DDR and DDR2 SDRAM ECC Reference Design

Version 1.0, June 2006 Application Note 415

Introduction

This application note describes an error-correcting code (ECC) block for use with the Altera DDR and DDR2 SDRAM controller MegaCore functions. Altera also supplies an ECC reference design, which uses the ECC block with the DDR2 SDRAM controller MegaCore function and a Micron MT9HTF3272AY-53EB3 DIMM at 200 MHz. The reference design runs on a Stratix® II High-Speed Development Board.

The Stratix II High-Speed Development Board is part of the High-Speed Development Kit, Stratix II Edition.

The ECC block comprises an encoder and a decoder-corrector, which can detect and correct single-bit errors and detect double-bit errors. It is a cost-efficient solution that is fast, has a low latency, has no detrimental effect on system performance, and uses almost no system resources. The ECC block uses an 8-bit ECC for each 64-bit message.

The application note also explores the idea of using ECCs on any bus, interfacing with any device, internal or external to Altera FPGAs.

Features

- Fully parameterized Hamming code ECC block with 8-bit ECC for 64-bit message
- Configurable latency:
  - 1 or 2 clock delay during writes for ECC computation
  - 2 or 3 clock delay during reads. Detect any single- and double-bit error on the first clock and correct any detected single-bit errors on the second clock
- Avalon-based high-speed architecture:
  - Performance of over 38 gigabits per second (Gbps)
  - Uses around 300 to 400 LEs
- Reference design:
  - Instantiated with Altera® DDR2 SDRAM Controller MegaCore function
  - Adaptable for all types of 64-bit memory controllers that are compatible with the Avalon® interface
  - Can run at memory controllers f_{MAX}
- VHDL tests to verify the ECC block:
  - IP functional simulation models for use in Altera-supported VHDL HDL simulators
  - ModelSim simulation script

General Description

In computer science and information theory, the issue of error correction and detection has great practical importance. ECCs permit detection and correction of errors that result from noise or other impairments during transmission from the transmitter to the receiver. Given some data, ECC methods enable you to check whether data has been corrupted, which can provide the difference between a functional and nonfunctional system. Error correction schemes permit error localization and also give the possibility of correcting them.

Error correction and detection schemes find use in implementations of reliable data transfer over noisy transmission links, data storage media (including dynamic RAM, compact discs), and other applications where the integrity of data is important. Error correction avoids retransmission of the data, which can degrade system performance.

RAM Devices

RAM devices do not as such support error control codes. There are no mandatory requirements for ECC support on RAM/DRAM devices. Memory suppliers are generally not in favor of implementing a complex logic function like ECC onto a RAM die. It is costly, inefficient, and leads to an expensive memory subsystem.

Where enhanced reliability is a requirement, the standard technique is to use a wider interface. In the context of SDRAMs, DIMMs come in two widths: 64 and 72 bits. The 72-bit DIMMs are targeted for use with ECCs, because of the extra 8 bits. The extra 8 bits are merely extra data bits, in reality you can use any of the bits. An extra 8 bits of parity on 64 bits of data allows you to employ a two-bit error detection. single bit correcting Hamming code.

The Hamming Code

Hamming Codes are relatively simple yet powerful ECC codes.

General Equations

For a 64-bit number there are 64 possible one bit errors. 64 different binary permutations can be recognized in a string of length 6 bits. However, another state is needed to represent the case when a detectable error has not occurred.

The number of parity bits, m, needed to detect and correct a single bit error in a data string of length n is given by the following equation:

$$m = \log_2 n + 1$$

The ECC block uses the Hamming code with an additional parity bit, which can detect single and double-bit errors, and correct single-bit errors. The extra parity bit applies to all bits after the Hamming code check bits have been added. This extra parity bit represents the parity of the codeword. If one error occurs, the parity changes, if two errors occur, the parity stays the same. In general the number of parity bits, m, needed to detect a double-bit error or detect and correct a single-bit error in a data string of length n, is given by the following equation:

$$m = \log_2 n + 2$$

Table 1 shows the parity bits for a double-bit error detection Hamming Code.

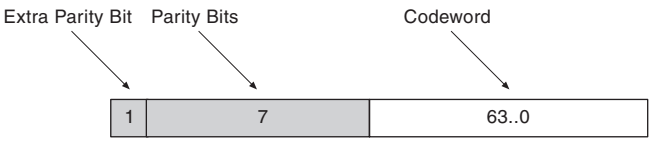
Table 1. Parity Bits for a Double Bit Error Detection Hamming Code

Data Bits (*n*)	Parity Bits (*m*)
1	2
3:2	3
7:4	4
15:8	5
31:16	6
63:32	7
127:64	8
255:128	9
511:256	10

Traditional Hamming Codes

The ECC block uses the traditional Hamming code and places the parity bits at the end of the codeword. The extra parity bit is placed at the end of the codeword. Figure 2 shows the location of the parity bits.

Figure 1. Parity Bit Location



Encoding is performed by multiplying the original message vector by the generator matrix; decoding is performed by multiplying the codeword vector by the parity check matrix H. All additions are performed modulo 2. In hardware, this process equates to XORing a particular set of data elements and is computationally inexpensive. If an error occurs and one of the parity or data bits change during transmission, the ECC decoder-corrector gives the bit syndrome of the data bit that is affected by recalculating the parity bits and XORing them with the transmitted parity bits (computing the syndrome). Then the decoder-corrector allows the correction of a single-bit error.

The Generator Matrix & The Parity Check Matrix

The generator matrix directly affects the encoding operation. The parity check matrix is generated automatically and is defined specifically to work with the generator matrix or traditional Hamming code.

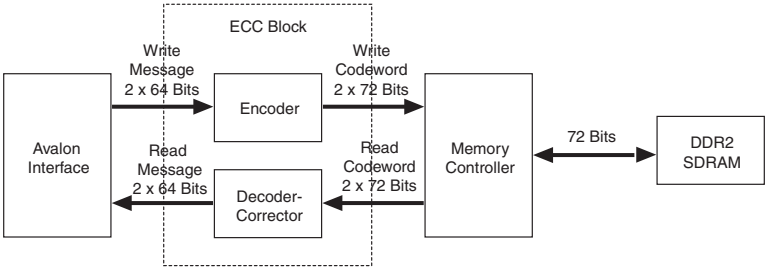
Functional Description

The ECC block operates between the Avalon interface and the memory controller. It is compatible with the Avalon interface on both sides.

The local data buses are 128 bits between the Avalon interface and the ECC block and 144 bits between the ECC block and the memory controller.

Figure 2 shows the ECC block diagram.

Figure 2. ECC Block Diagram



The ECC block comprises the following blocks:

- The encoder—the message is encoded to a codeword
- The decoder-corrector—the codeword is decoded and corrected if necessary

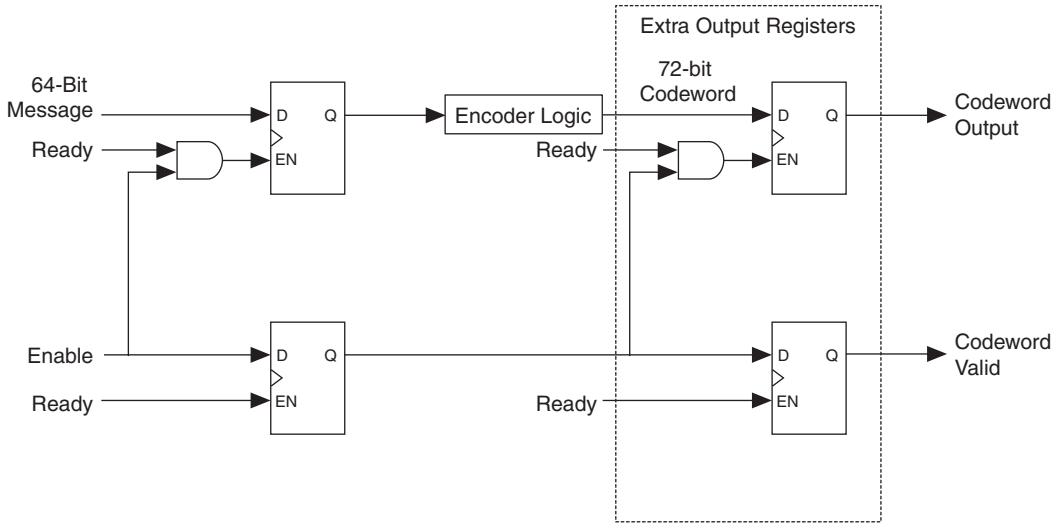
Each message must be encoded and decoded separately because it can be written and read on the memory controller separately. The 128-bit Avalon data bus is split into two 64-bit messages. They are encoded into two 72-bit codewords and concatenated into the 144-bit ECC data bus. Similarly, the 144-bit ECC read data bus is split into two 72-bit codewords. They are decoded and corrected into two 64-bit messages and then concatenated into the 128-bit Avalon read data bus.

Because the encoder contains up to two registers, all the control signals sent to the memory controller from the local interface are pipelined. These registers hold their data when the `ready` signal is deasserted.

Encoder

Figure 3 shows the encoder block diagram.

Figure 3. Encoder Block Diagram



The encoder runs for an input message of 64 bits and uses a Hamming code with an additional parity bit. The encoder takes the input data and encodes the message into a (64 + 8) bit codeword. The generator matrix, **matrix_g_71_64_v1**, which is in the **ecc_matrix_ref.vhd** library, creates the codeword.

The encoder includes two register levels of pipeline; the second one can be suppressed with the parameter `register_output_enc`. This register level allows a higher operational frequency. These registers maintain the data if the `ready` signal is deasserted and are enabled by the `en_encoder` signal.

Table 2 shows the encoder signals.

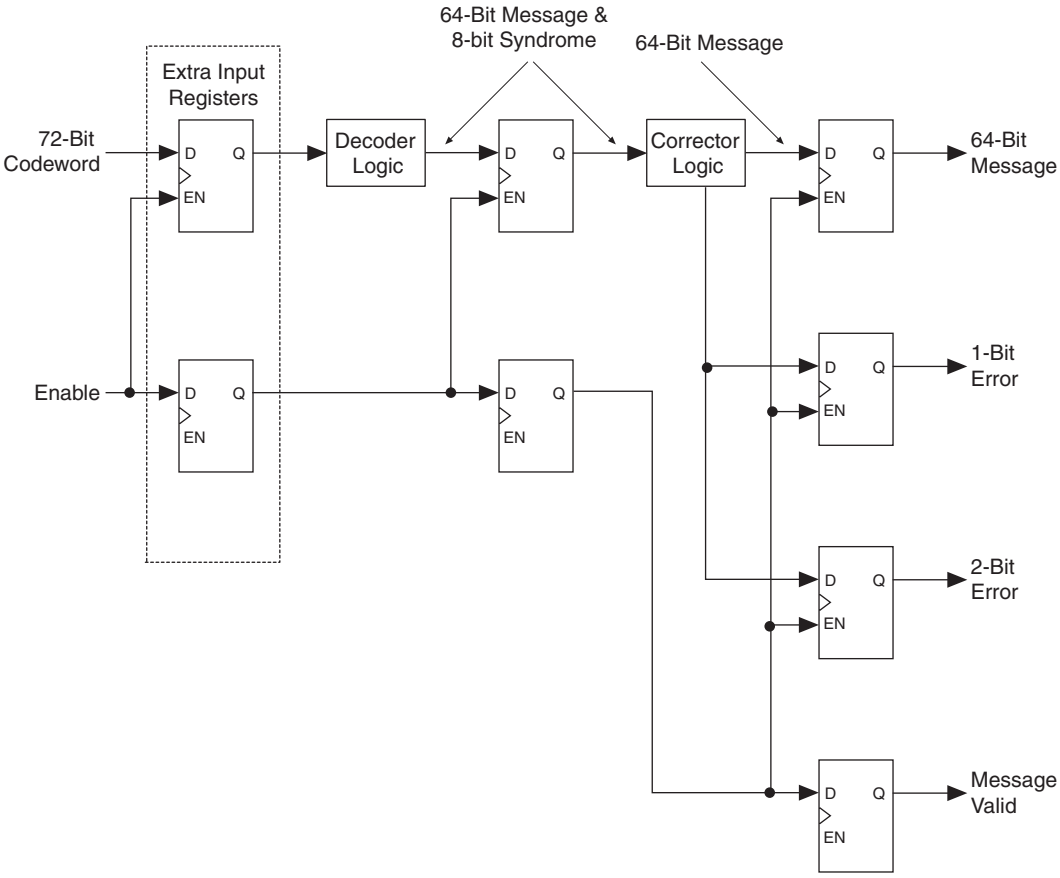
Table 2. Encoder Signals

Name	Direction	Description
`clk`	Input	System clock.
`reset_n`	Input	System reset, which can be asserted asynchronously but must be deasserted synchronous to the rising edge of the system clock.
`message_in[63:0]`	Input	Original data input to the encoder. This data is the message to be encoded in a codeword.
`en_encoder`	Input	Encoder enable signal, which is driven by the local interface master and enables the encoding of the data coming on the same clock edge. `en_encoder` is asserted when a valid message to encode is on the `message_in` bus.
`ready`	Input	The ready signal comes from the block receiving the codeword; it indicates if the memory controller can accept any more requests. If the `ready` signal is asserted in the clock cycle, it enables the encoder. If the signal is deasserted, it indicates to the registers of the encoder to keep their data.
`codeword_out[71:0]`	Output	Encoded codeword data from the encoder. The codeword is driven from the encoder to the memory interface.
`codeword_val`	Output	Data valid signal. `codeword_val` is asserted high for one clock cycle, whenever there is a valid output on `codeword[]`.

The Decoder-Corrector

Figure 4 shows the decoder-corrector block diagram.

Figure 4. Decoder-Corrector Block Diagram



The decoder-corrector is a self-contained entity for messages of 64 bits and uses Hamming code with additional parity bit.

The decoder creates a syndrome and sends it with the received message to the corrector. The corrector detects single- and double-bit errors and corrects the single bit errors of the received message according to the syndrome.

The decoder-corrector includes three register levels of pipeline The first level can be suppressed with the parameter `register_input_dec`.

Table 3 shows the decoder-corrector signals.

Table 3. Decoder-Corrector Signals

Name	Direction	Width	Description
clk	Input		System clock.
reset_n	Input		System reset, which can be asserted asynchronously but must be deasserted synchronous to the rising edge of the system clock.
codeword_in[]	Input	71:0	Codeword input to the decoder-corrector from the memory interface.
en_decoder	Input		Decoder-corrector enable signal. en_decoder is driven by the memory interface to the decoder-corrector. en_decoder is asserted when a valid codeword to decode or correct is on the codeword_in.
message_out[]	Output	63:0	Corrected message.
codeword_val	Output		Codeword valid signal. codeword_val is asserted high for one clock cycle, whenever there is a valid output on the codeword[] bus.
error_1bit	Output		If error_1bit is asserted, the message on message_out had one bit error on the received codeword. It comes out on the same clock edge as codeword_val.
error_2bit	Output		If error_2bit is asserted, the message on message_out had two bits error on the received codeword. It comes out on the same clock edge as the codeword_val.

Error Diagnosis

The ECC block can diagnose if a one- or two-bit error has occurred on the the output message. These flags, error_1bit and error_2bit, are provided with the message output on the decoder-corrector.

Table 4 shows the diagnosis.

Table 4. Diagnosis

error_1bit	error_2bit	Diagnosis
0	0	There is no error on the message on the output.
1	0	There was one error on the codeword; the message is equivalent to the original.
0	1	There are two errors on the codeword; no correction have been made.
1	1	Not possible.

Error Correction

The error correction is linked with the error detection.

- For no errors—no corrections are made.
- For one error—if the error occurs in the parity bit of the codeword, no correction is performed on the message output; if the error occurs in the message, the bit is corrected on the message output.
- For two errors—the ECC block cannot perform an error correction on more than one error. No correction is applied on the output message. It is possible that one or both of the double-bit errors are on the parity bits of the codeword, so either no or one error appears on the output message.
- For more than two errors—the ECC block cannot detect when more than two errors occur in the codeword. So, the ECC block either diagnoses no, one, or two errors. If more than two errors are diagnosed as a single error, the ECC block applies a correction as if the error detected was a one error. This method can lead to corrupt data but is a limitation of this form of ECC.

Signals

The ECC block signals are compatible with SOPC Builder.

For more information on implementing a DDR and DDR2 SDRAM Controller in SOPC Builder, refer to the *DDR and DDR2 SDRAM Controller Compiler User Guide*.

Table 5 shows the ECC block signals.

Table 5. Signals (Part 1 of 3)

Name	Direction	Width	Description
clk	Input		System clock.
reset_n	Input		System reset, which can be asserted asynchronously but must be deasserted synchronous to the rising edge of the system clock.
Local Interface Signals			

Table 5. Signals (Part 2 of 3)

Name	Direction	Width	Description
local_addr	Input	23:0	Memory address at which the burst should start. The width of this bus is sized using the following equation: For one chip select: width = bank bits + row bits + column bits – 1 For multiple chip selects: width = chip bits + bank bits + row bits + column bits – 1 The least significant bit (LSB) of the column address on the memory side is ignored, because the local data width is twice that of the memory data bus width. The order of the address bits is set in the clear-text part of the MegaCore function (**auk_ddr_sdram.vhd**). The order is: chips, bank, row, column, but you can change it if required.
local_burstbegin	Input		Avalon burst begin strobe, which indicates the beginning of an Avalon burst. The controller supports burst lengths of 1, 2, or 4 for DDR SDRAM and 2 for DDR2 SDRAM.
local_read_req	Input		Read request signal.
local_size[]	Input	1:0	The burst size of the requested access, which is encoded as a binary number. You may request any size up to the maximum burst length. For example, if you chose a memory burst length of 8, the local burst size is 4 and you may request bursts of length 1, 2, 3, or 4. Similarly, if you chose a memory burst length of 4, the local burst size is 2 and you may request bursts of length 1 or 2. If you chose a memory burst length of 2 (local burst size of 1) or use the Avalon interface as your local interface, the local_size[] signal is tied to 1 and is not visible on the controller interface.
local_wdata[]	Input	127:0	Write data bus. The width of local_wdata is twice that of the memory data bus.
local_write_req	Input		Write request signal.
local_rdata[]	Output	127:0	Read data bus. The width of local_rdata is twice that of the memory data bus.
local_rdata_valid	Output		Read data valid signal. The local_rdata_valid signal indicates that valid data is present on the read data bus. The timing of local_rdata_valid is automatically adjusted to cope with your choice of resynchronization and pipelining options.

Table 5. Signals (Part 3 of 3)

Name	Direction	Width	Description
local_ready	Output		The local_ready signal indicates that the ECC block is ready to accept request signals. If local_ready is asserted in the clock cycle that a read or write request is asserted, that request has been accepted. The local_ready signal is deasserted to indicate that the ECC cannot accept any more requests.
ECC Diagnosis Signals			
error_1bit_m1	Output		If error_1bit_m1 is asserted, the first message on local_rdata (63:0) had one bit error on the received codeword. It is presented on the same clock edge as local_rdata_valid.
error_2bit_m1	Output		If error_2bit_m1 is asserted, the first message on local_rdata (63:0) had two bits error on the received codeword. It is presented on the same clock edge as local_rdata_valid.
error_1bit_m2	Output		If error_1bit_m2 is asserted, the second message on local_rdata (127:64) had one bit error on the received codeword. It is presented on the same clock edge as local_rdata_valid.
error_2bit_m2	Output		If error_2bit_m2 is asserted, the second message on local_rdata (127:64) had two bits error on the received codeword. It is presented on the same clock edge as local_rdata_valid.

Table 5 shows the memory controller interface signals.

Table 6. Memory Controller Interface Signals (Part 1 of 2)

Name	Direction	Width	Description
ecc_addr	Output	23:0	Memory address at which the burst should start. The width of this bus is sized using the following equation: For one chip select: width = bank bits + row bits + column bits – 1 For multiple chip selects: width = chip bits + bank bits + row bits + column bits – 1 The least significant bit (LSB) of the column address on the memory side is ignored, because the local data width is twice that of the memory data bus width.
ecc_burstbegin	Output		Avalon burst begin strobe, which indicates the beginning of an Avalon burst. The controller supports burst lengths of 1, 2, or 4 for DDR SDRAM and 2 for DDR2 SDRAM.
ecc_read_req	Output		Read request signal.
ecc_size[]	Output		The burst size of the requested access, which is encoded as a binary You may request any size up to the maximum burst length, so for example if you chose a memory burst length of 8, the local burst size is 4 and you may request bursts of length 1, 2, 3 or 4. Similarly, if you chose a memory burst length of 4, the local burst size is 2 and you may request bursts of length 1 or 2. If you chose a memory burst length of 2 (local burst size of 1) or use the Avalon interface as your local interface, the ecc_size[] signal is tied to 1 and is not visible on the controller interface.
ecc_wdata[]	Output	143:0	Write data bus. The width of ecc_wdata is twice that of the memory data bus.
ecc_write_req	Output		Write request signal.
ecc_rdata[]	Input	143:0	Read data bus. The width of local_rdata is twice that of the memory data bus.

Table 6. Memory Controller Interface Signals (Part 2 of 2)			
Name	**Direction**	**Width**	**Description**
ecc_rdata_valid	Input		Read data valid signal. The ecc_rdata_valid signal indicates that valid data is present on the read data bus. The timing of ecc_rdata_valid is automatically adjusted to cope with your choice of resynchronization and pipelining options.
ecc_ready	Input		The local_ready signal indicates that the DDR or DDR2 SDRAM controller is ready to accept request signals. If local_ready is asserted in the clock cycle that a read or write request is asserted, that request has been accepted. The local_ready signal is deasserted to indicate that the DDR or DDR2 SDRAM controller cannot accept any more requests.

Parameters

You can specify the following latency parameters on the ECC block:

- For the encoder—use a 1 clock instead of 2 clock delay, which removes the extra output registers but may decrease the frequency
- For the decoder-corrector—use a 2 instead of 3 clock delay, which removes the extra input registers but may decrease the frequency

You can also implement individual blocks—either the encoder or decoder-corrector separately to make various 64-bit ECC encoder or decoder-correctors.

Reference Design

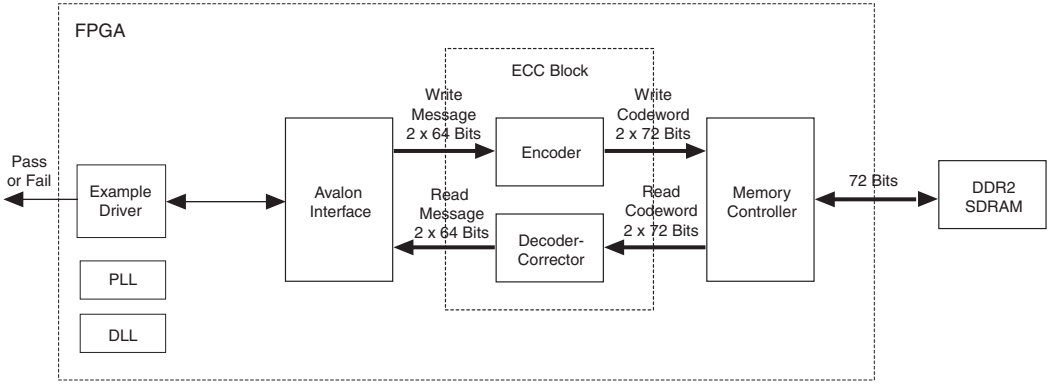
The reference design tests the complete transmission chain. The reference design implements an ECC block on a Stratix II EP2S60F device and includes a 72-bit DDR or DDR2 SDRAM controller with an Avalon interface.

For more information on the Altera DDR and DDR2 SDRAM Controller MegaCore functions, refer to the *DDR and DDR2 SDRAM Controller Compiler User Guide*.

The reference design contains a driver that generates and tests data, the ECC block, and a memory controller for Micron MT9HTF3272AY-53EB3 DIMM memory. The reference design operates at 200 MHz on the Stratix II High Speed Development board.

Figure 5 shows the reference design block diagram.

Figure 5. Reference Design Block Diagram



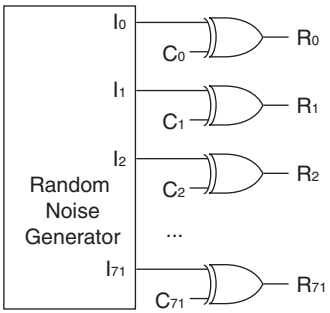
Random Noise Generator

Altera has carried out a wide variety of test on the ECC block to verify its functionality and provides simulation testing for the DDR SDRAM controller (driver with data generator). To test the ECC block (**ecc_block.vhd**), it includes a random noise generator, which produces an error on the output codeword and checks that the system can correct and detect these errors.

The random noise generator models the noisy communication channel over which data is transmitted from the source and the destination, thus you can show whether the errors have been detected and corrected correctly.

To generate an error, you pass the `codeword_out` to the random noise generator. The random noise generator generates an error of a certain number of bits. By asserting the `number_errors`, you can set the number of bits that are affected by noise to 0, 1, or 2, to test different predictable errors. To create an error, a random signal inverts data bits. Figure 6 shows the random noise generator.

Figure 6. Random Noise Generator



Getting Started

This section includes the following sections:

- "System Requirements"
- "Install the Design"
- "Generate the DDR2 SDRAM Controller"
- "Simulate Using the Testbench"
- "Compile the Design"
- "Hardware Test"

System Requirements

The instructions in this section require the following hardware and software:

- A computer running any of the following operating systems:
  - Windows 2000/XP
  - Red Hat Linux 8.0
  - Red Hat Enterprise Linux 3 WS (with support for 32-bit, AMD64, or Intel EM64T workstations)
  - Solaris 8 or 9 (32-bit or 64-bit)
- Quartus® II software version 6.0
- Altera® DDR and DDR2 SDRAM Controller Compiler v3.4.0
- ModelSim SE 6.1 (for simulation with VHDL and Verilog HDL code)

  If you have a VHDL DDR2 SDRAM model, you can use the ModelSim-Altera simulator.

Install the Design

To evaluate the ECC reference design for the DDR2 SDRAM Controller, unzip the **an415.zip** file.

Figure 7 shows the directory structure of the reference design.

Figure 7. Directory Structure

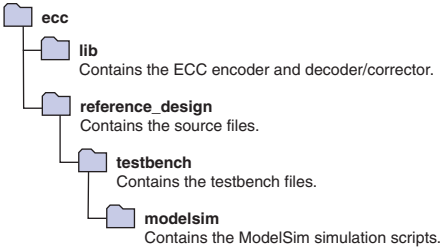


Table 2 shows the ECC reference design files.

Table 7. Reference Design Files

File	Description
ddr_stratix2.qpf	File that contains the Quartus II project.
ddr_stratix2.qsf	File that contains information for the Quartus II project.
/lib/	
ecc_module_ref.vhd	Top-level file of the ECC block.
ecc_enc_ref.vhd	ECC encoder block.
ecc_dec_corr_ref.vhd	ECC decoder-corrector top-level file.
ecc_decoder_ref.vhd	ECC decoder block.
ecc_corrector_ref.vhd	ECC corrector block.
ecc_matrix_ref.vhd	Package containing the ECC generator and functions.
/reference_design/	
ddr2_controller.vhd	Controller provided for the Micron MT9HTF3272AY-53EB3 DIMM.
ddr2_controller_128b_driver.vhd	The 128-bit driver is a self-test block that issues read and write commands to the controller and checks the read data to produce the pass/fail and test complete signals.
ddr_stratix2.vhd	ECC reference design top-level file. Within the file, you can choose the parameters for **ecc_module_ref.vhd** (the number of registers and error bits inserted).
ddr_pll_stratixii.vhd	File for the Stratix II PLL.
/reference_design/testbench/	
ddr_stratix2_ecc_tb.vhd	Testbench for the ECC reference design.

Table 7. Reference Design Files	
File	**Description**
generic_ddr2_sdram.vhd	DDR2 SDRAM simulation block that connects to the 8-bit RAM block.
generic_ddr_dimm_model.vhd	DDR2 DIMM simulation top-level file.
/reference_design/testbench/modelsim/	
wave_ecc.do	File that contains the ModelSim signal to trace for the reference design.
ddr2_controller_ddr_sdram_vsim_ecc.tcl	File that contains the ModelSim Tcl script for the simulation of the reference design

To add one extra register level on the ECC block (disabled by default) or to activate noise generation on the codewords created, you can change the `extra_register` and the `number_errors` parameters instantiated for the ECC block in the top-level file **ddr_stratix2.vhd**.

Generate the DDR2 SDRAM Controller

To generate the files in the Quartus II software, follow these steps:

1. Start the Quartus II software.
2. Open the **ddr_stratix2.qpf** project.
   a. On the File menu, click **Open Project**.
   b. Browse to the **ddr_stratix2.qpf** project and click **Open**.
3. On the Tool menu click **MegaWizard Plug-In Manager**
4. Select **Edit an existing custom megafunction variation** and click **Next**.
5. Browse to the **reference_design/ddr2_controller.vhd** file and click **Next**.
6. The DDR2 SDRAM controller for the reference design has the correct parameters. In IP Toolbench, click **Generate** to generate the files for the MegaCore function. Some files may be overwritten during the generation.

Simulate Using the Testbench

To simulate using the testbench, follow these steps:

1. Download and unzip the 8-bit width memory test model from Micron at:

   http://download.micron.com/downloads/models/verilog/sdram/ddr2/256Mb_ddr2.zip

2. Extract the file **ddr2.v** to:

   <*path*>**/reference_design/testbench/**

3. Add the following code in the **ddr2.v** file before the module definition:

```
`define sg5E;
`define x8;
```

4. Extract the file **ddr2_parameters.vh** to:

   <*path*>**/reference_design/testbench/modelsim**

5. Edit the simulation script:

   <*path*>**/reference_design/testbench/**
   /modelsim/ddr2_controller_ddr_sdram_vsim_ecc.tcl

6. Ensure the correct MegaCore function directory is referenced by editting the following line in the script:

"Set ddr_sdram_megacore_rootdir <*path to DDR SDRAM MegaCore function*>/ddr_ddr2_sdram-v3.4.0"

7. Open the ModelSim SE simulator.

8. On the File menu, click **Change Directory** and specify the ModelSim Tcl script directory of the reference design: <*path*>**/reference_design/testbench/modelsim/**

9. Type the following command:

   set memory_model <*model_name*>↵

   where <*model_name*> is the filename of the downloaded memory model.

10.

11. Launch the script with this function:

```
"source ddr2_controller_ddr_sdram_vsim_ecc.tcl"
```

Compile the Design

To compile the design, follow these steps:

1. Start the Quartus II software.
2. Run the **stratixii_hsio_pins.tcl** script, which is in the DDR2 SDRAM MegaCore directory.
3. On the Tools menu, click **Start compilation**.

Hardware Test

The reference design uses the Micron MT9HTF3272AY-53EB3 DIMM memory at the frequency of 200 MHz and can be implemented on a Stratix II EP2S60F1020C3 device.

✓ On the Tools menu click **Programmer** and program the board.

Performance

Table 8 shows typical expected performance with the Quartus II software, version 6.0.

Table 8. Performance

Device Family	LEs or ALUTs
Stratix II	573

Note to Table 8:
(1) Stratix II devices use adaptive look-up tables (ALUTs); other devices use logic elements (LEs).

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com