Designing with Low-Level Primitives

User Guide

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com

Software Version	7.1
Document Version:	3.0
Document Date:	April 2007

I.S. EN ISO 9001

UG-83105-3.0

Contents

About this User Guide

Document Revision History

The table below shows the revision history for this document.

Date and Document Version	Changes Made	Summary of Changes
April 2007 v3.0	Made changes to the Guide: ● Added examples 1–8, 2–4, 2–6, 2–8 ● Added these new sections: "ALT_INBUF_DIFF" on page 2–11 "ALT_OUTBUF_DIFF" on page 2–13 "ALT_OUTBUF_TRI_DIFF" on page 2–14 "ALT_IOBUF_DIFF" on page 2–19 "ALT_BIDIR_DIFF" on page 2–22 "ALT_BIDIR_BUF" on page 2–25 ● Removed most of the "Synthesis Attributes" on page 2–33 section, replaced with a reference to the *Quartus II Handbook*.	Technical changes to coincide with changes to the Quartus II software 7.0.0 release
May 2006 v2.0	Technical changes to coincide with changes to the Quartus II software 6.0.0 release	—
October 2005 v1.0	Initial Release	—

How to Contact Altera

For the most up-to-date information about Altera® products, go to the Altera world wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	Resource
Technical support	www.altera.com/mysupport/
Product literature	www.altera.com
Altera literature services	literature@altera.com *(1)*
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\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`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	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.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user’s work.
⚠	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

1. Low-Level Primitive Design

Introduction

Your hardware description language (HDL) coding style can have a significant effect on the quality of results that you achieve for programmable logic designs. Although synthesis tools optimize HDL code for both logic utilization and performance, sometimes the best optimizations require engineering knowledge of the design. Therefore, it is important to consider the HDL coding style that you adopt when creating your programmable logic design.

Low-level HDL design is the practice of using low-level primitives and assignments in your HDL code to dictate a particular hardware implementation for a piece of logic. Low-level primitives are small architectural building blocks that assist you in creating your design. With the Quartus® II software, you have the option of using low-level HDL design techniques that can help you to achieve better resource utilization or faster timing results.

Using low-level primitives in your design enables you to control the hardware implementation for a cone of logic in your design. These cones can be as small as an LCELL instantiation, which prevents the Quartus II synthesis engine from performing optimizations, to larger, more complex examples that specify the encoding method for a finite state machine (FSM).

The Quartus II software can synthesize and place and route designs that instantiate low-level primitives. This user guide describes the support that the Quartus II software offers for creating a design with primitives and includes the definition of each primitive, usage guidelines, and example designs.

Using the Quartus II software, you can instantiate a Quartus II primitive into your HDL design. The source files for Quartus II primitives are built into the Quartus II software.

Example 1–1 is a small Verilog example that shows an instantiation of a DFF primitive and an ALT_OUTBUF_TRI primitive.

Example 1–1. Instantiation of a DFF Primitive and alt_outbuf_tri Primitive, Verilog

```
module compinst (data, clock, clearn, presetn,
              a, b, q_out, t_out);
    input data, clock, clearn, presetn, a, b;
    output  q_out, t_out;

   dff dff_inst (.d (data), .q (q_out), .clk (clock),
//dff is a primitive
              .clrn(clearn),.prn (presetn));
alt_outbuf_tri tri_inst (.i(b), .oe(a), .o(t_out))
// alt_outbuf_tri is a primitive

endmodule
```

Low-Level Primitive Examples

The following sections provide examples of how you can implement low-level primitives:

- "LCELL Primitive"
- "Using I/Os" on page 1–6
- "Using Registers in Altera FPGAs" on page 1–7
- "Creating Memory for Your Design" on page 1–9
- "Look-Up Table Buffer Primitives" on page 1–13

For detailed specification of the primitive's ports used in these sections, refer to "Primitives" on page 2–1.

LCELL Primitive

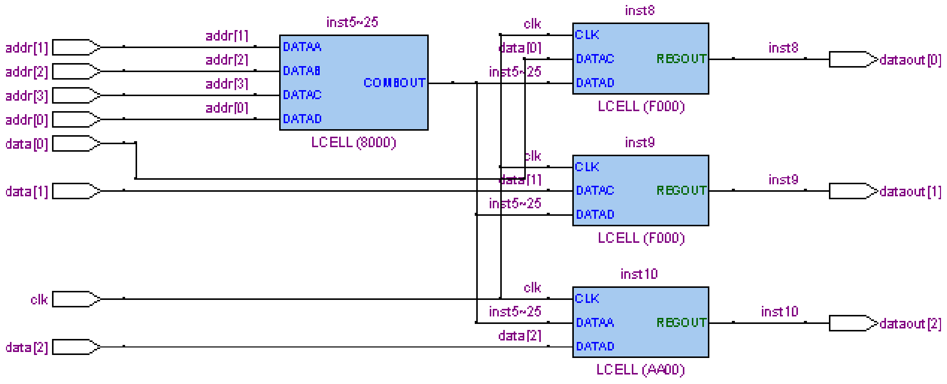
The LCELL primitive allows you to break up your design into manageable parts and prevents the Quartus II synthesis engine from merging logic. This is especially useful when you are trying to debug your design at the implementation level.

In Example 1–2, the LCELL primitive separates the logic in your design. The first code example and the resulting view from the Quartus II Technology Map Viewer (Figure 1–1) show that the logic is merged during the synthesis process.

Example 1–2. LCELL Primitive Separates Logic

```
module logic_merge(
    clk,
    addr,
    data,
    dataout
);

input clk;
input [3:0] addr;
input [2:0] data;
output[2:0] dataout;
reg [2:0] dataout;

wire temp_0;
wire temp_1;
wire temp_2;
wire temp_3;
wire temp_4;
wire temp_5;
wire temp_6;

assign temp_3 = addr[0] & addr[1] & addr[2] & addr[3];
assign temp_4 = addr[3] & addr[2] & addr[1] & addr[0];
assign temp_1 = addr[1] & addr[2] & addr[3];
assign temp_2 = temp_1 & addr[0];
assign temp_5 = temp_2 & data[0];
assign temp_6 = temp_3 & data[1];
assign temp_0 = temp_4 & data[2];
always@(posedge clk)
begin

    dataout[2] <= temp_0;
end

always@(posedge clk)
begin
    dataout[0] <= temp_5;
end

always@(posedge clk)
begin
    dataout[1] <= temp_6;
end

endmodule
```

Figure 1–1. Logic Merged During the Process



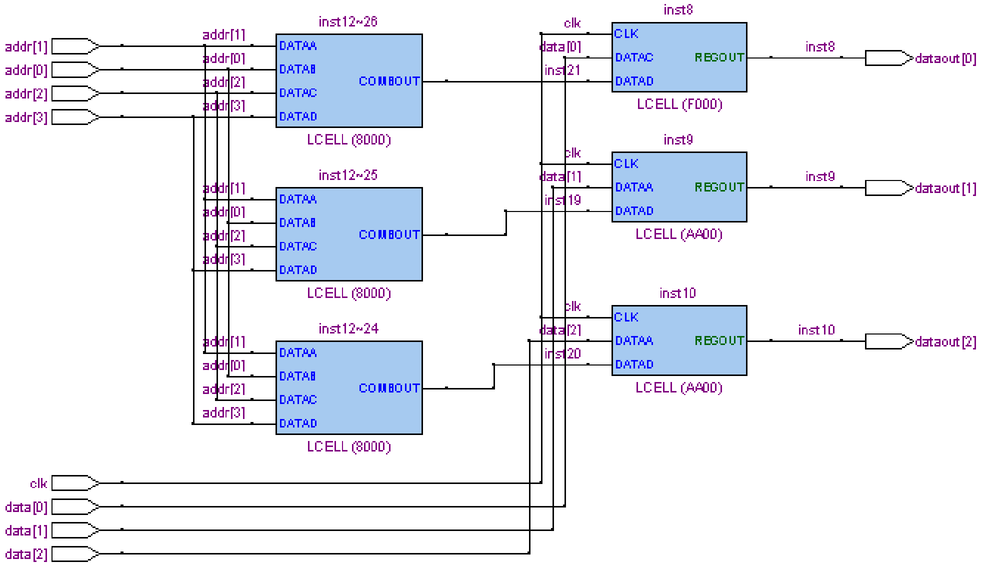
By strategically placing the LCELLs, you can control how the Quartus II synthesis engine splits your design into logic cells. This typically causes your design to use more logic resources, so this primitive should be used with care. In the following example, and the resulting view from the Quartus II Technology Map Viewer (Figure 1–2), three LCELL primitive instantiations are introduced between the combinational logic. Note that “LCELL” is also the name that the Technology Map Viewer gives to the logic cells in some device families, as shown in the figure.

In Example 1–3, the address decoder logic is not merged with the registers in the design.

Example 1–3. Address Decoder Logic Not Merged with Registers

```
module comb_logic_with_lcells(
    clk,
    addr,
    data,
    dataout
);

input clk;
input[3:0] addr;
input [2:0] data;
output [2:0] dataout;
reg [2:0] dataout;
wire temp_0;
wire temp_1;
wire temp_2;
wire temp_3;
wire temp_4;
wire temp_5;
wire temp_6;
wire temp_7;
wire temp_8;
wire temp_9;
assign temp_1 = addr[0] & addr[1] & addr[2] & addr[3];
assign temp_3 = temp_4 & addr[0];
assign temp_8 = temp_5 & data[0];
assign temp_9 = temp_6 & data[1];
assign temp_0 = temp_7 & data[2];
assign temp_4 = addr[1] & addr[2] & addr[3];
assign temp_2 = addr[3] & addr[2] & addr[1] & addr[0];
lcell inst1(.in(temp_1),
.out(temp_6));
lcell inst2(.in(temp_2),
.out(temp_7));
lcell inst3(.in(temp_3),
.out(temp_5));

always@(posedge clk)
begin
    dataout[2] <= temp_0;
end
always@(posedge clk)
begin
    dataout[0] <= temp_8;
end
always@(posedge clk)
begin
    dataout[1] <= temp_9;
end
endmodule
```

Figure 1–2. LCELL Primitive Instantiations



Using I/Os

With I/O primitives, you can make I/O assignments in your HDL file instead of making them through the Assignment Editor in the Quartus II software. Example 1–4 describes how to make an I/O standard assignment to an input pin using the `ALT_INBUF` primitive in Verilog HDL.

Example 1–4. Making an I/O Standard Assignment to an Input Pin Using the ALT_INBUF Primitive, Verilog

```
module io_primitives (data_in, data_out);
   input data_in;
wire internal_sig;
   output data_out;

   alt_inbuf my_inbuf (.i(data_in), .o(internal_sig));
      defparam my_inbuf.io_standard="1.8 V HSTL Class I";
   assign data_out = !internal_sig;

endmodule
```

For detailed specifications of the primitive's ports used in these sections, refer to "Primitives" on page 2–1.

I/O Attributes

There are no primitives available to define an I/O register that can be implemented as a fast input, fast output, or fast output enable register. However, registers associated with an input or output pin can be moved into I/O registers using the following assignments in the Quartus II software for those I/O pins:

- `fast_input_register`
- `fast_output_register`
- `fast_output_enable_register`

These assignments can be set by HDL synthesis attributes. Example 1–5 illustrates the `fast_output_register` synthesis attribute.

Example 1–5. The fast_output_register Synthesis Attribute

```
module fast_output(i,clk,o);
   input i;
   output o;
   reg o /* synthesis altera_attribute = ”FAST_OUTPUT_REGISTER”
      =ON */;
   always @(posedge clk)
   begin
      o <= i;
   end
endmodule
```

For more information, refer to "Synthesis Attributes" on page 2–33.

Using Registers in Altera FPGAs

The building blocks of FPGA architectures contain a combinational component along with a register component. Each register component in an Altera FPGA provides a number of secondary control signals (such as `clear`, `reset`, and `enable` signals) that you can use to implement control logic for each register without the use of extra logic cells. Device families vary in their support for secondary signals, so you must consult the device family data sheet to verify which signals are available in your target device. Download the device family data sheets from the Literature section of www.altera.com.

Inferring Registers Using HDL Code

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. Because of the layout of the architecture, the control signals have a certain priority. Therefore, your HDL code should follow that priority whenever possible. If you do not follow the signal priority, your synthesis tool emulates the control signals using additional logic resources. Therefore, creating functionally correct results is always possible. However, if your design requirements are flexible (in terms of which control signals are used and in what priority), you can match your design to the target device architecture to achieve the optimal performance and logic utilization results.

There are certain cases where using extra logic resources to emulate control signals can have an unintended impact. For example, a `clock_enable` signal has priority over a `synchronous_reset` or a `clear` signal in the device architecture. The `clock_enable` signal disables the clock line in the logic array block (LAB), and the `sclr` signal is synchronous. In the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a `synchronous clear` signal that has priority over a `clock enable` signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the `clock enable` port of a register, you cannot apply a **Clock Enable Multicycle** constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals because using a different priority causes unexpected results with an assignment to the `clock enable` signal.

The signal order is the same for all Altera device families, although, as mentioned earlier, not all device families provide every signal. In general, use the signal order shown in Table 1–1.

Table 1–1. Signal Order (from Highest to Lowest Priority)

Priority	Signal
1	`Asynchronous clear`
2	`Preset`
3	`Asynchronous load`
4	`Enable`
5	`Synchronous clear`
6	`Synchronous load`
7	`Data in`

The `sclr` signal is not inferred by Quartus Integrated Synthesis when there are a large number of registers with different `sclr` signals.This behavior makes it easier for the fitter to successfully route the design. If you would like to force the use of the `sclr` signals, you can use the following Quartus II synthesis settings.

For more details about these and other synthesis settings, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

- Force Use of Synchronous Clear Signals—Forces the compiler to utilize synchronous `clear` signals in normal mode logic cells. Turning on this option helps to reduce the total number of logic cells used in the design, but might negatively impact the fitting because synchronous control signals are shared by all the logic cells in a LAB.
- Allow Synchronous Control Signals—Allows the compiler to utilize synchronous `clear` and/or synchronous `load` signals in normal mode logic cells. Turning on this option helps to reduce the total number of logic cells used in the design, but might negatively impact the fitting because synchronous control signals are shared by all the logic cells in a LAB.

For more information about inference guidelines for registers and on secondary control signal inference rules, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Using the DFFEAS Primitive

The DFFEAS primitive allows you to directly instantiate a register in your design and gives you control over which secondary signals are used. The `DFFEAS` primitive instantiations are always adhered to unless the secondary control signals that you use are not supported by the device family architecture. If you instantiate a `DFFEAS` primitive with unsupported secondary control signals, they are converted into the equivalent logic.

For an example on instantiation of the `DFFEAS` primitive, refer to the *Primitive Reference and Synthesis Attributes* chapter in this user guide.

Creating Memory for Your Design

You can create RAM for your design in two ways. The first method involves creating HDL code that infers a memory function. The second method involves building a function using the MegaWizard® Plug-In Manager and instantiating the resulting custom megafunction variation file in your design.

Inferring RAM Functions from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the altsyncram or lpm_ram_dp megafunctions, depending on the targeted device family. The Quartus II software usually does not infer very small RAM blocks because they typically are implemented more efficiently by using the registers in regular logic.

If your design contains a RAM block that your synthesis tool does not recognize and infer, the design might require a large amount of system memory, which can potentially cause run-time compilation problems.

For RAM inference guidelines, refer to the *Recommended HDL Coding Styles* chapter of the *Quartus II Handbook*.

Using the MegaWizard Plug-In Manager

You can use the MegaWizard Plug-In Manager to create RAM functions. The MegaWizard Plug-In Manager, located in the Tools menu in the Quartus II software, allows you create or modify design files that contain custom megafunction variations, which you can then instantiate in a design file.

The GUI-based interface of the MegaWizard Plug-In Manager provides an easy and intuitive interface that allows you to parameterize complex functions such as memory. However, there are cases, particularly with memory, where you simply want to modify a small component of the megafunction. For example, your design can call for two types of memory functions: a 32, 8-bit word single-port memory function and a 64, 8-bit word single-port memory function. In this scenario, you can use the MegaWizard Plug-In Manager to create one function and then use the instantiation from the wizard-generated file to directly instantiate the second variation. However, directly instantiating memory functions should only be used when the modifications to the functions are minimal.

Example 1–6 shows a Verilog example for a 32, 8-bit word single-port memory function.

Example 1–6. A 32, 8-Bit Word Single-Port Memory Function, Verilog

```
   altsyncramalt syncram_component (
            .wren_a (wren),
            .clock0 (clock),
            .address_a (wraddress),
            .address_b (rdaddress),
            .data_a (data_in),
            .q_b (data_out),
            .aclr0 (1'b0),
            .aclr1 (1'b0),
            .clocken1 (1'b1),
            .clocken0 (1'b1),
            .q_a (),
            .data_b ({8{1'b1}}),
            .rden_b (1'b1),
            .wren_b (1'b0),
            .byteena_b (1'b1),
            .addressstall_a (1'b0),
            .byteena_a (1'b1),
            .addressstall_b (1'b0),
            .clock1 (1'b1));
   defparam
      altsyncram_component.address_aclr_a = "NONE",
      altsyncram_component.address_aclr_b = "NONE",
      altsyncram_component.address_reg_b = "CLOCK0",
      altsyncram_component.indata_aclr_a = "NONE",
      altsyncram_component.intended_device_family = "Stratix",
      altsyncram_component.lpm_type = "altsyncram",

      //This is where a 32, 8-bit word is modified.
      altsyncram_component.numwords_a = 32,
      altsyncram_component.numwords_b = 32,

      altsyncram_component.operation_mode = "DUAL_PORT",
      altsyncram_component.outdata_aclr_b = "NONE",
      altsyncram_component.outdata_reg_b = "CLOCK0",
      altsyncram_component.power_up_uninitialized = "FALSE",
      altsyncram_component.read_during_write_mode_mixed_ports  =
"DONT_CARE",

      altsyncram_component.widthad_a = 5,
      altsyncram_component.widthad_b = 5,

      //This is the width of the input port.
      altsyncram_component.width_a = 8,
      altsyncram_component.width_b = 8,
      altsyncram_component.width_byteena_a = 1,
      altsyncram_component.wrcontrol_aclr_a = "NONE";
```

Example 1–7 shows a Verilog example for a 64, 8-bit word single-port memory function.

Example 1–7. A 64, 8-Bit Word Single-Port Memory Function, Verilog

```
altsyncram altsyncram_component (
          .wren_a (wren),
          .clock0 (clock),
          .address_a (wraddress),
          .address_b (rdaddress),
          .data_a (data_in),
          .q_b (data_out),
          .aclr0 (1'b0),
          .aclr1 (1'b0),
          .clocken1 (1'b1),
          .clocken0 (1'b1),
          .q_a (),
          .data_b ({8{1'b1}}),
          .rden_b (1'b1),
          .wren_b (1'b0),
          .byteena_b (1'b1),
          .addressstall_a (1'b0),
          .byteena_a (1'b1),
          .addressstall_b (1'b0),
          .clock1 (1'b1));
defparam
   altsyncram_component.address_aclr_a = "NONE",
   altsyncram_component.address_aclr_b = "NONE",
   altsyncram_component.address_reg_b = "CLOCK0",
   altsyncram_component.indata_aclr_a = "NONE",
   altsyncram_component.intended_device_family = "Stratix",
   altsyncram_component.lpm_type = "altsyncram",

   //This is where a 64, 8-bit word is modified.
   altsyncram_component.numwords_a = 64,
   altsyncram_component.numwords_b = 64,

   altsyncram_component.operation_mode = "DUAL_PORT",
   altsyncram_component.outdata_aclr_b = "NONE",
   altsyncram_component.outdata_reg_b = "CLOCK0",
   altsyncram_component.power_up_uninitialized = "FALSE",
   altsyncram_component.read_during_write_mode_mixed_ports
   ="DONT_CARE",

   altsyncram_component.widthad_a = 6,
   altsyncram_component.widthad_b = 6,

   //This is the width of the input port.
   altsyncram_component.width_a = 8,
   altsyncram_component.width_b = 8,
   altsyncram_component.width_byteena_a = 1,
   altsyncram_component.wrcontrol_aclr_a = "NONE";
```

Look-Up Table Buffer Primitives

The look-up table (LUT) buffer primitives, `LUT_INPUT` and `LUT_OUTPUT`, specify a LUT function in your design. These primitives are single-input, single-output buffers that you use to create a LUT directly in your design. The `LUT_INPUT` acts as an input to the `LUT_OUTPUT`. If your design contains a `LUT_OUTPUT` primitive that is not properly driven, the `LUT_OUTPUT` is ignored. By using `LUT_INPUT` and `LUT_OUTPUT`, you can specify which LUT inputs are used. These primitives are similar to the `LCELL` primitive; they give you control over how the Quartus II synthesis engine breaks your design up into logic cells. Because they give you full control of the inputs and outputs to a logic cell, `LUT_INPUT` and `LUT_OUTPUT` primitives give you more control over the synthesis process, but you must have more understanding of the device architecture to use them successfully

Example 1–8 shows a primitive instantiation that creates a four-input LUT that implements the function `aw & bw ^ cw | dw`.

Example 1–8. A Primitive Instantiation that Creates a Four-Input LUT

```
module lut_function (a,b,c,d,o);
   input a,b,c,d;
   output o;
   wire aw,bw,cw,dw,o;

   lut_input lut_in1 (a, aw) ;
   lut_input lut_in2 (b, bw) ;
   lut_input lut_in3 (c, cw) ;
   lut_input lut_in4 (d, dw) ;
   lut_output lut_o (aw & bw ^ cw | dw, o) ;
endmodule
```

Example 1–9 is a more complex example using parameterized Verilog and the `LUT_INPUT` and `LUT_OUTPUT` primitive buffers. This example creates a LUT function to implement an arbitrary function in a single LUT. The value of “out” is generated by the `mask` signal and uses the `LUT_INPUT` and `LUT_OUTPUT` primitive buffers to build a single logic cell that implements the functionality given by the `LUT_MASK` parameter.

Example 1–9. Parameterized Verilog and LUT_INPUT and LUT_OUTPUT Primitive Buffers

```
module lut_sub (din,out);

   parameter LUT_SIZE = 4;
   parameter NUM_BITS = 2**LUT_SIZE;

   input [LUT_SIZE-1:0] din;
   parameter [NUM_BITS-1:0] mask = {NUM_BITS{1'b0}};
   output out;
   wire out;

   // buffer the LUT inputs
   wire [LUT_SIZE-1:0] din_w;
   genvar i;
   generate
      for (i=0; i<LUT_SIZE; i=i+1)
      begin : liloop
      lut_input li_buf (din[i],din_w[i]);
      end
   endgenerate
   // build up the pterms for the LUT
   wire [NUM_BITS-1:0] pterms;
   generate
   for (i=0; i<NUM_BITS; i=i+1)
      begin : ploop
      assign pterms[i] = ((din_w == i) & mask[i]);
      end
   endgenerate

   // assign the pterms to the LUT function
   wire result;
   assign result = | pterms;
   lut_output lo_buf (result,out);
   endmodule
```

Example 1–10 uses the LUT primitive to create LUTs that implement a 4-input `AND` and a 4-input `OR` function.

Example 1–10. Using the LUT Primitive

```
module luts (
       input [3:0] in1, in2,
       output out1, out2
       );

   lut_sub inst1 (in1, out1);
   defparam inst1.mask = 16'H8000; // AND function
   lut_sub inst2 (in2, out2);
   defparam inst2.mask = 16'HFFFE; // OR function
   endmodule
```

2. Primitive Reference

Primitives

Using primitives with HDL is an efficient way to make assignments to your design without using the Assignment Editor.

The Quartus® II software supports the following primitives, described on the corresponding pages:

- "ALT_INBUF"
- "ALT_OUTBUF" on page 2–3
- "ALT_OUTBUF_TRI" on page 2–6
- "ALT_IOBUF" on page 2–8
- "ALT_INBUF_DIFF" on page 2–11 *(1)*
- "ALT_OUTBUF_DIFF" on page 2–13 *(1)*
- "ALT_OUTBUF_TRI_DIFF" on page 2–14 *(1)*
- "ALT_IOBUF_DIFF" on page 2–19 *(1)*
- "ALT_BIDIR_DIFF" on page 2–22 *(1)*
- "ALT_BIDIR_BUF" on page 2–25 *(1)*
- "LCELL" on page 2–27
- "DFF" on page 2–28
- "CARRY and CARRY_SUM" on page 2–29
- "CASCADE" on page 2–30
- "LUT_INPUT" on page 2–31
- "LUT_OUTPUT" on page 2–32

Note to the above list:

(1) These I/O primitives are supported only in Stratix® III and Cyclone® III device families.

ALT_INBUF

The primitive allows you to make a location assignment, termination assignment, and also lets you determine whether to use weak pull up resistor, whether to enable bus-hold circuitry or an `io_standard` assignment to an input pin from a lower-level entity. Table 2–1 describes the input and output ports and the parameters associated with ALT_INBUF. If any other parameter is specified (for example, current_strength) an error will result.

Table 2–1. ALT_INBUF Ports and Parameters

Port/Parameter	Description/Value
Input Ports	
i	Connect this port to a chip input pin or to a wire to be connected to an input pin. There must be no logic between the i port and the chip input pin. If there is, an error results.
Output Ports	
o	Connect this port to the logic in the design to receive the input signal.
Parameter	
io_standard	A logic option that specifies the I/O standard of a pin. Different device families support different I/O standards, and restrictions apply to placing pins with different I/O standards together.
location	Any legal pin location for the current device.
enable_bus_hold	An option to enable bus-hold circuitry. Legal values are “on” and “off.”
weak_pull_up_resistor	An option to enable the weak pull-up resistor. Legal values are “on” and “off.”
termination	Any legal on-chip-termination value for the current device. For GX families, this parameter supports only regular termination, and not GXB termination. For GX families, you must use Assignment Editor in the Quartus II software to set the type of termination on high-speed transceivers. This parameter is supported for Stratix III, Stratix II, Stratix II GX, HardCopy® II and Cyclone II.

Example 2–1 shows a Verilog HDL example of an ALT_INBUF primitive instantiation.

Example 2–1. ALT_INBUF Primitive Instantiation, Verilog HDL

```
alt_inbuf my_inbuf (.i(in), .o(internal_sig));
   //in must be declared as an input pin
defparam my_inbuf.io_standard = "2.5 V";
defparam my_inbuf.location = "IOBANK_2";
defparam my_inbuf.enable_bus_hold = "on";
defparam my_inbuf.weak_pull_up_resistor = "off";
defparam my_inbuf.termination = "parallel 50 ohms";
```

Example 2–2 shows a VHDL component declaration for an ALT_INBUF primitive instantiation.

Example 2–2. ALT_INBUF Primitive Component Declaration, VHDL

```
COMPONENT ALT_INBUF
    GENERIC
     (IO_STANDARD                 :   STRING :="NONE";
         WEAK_PULL_UP_RESISTOR   :   STRING :="NONE";
         LOCATION                :   STRING :="NONE";
         ENABLE_BUS_HOLD         :   STRING :="NONE";
         WEAK_PULL_UP_RESISTOR   :   STRING :="NONE";
         TERMINATION             :   STRING :="NONE");
     PORT (i  : IN STD_LOGIC;
         o  : OUT STD_LOGIC);
END COMPONENT;
```

ALT_OUTBUF

The primitive allows you to make a location assignment, io_standard assignment, current_strength assignments, termination assignment, and also lets you determine whether to use weak pull-up resistor, whether to enable bus-hold circuitry and/or a slow_slew_rate assignment to an output pin from a lower-level entity.

Table 2–2 explains the ALT_OUTBUF input and output ports, and the parameter options. If any other parameter is specified, an error will result.

Table 2–2. ALT_OUTBUF Ports & Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	Connect this port to the logic in the design that generates the output signal.
Output Ports	
`o`	Connect this port to a chip output pin or a wire to be connected to a chip output. There must be no logic between the `o` port and the chip output pin. If there is, an error results.
Parameter	
`io_standard`	A logic option that specifies the I/O standard of a pin. Different device families support different I/O standards, and restrictions apply to placing pins with different I/O standards together.
`current_strength`	A logic option that sets the drive strength of a pin. Specific numerical strength settings are appropriate only for pins with certain I/O standards. You can specify any legal current strength value here (including “minimum current” or “maximum current”).The parameter name `current_strength_new` is also supported for backward compatibility.
`location`	Any legal pin location for the current device.
`enable_bus_hold`	An option to enable bus-hold circuitry. Legal values are “`on`” and “`off`.”
`weak_pull_up_resistor`	An option to enable the weak pull-up resistor. Legal values are “`on`” and “`off`.”
`termination`	Any legal on-chip-termination value for the current device. For GX families, this parameter supports only regular termination, and not GXB termination. This parameter is supported for Stratix III, Stratix II, Stratix II GX, HardCopy II, Cyclone III, and Cyclone II.
`slow_slew_rate`	A logic option that implements slow low-to-high or high-to-low transitions on output pins to help reduce switching noise. Current legal values are “`on`” and “`off`.” This assignment is ignored by the Fitter for Stratix II and Cyclone II devices. This parameter is not supported for Stratix II GX or HardCopy II devices.
`slew_rate`	The `slow_slew_rate` parameter is not available for Stratix III and Cyclone III. These two families support the `slew_rate` parameter instead. Accepts any positive integer value, including 0. The default value is –1, which is equivalent to not using this parameter. This parameter is available only for Stratix III and Cyclone III.

Example 2–3 shows a Verilog HDL example of an `ALT_OUTBUF` primitive instantiation.

Example 2–3. ALT_OUTBUF Primitive Instantiation, Verilog HDL

```
alt_outbuf my_outbuf (.i(internal_sig), .o(out));  //out must be declared as
an output pin
defparam my_outbuf.io_standard = "2.5 V";
defparam my_outbuf.slow_slew_rate = "on";
defparam my_outbuf.enable_bus_hold = "off";
defparam my_outbuf.weak_pull_up_resistor = "on";
defparam my_outbuf.termination = "series 25 ohms";
```

Example 2–4 shows a VHDL component declaration for an `ALT_OUTBUF` primitive instantiation.

Example 2–4. ALT_OUTBUF Primitive Instantiation, VHDL

```
COMPONENT alt_outbuf
GENERIC(
   IO_STANDARD : STRING :="NONE";
   CURRENT_STRENGTH : STRING :="NONE";
   SLOW_SLEW_RATE : STRING :="NONE";
   LOCATION : STRING :="NONE";
   ENABLE_BUS_HOLD : STRING :="NONE";
   WEAK_PULL_UP_RESISTOR : STRING :="NONE";
   TERMINATION : STRING :="NONE";
   SLEW_RATE:INTEGER := -1
);
PORT (
   i : IN STD_LOGIC;
   o : OUT STD_LOGIC
);
END COMPONENT;
```

ALT_OUTBUF_TRI

The primitive allows you to make a location assignment, `io_standard` assignment, `current_strength` assignment, termination assignment, whether or not to use weak pull-up resistor, and allows you to determine whether or not to enable bus-hold circuitry and/or a `slow_slew_rate` assignment to a tri-stated output pin from a lower-level entity. Table 2–3 explains the `ALT_OUTBUF_TRI` input and output ports, as well as the parameter options. If any other parameter is specified, an error will result.

Example 2–5 shows a Verilog HDL example of an `ALT_OUTBUF_TRI` primitive instantiation.

Example 2–5. ALT_OUTBUF_TRI Primitive Instantiation, Verilog HDL

```
alt_outbuf_tri my_outbuf tri (.i(internal_sig), .oe(enable_sig),
   .o(out));
   //out must be declared as an output pin
defparam my_outbuf_tri.io_standard = “1.8 V”;
defparam my_outbuf_tri.current_strength =
"maximum current";
defparam my_outbuf_tri.slow_slew_rate = “off”;
```

Example 2–6 shows a VHDL component declaration for an `ALT_OUTBUF_TRI` primitive instantiation.

Example 2–6. ALT_OUTBUF_TRI Primitive Component Declaration, VHDL

```
COMPONENT alt_outbuf_tri
GENERIC (
   IO_STANDARD : STRING :="NONE";
   CURRENT_STRENGTH : STRING :="NONE";
   SLOW_SLEW_RATE : STRING :="NONE";
   LOCATION : STRING :="NONE";
   ENABLE_BUS_HOLD : STRING :="NONE";
   WEAK_PULL_UP_RESISTOR : STRING :="NONE";
   TERMINATION : STRING :="NONE";
   SLEW_RATE:INTEGER := -1
);
PORT (
   i : IN STD_LOGIC;
   oe : IN STD_LOGIC;
   o : OUT STD_LOGIC
);

END COMPONENT;
```

Table 2–3. ALT_OUTBUF_TRI Ports & Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	Connect this port to the logic in the design that generates the output signal.
`oe`	Connect this port to the tristate output enable logic.
Output Ports	
`o`	Connect this port to a chip output pin or a wire to be connected to a chip output. There must be no logic between the `o` port and the chip output pin. If there is, an error results.
Parameters	
`io_standard`	A logic option that specifies the I/O standard of a pin. Different device families support different I/O standards, and restrictions apply to placing pins with different I/O standards together.
`current_strength`	A logic option that sets the drive strength of a pin. Specific numerical strength settings are appropriate only for pins with certain I/O standards. You can specify any legal current strength value here (including “minimum current” or “maximum current”). The parameter name `current_strength_new` is also supported for backward compatibility.
`location`	The location is any legal pin location for the current device.
`enable_bus_hold`	An option to enable bus-hold circuitry. Legal values are “`on`” and “`off`.”
`weak_pull_up_resistor`	An option to enable the weak pull-up resistor. Legal values are “`on`” and “`off`.”
`termination`	Any legal on-chip-termination value for the current device. For GX families, this parameter supports only regular termination, and not GXB termination. This parameter is supported for Stratix III, Stratix II, Stratix II GX, HardCopy II, Cyclone III, and Cyclone II.
`slow_slew_rate`	A logic option that implements slow low-to-high or high-to-low transitions on output pins to help reduce switching noise. Current legal values are “`on`” and “`off`.” This assignment is ignored by the Fitter for Stratix II and Cyclone II devices. This parameter is not supported for Stratix II GX or HardCopy II devices.
`slew_rate`	The `slow_slew_rate` parameter is not available for Stratix III and Cyclone III. These two families support the `slew_rate` parameter instead. Accepts any positive integer value including 0. Default value is –1, which is equivalent to not using this parameter. Available only for Stratix III and Cyclone III.

ALT_IOBUF

The primitive allows you to make a `location` assignment, `io_standard` assignment, `current_strength` assignment, termination assignment, and allows you to determine whether or not to use weak pull-up resistor, whether or not to enable bus-hold circuitry and/or a `slow_slew_rate` assignment to a bidirectional pin from a lower-level entity.

Table 2–4 lists the ports and parameters of the `ALT_IOBUF` primitive, and their respective descriptions and possible values.

Example 2–7 shows a Verilog HDL example of an `ALT_IOBUF` primitive instantiation.

Example 2–7. ALT_IOBUF Primitive Instantiation, Verilog HDL

```
alt_iobuf my_iobuf (.i(internal_sig1), .oe(enable_sig),
   .o(internal_sig2), .io(bidir));
//bidir must be declared as an inout pin
defparam my_iobuf.io_standard = "3.3-V PCI";
defparam my_iobuf.current_strength = "minimum current";
defparam my_iobuf.slow_slew_rate = "on";
defparam my_iobuf.location = "iobank_1";
```

Example 2–8 shows a VHDL component declaration for an `ALT_IOBUF` primitive instantiation.

Table 2–4. ALT_IOBUF Ports and Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	Connect this port to the logic in the design that generates the output signal.
`oe`	Connect this port to the tri-state output enable logic.
Output Ports	
`o`	Connect this port to the logic in the design that receives the input signal
Bidirectional Port	Connect this port to the chip `bidir` pin or an entity `bidir` port. There must be no logic between the `io` port and the chip `bidir` pin. If there is, an error results.
Parameter	
`io_standard`	A logic option that specifies the I/O standard of a pin. Different device families support different I/O standards, and restrictions apply to placing pins with different I/O standards together.
`current_strength`	A logic option that sets the drive strength of a pin. Specific numerical strength settings are appropriate only for pins with certain I/O standards. You can specify any legal current strength value here (including "minimum current" or "maximum current"). The parameter name `current_strength_new` is also supported for backward compatibility.
`location`	Any legal pin location for the current device.
`enable_bus_hold`	An option to enable bus-hold circuitry. Legal values are "`on`" and "`off`."
`weak_pull_up_resistor`	An option to enable the weak pull-up resistor. Legal values are "`on`" and "`off`."
`termination`	Any legal on-chip-termination value for the current device. For GX families, this parameter supports only regular termination, and not GXB termination. This parameter is supported for Stratix III, Stratix II, Stratix II GX, HardCopy II, Cyclone III, and Cyclone II. To set separate input and output values, use `input_termination` and `output_termination` parameters instead. This parameter can not be used along with `input_termination` or `output_termination`. This parameter sets both input termination value and output termination values for Stratix III and Cyclone III devices. However, note that the Fitter ignores any input termination parameters for Cyclone III devices. To set separate input and output values, use `input_termination` and `output_termination` parameters instead. This parameter can not be used with `input_termination` or `output_termination`.
`input_termination`	Any legal input on-chip termination value for the current device. This parameter cannot be used with the `termination` parameter. This parameter is supported in Stratix III devices only.

Table 2–4. ALT_IOBUF Ports and Parameters

Port/Parameter	Description/Value
`output_termination`	Any legal output on-chip termination value for the current device. This parameter cannot be used with the termination parameter. this parameter is supported in Stratix III and Cyclone III devices.
`slow_slew_rate`	A logic option that implements slow low-to-high or high-to-low transitions on output pins to help reduce switching noise. Current legal values are “`on`” and “`off`.” This assignment is ignored by the Fitter for Stratix II and Cyclone II devices. This parameter is not supported for Stratix II GX or HardCopy II devices.
`slew_rate`	The `slow_slew_rate` parameter is not available for Stratix III and Cyclone III. These two families support the `slew_rate` parameter instead. Accepts any positive integer value including 0. Default value is –1, which is equivalent to not using this parameter. Available only for Stratix III and Cyclone III.

Example 2–8. ALT_IOBUF Primitive Component Declaration, VHDL

```
COMPONENT alt_iobuf
GENERIC (
    IO_STANDARD : STRING :="NONE";
    CURRENT_STRENGTH : STRING :="NONE";
    SLOW_SLEW_RATE : STRING :="NONE";
    LOCATION : STRING :="NONE";
    ENABLE_BUS_HOLD : STRING :="NONE";
    WEAK_PULL_UP_RESISTOR : STRING :="NONE";
    TERMINATION : STRING :="NONE";
    INPUT_TERMINATION : STRING := "NONE" ;
    OUTPUT_TERMINATION : STRING := "NONE";
    SLEW_RATE:INTEGER := -1
);
PORT (
    i : IN STD_LOGIC;
    oe: IN STD_LOGIC;
    io : INOUT STD_LOGIC;
    o : OUT STD_LOGIC );

END COMPONENT;
```

ALT_INBUF_DIFF

This primitive allows you to name and connect positive and negative pins when a differential I/O standard is used for an input pin. You can assign I/O standard assignments, location assignments, termination assignments, control bus hold circuitry, and enable weak pull up on the input pins. An attempt to set any other parameter will result in an error.

Table 2–5 lists the ports and parameters of the `ALT_INBUF_DIFF` primitive, and their respective descriptions and possible values.

Table 2–5. ALT_INBUF_DIFF Ports and Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	This port represents the positive pin of a differential I/O standard. Connect this port either to the device's input pin or to a wire to be connected to an input pin. There must be no logic between the `i` port and the device's input pin. The input pin cannot have any other fan-out.
`ibar`	This port represents the negative pin of a differential I/O standard. Connect this port either to the device's input pin or to a wire to be connected to an input pin. There must be no logic between the `ibar` port and the chip input pin. The input pin cannot have any other fan-out.
Output Ports	
`o`	Connect this port to the logic in the design to receive the `input` signal.
Parameter Name	
`io_standard`	Any legal differential I/O standard value.
`location`	Any legal pin location for the current device.
`enable_bus_hold`	An option to enable bus-hold circuitry. Legal values are "`on`" and "`off`."
`weak_pull_up_resistor`	An option to enable the weak pull-up resistor. Current legal values are "`on`" and "`off`".
`termination`	Any legal on-chip-termination value for the current device. For GX families, this parameter supports only regular termination, and not GXB termination. This parameter is supported for Stratix III, Stratix II, Stratix II GX, HardCopy II, and Cyclone II. Note: The Fitter ignores this parameter for Cyclone III devices.

Each parameter also accepts the value "none". Assigning the value "none" to any parameter is equivalent to not setting the parameter. Note that the primitive requires that all three ports (`i`, `ibar`, and `o`) are connected. Also note that all parameters are optional.

Example 2–9 shows a VHDL component instantiation example of an `ALT_INBUF_DIFF` primitive.

Example 2–9. ALT_INBUF_DIFF Primitive, VHDL Component Instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.altera_primitives_components.all;

entity test_inbuf is
port (
   in1,in2,in3 : in std_logic;
   out1 : out std_logic
);
end test_inbuf;

architecture test of test_inbuf is

signal tmp1: std_logic;

begin

inst : ALT_INBUF_DIFF
generic map (
   IO_STANDARD => "LVDS",7
   LOCATION => "IOBANK_3"
)
port map (
   i => in1,
   ibar => in2,
   o => tmp1
) ;

out1 <= in3 and tmp1;

end test;
```

ALT_OUTBUF_DIFF

This primitive allows you to name and connect positive and negative pins when a differential I/O standard is used for an output pin. You can assign I/O standard, location, drive strength (current strength), slew rate, and termination assignments, control bus hold circuitry, and enable weak pull-up resistor on the output pins. An attempt to set any other parameter will result in an error.

Table 2–6 lists the ports and parameters of the `ALT_OUTBUF_DIFF` primitive, and their respective descriptions and possible values.

Table 2–6. ALT_OUTBUF_DIFF Ports and Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	Connect this port to the logic in the design that generates the output signal.
Output Ports	
`o`	This port represents the positive pin of a differential I/O standard. Connect this port to the device’s output pin or a wire to be connected to the device’s output. There must be no logic between the `o` port and the chip output pin. This port cannot have multiple fan-outs.
`obar`	This port represents the negative pin of a differential I/O standard. Connect this port to the device's output pin or a wire to be connected to the device's output. There must be no logic between the `obar` port and the chip output pin. This port cannot have multiple fan-outs.
Parameter Name	
`io_standard`	Any legal differential I/O standard value.
`current_strength`	Any legal value of the `current_strength_new` QSF assignment.
`location`	Any legal pin location for the current device.
`slew_rate`	Any legal slew rate value for the current device. This value must be a positive integer (including `0`).
`enable_bus_hold`	Whether or not to enable bus-hold circuitry. Current legal values are “`on`” and “`off`”.
`weak_pull_up_resistor`	Whether or not to enable the weak pull-up resistor. Current legal values are “`on`” and “`off`”.
`termination`	Any legal on-chip-termination value for the current device.

Each parameter except `slew_rate` also accepts the value “none”. Assigning the value “none” to any such parameter is equivalent to not setting the parameter. The parameter `slew_rate` accepts the value –1 as the default. Assigning –1 to `slew_rate` is equivalent to not setting the parameter. Note that the primitive requires that all three ports (`i`, `o`, and `obar`) are connected. Also note that all parameters are optional.

Example 2–10 shows a VHDL component instantiation example of an `ALT_OUTBUF_DIFF` primitive.

Example 2–10. ALT_OUTBUF_DIFF Primitive, VHDL Component Instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.altera_primitives_components.all;

entity test_outbuf is
port (
    in1, in2 : in std_logic;
    out1, out1_n : out std_logic
);
end test_outbuf;

architecture test of test_outbuf is

signal tmp: std_logic;

begin
inst : ALT_OUTBUF_DIFF
generic map (
        IO_STANDARD =>  "LVDS",
        LOCATION=>  "IOBANK_3"
)
port map (
        i => tmp,
        o => out1,
        obar => out1_ n
) ;

tmp <= in1 and in2;

end test;
```

ALT_OUTBUF_TRI_DIFF

This primitive allows you to name and connect positive and negative pins when a differential I/O standard is used for a tri-statable output pin. You can assign I/O standard, location, drive strength (current strength), slew rate, and termination assignments, control bus hold circuitry, and enable weak pull-up resistor on the output pins. An attempt to set any other parameter will result in an error.

Table 2–7 lists the ports and parameters of the ALT_OUTBUF_TRI_DIFF primitive, and their respective description and possible values.

Table 2–7. ALT_OUTBUF_TRI_DIFF Ports and Parameters

Port/Parameter	Description/Value
Input Ports	
`i`	Connect this port to the logic in the design that generates the output signal.
`oe`	Connect this port to the tri-state output enable logic.
Output Ports	
`o`	This port represents the positive pin of a differential I/O standard. Connect this port to the device's output pin or a wire to be connected to the device's output. There must be no logic between the `o` port and the chip output pin. This port cannot have multiple fan-outs.
`obar`	This port represents the negative pin of a differential I/O standard. Connect this port to the device's output pin or a wire to be connected to the device's output. There must be no logic between the `obar` port and the chip output pin. This port cannot have multiple fan-outs.
Parameter Name	
`io_standard`	Any legal differential I/O standard value.
`current_strength`	Any legal value of the `current_strength_new` QSF assignment.
`location`	Any legal pin location for the current device.
`slew_rate`	Any legal slew rate value for the current device. This value must be a positive integer (including `0`).
`enable_bus_hold`	Whether or not to enable bus-hold circuitry. Current legal values are "`on`" and "`off`".
`weak_pull_up_resistor`	Whether or not to enable the weak pull-up resistor. Current legal values are "`on`" and "`off`".
`termination`	Any legal on-chip-termination value for the current device.

Each parameter except `slew_rate` also accepts the value "`none`". Assigning the value "`none`" to any such parameter is equivalent to not setting the parameter. The parameter `slew_rate` accepts the value –1 as the default. Assigning –1 to `slew_rate` is equivalent to not setting the parameter. Note that the primitive requires that all four ports (`i`, `oe`, `o`, and `obar`) are connected. Also note that all parameters are optional.

Example 2–11 shows a Verilog HDL example and Example 2–12 shows a VHDL example of an ALT_OUTBUF_TRI_DIFF.

Example 2–11. ALT_OUTBUF_TRI_DIFF Primitive Instantiation, Verilog HDL

```
module test (
   datain_h,
   datain_l,
   oe,
   outclock,
   dataout,
   dataout_n
);

input datain_h;
input datain_l;
input outclock;
input oe;
output dataout;
output dataout_n;

wire tmp_out;
wire tmp_oe;

my_altddio_out altddio_out_inst (
       .outclock (outclock),
       .datain_h (datain_h),
       .datain_l (datain_l),
       .dataout (tmp_out),
       .aclr (1'b0),
       .aset (1'b0),
       .oe (oe),
       .outclocken (1'b1),
       .oe_out (tmp_oe),
       .sclr (1'b0)
);

ALT_OUTBUF_TRI_DIFF my_outbuf (
          .i (tmp_out),
          .oe (tmp_oe),
          .o(dataout),
          .obar(dataout_n)
          );
defparam my_outbuf.io_standard = "LVDS";

endmodule
```

Example 2–12. ALT_OUTBUF_TRI_DIFF Primitive, VHDL Component Instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.altera_primitives_components.all;
entity test_outbuf_tri is
port (
   datain_h, datain_l  : in std_logic_vector (0 downto 0);
   oe, outclock    : in std_logic;
   dataout, dataout_n : out std_logic
);
end test_outbuf_tri;

architecture test of test_outbuf_tri is

component altddio_out
generic (
       intended_device_family: STRING;
       lpm_type : STRING;
       power_up_high : STRING;
       width  : NATURAL
);
port (
   dataout : OUT STD_LOGIC_VECTOR (0 DOWNTO 0);
   outclock : IN STD_LOGIC ;
   oe   : IN STD_LOGIC ;
   datain_h : IN STD_LOGIC_VECTOR (0 DOWNTO 0);
   datain_l : IN STD_LOGIC_VECTOR (0 DOWNTO 0)
);
end component;

signal tmp_out : std_logic_vector (0 downto 0);
signal tmp_oe : std_logic;

begin

DDIO_OUT : altddio_out
generic map (
   intended_device_family => "Stratix II",
   lpm_type => "altddio_out",
   power_up_high => "OFF",
   width => 1
)
port map (
   outclock => outclock,
   oe => tmp_oe,
   datain_h => datain_h,
   datain_l => datain_l,
   dataout => tmp_out
);
```

```
inst : ALT_OUTBUF_TRI_DIFF
generic map (
       IO_STANDARD =>  "LVDS",
       LOCATION=>  "IOBANK_3"
)
port map (
       i    => tmp_out,
       oe   => tmp_oe,
       o    => dataout,
       obar => dataout_n
) ;

end test;
```

ALT_IOBUF_DIFF

This primitive allows you to name and connect positive and negative pins when a differential I/O standard is used for a bidirectional pin. You can assign I/O standard, location, drive strength (current strength), slew rate, and termination assignments, enable bus hold circuitry, and enable weak pull-up resistor on the `bidir` pins. An attempt to set any other parameter will result in an error.

Table 2–8 lists the ports and parameters of the `ALT_IOBUF_DIFF` primitive, and their respective descriptions and possible values.

Table 2–8. ALT_IOBUF_DIFF Ports and Parameters (Part 1 of 2)

Port/Parameter	Description/Value
Input Port	
`i`	Connect this port to the logic in the design that generates the output signal.
`oe`	Connect this port to the tri-state output enable logic.
Output Ports	
`o`	Connect this port to the logic in the design that receives the input signal.
Bidirectional Port	
`io`	This port represents the positive pin of a differential I/O standard. Connect this port to the device's `bidir` pin or an entity `bidir` port. There must be no logic between the `io` port and the chip `bidir` port.
`iobar`	This port represents the negative pin of a differential I/O standard. Connect this port to the device's `bidir` pin, or an entity `bidir` port. There must be no logic between the `iobar` port and the chip `bidir` port.
Parameter Name	
`io_standard`	Any legal differential I/O standard value.
`current_strength`	Any legal value of the `current_strength_new` QSF assignment.
`location`	Any legal pin location for the current device.
`slew_rate`	Any legal slew rate value for the current device. This value must be a positive integer (including 0).
`enable_bus_hold`	The ability to enable bus-hold circuitry. Current legal values are "`on`" and "`off`".
`weak_pull_up_resistor`	The ability to enable the weak pull-up resistor. Current legal values are "`on`" and "`off`".

Table 2–8. ALT_IOBUF_DIFF Ports and Parameters (Part 2 of 2)

Port/Parameter	Description/Value
`termination`	Any legal on-chip-termination value for the current device. This value is set as the input as well as the output termination value for the current device. To set separate input and output values, use `input_termination` and `output_termination` parameters instead. This parameter cannot be used with `input_termination` or `output_termination`.
`input_termination`	Any legal input on-chip-termination value for the current device. This parameter cannot be used with the `termination` parameter. Note: The Fitter ignores this parameter for Cyclone III devices.
`output_termination`	Any legal output on-chip-termination value for the current device. This parameter cannot be used with the `termination` parameter.

Each parameter except `slew_rate` also accepts the value "none". Assigning the value "none" to any such parameter is equivalent to not setting the parameter. The parameter `slew_rate` accepts the value –1 as the default. Assigning –1 to `slew_rate` is equivalent to not setting the parameter. Note that the primitive requires that all five ports (`i`, `oe`, `o`, `io`, and `iobar`) are connected. Also note that all parameters are optional.

Example 2–13 shows an example of a VHDL component instantiation, and Example 2–14 shows a Verilog HDL example of an `ALT_IOBUF_DIFF` primitive instantiation.

Example 2–13. ALT_IOBUF_DIFF Primitive, VHDL Component Instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.altera_primitives_components.all;
entity test_iobuf is
port (
   in1, in2, oe : in std_logic;
   bidir, bidir_n : inout std_logic;
   out : out std_logic
);
end test_iobuf;

architecture test of test_iobuf is

signal tmp1: std_logic;

tmp1 <= in1 and in2;

begin

inst : ALT_IOBUF_DIFF
generic map (
      IO_STANDARD =>  "LVDS",
      LOCATION=>  "IOBANK_3"
)
port map (
      i => tmp1,
      oe => oe,
      o => out,
      io => bidir,
      iobar => bidir_n
) ;

end test;
```

Example 2–14. ALT_IOBUF_DIFF Primitive Instantiation, Verilog HDL

```
module test(in1,in2,oe,out,bidir,bidir_n);
input in1;
input in2;
input oe;
inout bidir;
inout bidir_n;
output out;
wire tmp1;
and(tmp1,in1,in2);

ALT_IOBUF_DIFF inst(.i(tmp1), .oe(oe), .o(out), .io(bidir),
.iobar(bidir_n));
defparam inst.io_standard = "LVDS";
defparam inst.current_strength = "12mA";
endmodule
```

ALT_BIDIR_DIFF

This primitive allows you to name and connect positive and negative pins when the altddio_bidir megafunction is used your design. You can assign I/O standard, location, drive strength (current strength), slew rate, and termination assignments, control bus hold circuitry, and enable weak pull-up resistor on the `bidir` pins. An attempt to set any other parameter will result in an error.

Table 2–9 lists the ports and parameters of the `ALT_BIDIR_DIFF` primitive, and their respective description and possible values.

Table 2–9. ALT_BIDIR_DIFF Ports and Parameters (Part 1 of 2)

Port/Parameter	Description/Value
Input Ports	
`oe`	Connect this port to the `oe_out` port of the altddio_bidir megafunction.
Bidirectional Port	
`bidirin`	Connect this port to the `padio` port of the altddio_bidir megafunction. The `padio` port should not have any other fan-outs.
`io`	This port represents the positive pin of a differential I/O standard. Connect this port to the device's `bidir` pin or an entity `bidir` port. There must be no logic between the `io` port and the chip `bidir` port.
`iobar`	This port represents the negative pin of a differential I/O standard. Connect this port to the device's `bidir` pin or an entity `bidir` port. There must be no logic between the `iobar` port and the chip `bidir` port.

Table 2–9. ALT_BIDIR_DIFF Ports and Parameters (Part 2 of 2)

Port/Parameter	Description/Value
Parameter Name	
`io_standard`	Any legal differential I/O standard value.
`current_strength`	Any legal value of the `current_strength_new` QSF assignment.
`location`	Any legal pin location for the current device.
`slew_rate`	Any legal slew rate value for the current device. This value must be a positive integer (including 0).
`enable_bus_hold`	Whether to enable bus-hold circuitry. Current legal values are "`on`" and "`off`".
`weak_pull_up_resistor`	Whether or not to enable the weak pull-up resistor. Current legal values are "`on`" and "`off`".
`termination`	Any legal on-chip-termination value for the current device. This value is set as the input as well as the output termination value for the current device. To set separate input and output values, use the `input_termination` and `output_termination` parameters instead. This parameter cannot be used along with `input_termination` or `output_termination`.
`input_termination`	Any legal input on-chip-termination value for the current device. This parameter can not be used along with the "`termination`" parameter. Note: The Fitter ignores this parameter for Cyclone III devices.
`output_termination`	Any legal output on-chip-termination value for the current device. This parameter can not be used along with the "`termination`" parameter.

Each parameter except `slew_rate` also accepts the value "none". Assigning the value "none" to any such parameter is equivalent to not setting the parameter. The parameter `slew_rate` accepts the value –1 as the default. Assigning –1 to `slew_rate` is equivalent to not setting the parameter. Note that the primitive requires that all four ports (`oe`, `bidirin`, `io`, and `iobar`) are connected. Also note that all parameters are optional. This primitive can *only* be used with an altddio_bidir megafunction, and with the port connections described above; using it in any other configuration will result in an error.

Example 2–15 shows an example of a Verilog HDL primitive instantiation, and Example 2–16 shows a VHDL example of an `ALT_BIDIR_DIFF` component declaration.

Example 2–15. ALT_BIDIR_DIFF Primitive Instantiation, Verilog HDL

```
module ddio_top (aset, combout, datain_h, datain_l, inclock, sclr, oe,
outclock, bidir, bidir_n );
 input aset;
 input sclr;
 input datain_h;
 input datain_l;
 input inclock;
 input oe;
 input outclock;
 output combout;
 inout bidir;
 inout bidir_n;
 wire tmp_oe;
 wire tmp_padio;

 //myddio_bidir is an instance of the altddio_bidir megafunction
 myddio_bidir sample_ddio ( .aset(aset),
 .combout(combout),
 .datain_h (datain_h),
 .datain_l(datain_l),
 .inclock(inclock),
 .oe(oe),
 .outclock(outclock),
 .padio(tmp_padio),
 .oe_out_port(tmp_oe),
 .sclr(sclr)
 );

ALT_BIDIR_DIFF my_bidir (.bidirin (tmp_padio), .oe (tmp_oe), .io (bidir),
.iobar (bidir_n));
endmodule
```

Example 2–16. ALT_BIDIR_DIFF Primitive, VHDL Component Declaration

```
component ALT_BIDIR_DIFF
generic (
    IO_STANDARD : STRING := "none";
    LOCATION : STRING := "none";
    ENABLE_BUS_HOLD : STRING := "none";
    WEAK_PULL_UP_RESISTOR : STRING := "none";
    INPUT_TERMINATION : STRING := "none";
    OUTPUT_TERMINATION : STRING := "none" ;
    TERMINATION : STRING := "none"
);
port (
   bidirin : inout std_logic;
   oe: in std_logic;
   io : inout std_logic;
   iobar : inout std_logic
);
end component;
```

ALT_BIDIR_BUF

This primitive allows you to create a location assignment, io_standard assignment, drive strength (current strength) assignment and/or `slew_rate` assignment to the bidirectional pin connected to an altddio_bidir megafunction. The legal configuration of the primitive and the complete list of supported parameters are described in Table 2–10. If the primitive is used in any other configuration or with any other parameter, an error will be given.

Table 2–10. ALT_BIDIR_BUF Ports and Parameters (Part 1 of 2)

Port/Parameter	Description/Value
Input Port	
`oe`	Connect this port to the `oe_out` port of the altddio_bidir megafunction.
Bidirectional Port	
`bidirin`	Connect this port to the `padio` port of the altddio_bidir megafunction. The `padio` port should not have any other fan-outs.
`io`	Connect this port to the device's `bidir` pin or an entity `bidir` port. There must be no logic between the `io` port and the chip `bidir` port.
Parameter Name	
`io_standard`	Any legal I/O standard value.
`current_strength`	Any legal value of the `current_strength_new` QSF assignment.
`location`	Any legal pin location for the current device.

Table 2–10. ALT_BIDIR_BUF Ports and Parameters (Part 2 of 2)

Port/Parameter	Description/Value
`slew_rate`	Any legal slew rate value for the current device. This value must be a positive integer (including 0).
`enable_bus_hold`	Whether to enable bus-hold circuitry. Current legal values are "`on`" and "`off`".
`weak_pull_up_resistor`	Whether or not to enable the weak pull-up resistor. Current legal values are "`on`" and "`off`".
`termination`	Any legal on-chip-termination value for the current device. This value is set as the input as well as the output termination value for the current device. To set separate input and output values, use the `input_termination` and `output_termination` parameters instead. This parameter cannot be used with `input_termination` or `output_termination`.
`input_termination`	Any legal input on-chip-termination value for the current device. This parameter cannot be used along with the "`termination`" parameter. Note: The Fitter ignores this parameter for Cyclone III devices.
`output_termination`	Any legal output on-chip-termination value for the current device. This parameter cannot be used along with the "`termination`" parameter.

Each parameter except `slew_rate` also accepts the value "none". Assigning the value "none" to any such parameter is equivalent to not setting the parameter. The parameter `slew_rate` accepts the value –1 as the default. Assigning –1 to `slew_rate` is equivalent to not setting the parameter. Note that the primitive requires that all three ports (`oe`, `bidirin`, and `io`) are connected. Also note that all parameters are optional.

Example 2–17 shows a VHDL example of an `ALT_BIDIR_BUF` component declaration.

Example 2–17. ALT_BIDIR_BUF Primitive, VHDL Component Declaration

```
component ALT_BIDIR_BUF
generic (
    IO_STANDARD : STRING := "none";
    LOCATION : STRING := "none";
    ENABLE_BUS_HOLD : STRING := "none";
    WEAK_PULL_UP_RESISTOR : STRING := "none";
    SLEW_RATE : STRING := "none";
    CURRENT_STRENGTH : STRING := "none";
    INPUT_TERMINATION : STRING := "none";
    OUTPUT_TERMINATION : STRING := "none";
    TERMINATION : STRING := "none"
);
port (
    bidirin : inout std_logic;
    oe: in std_logic;
    io : inout std_logic;
);
end component;
```

LCELL

The instantiation of an `LCELL` primitive buffer allocates one logic cell for your design. When you instantiate an `LCELL` buffer in your design, the Quartus II software preserves the assignment and does not remove it during the synthesis process. The name that you assign the `LCELL` is also preserved.

You should not use `LCELL` primitives to create an intentional delay or asynchronous pulse in your design. The delay of these elements varies with temperature, power supply voltage, and device fabrication process. Race conditions may occur that result in an unreliable circuit.

When you turn on the **Implement as Output of Logic Cell** option, or use the synthesis attribute KEEP, an LCELL buffer is automatically inserted by the Quartus II synthesis engine into your design.

Example 2–18 shows a Verilog HDL example of an `LCELL` primitive instantiation.

Example 2–18. LCELL Primitive Instantiation, Verilog HDL

```
lcell <instance_name> (.in(<input_wire>), .out(<output_wire>);
```

Example 2–19 shows a VHDL component declaration for an `LCELL` primitive instantiation.

Example 2–19. LCELL Primitive Instantiation, VHDL Component Declaration

```
COMPONENT LCELL
   PORT (a_in : IN STD_LOGIC;
         a_out : OUT STD_LOGIC);
END COMPONENT;
```

DFF

The registers in an Altera FPGA support an assortment of configurations, and you have the option of instantiating the following configurations:

- A `DFFE` (data flipflop with enable) primitive
- A `DFFEA` (data flipflop with enable and asynchronous load) primitive with additional ALOAD asynchronous load and ADATA data signals
- A `DFFEAS` (data flipflop with enable and both synchronous and asynchronous load)

Example 2–20 shows a a Verilog HDL example of a `DFF` primitive instantiation.

Example 2–20. DFF Primitive Instantiation, Verilog HDL

```
dffeas <instance_name> (.d(<input_wire>), .clk(<input_wire>),
   .clrn(<input_wire>), .prn(<input_wire>), .ena(<input_wire>),
   .asdata(<input_wire>), .aload(<input_wire>), .sclr(<input_wire>),
   .sload(<input_wire>), .q(<output_wire>);
```

Example 2–21 shows a VHDL component declaration for a DFF primitive instantiation.

Example 2–21. DFF Primitive Instantiation, VHDL

```
COMPONENT DFFEAS
   PORT (d         : IN STD_LOGIC;
        clk        : IN STD_LOGIC;
        clrn       : IN STD_LOGIC;
        prn        : IN STD_LOGIC;
        ena        : IN STD_LOGIC;
        asdata     : IN STD_LOGIC;
        aload      : IN STD_LOGIC;
        sclr       : IN STD_LOGIC;
        sload      : IN STD_LOGIC;
        q          : OUT STD_LOGIC );
END COMPONENT;
```

CARRY and CARRY_SUM

The CARRY_SUM primitive is a two-input, two-output primitive that designates the carry-out and sum-out logic for a function. The cout port of the primitive acts as the carry-in for the next element of the carry chain. This CARRY function also implements fast carry-chain logic for functions such as adders and counters. The CARRY_SUM primitive does not generate the carry or sum logic, but indicates to the compiler that the wires connected to it should be placed on the fast carry-chain logic wires if possible.

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

- The cout port of the CARRY_SUM primitive can feed one or two cones of logic. If the CARRY_SUM primitive feeds two cones of logic, one and only one of the cones of logic must be buffered by another CARRY_SUM 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 that feeds the cin port of a CARRY_SUM primitive can have up to two inputs. A third input is allowed only if it is a CARRY_SUM input or a q feedback from the register.
- The cout port of the CARRY_SUM primitive cannot feed an output pin.
- The cin port of the CARRY_SUM primitive cannot be fed by an input pin.
- The cout port of two different CARRY_SUM primitives cannot feed the same gate.

The CARRY primitive is supported for backward-compatibility with old designs; new designs should use the CARRY_SUM primitive. If you use either primitive incorrectly, it is ignored, and the Quartus II software issues a warning message in the Message processor.

Example 2–22 shows a Verilog HDL example of a CARRY_SUM primitive instantiation.

Example 2–22. CARRY_SUM Primitive Instantiation, Verilog HDL

```
carry_sum <instance_name> (.sin(<input_wire1>), .cin(<input_wire2>),
.sout(<output_wire1>), .cout(<output_wire2>);
```

Example 2–23 shows a VHDL component declaration for a CARRY_SUM primitive instantiation.

Example 2–23. CARRY_SUM Primitive Instantiation, VHDL Component Declaration

```
COMPONENT CARRY_SUM
   PORT (sin, cin  : IN STD_LOGIC;
        sout, cout : OUT STD_LOGIC);
END COMPONENT;
```

CASCADE

The CASCADE buffer enables the cascade-out function from one logic cell and acts as a cascade-in to another logic cell. The cascade-in function allows a cascade, which is a fast output located on each combinational logic cell, to be OR'd or AND'd with the output of an adjacent combinational logic cell within the FPGA.

The CASCADE primitive is only supported with the FLEX 10K® and APEX™ family of FPGAs. If you attempt to use the CASCADE primitive with a non-supported FPGA family, an error message occurs.

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

- A CASCADE primitive can feed or be fed only 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` 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.
- When you turn on the **Auto Cascade Chains** logic option, the compiler automatically inserts `CASCADE` primitives during logic synthesis. When you turn on the **Ignore CASCADE Buffers** logic option, the compiler converts all `CASCADE` buffers to `wire` primitives.

Example 2–24 shows a Verilog HDL example of a `CASCADE` primitive instantiation.

Example 2–24. CASCADE Primitive Instantiation, Verilog HDL

```
cascade <instance_name> (.in(<input_wire>), .out(<output_wire>);
```

Example 2–25 shows a VHDL component declaration for a `CASCADE` primitive instantiation.

Example 2–25. CASCADE Primitive Instantiation, VHDL Component Declaration

```
COMPONENT CASCADE
   PORT (a_in : IN STD_LOGIC;
        a_out : OUT STD_LOGIC);
END COMPONENT;
```

LUT_INPUT

The `LUT_INPUT` buffer specifies the creation of a LUT function. The `LUT_INPUT` buffer marks input signals for a `LUT_INPUT`. The logical functionality of the `LUT_INPUT` and `LUT_OUTPUT` buffers is a simple wire, but together they identify LUT boundaries.

To make a LUT, you must use both input and output buffers that bound a cone of logic.

Example 2–26 shows a Verilog HDL example of a `LUT_INPUT` primitive instantiation.

Example 2–26. LUT_INPUT Primitive Instantiation, Verilog HDL

```
lut_input <instance_name> (.in(<input_wire1), .out(<output_wire>)
```

Example 2–27 shows a VHDL component declaration for a `LUT_INPUT` primitive instantiation.

Example 2–27. LUT_INPUT Primitive Instantiation, VHDL Component Declaration

```
COMPONENT LUT_INPUT
   PORT (a_in : IN STD_LOGIC;
         a_out: OUT STD_LOGIC);
END COMPONENT;
```

LUT_OUTPUT

The `LUT_OUTPUT` buffer specifies a LUT function. The `LUT_OUTPUT` buffer works like an LCELL buffer with the additional detail of specifying the inputs and without the requirement that the LUT function has become a hard output. The `LUT_INPUT` buffer is the input for a `LUT_OUTPUT` buffer. The logical functionality of the `LUT_OUTPUT` and `LUT_INPUT` buffers is a simple wire, but together they identify LUT boundaries.

Example 2–28 shows a Verilog HDL example of a `LUT_OUTPUT` primitive instantiation.

Example 2–28. LUT_OUTPUT Primitive Instantiation, Verilog HDL

```
lut_output <instance_name> (.in(<input_wire>), .out(<output_wire>)
```

Example 2–29 shows a VHDL component declaration for a `LUT_OUTPUT` primitive instantiation.

Example 2–29. LUT_OUTPUT Primitive Instantiation, VHDL Component Declaration

```
COMPONENT LUT_OUTPUT
   PORT (a_in : IN STD_LOGIC;
        a_out : OUT STD_LOGIC);
END COMPONENT;
```

Synthesis Attributes

Using synthesis attributes in HDL is an easy way to make assignments to your design instead of using the Assignment Editor. Synthesis attributes are also commonly called "pragmas".

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.