Contents

Chapter 1. About these Megafunctions
Device Family Support ................................................................. 1-1
Introduction ....................................................................................... 1-1
Features of the ALTECC_ENCODER Megafunction .............................. 1-3
   General Description of the ALTECC_ENCODER Megafunction ........ 1-3
Features of the ALTECC_DECODER Megafunction ............................ 1-5
   General Description of the ALTECC_DECODER Megafunction ........ 1-6

Chapter 2. Getting Started
Software and System Requirements ........................................... 2-1
MegaWizard Plug-In Manager Customization .................................... 2-1
MegaWizard Plug-In Manager Descriptions ..................................... 2-1
Instantiating Megafunctions in HDL Code or Schematic Designs .......... 2-9
   Generating a Netlist for EDA Tool Use ........................................ 2-9
   Using the Port and Parameter Definitions .................................... 2-10
Identifying a Megafunction after Compilation ............................... 2-10
Simulation ...................................................................................... 2-10
Quartus II Software Simulator .................................................... 2-10
EDA Simulator ............................................................................. 2-11
Design Example 1: ALTECC_ENCODER ......................................... 2-11
   Design Files ................................................................................ 2-11
   Configuration Settings .............................................................. 2-12
   Functional Simulation in the ModelSim-Altera Simulator ............ 2-12
   Understanding the Simulation Results for the ECC Encoder .......... 2-13
Design Example 2: ALTECC_DECODER ......................................... 2-17
   Design Files ................................................................................ 2-17
   Configuration Settings .............................................................. 2-18
   Functional Simulation in the ModelSim-Altera Simulator ............ 2-18
   Understanding the Simulation Results for the ECC Decoder .......... 2-19
Conclusion ..................................................................................... 2-25

Chapter 3. Specifications
Ports and Parameters ...................................................................... 3-1
Ports and Parameters for the ALTECC_ENCODER Megafunction .......... 3-1
Ports and Parameters for the ALTECC_DECODER Megafunction .......... 3-2

Additional Information
Revision History ........................................................................... Info–1
Referenced Documents .................................................................. Info–1
How to Contact Altera ................................................................. Info–1
Contents

Typographic Conventions ............................................................................................................... Info–2
Chapter 1. About these Megafunctions

Device Family Support

The ALTECC_ENCODER and ALTECC_DECODER megafunctions support the following target Altera® device families:

- Arria® GX
- Stratix® IV
- Stratix III
- Stratix II
- Stratix II GX
- Stratix
- Stratix GX
- Cyclone® III
- Cyclone II
- Cyclone
- HardCopy® II
- HardCopy Stratix
- MAX® II
- MAX 7000AE
- MAX 7000B
- MAX 7000S
- MAX 3000A
- ACEX 1K®
- APEX® II
- APEX 20KC
- APEX 20KE
- FLEX 10K®
- FLEX® 10KA
- FLEX 10KE
- FLEX 6000

Introduction

This megafuction user guide provides the following information:

- Definition of Error Correction Code (ECC)
- Why and when to use ECC
- General description of ECC functionality
- Description of the Altera ECC megafunctions and supporting features
- Configuration of the ALTECC_ENCODER and ALTECC_DECODER megafunctions using the MegaWizard® Plug-In Manager
Introduction

- Design examples for the ALTECC_ENCODER and ALTECC_DECODER megafunctions
- Ports and parameters for the ALTECC_ENCODER and ALTECC_DECODER megafunctions

ECC is a method of error detection and correction in digital data transmission. Its primary purpose is to detect corrupted data at the receiver side that occurs during data transmission. This method of error correction is best suited for situations where errors occur at random rather than in bursts.

ECC detects errors through the process of data encoding and decoding. For example, when ECC is applied in a transmission application, data read from the source are encoded before being sent to the receiver. The output (code word) from the encoder consists of the raw data appended with the number of parity bits. The exact number of parity bits appended depends on the number of bits in the input data. The generated code word is then transmitted to the destination.

The receiver receives the code word and decodes it. Information obtained by the decoder determines whether or not an error is detected. The decoder detects single-bit and double-bit errors, but can fix only single-bit errors in the corrupted data. This kind of ECC is called Single Error Correction Double Error Detection (SECDED).

Altera provides two megafunctions, ALTECC_ENCODER and ALTECC_DECODER, to implement the ECC functionality. The data input to the ALTECC_ENCODER megafunction is encoded to generate a code word that is a combination of the data input and the generated parity bits. The generated code word is transmitted to the ALTECC_DECODER megafunction for decoding just before reaching its destination block. The ALTECC_DECODER megafunction generates a syndrome vector to determine if there is any error in the received code word. It fixes the data if and only if the single-bit error is from the data bits. No signal is flagged if the single-bit error is from the parity bits. The megafunction also has flag signals to show the status of the data received and the action taken by the ALTECC_DECODER megafunction, if any.

These Altera megafunctions also have features such as pipelining, clock-enable, and asynchronous-clear support. Descriptions of these features can be found in Table 2–1 on page 2–5.

These parameterizeable megafunctions are optimized for Altera device architectures. Using megafunctions instead of coding your own ECC logic saves valuable design time. Also, the Altera-provided functions offer more efficient logic synthesis and device implementation.
About these Megafunctions

The ALTECC_ENCODER and ALTECC_DECODER megafonctions can be easily implemented and configured through the MegaWizard Plug-In Manager, which guides you in configuring the two ECC megafonctions. Details about the available options for each page of the wizard are provided in Chapter 2, Getting Started.

Figure 1–1 shows how the data from the source is transmitted to the destination through the ALTECC_ENCODER and ALTECC_DECODER megafonctions.

Figure 1–1. ALTECC_ENCODER and ALTECC_DECODER Megafonctions in General Data Transmission Environment

The arrows in the block diagram in Figure 1–1 represent the data flow and connection between each block. When the modules are successfully built, the connections between the blocks must be made manually.

Features of the ALTECC_ENCODER Megafonction

The ALTECC_ENCODER megafonction can be implemented and configured through the ALTECC MegaWizard Plug-In Manager. Configuration options for the ALTECC_ENCODER megafonction include:

- Wide range of input data widths, from 2 to 64 bits
- Data encoding: generate output (code word) according to the data width
- Pipelining to introduce output latency
- Asynchronous-clear and clock-enable features

General Description of the ALTECC_ENCODER Megafonction

The ALTECC_ENCODER megafonction takes in and encodes the data using the Hamming Coding scheme. The Hamming Coding scheme derives the parity bits and appends them to the original data to produce the output code word. The number of parity bits appended depends on the width of the data.
Refer to the section “Understanding the Simulation Results for the ECC Encoder” on page 2–13 for an example of how the ALTECC_ENCODER megafunction encodes data using the Hamming Coding scheme.

Table 1–1 shows the number of parity bits appended for different ranges of data widths. The Total Bits column represents the total number of input data bits and appended parity bits.

<table>
<thead>
<tr>
<th>Data Width</th>
<th>Number of Parity Bits</th>
<th>Total Bits (Code Word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–4</td>
<td>3+1</td>
<td>6–8</td>
</tr>
<tr>
<td>5–11</td>
<td>4+1</td>
<td>10–16</td>
</tr>
<tr>
<td>12–26</td>
<td>5+1</td>
<td>18–32</td>
</tr>
<tr>
<td>27–57</td>
<td>6+1</td>
<td>34–64</td>
</tr>
<tr>
<td>58–64</td>
<td>7+1</td>
<td>66–72</td>
</tr>
</tbody>
</table>

The parity bit derivation uses an even-parity checking. The additional 1 bit (shown in Table 1–1 as +1) is appended to the parity bits as the MSB of the code word. This ensures that the code word has an even number of 1’s. For example, if the data width is 4 bits, 4 parity bits are appended to the data to become a code word with a total of 8 bits. If 7 bits from the LSB of the 8-bit code word have an odd number of 1’s, the 8th bit (MSB) of the code word is 1 making the total number of 1’s in the code word even.

Figure 1–2 shows the generated code word and the arrangement of the parity bits and data bits in an example 8-bit data input.

The ALTECC_ENCODER megafunction accepts only 2 to 64 bits of input at one time. An input of 12 bits, 29 bits, or 64 bits generates output with 18 bits, 36 bits, and 72 bits, respectively. These output widths are ideally suited to Altera devices. The bit-selection limitation is controlled by the wizard.
About these Megafunctions

The ALTECC MegaWizard Plug-In Manager provides you the option of pipelining when you configure the ALTECC_ENCODER megafuction. This feature allows you to specify the output latency in clock cycles. If the pipeline is set to 1, the output is registered. The output receives the code word at the first rising edge of the clock. If the pipeline is set to 2, both the input and output are registered. The output receives the code word at the next rising edge of the clock. There is no output latency if you choose not to use the pipelining option.

The ALTECC_ENCODER megafuction also supports asynchronous-clear and clock-enable features. The asynchronous-clear signal is used to reset the registered port. The clock-enable feature allows you to enable the clock signal, which controls the data loading to and output from the ALTECC_ENCODER megafuction block. Details about these options can be found in Table 2–1 on page 2–5.

Figure 1–3 shows a block diagram of the ALTECC_ENCODER megafuction.

Figure 1–3. ALTECC_ENCODER Megafuction Block Diagram

Features of the ALTECC_DECODER Megafuction

The ALTECC_DECODER megafuction can be implemented and configured through the MegaWizard Plug-In Manager. Configuration options for the ALTECC_DECODER megafuction include:

- Wide range of input data (code word) widths, from 6 to 72 bits, except for 9-, 17-, 33-, and 65-bit input data widths
- Data decoding: extracts the data from the input code word and performs Single Error Correction Double Error Detection (SECDED)
- Flag signals to show the status of the data
- Pipelining to introduce output latency
- Asynchronous-clear and clock-enable features
General Description of the ALTECC_DECODER Megafunction

The ALTECC_DECODER megafunction accepts a wide range of code word widths, from 6 bits to 72 bits except 9-, 17-, 33-, and 65-bit widths. To calculate the width of the output data from the decoder, refer to Table 1–1 on page 1–4.

The ALTECC_DECODER megafunction decodes the input data (code word) by extracting the parity bits and data bits from the code word. The parity bits and data bits are recalculated based on the Hamming Coding scheme to generate a syndrome code. The generated syndrome code provides the status of the data received. The ECC detects single-bit and double-bit errors, but only single-bit errors are corrected. This ECC process is called SECDED.

Refer to the section “Understanding the Simulation Results for the ECC Decoder” on page 2–19 for an example of how the ALTECC_DECODER megafucntion decodes a code word using the Hamming Coding scheme.

The ALTECC_DECODER has flag signals (err_detected, err_corrected, and err_fatal) that reflect the status of the data received. From these signals, you can determine whether or not the data you receive is corrupted, or if a single-bit error has been corrected.

Table 1–2 lists the syndrome codes and their respective flag signals.

<table>
<thead>
<tr>
<th>Syndrome Code</th>
<th>Description</th>
<th>Flag Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-zero</td>
<td>No error occurs.</td>
<td>err_detected = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_corrected = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_fatal = 0</td>
</tr>
<tr>
<td>Non-zero and the MSB is 1</td>
<td>A single-bit error is detected, and the corrupted bit is flipped. The value of the syndrome vector (except the MSB) indicates the corrupted bit position within the code word.</td>
<td>err_detected = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_corrected = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_fatal = 0</td>
</tr>
<tr>
<td>Non-zero and the MSB is 0</td>
<td>A double-bit error is detected. No correction is made and the output data is incorrect.</td>
<td>err_detected = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_corrected = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>err_fatal = 1</td>
</tr>
</tbody>
</table>

Note to Table 1–2:
(1) Even if the generated syndrome code indicates a single-bit error, the err_detected and err_corrected signals are asserted only if the corrupted bit is from the data bits and not from the parity bits.
For $n$-bit errors where $n$ is more than 2, errors with an odd $n$ are treated as single-bit errors, while errors with an even $n$ are treated as double-bit errors due to the extra parity bit behavior. If the syndrome bits show all 0’s, no single-bit or double-bit error occurred. There is a slight possibility that the generated syndrome code shows all 0’s even if errors occur. For this reason, the ECC is not meant to detect errors of more than 2 bits.

The ALTECC MegaWizard Plug-In Manager provides a pipelining option when you configure the ALTECC_DECODER megafunction. This feature allows you to specify the output latency in clock cycles. If the pipeline is set to 1, the output is registered. The output receives the data at the first rising edge of the clock. If the pipeline is set to 2, both input and output are registered. The output receives the data at the next rising edge of the clock. There is no output latency if you choose not to use the pipelining option.

The ALTECC_DECODER megafunction also supports asynchronous-clear and clock-enable features. An asynchronous-clear signal is used to reset the registered port. The clock-enable signal enables the clock signal, which controls the data loading to and output from the ALTECC_DECODER megafunction block. Details about the options can be found in Table 2–1 on page 2–5.

Figure 1–4 shows a block diagram of the ALTECC_DECODER megafunction.
Features of the ALTECC_DECODER Megafunction
Chapter 2. Getting Started

Software and System Requirements

The instructions in this section require the following software:

- Quartus® II software 8.0 or later
- For operating system support information, refer to: http://www.altera.com/support/software/os_support/oss-index.html

MegaWizard Plug-In Manager Customization

The MegaWizard® Plug-In Manager creates or modifies design files that contain custom megafunction variations which can then be instantiated in a design file. These custom variations are based on Altera®-provided megafunctions that are optimized to use device resources in the most efficient manner.

The MegaWizard Plug-In Manager allows you to select the ALTECC_ENCODER megafunction or ALTECC_DECODER megafunction, depending on the functionality desired. You can configure the selected megafunction with the options provided, and complete the wizard to successfully build the megafunction.

Start the MegaWizard Plug-In Manager in one of the following ways:

- On the Tools menu, click MegaWizard Plug-In Manager.
- When working in the Block Editor, from the Edit menu, click Insert Symbol as Block, or right-click in the Block Editor, point to Insert, and click Symbol as Block. In the Symbol dialog box, click MegaWizard Plug-In Manager.
- Start the stand-alone version of the MegaWizard Plug-In Manager by typing the following command at the command prompt: qmegawiz

MegaWizard Plug-In Manager Descriptions

This section describes the options available on the individual pages of the ALTECC MegaWizard Plug-In Manager.

On page 1 of the MegaWizard Plug-In Manager, you can select Create a new custom megafunction variation, Edit an existing custom megafunction variation, or Copy an existing custom megafunction variation (Figure 2–1).
On page 2a of the MegaWizard Plug-In Manager, specify the megafuction, device family to use, the type of output file to create, and the name of the output file (Figure 2–2). Choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type.
Getting Started

On page 3 of the MegaWizard Plug-In Manager, select the device family, select an encoder (ALTECC_ENCODER megafunction) or decoder (ALTECC_DECODER megafunction) to configure the megafunction, specify the width of the data input bus, and select the pipelining options.

Figures 2–3 and 2–4 show page 3 of the ALTECC MegaWizard Plug-In Manager when configuring the megafunction as an ECC encoder and an ECC decoder, respectively.

Figure 2–3. ALTECC MegaWizard Plug-In Manager for ECC Encoder [page 3 of 5]
Figure 2–4. ALTECC MegaWizard Plug-In Manager for ECC Decoder [page 3 of 5]
Table 2–1 shows the options available on page 3 of the ALTECC MegaWizard Plug-In Manager.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Currently selected device family</td>
<td>Specify the device family to use. Select the device from the available list.</td>
</tr>
</tbody>
</table>
| How do you want to configure this module?     | If you want to use the ECC encoder (ALTECC_ENCODER megafunction), select **Configure this module as an ECC encoder**. The data port functions as the data word input in this mode. The data input is encoded to generate a code word output that is the combination of the raw data and appended parity bits. *(1)*  

If you want to use the ECC decoder (ALTECC_DECODER megafunction), select **Configure this module as an ECC decoder**. The data port functions as the code word input in this mode. The code word input is decoded to generate the syndrome vector, extract the raw data, and perform Single Error Correction Double Error Detection (SECDED). *(2)* |
| How wide should the data be?                  | Specify the width of the input data bus. You can manually enter a number that is not in the list. The width selection or limitation is controlled by the MegaWizard Plug-In Manager.  

If you select **ECC encoder**, the supported data (data word input) width is from 2 to 64 bits.  
If you select **ECC decoder**, the supported data (code word input) width is from 6 to 72 bits. However, 9-, 17-, 33-, and 65-bit data widths are not supported. *(3)* |
| Do you want to pipeline the function?         | Specify the output latency of the ECC megafunctions. Select **Yes** if you want to have an output latency, and specify a value of either 1 or 2 clock cycles. The wizard prompts you with an error message if you enter an invalid value.  

If you set the output latency to 1, the output is registered, and the output gets the data at the first rising edge of the clock. If you set the output latency to 2, both input and output are registered, and the output gets the data at the next rising edge of the clock. Select **No** if you do not want any output latency. |
| Create an 'aclr' asynchronous clear port      | Turn on this option to create an asynchronous-clear signal. The module asynchronously clears all the registered ports when the signal is asserted. This feature is supported only if you selected **Yes** to pipeline the function. |
Table 2–1. ALTECC MegaWizard Plug-In Manager [Page 3] Options (Part 2 of 2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a ‘clocken’ clock enable clock</td>
<td>Turn on this option to create a clock-enable signal. Together with the clock signal, the clock-enable signal controls the input loading and output operations. This feature is supported only if you selected Yes to pipeline the function.</td>
</tr>
</tbody>
</table>

Notes to Table 2–1:
(1) Refer to the section “General Description of the ALTECC_ENCODER Megafuction” on page 1–3.
(2) Refer to the section “General Description of the ALTECC_DECODER Megafuction” on page 1–6.
(3) For the ECC decoder, make sure that the width of the data input (code word input) matches the width of the generated code word from the ECC encoder. For the width of code word generated for different input data widths, refer to Table 1–1 on page 1–4.

Page 4 of the ALTECC MegaWizard Plug-In Manager shows the simulation model file(s) needed to properly simulate the generated design file. For the ALTECC_ENCODER megafuction, no simulation model file is needed. For the ALTECC_DECODER megafuction, the LPM megafuction simulation library is needed. Page 4 also provides an option to generate a synthesis area and timing estimation netlist. By using the netlist, you can enable some third-party synthesis tools to estimate the timing and resource usage for the megafuctions.

For more information about the netlist files for other synthesis tools, refer to the Recommended HDL Coding Styles chapter in volume 1 of the Quartus II Handbook.
Figure 2–5 shows page 4 of the ALTECC MegaWizard Plug-In Manager.

Figure 2–5. ALTECC MegaWizard Plug-In Manager (page 4 of 5)

Page 5 of the ALTECC MegaWizard Plug-In Manager displays the types of files to be generated. The Variation file, which is automatically generated, contains wrapper code in the language you specified on page 2a. On page 5 of the ALTECC MegaWizard Plug-In Manager, specify the types of files to be generated. You can choose from the following types of files:

- AHDL Include file (<function name>.inc)
- VHDL component declaration file (<function name>.cmp)
- Quartus II symbol file (<function name>.bsf)
- Instantiation template file (<function name>_inst.v)
- Verilog HDL black-box file (<function name>_bb.v)
If you selected **Generate netlist** on page 4, the file for that netlist is also available. A gray checkmark indicates a file that is automatically generated and a red checkmark indicates an optional file (Figure 2–6).

**Figure 2–6. ALTECC MegaWizard Plug-In [page 5 of 5]**

For more information about the ports and parameters for the ALTECC_ENCODER and ALTECC_DECODER megafunctions, refer to Chapter 3, Specifications.
Instantiating Megafuncions in HDL Code or Schematic Designs

When you use the MegaWizard Plug-In Manager to customize and parameterize a megafunction, it creates a set of output files that allows you to instantiate the customized function in your design. Depending on the language you choose in the MegaWizard Plug-In Manager, the wizard instantiates the megafunction with the correct parameter values and generates a megafunction variation file (wrapper file) in Verilog HDL (.v), VHDL (.vhd), or AHDL (.tdf), along with other supporting files.

The MegaWizard Plug-In Manager provides options to create the following files:

- A sample instantiation template for the language of the variation file (_inst.v, _inst.vhd, or _inst.tdf)
- Component Declaration File (.cmp) that can be used in VHDL Design Files
- ADHL Include File (.inc) that can be used in Text Design Files (.tdf)
- Quartus II Block Symbol File (.bsf) that can be used in schematic designs
- Verilog HDL module declaration file that can be used when instantiating the megafunction as a black box in a third-party synthesis tool (_bb.v)

For more information about the wizard-generated files, refer to the Quartus II Help or to the Recommended HDL Coding Styles chapter in volume 1 of the Quartus II Handbook.

Generating a Netlist for EDA Tool Use

If you use a third-party EDA synthesis tool, you can instantiate the megafunction variation file as a black box for synthesis. Use the VHDL component declaration or Verilog HDL module declaration black-box file to define the function in your synthesis tool, and then include the megafunction variation file in your Quartus II project.

If you enable the option to generate a synthesis area and timing estimation netlist in the MegaWizard Plug-In Manager, the wizard generates an additional netlist file (_syn.v). The netlist file is a representation of the customized logic used in the Quartus II software. The file provides the connectivity of the architectural elements in the megafunction but may not represent true functionality. This information enables certain third-party synthesis tools to better report area and timing estimates. In addition, synthesis tools can use the timing information to focus timing-driven optimizations and improve the quality of results.

For more information about using megafuinctions in your third-party synthesis tool, refer to the appropriate chapter in the Synthesis section in volume 1 of the Quartus II Handbook.
Identifying a Megafunction after Compilation

Using the Port and Parameter Definitions

Instead of using the MegaWizard Plug-In Manager, you can instantiate the megafunction directly in your Verilog HDL, VHDL, or AHDL code by calling the megafunction and setting its parameters as you would any other module, component, or subdesign.

Altera strongly recommends that you use the MegaWizard Plug-In Manager for complex megafunctions. The MegaWizard Plug-In Manager ensures that you set all megafunction parameters properly.

For a list of the megafunction ports and parameters, refer to Chapter 3, Specifications.

Identifying a Megafunction after Compilation

During compilation with the Quartus II software, analysis and elaboration are performed to build the structure of your design. To locate your megafunction in the Project Navigator window, expand the compilation hierarchy and find the megafunction by its name.

To search for node names within the megafunction (using the Node Finder), click Browse in the Look in box and select the megafunction in the Hierarchy box.

Simulation

The Quartus II Simulator provides an easy-to-use, integrated solution for performing simulations. The following sections describe the simulation options.

Quartus II Software Simulator

With the Quartus II Simulator, you can perform two types of simulations: functional and timing. A functional simulation enables you to verify the logical operation of your design without taking into consideration the timing delays in the FPGA. This simulation is performed using only your RTL code. When performing a functional simulation, add only signals that exist before synthesis. You can find these signals in the Node Finder by using any of the following Filter options: Registers: Pre-Synthesis, Design Entry, or Pins. The top-level ports of megafunctions are found using these three filters.

In contrast, the timing simulation in the Quartus II software verifies the operation of your design with annotated timing information. This simulation is performed using the post-place-and-route netlist. When performing a timing simulation, add only signals that exist after place-and-route. These signals are found with the post-compilation filter.
of the Node Finder. During synthesis and place-and-route, the names of RTL signals change. Therefore, it may be difficult to find signals from your megafunction instantiation in the post-compilation filter.

To preserve the names of your signals during the synthesis and place-and-route stages, use the synthesis attributes keep or preserve. These are Verilog HDL and VHDL synthesis attributes that direct analysis and synthesis to keep a particular wire, register, or node intact. Use these synthesis attributes to keep a combinational logic node so you can observe the node during simulation.

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

EDA Simulator

The Quartus II Handbook chapters describe how to perform functional and gate-level timing simulations that include the megafunctions, with details about the files that are needed and the directories where the files are located.

Depending on which simulation tool you are using, refer to the appropriate chapter in the Simulation section in volume 3 of the Quartus II Handbook.

Design Example 1: ALTECC_ENCODER

The objective of this example is to implement and instantiate an ECC encoder using the MegaWizard Plug-In Manager. This example illustrates how the ECC encoder encodes an 8-bit wide input data to generate 13 bits of output code word. The clock-enable signal is created to show how it controls the registered port together with the clock. Verify the results you obtain at the end of this example with the expected simulation results provided.

Design Files

The example design files are available in the User Guides section on the Literature page of the Altera® website (www.altera.com).
Design Example 1: ALTECC_ENCODER

Configuration Settings

In the ALTECC MegaWizard Plug-In Manager pages, select or verify the configuration settings shown in Table 2–2. Click Next to advance from one page to the next.

<table>
<thead>
<tr>
<th>MegaWizard Plug-In Manager Page</th>
<th>MegaWizard Plug-In Manager Configuration Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Currently selected device family</td>
<td>Stratix III</td>
</tr>
<tr>
<td></td>
<td>How do you want to configure this module?</td>
<td>Configure this module as an ECC encoder</td>
</tr>
<tr>
<td></td>
<td>How wide should the data be?</td>
<td>8 bits</td>
</tr>
<tr>
<td></td>
<td>Do you want to pipeline the function?</td>
<td>Yes, I want an output latency of 2 clock cycles (1)</td>
</tr>
<tr>
<td></td>
<td>Create an ‘aclr’ asynchronous clear port</td>
<td>Not selected</td>
</tr>
<tr>
<td></td>
<td>Create a ‘clocken’ clock enable clock</td>
<td>Selected</td>
</tr>
</tbody>
</table>

Note to Table 2–2:
(1) The generated code word appears at the output port during the second rising edge of the clock.

Functional Simulation in the ModelSim-Altera Simulator

Simulate the design in the ModelSim®-Altera software to generate a waveform display of the device behavior.

You should be familiar with the ModelSim-Altera software before trying out the design example. If you are unfamiliar with the ModelSim-Altera software, refer to the support page for software products on the Altera website (www.altera.com). On the support page, there are links to such topics as installation, usage, and troubleshooting.

Set up and simulate the design in the ModelSim-Altera software by performing the following steps:

1. Unzip the DE1_ALTECC_ENCODER.zip file to any working directory on your PC.

2. Start the ModelSim-Altera software.


4. Select the folder in which you unzipped the files.

5. Click OK.
6. On the Tools menu, click **Execute Macro**.

7. Select the **DE1_ALTECC_ENCODER.do** file and click **Open**. The **DE1_ALTECC_ENCODER.do** file is a script file for the ModelSim-Altera software to automate all the necessary settings for the simulation.

View the simulation results in the Wave window. Figure 2–7 shows the expected simulation results in the ModelSim-Altera software.

![Figure 2–7. Design Example 1: Simulation Waveform for the ECC Encoder](image)

### Understanding the Simulation Results for the ECC Encoder

In this example, you configured the ECC encoder to have the following properties:

- 8 bits of input data width
- Enabled pipelining with an output latency of 2 clock cycles
- Created a clock-enable signal
Design Example 1: ALTECC_ENCODER

Figure 2–8 shows the encoding of input data F0 to code word 14F0.

![Figure 2–8. Design Example 1: ECC Encoder Encodes Input Data F0 to Output Code Word 14F0](image)

At the beginning, data F0 is fed to the ECC encoder. The data is encoded at the rising edge of the clock when the clock-enable signal (clocken) is high at 15 ns. Because pipelining is enabled to have an output latency of 2 clock cycles, the encoded code word appears at the output port q during the next rising edge of the clock at 25 ns. If you choose to have an output latency of 1 clock cycle for this example, the code word appears at the first rising edge of the clock at 15 ns.

The 8-bit input data (F0) is encoded to generate a 13-bit output code word (14F0). The input data is appended with 5 parity bits.

For the number of parity bits appended for different data widths, refer to Table 1–1 on page 1–4.

The ECC encoder encodes the data based on the Hamming Code scheme. The following steps describe the Hamming Code algorithm and explain how the ECC encoder encodes input data F0 to generate the output code word of 14F0:

1. In a 13-bit code word, there are 13 locations (bit positions), and each location holds 1 bit. There are 8 bits of original data, and the appended 5 parity bits. The locations (bit positions) for the bits must be defined. Table 2–3 shows the bit positions, and the position of the parity bits of a 13-bit code word. P5* is the extra parity bit added. The prefix P denotes parity.

<table>
<thead>
<tr>
<th>(position)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bits</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P5*</td>
</tr>
</tbody>
</table>
2. All other bit positions are for the data to be encoded. The least significant bit (LSB) of the data bit fills the lowest bit position. In this case, starting from the LSB of the data, F0 (1111 0000 in binary) fills the empty bit positions, starting from position 3, as shown in Table 2–4. The prefixes P and D denote parity and data, respectively.

For the standard Hamming Code algorithm, the most significant bit (MSB) of the data bit fills the lowest bit position, unlike the Altera ECC megafunction, which fills up the lowest bit position starting with the LSB. This bit order reduces the complexity of the circuit design.

| Table 2–4. Design Example 1: Filling of Data Bits (1111 0000) for a 13-Bit Code Word |
|-----------------------------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| (position)                              | (1)   | (2)   | (3)   | (4)   | (5)   | (6)   | (7)   | (8)   | (9)   | (10)  | (11)  | (12)  | (13)  |
| Parity bits and data bits              | P1    | P2    | D1    | P3    | D2    | D3    | D4    | P4    | D5    | D6    | D7    | D8    | P5*   |
| Parity bits at position-n = Skip (n-1) bit, check n bit, skip n bit, check n bit, skip n bit... |

3. Each parity bit calculates the parity (even parity) for some of the bits in the code word using the following formula:

(1) Parity bit at position-n = Skip (n-1) bit, check n bit, skip n bit, check n bit, skip n bit...
In this example, the parity bits are derived as shown in Table 2–5.

### Table 2–5. Design Example 1: Calculation of Parity Bits

<table>
<thead>
<tr>
<th>(position)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bits and data bits</td>
<td>P1</td>
<td>P2</td>
<td>D1</td>
<td>P3</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
<td>P4</td>
<td>D5</td>
<td>D6</td>
<td>D7</td>
<td>D8</td>
<td>P5*</td>
</tr>
<tr>
<td>Calculate P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Calculate P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calculate P3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calculate P4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calculate P5*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Parity bit P1 is calculated using an even parity checking on bit positions (3), (5), (7), (9), and (11).

Parity bit P2 is calculated using an even parity checking on bit positions (3), (6), (7), (10), and (11).

The same calculation method using Equation 1 is applied to other parity bits.

4. Calculate the additional parity bits using an even parity checking on all the bits, including the calculated parity bits.

The additional parity bit P5* is calculated with an even parity checking on all the bits from position (1) to position (12), as shown in Table 2–5.

5. The generated code word is rearranged so that the data is at the LSB and the parity bits are at the MSB. In this example, the generated code word is rearranged as shown in Figure 2–9.
Therefore, the encoded input data for F0 is 14F0 (1 0100 1111 0000 in binary). This value matches the result shown in Figure 2–8 on page 2–14.

Figure 2–10 shows the encoding of input data FA and FB.

The input data FA is encoded at 65 ns. The result appears as 06FA on the output port at the next rising edge of the clock at 75 ns with the clock-enable signal set to high. Also, FB is encoded at 75 ns, but it does not appear on the output port at the next rising edge of the clock at 85 ns because the clock-enable signal is low. With the clock-enable feature, you can control when the data should be encoded and when it should appear on the output port.

The objective of this example is to implement and instantiate an ECC decoder using the MegaWizard Plug-In Manager. This example illustrates how the ECC decoder decodes an input code word of 13-bit width (using the code word generated by the ALTECC_ENCODER from Design Example 1) to generate 8 bits of output data. An asynchronous-clear signal is created to illustrate how the signal affects the registered ports. Verify the results you obtain at the end of this example with the expected simulation results provided.
Design Files

The example design files are available in the User Guides section on the Literature page of the Altera® website (www.altera.com).

Configuration Settings

In the ALTECC MegaWizard Plug-In Manager pages, select or verify the configuration settings shown in Table 2–6. Click Next to advance from one page to the next.

Table 2–6. Design Example 2: Configuration Settings

<table>
<thead>
<tr>
<th>MegaWizard Plug-In Manager Page</th>
<th>MegaWizard Plug-In Manager Configuration Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Currently selected device family</td>
<td>Stratix III</td>
</tr>
<tr>
<td></td>
<td>How do you want to configure this module?</td>
<td>Configure this module as an ECC decoder</td>
</tr>
<tr>
<td></td>
<td>How wide should the data be?</td>
<td>13 bits</td>
</tr>
<tr>
<td></td>
<td>Do you want to pipeline the function?</td>
<td>Yes, I want an output latency of 2 clock cycles (1)</td>
</tr>
<tr>
<td></td>
<td>Create an ‘aclr’ asynchronous clear port</td>
<td>Selected</td>
</tr>
<tr>
<td></td>
<td>Create a ‘clocken’ clock enable clock</td>
<td>Not selected</td>
</tr>
</tbody>
</table>

Note to Table 2–6:
(1) The output data appears at the output port at the second rising edge of the clock.

Functional Simulation in the ModelSim-Altera Simulator

Simulate the design in the ModelSim-Altera software to generate a waveform display of the device behavior.

You should be familiar with the ModelSim-Altera software before trying out the design example. If you are unfamiliar with the ModelSim-Altera software, refer to the support page for software products on the Altera website (www.altera.com). On the support page, there are links to such topics as installation, usage, and troubleshooting.

Set up and simulate the design in the ModelSim-Altera software by performing the following steps:

1. Unzip the DE2_ALTECC_DECODER.zip file to any working directory on your PC.

2. Start the ModelSim-Altera software.
Getting Started

3. On the File menu, click **Change Directory**.
4. Select the folder in which you unzipped the files.
5. Click **OK**.
6. On the Tools menu, click **Execute Macro**.
7. Select the **DE2_ALTECC_DECODER.do** file and click **Open**. The **DE2_ALTECC_DECODER.do** file is a script file for the ModelSim-Altera software to automate all the necessary settings for the simulation.

View the simulation results in the Wave window. **Figure 2–11** shows the expected simulation results in the ModelSim-Altera software.

**Figure 2–11. Design Example 2: Simulation Waveform for the ECC Decoder**

![Simulation Waveform for the ECC Decoder](image)

**Understanding the Simulation Results for the ECC Decoder**

In this example, you configured the ECC decoder to have the following properties:

- Accepts 13 bits of input code word
- Pipelined with an output latency of 2 clock cycles
- Has an asynchronous-clear signal

**Figure 2–12** shows the decoding process with no errors. The non-corrupted input code word 14F0 is decoded to generate the output F0.
Design Example 2: ALTECC_DECODER

Figure 2–12. Design Example 2: Decoder Decodes Non-Corrupted Input Code Word 14F0 to Output Data F0

The decoder decodes the code word 14F0 at the first rising edge of the clock at 5 ns. In this case, the input code word is not corrupted. Pipelining has been enabled to have an output latency of 2 clock cycles. Therefore, the decoded data F0 only appears at the output port $q$ at the next rising edge of the clock at 15 ns. If you choose to have an output latency of 1 clock cycle for this example, the data appears at the first rising edge of the clock at 5 ns.

The 13-bit input code word 14F0 (1 0100 1111 0000 in binary) is decoded to generate an 8-bit output data of F0. Figure 2–13 shows the arrangement of parity bits and data bits in the code word 14F0. The prefixes P and D denote parity and data, respectively.

Figure 2–13. Design Example 2: Arrangement of Parity Bits and Data Bits in Code Word 14F0

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5*</td>
<td>P4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The ECC decoder decodes the code word based on the Hamming Code scheme. The following describes the Hamming Code algorithm and explains how the ECC decoder decodes input code word 14F0 to generate output data F0:
Getting Started

1. All bits have their bit positions, and bit positions that are powers of 2 are used as parity bits (positions 1, 2, 4, 8 …). Table 2–7 shows the bit positions, and the positions of the parity bits in a 13-bit code word.

Table 2–7. Design Example 2: Position of Parity Bits for a 13-Bit Code Word

<table>
<thead>
<tr>
<th>(position)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bits</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P5*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

2. All other bit positions are for the data bits. The LSB of the data bit fills the lowest bit position. In this case, starting from the LSB of the data, F0 (1111 0000 in binary) fills the empty bit positions, starting from position (3), as shown in Table 2–8.

Table 2–8. Design Example 2: Filling of Data Bits (1111 0000) for a 13-Bit Code Word

<table>
<thead>
<tr>
<th>(position)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bits and data bits</td>
<td>P1</td>
<td>P2</td>
<td>D1</td>
<td>P3</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
<td>P4</td>
<td>D5</td>
<td>D6</td>
<td>D7</td>
<td>D8</td>
<td>P5*</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
3. Recalculate parity bits to generate the syndrome code. Each syndrome bit calculates the parity (even parity) for some of the bits in the code word using Equation 1.

Table 2–9 shows how the syndrome bits are derived.

<table>
<thead>
<tr>
<th>(position)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
<th>(11)</th>
<th>(12)</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity bits and data bits</td>
<td>P1</td>
<td>P2</td>
<td>D1</td>
<td>P3</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
<td>P4</td>
<td>D5</td>
<td>D6</td>
<td>D7</td>
<td>D8</td>
<td>P5*</td>
</tr>
<tr>
<td>Calculate S1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1=0</td>
</tr>
<tr>
<td>Calculate S2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S2=0</td>
</tr>
<tr>
<td>Calculate S3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S3=0</td>
</tr>
<tr>
<td>Calculate S4</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S4=0</td>
</tr>
<tr>
<td>Calculate S5*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Syndrome bit S1 is calculated using an even parity checking on bit positions (1), (3), (5), (7), (9), and (11).

Syndrome bit S2 is calculated using an even parity checking on bit positions (2), (3), (6), (7), (10), and (11).

The same calculation method using Equation 1 is applied to generate other syndrome bits.

4. Calculate the additional syndrome bit using an even parity checking on all the bits in the code word. In this example, the additional syndrome bit S5* is calculated using an even parity checking on all the bits from position (1) to position (13) as shown in Table 2–9.

The generated syndrome code gives the status of the data, whether an error has occurred, and if so, whether it is a single-bit or double-bit error.
Getting Started

For more information about the generated syndrome code, refer to Table 1–2 on page 1–6.

In this case, the syndrome code is zero (S5*S4S3S2S1=0 0000). No error is detected and no correction is needed on the retrieved data F0 (D8D7D6D5D4D3D2D1=1111 0000) based on the generated syndrome code. Therefore, the flag signals err_detected, err_corrected, and err_fatal are deasserted. The decoding for 14F0 is F0 (1111 0000 in binary), which matches the result shown in Figure 2–12 on page 2–20 and indicates that the data is not corrupted.

Even if the generated syndrome code indicates a single-bit error, the err_detected and err_corrected signals are asserted only if the corrupted bit is from the data bits and not from the parity bits.

Figure 2–14 shows a single-bit error in the input code word that changes the code word to 14F1.

**Figure 2–14. Design Example 2: ECC Decoder Decodes Code Word with Single-Bit Error**

In this case, assume that one of the data bits, the LSB, is corrupted and is inverted from 0 to 1. This causes the code word to become 14F1.

With the same method of decoding using the Hamming Code scheme, the generated syndrome code is 1 0011. S5* equals to 1 (single error detected), and S4S3S2S1 equals to 0011 (the bit at position 3 is corrupted).

As only one of the data bits is corrupted, the decoder is able to correct it by flipping the error bit. Therefore, the corrupted data F1 is decoded as F0. When F0 is shown at the output port at the next rising edge of the...
Design Example 2: ALTECC_DECODER

clock at 25 ns, the err_detected and err_corrected signals are asserted to show that an error is detected and the single-bit error is corrected.

Figure 2–15 shows a double-bit error in the input code word that changes the code word to 14F3.

Figure 2–15. Design Example 2: ECC Decoder Decodes Code Word with Double-Bit Error

In this case, assume that two of the data bits (bit-0 and bit-1) are corrupted and are inverted from 0 to 1. This causes the code word to become 14F3.

The decoder decodes the code word 14F3 at 25 ns and shows the data F3 at 35 ns. Since the ECC decoder can perform only Single Error Correction Double Error Detection (SECD), it does not fix the corrupted data as it contains double-bit errors. In this case, the err_fatal signal is asserted together with the err_detected signal.

Figure 2–16 shows the effects of the asynchronous-clear signal on the registered ports.
If you do not want to use the corrupted data when the `err_fatal` signal is asserted, you can assert the asynchronous-clear signal (`aclr`) to clear the output port `q` and other status signals that are registered. You must enable the pipelining option in the wizard to use this feature.

**Figure 2–16** shows that when the `aclr` signal is asserted at 42 ns, the output and status signals are immediately cleared.

## Conclusion

Error Correction Code (ECC) is a method of error detection and correction in digital data transmission. Altera’s ALTECC_DECODER megafunction can perform Single Error Correction Double Error Detection (SECDED). This method of error correction is best suited for applications where errors occur at random rather than in bursts.

The Quartus II software provides parameterizable ECC encoder and ECC decoder megafunctions through the MegaWizard Plug-In Manager. With these megafunctions, you can easily configure your ECC design with other supported features, such as pipelining, clock enable, and asynchronous clear. These megafunctions are performance-optimized for Altera devices. Therefore, they provide more efficient logic synthesis and device implementation because they automate the coding process and save valuable design time. In addition, it is easy to reconfigure the characteristics of your ECC design through the easy-to-use GUI. Altera recommends that you use these functions during design implementation so you can consistently meet your design goals.
### Ports and Parameters for the ALTECC_ENCODER Megafunction

Table 3–1 shows the input and output ports of the ALTECC_ENCODER megafunction. Table 3–2 shows the parameters of the ALTECC_ENCODER megafunction.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Port Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[]</td>
<td>Input</td>
<td>Yes</td>
<td>Data input port. Input port [WIDTH_DATAWORD–1..0] wide. Contains the raw data to be encoded.</td>
</tr>
<tr>
<td>clock</td>
<td>Input</td>
<td>No</td>
<td>Clock input port. Clock signal to synchronize the encoding operation.</td>
</tr>
<tr>
<td>clocken</td>
<td>Input</td>
<td>No</td>
<td>Clock-enable signal.</td>
</tr>
<tr>
<td>aclr</td>
<td>Input</td>
<td>No</td>
<td>Reset signal that clears the registered ports asynchronously.</td>
</tr>
<tr>
<td>q[]</td>
<td>Output</td>
<td>Yes</td>
<td>Encoded code word output port. Output port [WIDTH_CODEWORD–1..0] wide.</td>
</tr>
</tbody>
</table>
Table 3–2. Parameters for the ALTECC_ENCODER Megafunction

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDTH_DATAWORD</td>
<td>Integer</td>
<td>Yes</td>
<td>Specifies the width of the raw data. Values are from 2 to 64. If omitted, the default is 8.</td>
</tr>
<tr>
<td>WIDTH_CODEWORD</td>
<td>Integer</td>
<td>Yes</td>
<td>Specifies the width of the corresponding code word. Valid values are from 6 to 72 except 9, 17, 33, and 65. If omitted, the default is 13.</td>
</tr>
<tr>
<td>LPM_PIPELINE</td>
<td>Integer</td>
<td>No</td>
<td>Specifies the pipeline for the circuit. Values are from 0 to 2. If the value is 0, the ports are not registered. If the value is 1, the output ports are registered. If the value is 2, the input and output ports are registered. If omitted, the default is 0.</td>
</tr>
</tbody>
</table>

Table 3–3 shows the input and output ports of the ALTECC_DECODER megafunction. Table 3–4 shows the parameters of the ALTECC_DECODER megafunction.

Table 3–3. Input and Output Ports for the ALTECC_DECODER Megafunction

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Port Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[]</td>
<td>Input</td>
<td>Yes</td>
<td>Data input port. Input port [WIDTH_CODEWORD–1 .. 0] wide. Contains the encoded data to be decoded.</td>
</tr>
<tr>
<td>clock</td>
<td>Input</td>
<td>No</td>
<td>Clock input port. Clock signal to synchronize the decoding operation.</td>
</tr>
<tr>
<td>clocken</td>
<td>Input</td>
<td>No</td>
<td>Enable to clock signal.</td>
</tr>
<tr>
<td>aclr</td>
<td>Input</td>
<td>No</td>
<td>Reset signal that clears the registered ports asynchronously.</td>
</tr>
<tr>
<td>q[]</td>
<td>Output</td>
<td>Yes</td>
<td>Decoded data output port. Output port [WIDTH_DATAWORD–1 .. 0] wide.</td>
</tr>
<tr>
<td>err_detected</td>
<td>Output</td>
<td>Yes</td>
<td>Specifies error found.</td>
</tr>
<tr>
<td>err_corrected</td>
<td>Output</td>
<td>Yes</td>
<td>Denotes single-bit error found and corrected. You can use the data because it is already corrected.</td>
</tr>
<tr>
<td>err_fatal</td>
<td>Output</td>
<td>Yes</td>
<td>Denotes double-bit error found, but not corrected. You should not use the data if this signal is asserted.</td>
</tr>
</tbody>
</table>
### Table 3–4. Parameters for the ALTECC_DECODER Megafucntion

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDTH_DATAWORD</td>
<td>Integer</td>
<td>Yes</td>
<td>Specifies the width of the output data. Values are from 2 to 64. If omitted, the default is 8.</td>
</tr>
<tr>
<td>WIDTH_CODEWORD</td>
<td>Integer</td>
<td>Yes</td>
<td>Specifies the width of the corresponding input code word. Valid values are from 6 to 72 except 9, 17, 33, and 65. If omitted, the default is 13.</td>
</tr>
<tr>
<td>LPM_PIPELINE</td>
<td>Integer</td>
<td>No</td>
<td>Specifies the pipeline for the circuit. Values are from 0 to 2. If the value is 0, the ports are not registered. If the value is 1, the output ports are registered. If the value is 2, the input and output ports are registered. If omitted, the default is 0.</td>
</tr>
</tbody>
</table>
Ports and Parameters
Additional Information

Revision History
The following table shows the revision history for this user guide.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 2008 v2.0</td>
<td>Updated, consolidated, and reorganized the following sections to support the Quartus® II software version 8.0:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● “Introduction”</td>
<td>Updated document to support the Quartus II software version 8.0</td>
</tr>
<tr>
<td></td>
<td>● “Features of the ALTECC_DECODER Megafuction”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● “MegaWizard Plug-In Manager Descriptions”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● “Design Example 1: ALTECC_ENCODER”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● “Design Example 2: ALTECC_DECODER”</td>
<td></td>
</tr>
<tr>
<td>May 2007 v1.0</td>
<td>Initial release</td>
<td>—</td>
</tr>
</tbody>
</table>

Referenced Documents
This user guide references the following documents:

- **Quartus II Integrated Synthesis** chapter in volume 1 of the **Quartus II Handbook**
- **Recommended HDL Coding Styles** chapter in volume 1 of the **Quartus II Handbook**
- **Simulation** section in volume 3 of the **Quartus II Handbook**
- **Synthesis** section in volume 1 of the **Quartus II Handbook**

How to Contact Altera
For the most up-to-date information about Altera® products, refer to the following table.

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Contact Method</th>
<th>Contact (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical support</td>
<td>Website</td>
<td><a href="http://www.altera.com/support">www.altera.com/support</a></td>
</tr>
<tr>
<td>Technical training</td>
<td>Website</td>
<td><a href="http://www.altera.com/training">www.altera.com/training</a></td>
</tr>
<tr>
<td></td>
<td>Email</td>
<td><a href="mailto:custrain@altera.com">custrain@altera.com</a></td>
</tr>
<tr>
<td>Product literature</td>
<td>Website</td>
<td><a href="http://www.altera.com/literature">www.altera.com/literature</a></td>
</tr>
</tbody>
</table>
Typographic Conventions

<table>
<thead>
<tr>
<th>Information Type</th>
<th>Contact Method</th>
<th>Contact (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-technical support (General) Email</td>
<td><a href="mailto:nacomp@altera.com">nacomp@altera.com</a></td>
<td></td>
</tr>
<tr>
<td>(Software Licensing) Email</td>
<td><a href="mailto:authorization@altera.com">authorization@altera.com</a></td>
<td></td>
</tr>
</tbody>
</table>

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 in the following table.

<table>
<thead>
<tr>
<th>Visual Cue</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold Type with Initial Capital Letters</strong></td>
<td>Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <strong>Save As</strong> dialog box.</td>
</tr>
<tr>
<td><strong>bold type</strong></td>
<td>External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: ( f_{\text{MAX}} ), \qdesigns\ directory, \d: \ drive, chiptrip.gdf file.</td>
</tr>
<tr>
<td><strong>Italic Type with Initial Capital Letters</strong></td>
<td>Document titles are shown in italic type with initial capital letters. Example: \AN 75: High-Speed Board Design.\</td>
</tr>
<tr>
<td><strong>Italic type</strong></td>
<td>Internal timing parameters and variables are shown in italic type. Examples: ( t_{\text{PIA}} ), ( n + 1 ). Variable names are enclosed in angle brackets (&lt;&gt;) and shown in italic type. Example: &lt;file name&gt;, &lt;project name&gt;.pof file.</td>
</tr>
<tr>
<td><strong>Initial Capital Letters</strong></td>
<td>Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.</td>
</tr>
<tr>
<td><strong>“Subheading Title”</strong></td>
<td>References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”</td>
</tr>
<tr>
<td><strong>Courier type</strong></td>
<td>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:\qdesign\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.</td>
</tr>
<tr>
<td><strong>1., 2., 3., and a., b., c., etc.</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Bullets are used in a list of items when the sequence of the items is not important.</td>
</tr>
<tr>
<td></td>
<td>The checkmark indicates a procedure that consists of one step only.</td>
</tr>
<tr>
<td></td>
<td>The hand points to information that requires special attention.</td>
</tr>
</tbody>
</table>
## Visual Cue Meaning

<table>
<thead>
<tr>
<th>Visual Cue</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Caution" /></td>
<td>A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.</td>
</tr>
<tr>
<td><img src="#" alt="Warning" /></td>
<td>A warning calls attention to a condition or possible situation that can cause injury to the user.</td>
</tr>
<tr>
<td><img src="#" alt="Angled Arrow" /></td>
<td>The angled arrow indicates you should press the Enter key.</td>
</tr>
<tr>
<td><img src="#" alt="Feet" /></td>
<td>The feet direct you to more information about a particular topic.</td>
</tr>
</tbody>
</table>