX 9000, FLEX 8000, FLEX 10K, and FLASHlogic devices.

Altera also offers Configuration EPROM devices which are used to configure FLEX 8000 and FLEX 10K devices.

device family A group of Altera programmable logic devices with the same fundamental architecture. Altera families include the Classic, MAX 5000, MAX 7000, MAX 9000, FLEX 8000, FLEX 10K, and FLASHlogic device families.

dual I/O feedback A combination of pin feedback and register or combinatorial feedback on the same logic cell.

dual-range group (or bus) name The name of a group (or bus) of up to 256 nodes, consisting of up to 32 name characters, followed by a two ranges of numbers or arithmetic expressions in brackets. The start and end of the ranges are separated by two periods. Each set of numbers in the two ranges represents an individual node (or "bus bit").

Example: group `a[2..1][5..3]` consists of the nodes `a2_5`, `a2_4`, `a2_3`, `a1_5`, `a1_4`, and `a1_3`.

In a Graphic Editor files, a sequential bus name can also include one or more single- or dual-range bus names in a series. The first node of the series or the first node in the first range is the most significant bit of the bus; the last node of the series or the last node in the last range is the least significant bit.

Example: `a[8..0][2..0], b1, dout[6..4]`

E

EAB *see* Embedded Array Block.

EC *see* embedded cell.

EDIF Electronic Design Interchange Format. An industry-standard format for the transmission of design data.

You can generate an EDIF 2 0 0 or 3 0 0 netlist file from a schematic design or from a VHDL or Verilog HDL design that has been processed with an appropriate industry-standard synthesis tool and then import the file into MAX+PLUS II as an EDIF Input File (**.edf**). MAX+PLUS II supports EDIF Input Files that contain functions from the Library of Parameterized Modules (LPM). The MAX+PLUS II Compiler can also generate one or more EDIF Output Files (**.edo**) in either EDIF 2 0 0 or 3 0 0 format that contain functional or timing information for simulation with a standard EDIF simulator.

EDIF Input File (.edf) An EDIF version 2 0 0 or 3 0 0 netlist file generated by any standard EDIF netlist writer. EDIF Input Files (with the extension **.edf**) can be compiled by the MAX+PLUS II Compiler. MAX+PLUS II supports EDIF Input Files that contain functions from the Library of Parameterized Modules (LPM).

EDIF Output File (.edo) An EDIF version 2 0 0 or 3 0 0 netlist file (with the extension **.edo**) generated by the EDIF Netlist Writer module of the Compiler. This file can be exported to an industry-standard workstation or PC environment for simulation.

Embedded Array Block (EAB) A physically grouped set of 8 embedded cells that implement memory (RAM or ROM) or combinatorial logic in a FLEX 10K device. An EAB consists of an embedded cell array, with data, address, and control signal inputs and data outputs that are optionally registered.

A single EAB can implement a memory block of 256 × 8, 512 × 4, 1,024 × 2, or 2,048 × 1 bits. Each embedded cell within the EAB implements up to 256 bits of memory. For memory blocks of these sizes, an EAB has 8, 4, 2, or 1 outputs, respectively. Multiple EABs can be combined to create larger memory blocks.

The EAB is fed by row interconnect paths and a dedicated input bus.

embedded cell (EC) A memory element that exists in the embedded array of a FLEX 10K device, and which can implement memory (RAM or ROM) or combinatorial logic. An Embedded Array Block (EAB) consists of a group of 8 embedded cells that can implement a memory block of 256 × 8, 512 × 4, 1,024 × 2, or 2,048 × 1 bits. Each embedded cell within an EAB implements up to 256 bits of memory. Depending on the depth of the memory, up to 8 of the embedded cells in an EAB have outputs. For memory blocks of 256 × 8, 512 × 4, 1,024 × 2, or 2,048 × 1 bits, an EAB has 8, 4, 2, or 1 outputs, respectively.

Embedded cells have "numbers" of the format EC<*number*>_<*row letter*>, where <*number*> ranges from 1 to 8 and <*row letter*> consists of the row letter of the EAB.

EPLD Erasable Programmable Logic Device, i.e., an Altera device that is a member of the Classic, MAX 5000, MAX 7000, or MAX 9000 device families.

evaluated function An mathematical function that evaluates an arithmetic expression and returns a value based on one or more arguments. The AHDL Define Statement can be used to create evaluated functions. The following example shows the definition of the evaluated function MAX:

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

excitation equation Combinatorial logic that directs state transitions in a state machine.

expander product term A single product term with an inverted output that feeds back into the Logic Array Block (LAB) of a MAX 5000, MAX 7000, or MAX 9000 device.

An uncommitted expander product term that can be shared with other logic cells in the same LAB is called a shareable expander; a product term that has been shared in this manner is called a shared expander.

In MAX 7000 and MAX 9000 devices only, an expander product term that is "borrowed" from an adjacent logic cell in the same LAB is called a parallel expander.

extension *see* filename extension.

F

family-specific mega- or macrofunction An Altera-provided mega- or macrofunction that contains logic optimized for the architecture of a specific device family.

The functionality of a family-specific mega- or macrofunction is always the same, regardless of the device family for which it is designed. However, the actual primitives and nodes used within the mega- or macrofunction file can vary from family to family to take advantage of different device architectures, thus providing higher performance and/or more efficient implementation.

fan-in and **fan-out** Fan-in refers to input signals that feed the input equations of a logic cell.

Fan-out refers to output signals that are fed by the output equations of a logic cell.

filename The name of a design file, ancillary file, or other file, without the extension.

A single filename can contain up to 32 name characters, plus a 3-character filename extension. A full pathname plus filename and extension can contain up to 128 characters.

Since Windows 3.1 and Windows for Workgroups 3.11 support only 8-character filenames, MAX+PLUS II maps longer filenames on these operating systems to 8-character filenames. These filename mappings are stored in the **maxplus2.idx** file in each directory that contains long filenames.

In the Hierarchy Display window, a filename, along with the file icon and filename extension, represents a file in the current hierarchy tree.

filename extension The one, two, or three-letter extension of a filename that follows a period (.).

In the Hierarchy Display window, a filename extension, along with the filename and the file icon, represents a file in the current hierarchy or the current project.

Fit File (.fit) An ASCII file (with the extension **.fit**) generated by the Compiler that documents pin, logic cell, I/O cell, chip, and device assignments made during the last compilation. Assignments are recorded in Assignment & Configuration File (**.acf**) syntax.

The Fit File can be used for back-annotation and for functional testing in the Simulator and Programmer. To preserve assignments permanently, Fit File assignments can be back-annotated into a project's ACF with the **Back-Annotate Project** command (Assign menu).

You can also display a read-only version of Fit File information from the most recent project compilation in the Floorplan Editor.

FLASHlogic (formerly **FLEXlogic**) An Altera device family consisting of SRAM-based devices with shadow EPROM or shadow FLASH memory. The high-performance FLASHlogic device family includes the EPX8160, EPX880, EPX780, and EPX740 devices.

MAX+PLUS II provides programming-only support for FLASHlogic devices. Full compilation, simulation, timing analysis, and programming support for all FLASHlogic devices will be available in a future version of MAX+PLUS II.

FLEX 8000 An Altera device family based on Flexible Logic Element MatriX architecture. This SRAM-based family offers high-performance, register-intensive, high-gate-count devices. The FLEX 8000 device family includes the EPF8282, EPF8282V, EPF8282A, EPF8452, EPF8452A, EPF8636A, EPF8820, EPF8820A, EPF81188, EPF81188A, EPF81500, EPF81500A, and EPF8050M devices.

FLEX 8000A devices provide the same architectural features as equivalent FLEX 8000 devices, but offer faster speeds and smaller die sizes.

The EPF8050M device has a multi-chip module architecture. When you compile a project for the EPF8050M, the Report File (**.rpt**) includes information for up to four chips, each of which is labeled EPF8050M/4.

FLEX 10K An Altera device family based on Flexible Logic Element MatriX architecture. This SRAM-based family offers high-performance, register-intensive, high-gate-count devices with embedded arrays. The FLEX 10K device family includes the EPF10K50 device.

flipflop or **register** An edge-triggered, clocked storage unit that stores a single bit of data. A low-to-high transition on the Clock signal changes the output of the flipflop, based on the value of the data input(s). This value is maintained until the next low-to-high transition of the Clock, or until the flipflop is preset or cleared.

Depending on the architecture of the device family, a register can be programmed as a level-sensitive flow-through latch or as an edge-triggered D,T, JK, or SR flipflop.

G

GDF *see* Graphic Design File.

global signal A signal from a dedicated input pin that does not pass through the logic array before performing its specified function. Clock, Preset, Clear, and Output Enable signals can be global signals.

A global signal can be designated during design entry with a `GLOBAL` primitive in a Graphic Design File (**.gdf**), Text Design File (**.tdf**), or VHDL Design File (**.vhd**). Or, when the appropriate *Automatic Global* option in the **Global Project Logic Synthesis** dialog box (Assign menu) is turned on, the Compiler chooses the signal that feeds the most flipflops as a global Clock, Preset, or Clear, and the signal that feeds the most `TRI` buffers is chosen as the global Output Enable.

GND A low-level input voltage.

`GND` is the default inactive node value. In an AHDL Text Design File (**.tdf**), `GND` is used as a predefined constant and keyword. In a VHDL Design File (**.vhd**), `GND` is represented by `'0'`. In a Graphic Editor file, `GND` is a primitive symbol. `GND` is represented as a low (0) logic level in the Simulator and Waveform Editor.

Graphic Design File (.gdf) A schematic design file (with the extension **.gdf**) created with the MAX+PLUS II Graphic Editor.

An OrCAD Schematic File (**.sch**) is automatically translated into a GDF and treated as a GDF in the MAX+PLUS II Graphic Editor and Compiler.

Gray code A counting scheme in which only one bit at a time changes value between consecutive count values. In contrast, a binary count sequence does not preclude more than one bit changing at consecutive count values. When only one bit changes, noise susceptibility is reduced in the circuit.

group In AHDL, a group is a collection of up to 256 symbolic names that are treated as a unit. A group name can be specified with a single-range group name, dual-range group name, or sequential group name format.

In the Waveform Editor and Simulator, a group is a collection of up to 256 nodes that are treated as a unit. In these applications, a group name can be specified with an arbitrary group name or single-range group name format.

group name *see* bus name.

H

hard logic function A logic function in a design file that is not removed during standard logic synthesis and therefore can be assigned to a physical resource such as a specific device, pin, logic cell, or I/O cell.

In Graphic Design Files (**.gdf**) and Text Design Files (**.tdf**), hard logic primitives/ports include `INPUT`, `INPUTC`, `OUTPUT`, `OUTPUTC`, `BIDIR`, `BIDIRC`, `LCELL`, `MCELL`, `DFF`, `DFFE`, `TFF`, `TFFE`, `JKFF`, `JKFFE`, `SRFF`, `SRFFE`, and `LATCH`. However, `INPUT` and `INPUTC` primitives that do not affect project outputs are not considered to be hard logic functions. When `SOFT`, `TRI`, and `OPNDRN` primitives are not removed during logic synthesis, they are also hard logic primitives. A macrofunction that contains a hard logic primitive is considered to be a hard logic function.

In Waveform Design Files (**.wdf**), hard logic functions are input nodes and output and buried nodes with registered and combinatorial node types.

hexadecimal The base 16 number system (radix). Hexadecimal digits are 0 through 9 and A through F.

In AHDL, hexadecimal numbers are indicated with the following notation:

X"*<series of digits 0 to 9, A to F>***"** *or*
H"*<series of digits 0 to 9, A to F>***"**

Example: `H"123AECF"`

Hexadecimal (Intel-format) File (.hex) A hexadecimal file (with the extension **.hex**) in the Intel Hex format.

The MAX+PLUS II Compiler and Simulator can use Hex Files as inputs to specify the initial contents of a memory (e.g., a ROM).

The MAX+PLUS II Compiler automatically creates output Hex Files containing configuration data for the Active Parallel Up (APU) configuration scheme for a FLEX 8000 devices, and the Passive Serial (PS) configuration scheme for FLEX 10K devices.

After compilation, you can also create Hex Files that support other configuration schemes for FLEX 8000 and FLEX 10K devices.

☞ If your project uses memory and you use a Hex File to specify its initial contents, you should name the file with a name that is not the same as the project name or any chip name within the project. Because the Compiler automatically generates Hex Files as outputs for FLEX 8000 and FLEX 10K devices, these output files may overwrite your initial memory content files.

hierarchical node or symbol name The unique name for a node or symbol that is based on its location in the hierarchy of design files and the net ID number or the AHDL or VHDL instance name of the logic function to which it is connected.

Every node and symbol in a project has a hierarchical name; you can also assign a node name or a probe name to a node.

Hierarchy Interconnect File (.hif) An ASCII file (with the extension **.hif**) created by the Compiler's Netlist Extractor module. This file specifies the hierarchical interconnections between design files in a project.

I

I/O cell An I/O cell is a register that exists on the periphery of a FLEX 10K, FLEX 8000, or MAX 9000 device (also known as an I/O element) or a fast input-type logic cell that is associated with an I/O pin in a MAX 7000E device. I/O cells permit short setup time.

☞ In pre-version 5.0 releases of MAX+PLUS II, I/O cells were known as peripheral registers.

I/O feedback Feedback from the output pin on an Altera device. It allows an output pin to be also used as an input pin.

Include File (.inc) An ASCII text file (with the extension **.inc**) that can be imported into a Text Design File (**.tdf**) by an AHDL Include Statement. The Include File replaces the Include Statement that calls it.

Include Files can contain Function Prototype, Define, Parameters, or Constant Statements. Include Files that contain Function Prototypes for Altera-provided mega- and macrofunctions are located in the **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories created during installation, respectively. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

insertion point The location at which text or graphics are inserted.

In a dialog box or in the Text Editor window, the insertion point appears as a flashing vertical bar. In the Graphic or Symbol Editor, it appears as a flashing square. In the Waveform Editor, an insertion point in the waveform drawing area appears as a short horizontal line that extends to the right of the Time cursor. In the node/group information area, a name or blank space that is selected is interpreted as an insertion point.

When you type text, it appears to the left of the insertion point, which moves to the right as you type. When you enter or paste symbols or waveforms, the upper left corner of the item(s) appears at the insertion point.

instance The use of a logic function in a design file. In the Graphic Editor, the instance is represented by the symbol (net) ID number in the lower left corner; in the Waveform Editor, it is the name of the node. In AHDL, instances are declared in one of two forms: an Instance Declaration that declares a variable of the type *<primitive>*, *<megafunction>*, or *<macrofunction>*, or an in-line logic function reference.

In the Hierarchy Display, an instance of a mega- or macrofunction is represented by the function name, followed by a colon (:) and a net ID number. In an AHDL Variable Declaration, an instance is represented by the instance name followed by a colon and the function name.

K

keyword Words that are reserved for implementing syntax in files used as inputs to MAX+PLUS II, including AHDL Text Design Files (**.tdf**), Assignment & Configuration Files (**.acf**), Command Files (**.cmd**), EDIF Command Files (**.edc**), Library Mapping Files (**.lmf**), VHDL Design Files (**.vhd**), and Vector Files (**.vec**). For example, the keyword `OF` cannot be used as an unquoted symbolic name in an AHDL file.

L

LAB *see* Logic Array Block.

latch A level-sensitive clocked storage unit that stores a single bit of data. A high-to-low transition on the Latch Enable signal fixes the contents of the latch at the value of the data input until the next low-to-high transition of the Latch Enable.

Latch Enable A level-sensitive signal that controls a latch. When it is high, the input flows through the output; when it is low, the output holds its last value.

LC *see* logic cell.

least significant bit (LSB) The bit of a binary number that contributes the smallest quantity to the value of that number, i.e., the last member in a bus or group name. For example, the LSB for a bus or group named a[31..0] is a[0] (or a0).

Library of Parameterized Modules (LPM) A technology-independent library of logic functions that are parameterized to achieve scalability and adaptability. Altera has implemented parameterized modules (also called "parameterized functions") from LPM version 2.0.1/2.1.0 that offer architecture-independent design entry for all MAX+PLUS II-supported devices. The MAX+PLUS II Compiler includes built-in compilation support for LPM functions used in schematic, AHDL, and EDIF input files.

logic function or Design Entity A primitive, megafunction, macrofunction, or state machine, which may be represented as either a name or a symbol in a design file.

Logic Array Block (LAB) A physically grouped set of logic resources in an Altera device. An LAB consists of a logic cell array and, in some device families, an expander product term array. Any signal that is available to any one logic cell in the LAB is available to the entire LAB.

In Classic devices, the logic in the LAB shares a global Clock signal. The LAB is fed by a global bus and a dedicated input bus. (In an EP1810 device, an LAB is synonymous with a quadrant.) In MAX 5000 and MAX 7000 devices, the LAB is fed by a Programmable Interconnect Array (PIA) and a dedicated input bus. In FLEX 8000, MAX 9000, and FLEX 10K devices, the LAB is fed by row interconnect paths and a dedicated input bus.

logic cell (LC) The generic term for a basic building block of an Altera device. In Classic, MAX 5000, MAX 7000, and MAX 9000 devices, a logic cell (also called a macrocell) consists of two parts: combinatorial logic and a configurable register. The combinatorial logic allows a wide variety of logic functions. In FLEX 8000 and FLEX 10K devices, a logic cell (also called a logic element) consists of a look-up table (LUT), i.e., a function generator that quickly computes any function of four variables, and a programmable register to support sequential functions.

The register can be programmed as a flow-through latch; as a D, T, JK, or SR flipflop; or bypassed entirely for pure combinatorial logic. The register can feed other logic cells or feed back to the logic cell itself. Some logic cells feed output or bidirectional I/O pins on the device.

You can assign a logic function to a specific logic cell. You can also assign a logic function to a logic array block (LAB), a row, or a column to ensure that the function is implemented in a logic cell in a particular LAB, row, or column.

In FLEX 10K, FLEX 8000, and MAX 9000 devices, logic cells have "numbers" of the format LC*<number>*_*<LAB name>*, where *<number>* ranges from 1 to 8 and *<LAB name>* consists of the row letter and column number of the LAB. In Classic, MAX 5000, and MAX 7000 devices, logic cells have numbers of the format LC*<number>*, where *<number>* may consist of both digits and letters.

☞ FLEX 10K, FLEX 8000, MAX 9000, and MAX 7000E devices have specialized logic cells, called I/O cells, on the periphery of the device.

logic element *see* logic cell.

logic level The input and output logic levels of nodes and groups are defined with the following characters:

Character:	**Logic Level:**
`0`	Logic low (GND)
`1`	Logic high (VCC)
`X`	Undefined/Don't Care (not permitted for initialization)
`Z`	High impedance (no input to pin); e.g., used for the "output" part of a bidirectional pin when the "input" part of the pin is driving in.
`0` to `9`, `A` to `F`	Used for groups and interpreted as binary, decimal, hexadecimal, or octal values according to the current radix. The most significant bit is first; the least significant bit is last.

logic option An option that controls the logic synthesis process on one or more logic functions.

A variety of logic options are available. Logic option assignments can be applied to individual logic functions; a group of logic option assignments, called a logic synthesis style, can be applied to individual logic functions. A default logic synthesis style is also applied to the project as a whole. The logic cell Turbo Bit logic option can also be turned on or off on a device-by-device basis.

Logic options can also be assigned as parameters for a megafunction or macrofunction.

☞ Some logic options are not available with standard synthesis; all logic options are available with multi-level synthesis.

logic synthesis style A combination of logic synthesis option settings that are saved under a single name.

A logic synthesis style can be individually tailored for different device families, so that the logic synthesis option settings vary according to the architecture of the target device family.

LPM *see* Library of Parameterized Modules.

LSB *see* least significant bit.

M

macrocell *see* logic cell.

macrofunction A high-level building block that can be used together with gate and flipflop primitives and/or megafunctions in MAX+PLUS II design files.

☞ In general, Altera recommends using megafunctions in preference to equivalent macrofunctions in all new projects. Megafunctions are easier to scale to different sizes and may offer more efficient logic synthesis and device implementation.

Altera provides a library of over 300 old-style macrofunctions in the **\maxplus2\max2lib** directory and its subdirectories created during installation. AHDL Include Files (**.inc**) for these macrofunctions are located in the **\maxplus2\max2inc** directory; VHDL Component Declarations for macrofunctions supported by VHDL are provided in the `maxplus2` package in the **altera** library, which is located in the **\maxplus2\max2vhdl** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

To view the file that contains the logic for a macrofunction, select the macrofunction symbol in the Graphic Editor or macrofunction name in the Text Editor and choose **Hierarchy Down** (File menu).

MAX 5000 An Altera device family based on the first generation of Multiple Array MatriX architecture. This EPROM-based device family includes the EPM5016, EPM5032, EPM5064, EPM5128, EPM5128A, EPM5130, EPM5192, and EPS464 devices.

MAX 7000 (and **MAX 7000E**) An Altera device family based on the second generation of Multiple Array MatriX architecture that includes MAX 7000 and MAX 7000E devices. These EPROM- and EEPROM-based devices include EPM7032, EPM7032V, EPM7064, EPM7096, EPM7128E, EPM7128, EPM7160E, EPM7160, EPM7192E, EPM7192, and EPM7256E devices.

MAX 7000E devices are enhanced versions of MAX 7000 devices and are function-, pin-, and programming-file-compatible with MAX 7000 devices. MAX 7000E devices differ from MAX 7000 devices in that they offer up to six pin- or logic-driven Output Enable signals, fast input setup times to logic cells, and multiple global Clocks with optional inversion.

☞ Altera recommends using MAX 7000E devices rather than MAX 7000 devices for new designs.

MAX 9000 An Altera device family based on the third generation of Multiple Array MatriX architecture. These EEPROM-based devices include the EPM9560, EPM9480, EPM9400, and EPM9320 devices.

MAX+PLUS (DOS) Altera's DOS-based Multiple Array MatriX Programmable Logic User System. MAX+PLUS is a set of computer programs and hardware support products for designing and implementing custom logic circuits with Altera Classic and MAX 5000 devices. Graphic Design Files (**.gdf**) created for MAX+PLUS are automatically converted and processed with the MAX+PLUS II Compiler; AHDL Text Design Files (**.tdf**) are compiled directly. The MAX+PLUS II Programmer can program Classic and MAX 5000 devices with JEDEC Files (**.jed**) and Programmer Object Files (**.pof**) created by MAX+PLUS.

☞ MAX+PLUS is no longer offered by Altera. All new designs should be created with MAX+PLUS II.

Mealy state machine A type of state machine in which the outputs are a function of the inputs and the current state.

Mealy, George H., A Method for Synthesizing Sequential Circuits, in *The Bell System Technical Journal*, Vol. 34, American Telephone and Telegraph Company (September 1955).

megafunction A complex or high-level building block that can be used together with gate and flipflop primitives and/or old-style macrofunctions in MAX+PLUS II design files.

Altera provides a library of megafunctions, including functions from the Library of Parameterized Modules (LPM), in the **\maxplus2\max2lib\mega_lpm** directory created during installation. AHDL Include Files (**.inc**) for these megafunctions are also located in the **\maxplus2\max2lib\mega_lpm** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

To view the file that contains the logic for a megafunction, select the megafunction symbol in the Graphic Editor or megafunction name in the Text Editor and choose **Hierarchy Down** (File menu).

memory bit and **memory word** A memory bit is an individual memory address in a memory (i.e., RAM or ROM) block.

A memory word is a group of memory bits in a RAM or ROM block.

For example, the `content5_[4..0]` memory word defines a byte of memory in which the individual memory bits are `content5_4`, `content5_3`, `content5_2`, `content5_1`, and `content5_0`.

Memory Initialization File (.mif) An ASCII file (with the extension **.mif**) that specifies the initial content of a memory block (RAM or ROM), i.e., the initial values for each address. This file is used during project compilation and/or simulation.

Moore state machine A state machine in which the present state depends only on its previous input and previous state, and the present output depends only on the present state.

Moore, Edward F., Gedanken-Experiments on Sequential Machines, in *Automata Studies, Annals of Mathematics Studies Number 34*, ed. C. E. Shannon and J. McCarthy, Princeton: Princeton University Press (1956).

most significant bit (MSB) The bit of a binary number that contributes the greatest quantity to the value of that number, and the first member in a bus or group name. For example, the MSB for a bus named `a[31..0]` is `a[31]`.

N

name characters The characters `A` to `Z`, `a` to `z`, `0` to `9`, slash (`/`), dash (`-`), and underscore (`_`) are legal for MAX+PLUS II breakpoint, chip, clique, file, group (bus), node, parameter, pin, pinstub, probe, logic synthesis style, and quoted and unquoted symbolic names, with the exceptions listed below. Case is not significant.

Item:	**Name Character Exception:**
filename	No slash (`/`) is permitted. Case is significant on UNIX workstations.

Item:	Name Character Exception:
single-range group (bus) name	No slash (`/`) is permitted. The name is followed by a range of numbers or arithmetic expressions in brackets. The start and end of the range are separated by two periods. For example, group `a[3..1]` consists of the nodes `a3`, `a2`, and `a1`. In Graphic Editor files only, sequential bus names can also include a series of single-range bus names. For example, `a[8..0],d[6..4]`.
dual-range group (bus) name	Same as single-range group names, with two ranges of numbers or arithmetic expressions in brackets. For example, `a[6..3][4..0]`.
sequential group (bus) name	The name consists of a series of comma-separated node names enclosed in parentheses. For example, group `(a, b, c)` consists of the nodes `a`, `b`, and `c`. In Graphic Editor files, parentheses are not used.
unquoted symbolic name (AHDL)	No dash (`-`) is permitted. Names cannot consist entirely of digits. AHDL keywords cannot be used.
VHDL names	No slash (`/`) or dash (`-`) is permitted. The name must start with a letter, cannot end with an underscore (_), and cannot contain two underscores (_ _) in a row. VHDL keywords cannot be used.
ACF names	Names that contain slash (`/`), dash (`-`), vertical bar (`|`), colon (`:`), and/or period (`.`) characters must be enclosed in double quotation marks (`"`).

nesting The repetition of an element or statement within an AHDL statement, e.g., an If Statement within an If Statement.

net ID number *see* symbol ID number.

node A node represents a wire carrying a signal that travels between different logical components of a design file.

In the Graphic Editor files, nodes are represented as lines; in text files, they are symbolic names; in Waveform Editor files, they are waveforms.

node name The name given to a signal in a design file. A node name can contain up to 32 of the following name characters: `A` to `Z`, `a` to `z`, `0` to `9`, slash (`/`), dash (`-`), and underscore (_). Hierarchical node names can contain 128 characters, including vertical bar (|), colon (:), and period (.). Case is not significant.

Some restrictions apply to names in VHDL Design Files (**.vhd**) and unquoted port and symbolic names in AHDL Text Design Files (**.tdf**).

O

octal The base 8 number system (radix). Octal digits are 0 though 7.

In AHDL, octal numbers are indicated with the following notation:

`O"`*<series of digits 0 to 7>*`"` *or*
`Q"`*<series of digits 0 to 7>*`"`

Example: `Q"4671223"`

one-hot encoding A type of binary coding in which one and only one bit of a value is set to 1. For example, the four legal values `0001`, `0010`, `0100`, and `1000` together comprise a "one-hot" code sample because in each of these four values a single bit is set to 1.

You can manually implement one-hot encoding. In addition, the **Global Project Logic Synthesis** dialog box (Assign menu) includes a *One-Hot State Machine Encoding* option to allow the Compiler to automatically implement one-hot encoding for the entire project. Altera strongly recommends using the *One-Hot State Machine Encoding* option rather than manual one-hot encoding to implement one-hot encoding.

one's complement A system of representing binary numbers in which the negative of a number is obtained by inverting each bit individually.

operand A node, group, or number that is acted upon in an operation.

operator A symbol that signifies the action of an operation. AHDL and VHDL offer both logical and arithmetic operators.

Output Enable A high logic level on the Output Enable signal enables the output.

In MAX 7000 devices (not including MAX 7000E devices), the signal from the active-low global Output Enable pin must be inverted and connected to the active-high Output Enable input of the `TRI` primitive. In all other device families, either active-high or active-low polarity can be used.

In MAX 9000 devices, the Fitter automatically inserts additional `LCELL` primitives to provide the correct polarity for a non-global Output Enable pin or an Output Enable signal driven by a logic cell.

P

parameter or parameterized A parameter is an attribute of a logic function that determines the logic created or used to implement the function, i.e., a characteristic that determines the size, behavior, or silicon implementation of a function. The parameter information can be used to determine the actual primitives and other subdesigns needed to implement the logic of the function.

A parameterized function is a function whose behavior is controlled by one or more parameters. Some logic functions, such as the functions in the Library of Parameterized Modules (LPM), are inherently parameterized and require parameter values to be assigned.

Parameters can be assigned to any individual instance of a megafunction in

MAX+PLUS II to control its size or implementation. Some parameters can also be applied to old-style macrofunctions to determine their style of implementation. MAX+PLUS II also allows you to assign global, project-wide default values for parameters.

parameterized module A logic function that uses parameters to achieve scalability, adaptability, and efficient silicon implementation. MAX+PLUS II supports a variety of parameterized modules (also called "parameterized functions"), including functions belonging to the Library of Parameterized Modules (LPM).

LPM functions provide architecture-independent design entry for all MAX+PLUS II-supported devices. The MAX+PLUS II Compiler includes built-in compilation support for LPM functions used in schematic, AHDL, and EDIF input files.

pin A pin is an actual input or I/O pin on an Altera device.

In Graphic Editor files, a pin is represented by an `INPUT`, `INPUTC`, `OUTPUT`, `OUTPUTC`, `BIDIR`, or `BIDIRC` symbol. In a Text Design File (**.tdf**), a pin is represented as an `INPUT`, `OUTPUT`, or `BIDIR` port. In a VHDL Design File (**.vhd**), a pin is represented as an `IN`, `OUT`, or `INOUT` port. In a Waveform Design File (**.wdf**), a pin is represented as a node with an input, output, or bidirectional I/O type and a pin input, registered, or combinatorial node type.

You can assign a logic function to a specific pin number. You can also assign a logic function to a row or a column to ensure that the function is implemented in a pin on a particular row or column.

pin number A number used to assign an input or output signal in a design file, which corresponds to the pin number on an actual device.

Both letters and digits are used to specify pin numbers for PGA-package devices.

pinstub In the Graphic and Symbol Editors, a pinstub is the location on the boundary of a symbol represented by an "**x**" in a Symbol File (**.sym**) and a name that represents an input or output of the primitive or of the megafunction or macrofunction design file that the symbol represents. A line (node) drawn in a schematic must connect to this pinstub to be recognized by the Compiler as a connection between the logic in the current file and the logic in the primitive, megafunction, or macrofunction.

You can specify whether or not to use an optional pinstub when you edit a symbol instance in a Graphic Editor file.

Pinstubs in Graphic Editor files are synonymous with ports in AHDL Function Prototypes and VHDL components. They are also synonymous with ports listed in the Subdesign Sections of lower-level Text Design Files (**.tdf**), and in Entity Declarations of lower-level VHDL Design Files (**.vhd**).

pinstub name A symbolic name that identifies an input or output of a logic function.

In the Symbol Editor, the "visible" pinstub name appears both inside and outside of the symbol. This "visible" pinstub name can be an abbreviation or an alias for the "full" pinstub name, which represents the full name of the original input, output, or bidirectional pin in a mega- or macrofunction design file or primitive Function Prototype.

You can specify whether or not to display the "visible" pinstub name in a Graphic Editor file when you create a pinstub in the Symbol Editor. The use or non-use of a particular pinstub (and hence its visibility) can be customized when you edit a symbol instance in the Graphic Editor with **Edit Ports/Parameters** (Symbol menu).

Pinstubs in Graphic Editor files are synonymous with ports in AHDL Function Prototypes and signals listed in the Subdesign Sections of lower-level Text Design Files (**.tdf**).

port A symbolic name that represents an input or output of a primitive or of a design file.

In AHDL, a port name in the Subdesign Section represents an input or output of the current file. This port name also appears in the Function Prototype for the function. When an instance of a primitive or lower-level design file is implemented with an Instance Declaration or an in-line reference, its ports are used to connect it to other functions in the TDF. After an instance is declared, its inputs and outputs are expressed as names in the format *<instance name>*. *<port name>* in the Logic Section. When an in-line reference is used, either named port association or positional port association can be used to connect the function's ports to other functions in the TDF.

A port name in an AHDL Subdesign Section or VHDL Entity Declaration is synonymous with a pin name in a Graphic Design File (**.gdf**) or Waveform Design File (**.wdf**). A port name that is appended to an instance name is synonymous with the full pinstub name in an instance of a symbol in a Graphic Editor file.

Preset An input signal that asynchronously sets the output of a register to a logic high (`1`), regardless of other inputs.

primitive One of the basic functional blocks used to design circuits with MAX+PLUS II software. Primitives are used in Graphic Design Files (**.gdf**), AHDL Text Design Files (**.tdf**), and VHDL Design Files (**.vhd**).

Graphic Editor primitives include buffers, flipflops, a latch, input and output primitives, and logic primitives. Primitive symbols for Graphic Editor files are provided in the **\maxplus2\max2lib\prim** directory created during installation.

AHDL and VHDL primitives, which include buffers, flipflops, and a latch, are a subset of the primitive symbols used in Graphic Editor files. Other functions are represented by logical operators, ports, and other constructs. Function Prototypes for AHDL primitives are built into the MAX+PLUS II software; Component Declarations for VHDL primitives are provided in the `maxplus2` package in the **altera** library, which is located in the **\maxplus2\max2vhdl** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

probe A unique name assigned to any node, e.g., the input or output of a primitive or macrofunction, which can be used instead of the full hierarchical node name throughout MAX+PLUS II. A probe name thus provides a short name to identify a node.

product term Two or more factors in a Boolean expression combined with an AND operator constitute a product term, where "product" means "logic product."

project A project consists of all files that are associated with a particular design, including all subdesign files and related ancillary files created by the user or by MAX+PLUS II software. The project name is the same as the name of the top-level design file in the project, without the filename extension.

MAX+PLUS II performs compilation, simulation, timing analysis, and programming on only one project at a time.

R

radix A number base. Group logic level and numerical values are entered and displayed in binary, decimal, hexadecimal, or octal radix in MAX+PLUS II.

range A sequence of numbers or arithmetic expressions that define the width of a group (bus) in a Graphic Editor or AHDL file. A range is enclosed in brackets; the most significant bit (MSB) of the range is shown first; the least significant bit (LSB) is shown last. The start and end of the range are separated by two periods.

Example: group `a[2..0]` consists of the nodes `a2`, `a1`, and `a0`; the MSB is `a2`; and the LSB is `a0`.

register *see* flipflop.

registered feedback Feedback that is the output of a flipflop or latch.

registered output The output of a flipflop or latch, which can feed an output pin on the device.

Reset An active-high input signal that asynchronously resets the output of a register to a logic low (`0`) or a state machine to its initial state, regardless of other inputs.

resource A resource is a portion of an Altera device that performs a specific, user-defined task (e.g., pins, logic cells).

resource assignment An assignment of a logic function in a project to a particular pin, logic cell, I/O cell, embedded cell, logic array block (LAB), embedded array block (EAB), row, column, or chip. This type of resource assignment assigns a logic function to a physical resource in a device.

A resource assignment can also consist of a clique, logic option, connected pin, or timing requirement assignment to a particular logic function in a project. This type of resource assignment assigns a compilation resource to a logic function.

S

SDF Output File *see* Standard Delay Format Output File.

secondary input The Clock, Preset, and Reset (Clear) inputs to a register or a state machine in a design file.

sign-extend To extend a two's complement binary number by padding to the left with 0's if the number is positive, or with 1's if the number is negative.

single-range group (or bus) name The name of a group (or bus) of up to 256 nodes, consisting of up to 32 name characters, followed by a range of numbers or arithmetic expressions in brackets. The start and end of the range are separated by two periods. Each number in the sequence represents an individual node (or "bus bit").

Example: group `a[4..1]` consists of the nodes `a4`, `a3`, `a2`, and `a1`.

In a Graphic Editor files, a sequential bus name can also include one or more single-range bus names in a series. The first node of the series or the first node in the first range is the most significant bit of the bus; the last node of the series or the last node in the last range is the least significant bit.

Example: `a[8..0], b1, dout[6..4]`

SMF *see* State Machine File.

Standard Delay Format Output File (.sdo) An optional output file (with the extension **.sdo**) containing timing delay information that allows you to perform back-annotation for simulation with VHDL simulators that use VITAL-compliant simulation libraries; back-annotation for simulation in Verilog simulators; and timing analysis and resynthesis with EDIF simulation and synthesis tools.The Standard Delay Format (SDF) is an industry-standard format.

The MAX+PLUS II Compiler's EDIF, VHDL, and Verilog Netlist Writer modules of the MAX+PLUS II Compiler can generate SDF Output Files in SDF version 2.1 or 1.0 format.

state A state is implemented in a device as a pattern of 1's and 0's (bits) that are the outputs of multiple flipflops (collectively called a state machine state register). States can be defined in an AHDL Text Design File (**.tdf**), a Waveform Design File (**.wdf**), a Vector File (**.vec**), a VHDL Design File (**.vhd**), or a State Machine File (**.smf**), and are reported in the State Machine Assignments Section of the Report File (**.rpt**).

state bit An output of a flipflop used by a state machine to store one bit of the value of the state machine.

state control equation An AHDL equation that assigns a value to the Clock, Clock Enable, or Reset port(s) of the D or T flipflop(s) on which a state machine is implemented.

state machine A sequential circuit that advances through a number of states. A state machine can be defined in a Waveform Design File (**.wdf**), State Machine File (**.smf**), Vector File (**.vec**), VHDL Design File (**.vhd**), or in a State Machine Declaration in an AHDL Text Design File (**.tdf**).

State Machine File (.smf) An ASCII file (with the extension **.smf**) that contains a state machine design created for use with Altera's A+PLUS or SAM+PLUS software. This file contains a symbolic representation of the data for a circuit in terms of inputs, outputs, and transitions between states. The MAX+PLUS II Compiler automatically translates an SMF into an Altera Design File (**.adf**) and a Compiler Netlist File (**.cnf**) during compilation.

state name A symbolic name that represents the state of a state machine.

state transition A conditional assignment of a state to the state machine variable. State transitions are created by conditionally assigning the state variables with a single behavioral construct.

In AHDL, state transitions are created with Case or Truth Table Statements. State transitions occur on the rising edge of the Clock.

In VHDL, state transitions are created with Case Statements. You must also provide a Wait Statement to cause each state transition to occur on a Clock edge.

subdesign A lower-level design file in a MAX+PLUS II project, i.e., an Altera-provided or user-created megafunction or macrofunction.

Altera provides libraries of mega- and macrofunctions in the **mega_lpm** and **mf** subdirectories of the **\maxplus2\max2lib** directory. AHDL Include Files (**.inc**) for these functions are located in the **\maxplus2\max2lib\mega_lpm** and **\maxplus2\max2inc** directories, respectively. Component Declarations for functions supported by VHDL are provided in the `maxplus2` package in the **altera** library, which is located in the **\maxplus2\max2vhdl** directory. (On a UNIX workstation, the **maxplus2** directory is a subdirectory of the **/usr** directory.)

subdesign name A name that represents the name of a subdesign. In AHDL, the subdesign name is a quoted or unquoted symbolic name that must be the same as the Text Design File (**.tdf**) filename.

Unquoted subdesign name:

Maximum length:	32 characters
Legal characters:	`a-z`, `A-Z`, `0-9`, and underscore (_) An unquoted subdesign name cannot be a reserved AHDL identifier or keyword.

Quoted subdesign name:

Maximum length:	32 characters
Legal characters:	`a-z`, `A-Z`, `0-9`, dash (`-`), and underscore (_)

☞ In the UNIX workstation environment, filenames and hence subdesign names are case-sensitive.

sum-of-products A Boolean expression is said to be in sum-of-products form if it consists of product terms combined with the `OR` operator.

symbol ID number or **net ID number** A number that uniquely identifies every node and symbol in a design file.

In the Graphic Editor, this number appears inside the bottom left corner of a symbol and reflects the order in which symbols are entered in a Graphic Editor file. In other types of design files, the Compiler assigns ID numbers to nodes when the project is compiled. In the Hierarchy Display window, the name of each lower-level design file is appended with a colon (`:`) plus the ID number or an AHDL or VHDL mega- or macrofunction instance name.

symbolic name A user-defined name in AHDL.

Unquoted symbolic name:

Maximum length:	32 characters
Legal characters:	a-z, A-Z, 0-9, slash (/), and underscore (_) An unquoted symbolic name cannot consist entirely of digits and cannot be a reserved identifier or keyword.

Quoted symbolic name:

Maximum length:	32 characters
Legal characters:	a-z, A-Z, 0-9, slash (/), dash (-), and underscore (_) A quoted symbolic name cannot be a reserved identifier.

T

TDF *see* Text Design File.

ternary operator An operator that selects between two expressions within an AHDL arithmetic expression. The ternary operator is used in the following format:

<expn 1> ? *<expn 2>* : *<expn 3>*

If the first expression is non-zero (true), the second expression is evaluated and given as the result of the ternary expression. Otherwise, the third expression is evaluated and given as the result of the ternary expression.

Text Design Export File (.tdx) An ASCII text file (with the extension **.tdx**) in AHDL that is optionally generated when you compile a Xilinx Netlist Format File (**.xnf**). It contains the same logic as the XNF File.

A Text Design Export File can be saved as a Text Design File (**.tdf**) and used to replace the corresponding XNF File in the hierarchy of a project.

Text Design File (.tdf) An ASCII text file (with the extension **.tdf**) written in AHDL. Text Design Export Files (**.tdx**) and Text Design Output Files (**.tdo**) can be saved as TDFs and compiled with MAX+PLUS II.

Text Design Output File (.tdo) An ASCII text file (with the extension **.tdo**), generated by the MAX+PLUS II Compiler, that contains the AHDL equivalent of the fully optimized logic for a device in the project.

The Compiler generates a TDO File, as well as an Assignment & Configuration Output File (**.aco**) when you compile a project if you turn on the **Generate AHDL TDO File** command (Processing menu).

You can save a TDO File as a Text Design File (**.tdf**) and recompile it. (You must also save the Assignment & Configuration Output File (**.aco**) as an Assignment & Configuration File (**.acf**) if you wish to preserve the assignments for the device.) TDO Files facilitate back-annotation and preserve the existing logic synthesis in the project.

tri-state buffer A buffer with an input, output, and controlling Output Enable signal. If the Output Enable input is high, the output signal equals the input. If the Output Enable input is low, the output signal is in a state of high impedance. The tri-state buffer is implemented with the `TRI` primitive.

Tri-state buses can be implemented by tying multiple nodes together in a Graphic Editor file and with the `TRI_STATE_NODE` variable in an AHDL file.

truncate In AHDL, to shorten a binary number by subtracting digits from the left.

two's complement A system of representing binary numbers in which the negative of a number is equal to its inverse plus 1. Arithmetic operators in AHDL assume that groups they operate on are a two's complement binary number.

U

unary An arithmetic operator that operates only on one operand.

user libraries One or more directories that contain your own megafunctions, macrofunctions, Symbol Files (**.sym**), AHDL Include Files (**.inc**), or precompiled, user-defined VHDL packages.

The Compiler automatically searches for these user-specified libraries when it compiles a project. The Compiler's **VHDL Netlist Settings** command (Interfaces menu) specifies VHDL design libraries for the current project. You can specify which directories contain your other user libraries with the **User Libraries** command (Options menu) in any MAX+PLUS II application.

V

variable A name that represents a node. In AHDL, a variable can also represent a state machine or an instance of a primitive, megafunction, or macrofunction and is declared in the Variable Section. In VHDL, variables have a single current value, and are declared and used only in processes and subprograms. A VHDL variable is declared with a Variable Declaration; the value of a variable can be modified with a Variable Assignment Statement.

VCC A high-level input voltage represented as a high (1) logic level in binary group values.

In an AHDL Text Design File (**.tdf**), `VCC` is a predefined constant and keyword, and the default active node value. In a VHDL Design File (**.vhd**), `VCC` is represented by `'1'`. In a Graphic Editor file, `VCC` is a primitive symbol. `VCC` is represented as a high (1) logic level in the Simulator and Waveform Editor.

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.

You can create a VHDL Design File (**.vhd**) with the MAX+PLUS II Text Editor or any standard text editor and compile it directly with MAX+PLUS II. You can also generate an EDIF 2 0 0 or 3 0 0 netlist file from a VHDL design that has been processed with a VHDL synthesis tool, then import the file into MAX+PLUS II as an EDIF Input File (**.edf**). The MAX+PLUS II Compiler can also generate a VHDL Output File (**.vho**) that contains functional and timing information for simulation with a standard VHDL simulator.

VHDL Design File (.vhd) An ASCII text file (with the extension **.vhd**) written in VHDL. VHDL Design Files can be compiled by the MAX+PLUS II Compiler.

X

Xilinx Netlist Format (XNF) File (.xnf) A netlist file (with the extension **.xnf**) generated by Xilinx software. XNF Files that are generated by running the Xilinx LCA2XNF utility can be compiled directly by the MAX+PLUS II Compiler. An XNF File can define all logic in a project, or be incorporated at the bottom level in a hierarchical project.

Index

All index references are to the ***MAX+PLUS II AHDL*** manual. Definitions of technical terms are given in *Glossary*.

Symbols

- character 11, 95, 103, 109, 112
-- characters 95, 191
! character 95, 103, 107, 108, 112
!# characters 96, 104, 107, 108, 112
!$ characters 96, 104, 107, 112
!& characters 96, 107, 108, 112
!= characters 11, 95, 103, 111, 112
" character 94, 141, 145, 190
# character 96, 104, 107, 108, 112
$ character 96, 104, 107, 108, 112
% character 94, 191
& character 96, 104, 107, 108
' character 94
( ) characters 94, 99, 112, 190
* character 103
+ character 11, 95, 103, 109, 112
, character 43, 94, 100, 169, 190
. character 94, 133, 134
.. characters 94, 100
/ character 42, 94, 192
: character 94, 190

; character 94, 190
< character 11, 96, 104, 111, 112
<= characters 96, 104, 111, 112
= character 94
== characters 11, 95, 103, 111, 112
=> characters 33, 36, 94, 172, 183
> character 11, 95, 104, 111, 112
>= characters 11, 95, 104, 111, 112
? character 96, 104
@ character 94
[] characters 28, 94, 99
\ character 94
^ character 103
_ character 94, 192
~ character 85, 100, 162

A

a6502 megafunction 130
ACF 2, 11
active-low signals 41, 173, 192
ADF 3
AHDL
- automatic LPM substitution for operators & comparators 11
- comments 191
- elements 91
- examples 9, 24
- file structure 4
- formatting guidelines 188
- general description 2
- golden rules 14
- syntax coloring 10
- templates 9, 22
- using 17

AHDL Template command 22
ahdl.tpl file 23
Altera Design Files (**.adf**) 3
AND operator 104, 107, 108
arithmetic comparators 111, 112
arithmetic expressions 100, 103
arithmetic functions 20
arithmetic operators 109, 112
ASSERT keyword 155

Assert Statement
- general description 155
- implementing 21, 89
- position in a TDF 5
- sample files 21, 90

Assign menu commands 2, 11
Assignment & Configuration File (**.acf**) 2, 11
assignments, resource & device 11
Auto-Indent command 188, 193

B

Backus-Naur Form xvii
`BEGIN` keyword 168
`BIDIR` keyword (*see* ports)
bidirectional pins 43
binary numbers 18, 102
`BITO` option (*see* Options Statement)
BNF xvii
Boolean equations
- control equations 57, 171
- general description 168
- general rules 169
- implementing 25
- position in a TDF 5
- priority rules 112
- using groups 28

Boolean expressions 108, 112
- arithmetic operators 109
- comparators 111
- general description 106
- implementing 25
- in Case Statements 34
- in If Then Statements 31, 34, 176
- logical operators 107, 108
- naming operators & comparators 84

buffer primitives 114–125
buses, tri-state (*see* groups *and* tri-state buses)

C

Carry Chain logic option 115

CARRY primitive
 general description 114
 primitive/port interconnections 127
 primitive/port to register connections 128
carry-out signal 110, 169
Cascade Chain logic option 116
CASCADE primitive
 general description 116
 primitive/port interconnections 127
 primitive/port to register connections 128
CASE keyword 172
case sensitivity 146
Case Statement
 compared with If Then Statement 34
 formatting guidelines 194
 general description 172
 implementing 31, 32
 position in a TDF 5
 recovering from illegal states 67
 sample files 33, 55, 57, 68, 78
 specifying default values for variables 39, 173
 specifying output values for states 56
 state transitions 56
CEIL function 105
chip assignments 11
Clear signal 118
clique assignments 11
Clique command 11
CLIQUE keyword 92
Clock Enable signal 48, 57, 171
Clock signal 118
 global 50
 in state machines 57, 60, 64, 171
 loading a register 52
 synchronizing state machine outputs 63
Color Palette command 10
column assignments 11
combinatorial feedback 122
combinatorial logic 25–45, 124
comments 191
comparators 25, 84, 103, 111, 112, 190
 context-sensitive help 10
 substituting LPM functions 11
 syntax coloring 10
conditional logic 31–35, 172, 176

conditionally generated logic 87
Connected Pins command 11
CONNECTED_PINS keyword 92
CONSTANT keyword 147
CONSTANT primitive 113
Constant Statement 101, 103
  general description 147
  implementing 19, 21
  position in a TDF 4
  sample files 20, 21
constants, in Boolean expressions & equations 25
control equations (*see* Boolean equations)
counters 51
Create Default Include File command 8, 70, 76, 77, 145, 153
Create Default Symbol command 76
creating evaluated functions 149
csdpram megafunction 83, 130
csfifo megafunction 83, 130

D

decimal numbers 18, 102
decoders 19, 32, 35, 36, 37, 70
Decrease Indent command 188, 193
default pin/port values 157
DEFAULTS keyword 173
Defaults Statement
  general description 173, 185
  implementing 39
  position in a TDF 5
  sample files 39, 40, 42
DEFINE keyword 149
Define Statement
  general description 149
  implementing 19, 21
  position in a TDF 4
  sample files 21
delay chains 121
Design Doctor Settings command 121
DESIGN keyword 92
device assignments 11
DEVICE keyword 92
DEVICE_FAMILY parameter 87
DFF primitive 50, 51, 63, 93, 126, 127
DFFE primitive 48, 50, 51, 93, 126, 127

DIV operator 103
documentation conventions xiv
dual-range group names 99

E

EDIF Input Files (**.edf**) 3
Edit Ports/Parameters command 137
ELSE GENERATE keywords 178
ELSE keyword 176
ELSIF keyword 176
embedded cell assignments 11
Enable signal 48, 171
END CASE keywords 172
END DEFAULTS keywords 173
END GENERATE keywords 178, 179
END IF keywords 176
END keyword 168
END TABLE keywords 183
Enter Parameters command 137
evaluated expression 103
evaluated functions
    creating 149
    implementing 19
    testing 155
examples xix, 9
EXP primitive
    conversion to LCELL buffer 117
    general description 117
    primitive/port connections 127
    primitive/port to register connections 128
expander product terms 117
exponent operator 103

F

Fitter Settings command 122
flipflops 125
FLOOR function 105
Floorplan Editor 11, 12

For Generate Statement
  formatting guidelines 194
  general description 179
  implementing 86
  position in a TDF 5
  sample files 86
FOR keyword 179
formatting guidelines 188–197
FUNCTION keyword 151
Function Prototype Statement
  general description 151
  implementing megafunctions & macrofunctions 69
  position in a TDF 4
  sample files 44, 71, 79

G

GDF 2
Generate AHDL TDO File command 13
GENERATE keyword 178, 179
GLOBAL primitive 50
  general description 118
  primitive/port interconnections 127
  primitive/port to register connections 128
Global Project Logic Synthesis command 50, 58, 67, 123, 137
Global Project Parameters command 137
global signals 50, 118
GND 30, 39, 40, 102, 127, 128, 157, 173
golden rules 14
Graphic Design File (**.gdf**) 2
Graphic Editor 137
groups
  bit-ordering 154
  default values 39
  defining 28
  general description 99
  in Boolean equations 25, 28, 169
  in Boolean expressions 108, 109, 111
  in truth tables 184
  names 99
  ranges 100, 103
  set equal to VCC & GND 30
  sign-extending 30, 110

H

Help
- context-sensitive 10
- examples 24
- updates xviii

HELP_ID keyword 155
hexadecimal numbers 18, 102
hierarchical designs
- general description 3, 7
- implementing in AHDL 69–80

Hierarchy Down command 16

I

identifiers, reserved 93
If Generate Statement
- formatting guidelines 194
- general description 178
- implementing 87
- position in a TDF 5
- sample files 87, 90

IF keyword 176, 178
If Then Statement
- compared with Case Statement 34
- Compiler interpretation 34
- creating counters 51
- formatting guidelines 193
- general description 176
- implementing 31
- position in a TDF 5
- sample files 31, 42
- specifying default values for variables 39, 173
- specifying output values for states 56

Include File (**.inc**) 2, 7, 70, 74, 76, 145
INCLUDE keyword 145
Include Statement
- general description 145
- implementing 70
- position in a TDF 4

Increase Indent command 188, 193

in-line logic function reference
    general description 180
    implementing 69
    named port association 71
    position in a TDF 5
    positional port association 71
    RETURNS clause 72
INPUT keyword (*see* ports)
Instance Declaration 80
    general description 160
    implementing 69
    position in a TDF 5
IS keyword 172
iteratively generated logic 86

J

JKFF primitive 48, 93, 126, 127, 128
JKFFE primitive 48, 93, 126, 127, 128

K

keywords
    context-sensitive help 10
    reserved AHDL keywords 92, 93
    syntax coloring 10

L

LATCH primitive 93, 126, 127, 128
LATENCY parameter 138
LCELL primitive
    converting EXP buffer 117
    converting SOFT buffer 123
    general description 120
    implementing 81
    primitive/port interconnections 127
    primitive/port to register connections 128
Library of Parameterized Modules (LPM) 2, 11, 129
location assignments 11
LOG2 operator 103, 105

logic
    active-low 41
    combinatorial 25–45
    conditional 31–35, 172, 176
    sequential 47–53
Logic Array Block 117
logic cells
    assignments 11
    converting `SOFT` buffer 123
logic expansion 124
logic option assignments 11
    Carry Chain logic option 115
    Cascade Chain logic option 116
    inserting `LCELL` buffers 122
    inserting `SOFT` buffers 124
    used as parameters 138
Logic Options command 11
Logic Section
    Assert Statement 89, 155
    Boolean control equations 57, 171
    Boolean equations 168
    Case Statement 31, 32, 34, 39, 56, 67, 172, 173, 194
    Defaults Statement 39, 173
    For Generate Statement 179
    formatting guidelines 193
    general description 168
    If Generate Statement 178
    If Then Statement 31, 34, 39, 51, 56, 173, 176, 193
    in-line logic function reference 69, 180
    ports 132
    position in a TDF 5
    Truth Table Statement 35, 39, 56, 60, 61, 63, 173, 183
Logic Synthesizer 60, 62, 117, 123
logic, reducing complexity 124
logical comparators 111, 112
logical operators 25, 84, 106, 107, 108, 112
LPM 2, 11, 111, 129
`lpm_add_sub` 11, 74
`lpm_compare` 11, 35, 111
`lpm_counter` 51
`lpm_decode` 35
`lpm_dff` 48
`lpm_ram_dq` 83
`lpm_ram_io` 83

lpm_rom 83
LSB keyword (*see* Options Statement)

M

Machine Alias Declaration
    general description 166
    implementing 77
    position in a TDF 5
    sample files 79
MACHINE INPUT keywords 77, 157, 166
MACHINE keyword 151, 165, 166
MACHINE OUTPUT keywords 77, 157, 166
macrofunctions
    context-sensitive help 10
    formatting guidelines 188
    general description 131
    implementing 69–80, 159, 160, 180
    in-line reference 69
    Instance Declaration 69
    instances 159, 160
    ports 70, 72
    prototypes 151
    syntax coloring 10
    unconnected inputs 72
    user-defined 76
MAX+PLUS II Floorplan Editor 11, 12
MAX+PLUS II Graphic Editor 137
MAX+PLUS II manuals
    documentation conventions xiv
    help updates xviii
    list of documents xii
MAX+PLUS II Symbol Editor 137
MAX+PLUS II Text Editor
    AHDL templates & examples 9
    context-sensitive help 10
    error location 12
    resource & device assignments 11
max2inc directory 8, 131, 145
max2lib directory 8, 131, 145
max2work\ahdl directory 9, 24
maxplus2.idx file 16
maxplus2.ini file 16
maxplus2\max2inc directory 8, 131, 145
maxplus2\max2lib directory 8, 131, 145

MCELL primitive 93, 121
Mealy state machines 64
mega_lpm directory 129, 145
megafunctions
- context-sensitive help 10
- formatting guidelines 188
- general description 129
- implementing 69–80, 159, 160, 180
- in-line reference 69
- Instance Declaration 69
- instances 159, 160
- ports 70, 72
- prototypes 151
- syntax coloring 10
- unconnected inputs 73
- user-defined 76

MEMORY identifier 93
message location 12
MOD operator 103
Moore state machines 60
MSB keyword (*see* Options Statement)
multiplication operator 103

N

named port association 71, 135
names
- formatting guidelines 189
- group 99
- naming conventions 192
- on logical operators & comparators 84
- ports 97, 133
- subdesign 97
- symbolic 97, 132

NAND operator 104, 107, 108
net ID numbers 85
Node Declaration
- general description 162
- implementing 27, 81
- position in a TDF 5
- sample files 82

NODE keyword 159, 162

nodes
- active-low 41, 173
- creating 162
- declaring 27, 162
- default values 39
- grouping 28, 99
- in Boolean equations 25, 168, 169
- in Boolean expressions 108, 111
- in truth tables 184
- names 42

NOR operator 104, 107, 108
NOT operator 107, 108
ntsc megafunction 130
numbers 149, 169
- general description 102
- implementing 18, 19
- in Boolean equations 112
- in Boolean expressions 108, 109
- replacing with constants 147
- sign-extended 169
- truncated 169

numeric equality operator 104

O

obsolete syntax 140
octal numbers 18, 102
OF BITS keywords 165
old-style macrofunctions (*see* macrofunctions)
one-hot code 32
one-hot encoding 67
One-Hot State Machine Encoding option 58, 67
open-drain pins 122
operands 106
operator and comparator priorities 112
operators 25, 103, 106, 107, 108, 109, 112, 190
- context-sensitive help 10
- substituting LPM functions 11
- syntax coloring 10

OPNDRN primitive
- general description 122
- primitive/port interconnections 127
- primitive/port to register connections 128

OPTIONS keyword 154

Options Statement
- `BITO` option 100
- general description 154
- implementing 28
- position in a TDF 4
- sample files 29

`OR` operator 104, 107, 108
OrCAD Schematic Files (**.sch**) 3
Output Enable signal 118
`OUTPUT` keyword (*see* ports)

P

`PARAM` primitive 113
parameters 20
- creating 142
- general description 137
- in Assert Statements 89
- in Function Prototypes 151
- in Instance Declarations & in-line references 73
- testing 155

`PARAMETERS` keyword 142
Parameters Statement
- Compiler parameter value search order 137
- formatting guidelines 189
- general description 142
- position in a TDF 4

pin assignments 11
Pin/Location/Chip command 11
pins
- bidirectional 43
- default values 157
- naming 132

placeholders 43, 100
`pll` megafunction 130
ports
- bidirectional 43
- commonly used ports 135
- default values 157
- formatting guidelines 188
- general description 132
- groups 99
- named association 71
- names 97
- positional association 71

- primitive/port interconnections 127
- primitives 163
- replacing INPUT, OUTPUT & BIDIR primitives 113
- specifying the port type 157
- testing usage with USED 87, 155
- using 47, 70, 72, 160

positional port association 71
Preset signal 118
primitives
- context-sensitive help 10
- flipflops and latch 125–126
- general description 113
- implementing 50, 81, 159, 160, 163, 180
- Instance Declaration 82
- legal interconnections 127
- ports 47, 163
- prototypes 151
- Register Declaration 47, 50, 51
- syntax coloring 10
- unconnected inputs 72

priority rules, in Boolean equations 112
product terms
- expanders 117
- reducing number 123

Project Archive command 16
Project Name command 16
Project Save & Check command 13, 77, 146
Project Save & Compile command 13
Project Set Project to Current File command 16
punctuation
- context-sensitive help 10
- syntax coloring 10

Q

quoted names 97

R

radixes 102
RAM, implementing 83
ranges 99, 100, 103
- group 154

read.me file xviii

Register Declaration
- creating counters 51
- declaring registered outputs 50
- general description 163
- implementing 47
- position in a TDF 5

REPORT keyword 155
reserved identifiers 93
reserved keywords 92
Reset signal 57, 171
resource assignments 11
RETURNS clause 72
RETURNS keyword 151
ROM, implementing 83
row assignments 11

S

Save As command 16
SCH file 3
SCLK primitive 118
SEGMENTS keyword 92
sequential group names 100
sequential logic 47–53
SEVERITY keyword 155
Shift+F1 keys 10
signals, active-low 41, 192
sign-extended numbers 169
single-range group names 99
SMF 3
SOFT primitive
- comparison to LCELL primitive 121
- general description 123
- implementing 81
- primitive/port interconnections 127
- primitive/port to register connections 128

SRFF primitive 48, 93, 126, 127
SRFFE primitive 48, 93, 126, 127
State Machine Declaration
- assigning state bits & values 58
- formatting guidelines 195
- general description 165
- implementing 55
- Mealy state machines 64
- Moore state machine 60

T

TABLE keyword 183
TDO file 2
TDX file 2
templates, AHDL 9, 22
ternary operator 104
Text Design Export File (**.tdx**) 2
Text Design File (**.tdf**)
    checking syntax 13
    comments 191
    compiling 13
    formatting guidelines 188–195
    in hierarchical designs 7
    structure 4
Text Design Output File (**.tdo**) 2, 13
Text Editor (*see* MAX+PLUS II Text Editor)
TFF primitive 48, 93, 126, 127
TFFE primitive 48, 93, 126, 127
THEN keyword 176
timing assignments 11
Timing Requirements command 11
Title Block 113
TITLE keyword 141
Title Statement
    general description 141
    naming conventions 192
    position in a TDF 4
TO keyword 179
TOK file 16
TRI primitive 43
    conversion of OPNDRN buffer 122
    general description 125
    primitive/port interconnections 127
    primitive/port to register connections 128
TRI_STATE_NODE keyword 27, 159, 162
tri-state buffer (*see* TRI primitive)
tri-state buses 27, 43, 45
tri-state nodes 162
truncated numbers 169

position in a TDF 5
sample files 55, 57, 59, 61, 63, 65, 68, 78
specifying machine outputs 60
State Machine Files (**.smf**) 3
State Machines
state machines
assigning state bits & values 54, 55, 56, 58, 60, 61, 67, 165
Clock, Reset & Enable signals 57
control equations 171
dummy states 67
general description 54
implementing 55–68
importing & exporting 77
logic minimization 62
machine alias 77, 166
Mealy state machines 64
Moore state machines 60
naming 165
one-hot encoding 58
ports 47
processing 54
recovering from illegal states 66
Reset state 165
sample files 55, 57, 59, 61, 63, 65, 68, 78
state transitions 56, 60
synchronizing outputs 60, 63
with asynchronous outputs 64
with synchronous outputs 60
string equality operator 104
Style Guide 187
SUBDESIGN keyword 157
subdesign names 97
Subdesign Section
formatting guidelines 189, 193
general description 157
ports 132
position in a TDF 4
subranges 100
Symbol Editor 137
Symbol File (**.sym**) 138
symbolic names 97, 132, 192
symbols 94
syntax checking 13
Syntax Coloring command 10, 14, 24, 187
syntax, obsolete 140

Truth Table Statement
- formatting guidelines 194
- general description 183
- implementing 35
- position in a TDF 5
- sample files 36, 37, 59, 61, 63, 65
- specifying default values for variables 39, 173
- state transitions 56, 61, 63

U

unary + and - operators 110
undefined (X) logic levels 35, 93
unquoted names 97
Use LPM for AHDL Operators logic option 11, 111
USED evaluated function 87
User Libraries command 76
usr/maxplus2 directory (*see* **maxplus2** directory)

V

VARIABLE keyword 159
Variable Section
- formatting guidelines 195
- general description 159
- If Generate Statement 178
- Instance Declaration 69, 160
- Machine Alias Declaration 77, 166
- Node Declaration 27, 81, 162
- position in a TDF 5
- Register Declaration 47, 50, 51, 163
- State Machine Declaration 55, 60, 165, 195

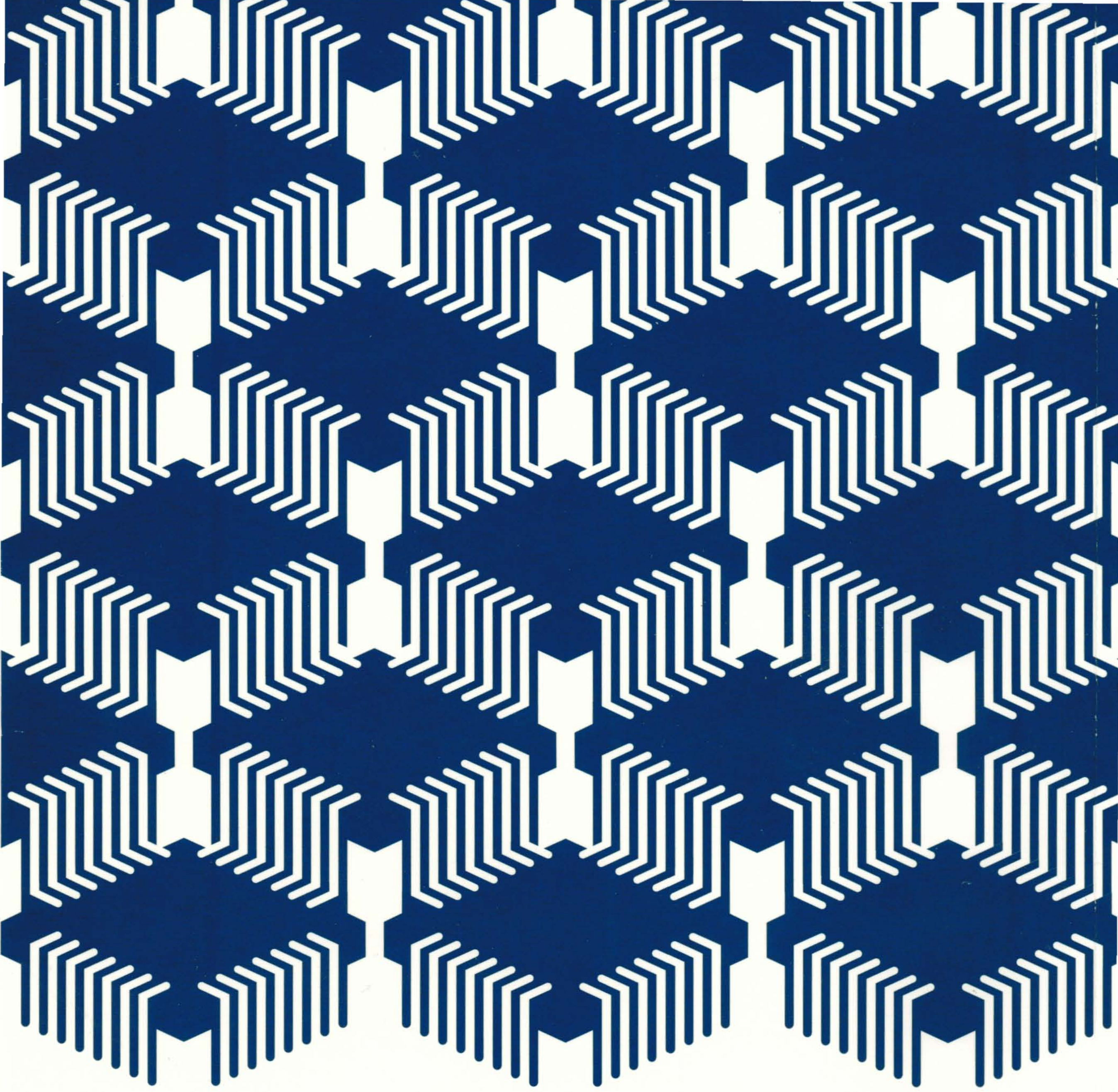
VCC 30, 39, 40, 102, 127, 128, 157, 173
VHDL Design Files (**.vhd**) 3

W

Waveform Design Files (**.wdf**) 3
WHEN keyword 172
WHEN OTHERS keywords 172
white space 190
WITH clause 73
WITH keyword 151, 180
WITH STATES keywords 165

X

X character 38, 183
Xilinx Netlist Format Files (**.xnf**) 3
XNOR operator 108
XOR operator 104, 107, 108



ALTERA CORPORATION

101 INNOVATION DRIVE

SAN JOSE, CALIFORNIA

95134 USA

P25-04802-02