Reed-Solomon Compiler

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

MegaCore Version: 11.0
Document Date: May 2011

UG-RSCOMPILER-11.0

Contents

Chapter 1. About This Compiler

Features . . . 1–1
Release Information . . . 1–2
Device Family Support . . . 1–2
Performance and Resource Utilization . . . 1–3
Installation and Licensing . . . 1–5
  OpenCore Plus Evaluation . . . 1–6
  OpenCore Plus Time-Out Behavior . . . 1–6

Chapter 2. Getting Started

Design Flows . . . 2–1
DSP Builder Flow . . . 2–1
MegaWizard Plug-In Manager Flow . . . 2–2
  Parameterize the MegaCore Function . . . 2–3
  Set Up Simulation . . . 2–6
  Generate the MegaCore Function . . . 2–7
Simulate the Design . . . 2–9
Compile the Design . . . 2–10
Program a Device . . . 2–10

Chapter 3. Functional Description

Background . . . 3–1
  Erasures . . . 3–2
  Shortened Codewords . . . 3–2
  Variable Encoding and Decoding . . . 3–3
RS Encoder . . . 3–3
RS Decoder . . . 3–4
  Error Symbol Output . . . 3–5
  Bit Error Count . . . 3–6
Interfaces . . . 3–6
Parameters . . . 3–7
Signals . . . 3–8
Throughput Calculator . . . 3–10

Appendix A. Using the RS Encoder or Decoder in a CCSDS System

Introduction . . . A–1
Test Patterns . . . A–1

Additional Information

Revision History . . . Info–1
How to Contact Altera . . . Info–1
Typographic Conventions . . . Info–2

1. About This Compiler

This document describes the Altera® Reed-Solomon (RS) Compiler. The Altera RS Compiler comprises a fully parameterizable encoder and decoder for forward error correction applications. RS codes are widely used for error detection and correction in a wide range of DSP applications for storage, retrieval, and transmission of data. The RS Compiler has the following options:

- Erasures-supporting option—the RS decoder can correct symbol errors up to the number of check symbols, if you give the location of the errors to the decoder. Refer to "Erasures" on page 3–2.
- Variable encoding or decoding—you can vary the total number of symbols per codeword and the number of check symbols, in real time, from their minimum allowable values up to their selected values, when you are encoding or decoding.
- Error symbol output—the RS decoder finds the error values and location and adds these values in the Galois field to the input value.
- Bit error output—either split count or full count

Features

The Altera Reed-Solomon Compiler supports the following features:

- High-performance encoder/decoder for error detection and correction
- Fully parameterized RS function, including:
  - Number of bits per symbol
  - Number of symbols per codeword
  - Number of check symbols per codeword
  - Field polynomial
  - First root of generator polynomial
  - Space between roots in generator polynomial
- Decoder features:
  - Variable option
  - Erasures-supporting option
- Encoder features variable architectures
- Support for shortened codewords
- Conforms to Consultative Committee for Space Data Systems (CCSDS) *Recommendations for Telemetry Channel Coding*, May 1999
- Easy-to-use IP Toolbench interface:
  - Generates parameterized encoder or decoder
  - Generates customized testbench and customized Tcl script

- DSP Builder ready
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore Plus evaluation

Release Information

Table 1–1 provides information about this release of the Reed-Solomon (RS) Compiler.

Table 1–1. RS Compiler Release Information

Item	Description
Version	11.0
Release Date	May 2011
Ordering Codes	IP-RSENC (Encoder) IP-RSDEC (Decoder)
Product IDs	0039 0041 (Encoder) 0080 0041 (Decoder)
Vendor ID	6AF7

For more information about this release, refer to the *MegaCore IP Library Release Notes and Errata*.

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore® function. The *MegaCore IP Library Release Notes and Errata* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release.

Device Family Support

Table 1–2 defines the device support levels for Altera IP cores.

Table 1–2. Altera IP Core Device Support Levels

FPGA Device Families	HardCopy Device Families
Preliminary support—The IP core is verified with preliminary timing models for this device family. The IPcore meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.	**HardCopy Companion**—The IP core is verified with preliminary timing models for the HardCopy companion device. The IP core meets all functional requirements, but might still be undergoing timing analysis for the HardCopy device family. It can be used in production designs with caution.
Final support—The IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.	**HardCopy Compilation**—The IP core is verified with final timing models for the HardCopy device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 1–3 shows the level of support offered by the RS Compiler to each of the Altera device families.

Table 1–3. Device Family Support

Device Family	Support
Arria® GX	Final
Arria II GX	Final
Arria II GZ	Final
Cyclone®	Final
Cyclone II	Final
Cyclone III	Final
Cyclone III LS	Final
Cyclone IV GX	Final
HardCopy® II	HardCopy Compilation
HardCopy III	HardCopy Compilation
HardCopy IV E	HardCopy Compilation
HardCopy IV GX	HardCopy Compilation
Stratix®	Final
Stratix II	Final
Stratix II GX	Final
Stratix III	Final
Stratix IV GT	Final
Stratix IV GX/E	Final
Stratix V	Preliminary
Stratix GX	Final
Other device families	No support

Performance and Resource Utilization

Table 1–4 shows the typical performance using the Quartus II software for Cyclone III (EP3C10F256C6) devices.

☞ Cyclone III devices use combinational look-up tables (LUTs) and logic registers; Stratix III and Stratix IV devices use combinational adaptive look-up tables (ALUTs) and logic registers.

Table 1–4. Performance—Cyclone III Devices (Part 1 of 2)

Parameters									
Options	**Keysize**	**Bits** *(1)*	**Symbols** *(2)*	**Check** *(3)*	**LUTs**	**Logic Registers**	**Memory (M9K)**	**f_{MAX} (MHz)**	**Throughput (Mbps)**
Standard decoder	Half	4	15	6	541	365	5	230	216
Standard decoder	Half	8	204	16	1,720	995	5	202	1,613
Split bit error decoder	Half	8	204	16	1,765	1,057	5	194	1,552

Table 1–4. Performance—Cyclone III Devices (Part 2 of 2)

Parameters									
Options	**Keysize**	**Bits** *(1)*	**Symbols** *(2)*	**Check** *(3)*	**LUTs**	**Logic Registers**	**Memory (M9K)**	**f_{MAX} (MHz)**	**Throughput (Mbps)**
Full bit error decoder	Half	8	204	16	1,778	1,058	5	190	1,519
Standard decoder	Half	8	255	32	2,972	1,676	5	193	1,213
Variable decoder	Half	8	204	16	1,886	1,074	5	202	1,620
Erasures decoder	Half	8	204	16	3,151	1,561	5	188	1,500
Erasures and variable decoder	Half	8	204	16	3,465	1,704	6	191	1,527
Standard encoder	—	8	204	16	256	210	—	324	2,593
Variable encoder	—	8	204	16	1,048	313	—	237	1,897
Variable encoder	—	8	204	32	2,341	580	—	227	1,813

Notes to Table 1–4:

(1) The number of bits per symbol (*m*).
(2) The number of symbols per codeword (*N*).
(3) The number of check symbols per codeword (*R*).

Table 1–5 shows the typical performance using the Quartus II software for Stratix III (EP3SE50F780C2) devices.

Table 1–5. Performance—Stratix III Devices

Parameters									
Options	**Keysize**	**Bits** *(1)*	**Symbols** *(2)*	**Check** *(3)*	**ALUTs**	**Logic Registers**	**Memory (M9K)**	**f_{MAX} (MHz)**	**Throughput (Mbps)**
Standard decoder	Half	4	15	6	417	366	5	403	378
Standard decoder	Half	8	204	16	1,139	998	5	358	2,865
Split bit error decoder	Half	8	204	16	1,196	1,060	5	336	2,686
Full bit error decoder	Half	8	204	16	1,181	1,065	5	328	2,624
Standard decoder	Half	8	255	32	2,027	1,685	5	319	2,011
Variable decoder	Half	8	204	16	1,273	1,082	5	359	2,871
Erasures decoder	Half	8	204	16	2,092	1,564	5	309	2,469
Erasures and variable decoder	Half	8	204	16	2,200	1,708	6	311	2,490
Standard encoder	—	8	204	16	204	210	—	621	4,969
Variable encoder	—	8	204	16	779	313	—	397	3,179
Variable encoder	—	8	204	32	1,650	581	—	365	2,923

Notes to Table 1–5:

(1) The number of bits per symbol (*m*).
(2) The number of symbols per codeword (*N*).
(3) The number of check symbols per codeword (*R*).

Table 1–6 shows the typical performance using the Quartus II software for Stratix IV (EP4SGX70DF29C2X) devices.

Table 1–6. Performance—Stratix IV Devices

Parameters					ALUTs	Logic Registers	Memory		f_{MAX} (MHz)	Throughput (Mbps)
Options	**Keysize**	**Bits** *(1)*	**Symbols** *(2)*	**Check** *(3)*			**ALUTs**	**M9K**		
Standard decoder	Half	4	15	6	426	382	8	3	413	387
Standard decoder	Half	8	204	16	1,220	1,034	64	3	368	2,945
Split bit error decoder	Half	8	204	16	1,273	1,092	64	3	340	2,719
Full bit error decoder	Half	8	204	16	1,255	1,092	64	3	325	2,603
Standard decoder	Half	8	255	32	2,100	1,713	64	3	324	2,038
Variable decoder	Half	8	204	16	1,362	1,119	64	3	356	2,850
Erasures decoder	Half	8	204	16	2,170	1,596	64	3	314	2,510
Erasures and variable decoder	Half	8	204	16	2,322	1,746	96	3	310	2,480
Standard encoder	—	8	204	16	204	210	—	—	620	4,960
Variable encoder	—	8	204	16	777	313	—	—	387	3,099
Variable encoder	—	8	204	32	1,651	582	—	—	347	2,775

Notes to Table 1–6:

(1) The number of bits per symbol (*m*).

(2) The number of symbols per codeword (*N*).

(3) The number of check symbols per codeword (*R*).

The throughput in megabits per second (Mbps) is derived from the formulas in Table 3–9 on page 3–10 and maximum frequency at which the design can operate.

Overall resource requirements vary widely depending on the parameter values used. The number of logic elements (LEs) or combinational ALUTs required to implement the function is linearly dependent on both the field size and the number of check symbols. More memory is required for 9, 10, 11, or 12 bits per symbol. Specifying the erasures-supporting and the variable option also increases the memory required.

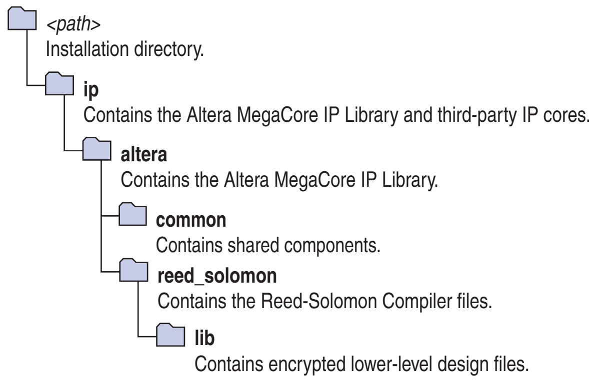
Installation and Licensing

The RS Compiler is part of the MegaCore® IP Library, which is distributed with the Quartus® II software and can be downloaded from the Altera® website, www.altera.com.

For system requirements and installation instructions, refer to the *Altera Software Installation and Licensing* manual.

Figure 1–1 shows the directory structure after you install the RS Compiler, where <*path*> is the installation directory for the Quartus II software. The default installation directory on Windows is **c:\altera**<*version*> and on Linux is **/opt/altera**<*version*>.

Figure 1–1. Directory Structure



OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP^SM megafunction) within your system.
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily.
- Generate time-limited device programming files for designs that include megafunctions.
- Program a device and verify your design in hardware.

You only need to purchase a license for the RS Compiler when you are completely satisfied with its functionality and performance, and want to take your design to production. After you purchase a license, you can request a license file from the Altera website at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

For more information about OpenCore Plus hardware evaluation, refer to *AN320: OpenCore Plus Evaluation of Megafunctions*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation supports the following operation modes:

- *Untethered*—the design runs for a limited time.
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time-out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior might be masked by the time-out behavior of the other megafunctions.

The untethered time-out for a RS Compiler MegaCore function is one hour; the tethered time-out value is indefinite.

Your design stops working after the hardware evaluation time expires and the data output `rsout` remains low.

2. Getting Started

Design Flows

The RS Compiler supports the following design flows:

- **DSP Builder**: Use this flow if you want to create a DSP Builder model that includes a RS Compiler variation.
- **MegaWizard™ Plug-In Manager**: Use this flow if you would like to create a RS Compiler variation that you can instantiate manually in your design.

This chapter describes how you can use a RS Compiler in either of these flows. The parameterization provides the same options in each flow and is described in "Parameterize the MegaCore Function" on page 2–3.

After parameterizing and simulating a design in either of these flows, you can compile the completed design in the Quartus II software.

DSP Builder Flow

Altera's DSP Builder product shortens digital signal processing (DSP) design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

DSP Builder integrates the algorithm development, simulation, and verification capabilities of The MathWorks MATLAB® and Simulink® system-level design tools with Altera Quartus II software and third-party synthesis and simulation tools. You can combine existing Simulink blocks with Altera DSP Builder blocks and MegaCore function variation blocks to verify system-level specifications and perform simulation.

In DSP Builder, a Simulink symbol for the MegaCore function appears in the MegaCore Functions library of the Altera DSP Builder Blockset in the Simulink library browser.

To use the RS Compiler in the MATLAB/Simulink environment, follow these steps:

1. Create a new Simulink model.
2. Select the `reed_solomon_`<*version*> block from the MegaCore Functions library in the Simulink Library Browser, add it to your model, and give the block a unique name.
3. Double-click on the `reed_solomon_`<*version*> block in your model to display the RS Compiler parameter editor and parameterize your RS Compiler variation. For an example of setting parameters for the RS Compiler, refer to "Parameterize the MegaCore Function" on page 2–3.
4. Click **Finish** in the parameter editor to complete the parameterization and generate your RS Compiler MegaCore function variation. For information about the generated files, refer to Table 2–1 on page 2–8.
5. Connect your RS Compiler MegaCore function variation to the other blocks in your model.

6. Simulate the MegaCore function variation in your DSP Builder model.

For more information about the DSP Builder flow, refer to the *Using MegaCore Functions* chapter in the *DSP Builder User Guide*.

When you are using the DSP Builder flow, device selection, simulation, Quartus II compilation and device programming are all controlled in the DSP Builder environment.

DSP Builder supports integration with SOPC Builder using Avalon® Memory-Mapped (Avalon-MM) master/slave and Avalon Streaming (Avalon-ST) source/sink interfaces.

For more information about the Avalon-MM and Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

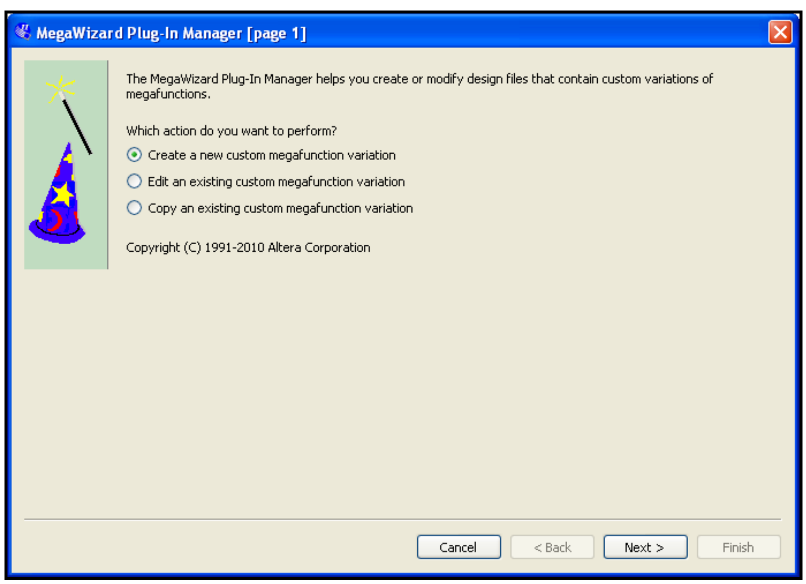
MegaWizard Plug-In Manager Flow

The MegaWizard Plug-in Manager flow allows you to customize a RS Compiler MegaCore function, and manually integrate the MegaCore function variation in a Quartus II design.

Follow the steps below to use the MegaWizard Plug-in Manager flow.

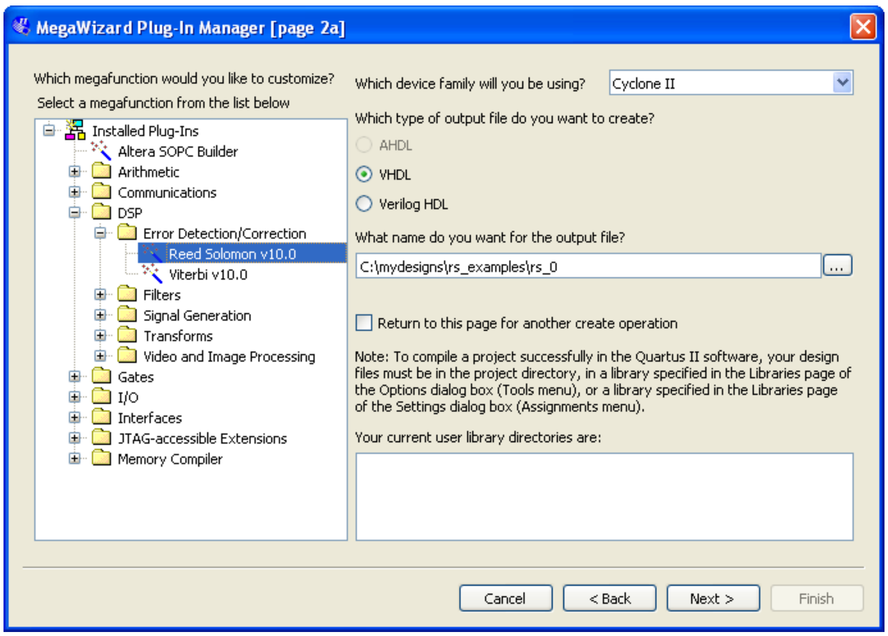
1. Create a new project using the **New Project Wizard** available from the File menu in the Quartus II software.
2. Launch **MegaWizard Plug-in Manager** from the Tools menu, and select the option to create a new custom megafunction variation (Figure 2–1).

Figure 2–1. MegaWizard Plug-In Manager



3. Click **Next** and select **Reed Solomon** <*version*> from the **DSP>Error Detection/Correction** section in the **Installed Plug-Ins** tab.
4. Verify that the device family is the same as you specified in the **New Project Wizard**.
5. Select the top-level output file type for your design; the wizard supports VHDL and Verilog HDL.
6. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files <*project path*>\<*variation name*>. Figure 2–2 shows the parameter editor after you specify these settings.

Figure 2–2. Select the Megafunction



7. Click **Next** to launch IP Toolbench.

Parameterize the MegaCore Function

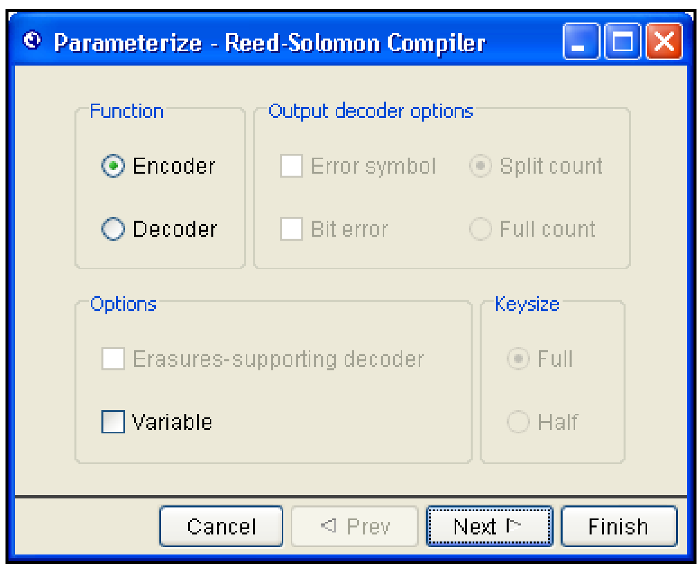
To parameterize your MegaCore function, follow these steps:

1. Click **Step 1: Parameterize** in IP Toolbench (Figure 2–3 on page 2–4).

Figure 2–3. IP Toolbench—Parameterize

2. Select **Encoder** or **Decoder** (Figure 2–4).

Figure 2–4. Select the Encoder or Decoder



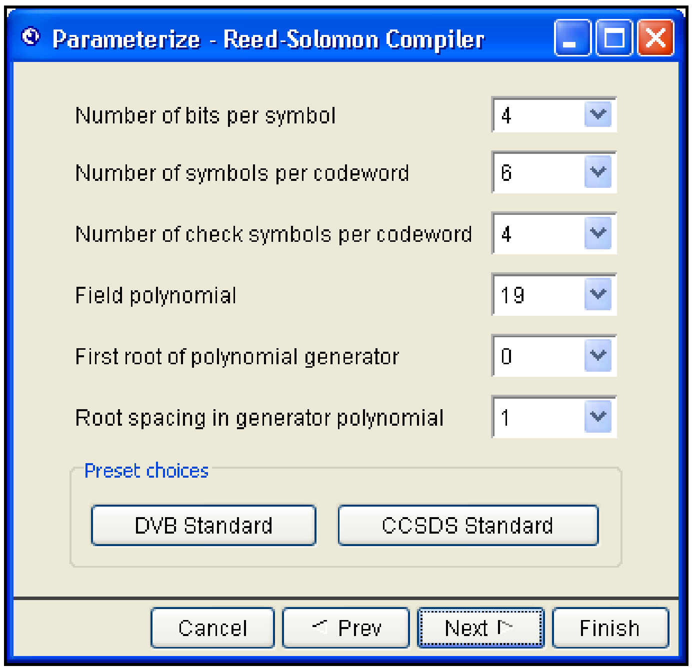
3. If you select **Encoder**, you can also turn on the **Variable** option.

   For more information about the variable option, refer to "Variable Encoding and Decoding" on page 3–3.

4. If **Decoder** is selected, the following controls are available:
   a. You can turn on the **Erasures-supporting decoder** or **Variable** options.
   b. You can select **Full** or **Half** keysize.
   c. You can turn on the **Error Symbol** or **Bit Error** outputs. For the bit error output, you can select **Split Count** or **Full Count**.

   For more information about these parameters, refer to Table 3–2 on page 3–7.
5. Click **Next**.
6. Select the parameters that define the specific RS codeword that you wish to implement (Figure 2–5).

Figure 2–5. Choose the Parameters

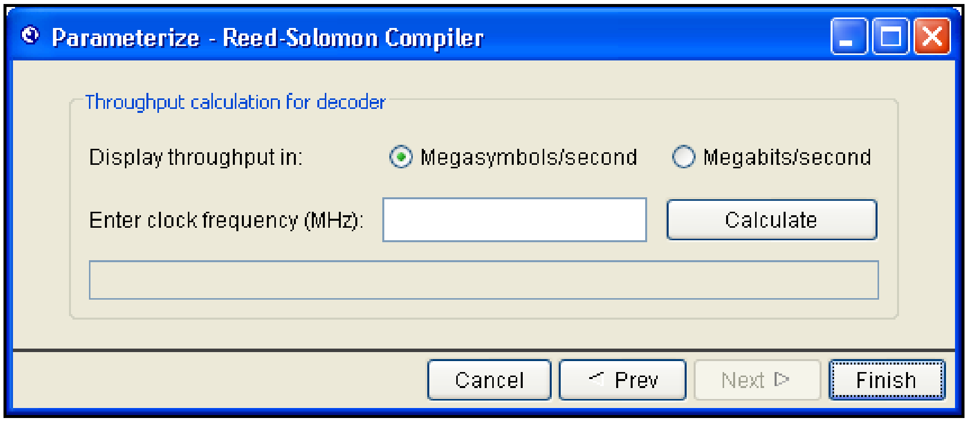


You can enter the parameters individually, or click **DVB Standard** to use digital video broadcast (DVB) standard values, or **CCSDS Standard** to use the CCSDS standard values.

For more information about these parameters, refer to Table 3–3 on page 3–8.

7. Click **Next**.
8. For a decoder throughput calculation, enter the frequency in MHz, select the desired units, and click **Calculate**. Figure 2–6 shows the decoder throughput calculation page.

Figure 2–6. Throughput Calculator



For more information about the throughput calculator, refer to "Throughput Calculator" on page 3–10.

9. Click **Finish**.

For more information about the RS Compiler parameters, refer to "Parameters" on page 3–7.

Set Up Simulation

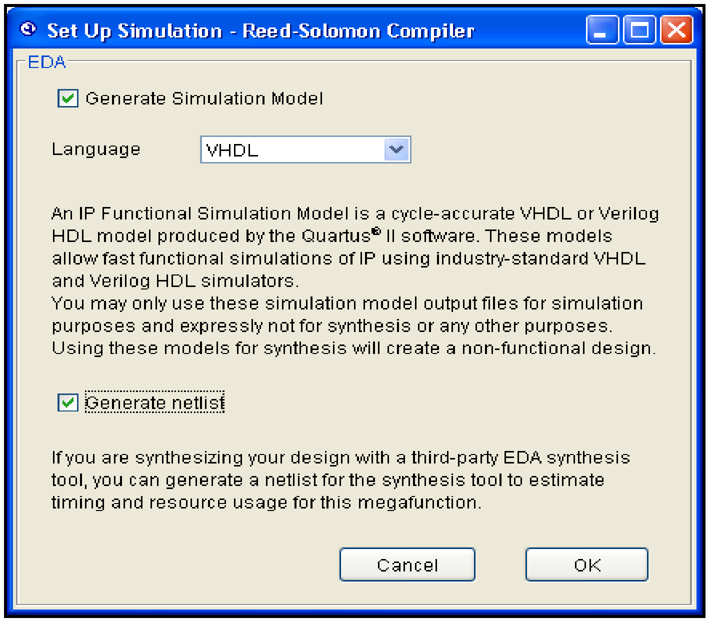
An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

To generate an IP functional simulation model for your MegaCore function, follow these steps:

1. Click **Step 2: Set Up Simulation** in IP Toolbench. Figure 2–3 on page 2–4 shows the IP Toolbench.
2. Turn on **Generate Simulation Model**, as shown in Figure 2–7.
3. Choose the required language in the **Language** list.
4. Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.
5. Click **OK**.

Figure 2–7. Generate Simulation Model



Generate the MegaCore Function

To generate your MegaCore function, follow these steps:

1. Click **Step 3:** Generate in IP Toolbench (Figure 2–3 on page 2–4).

   The generation phase may take several minutes to complete. The generation progress and status is displayed in a report window.

   Figure 2–8 shows the generation report.

Figure 2–8. Generation Report

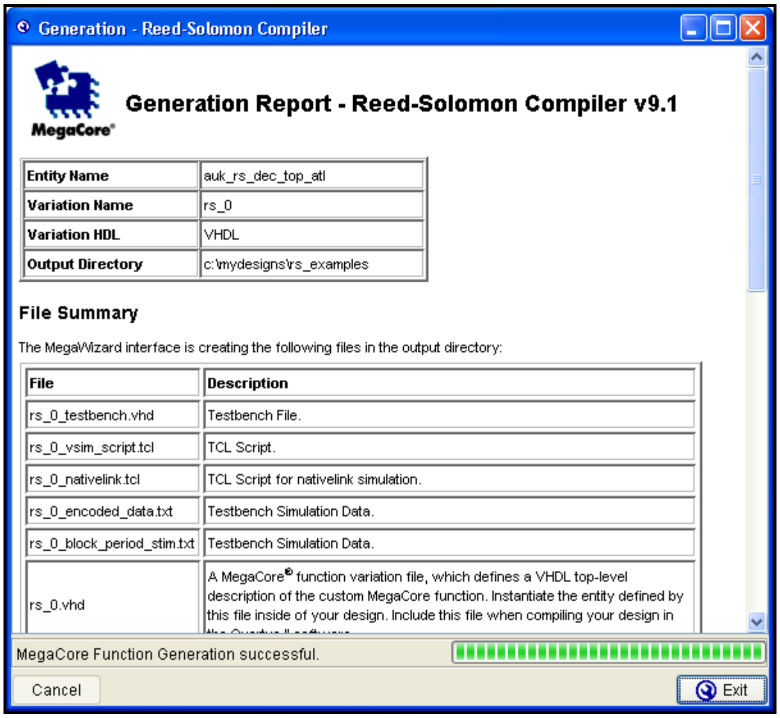


Table 2–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

Table 2–1. Generated Files (Part 1 of 2) *(Note 1)*

Filename	Description
<*variation name*>**.bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation name*>**.vo or .vho**	VHDL or Verilog HDL IP functional simulation model.
<*variation name*>**.vhd**, or **.v**	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<*variation name*>**.cmp**	A VHDL component declaration for the custom MegaCore function. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
<*variation name*>**_nativelink.tcl**	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools.
<*variation name*>**_syn.v** or **.vhd**	A timing and resource netlist for use in some third-party synthesis tools.
<*variation name*>**_testbench.vhd**	The testbench variation file, which defines the top-level testbench that runs the simulation. This file instantiates the function variation file and the testbench from the **reed_solomon\lib** directory.
<*variation name*>**_vsim_script.tcl**	Starts the MegaCore function simulation in the ModelSim simulator.
<*variation name*>**_block_period_stim.txt**	The testbench stimuli includes information such as number of codewords, number of symbols, and check symbols for each codeword

Table 2–1. Generated Files (Part 2 of 2) *(Note 1)*

Filename	Description
<*variation name*>**_encoded_data.txt**	Contains the encoded test data.
<*variation name*>**.html**	A MegaCore function report file in hypertext markup language format.
<*variation name*>**.qip**	A single Quartus II IP file is generated that contains all of the assignments and other information required to process your MegaCore function variation in the Quartus II compiler. You are prompted to add this file to the current Quartus II project when you exit the parameter editor.

Notes to Table 2–1:

(1) <*variation name*> is the variation name.

2. After you review the generation report, click **Exit** to close IP Toolbench. Then click **Yes** on the **Quartus II IP Files** prompt to add the **.qip** file describing your custom MegaCore function to the current Quartus II project.

Refer to the Quartus II Help for more information about the MegaWizard Plug-In Manager.

You can now integrate your custom variation into your design and simulate and compile.

Simulate the Design

IP Toolbench-generated Tcl scripts drive the simulation. For the decoder, the testbench includes a channel and the instantiated decoder. Data is read from an IP Toolbench-generated file. For the encoder, the testbench reads the same data file and just compares the encoder output with a data file. In the channel, some errors are introduced at various locations of the RS codeword. The testbench then receives the data decoded by the RS decoder and compares it with the originally transmitted data.

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.

For more information about NativeLink, refer to the *Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook*.

You can use the Tcl script file <*variation name*>**_nativelink.tcl** to assign default NativeLink testbench settings to the Quartus II project.

To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation but ensure you specify your variation name to match the Quartus II project name.
2. Check that the absolute path to your third-party simulator executable is set. On the Tools menu click **Options** and select **EDA Tools Options**.
3. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
4. On the Tools menu click **Tcl scripts**. Select the <*variation name*>**_nativelink.tcl** Tcl script and click **Run**. Check for a message confirming that the Tcl script was successfully loaded.

5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name**.
6. On the Tools menu point to **EDA Simulation Tool** and click **EDA RTL Simulation**.

Compile the Design

You can use the Quartus II software to compile your design. Refer to Quartus II Help for instructions on performing compilation.

Program a Device

After you have compiled your design, program your targeted Altera device and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate an RS MegaCore function before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model and produce a time-limited programming file.

For more information about IP functional simulation models, refer to the *Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook*.

You can simulate an RS MegaCore function in your design and perform a time-limited evaluation of your design in hardware.

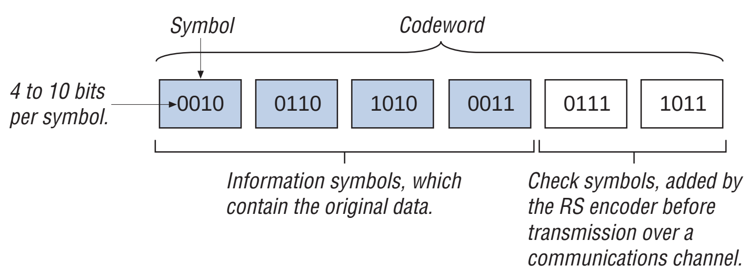
Fore more information about OpenCore Plus hardware evaluation using the RS Compiler, refer to "OpenCore Plus Evaluation" on page 1–6.

3. Functional Description

Background

To use Reed-Solomon (RS) codes, a data stream is first broken into a series of codewords. Each codeword consists of several information symbols followed by several check symbols (also known as parity symbols or redundant symbols). Symbols can contain an arbitrary number of bits. In an error correction system, the encoder adds check symbols to the data stream prior to its transmission over a communications channel. When the data is received, the decoder checks for and corrects any errors (Figure 3–1).

Figure 3–1. RS Codeword Example



RS codes are described as (N,K), where N is the total number of symbols per codeword and K is the number of information symbols. R is the number of check symbols ($N - K$). Errors are defined on a symbol basis. Any number of bit errors within a symbol is considered as only one error.

RS codes are based on finite-field (i.e., Galois field) arithmetic. Any arithmetic operation (addition, subtraction, multiplication, and division) on a field element gives a result that is an element of the field. The size of the Galois field is determined by the number of bits per symbol—specifically, the field has 2^m elements, where m is the number of bits per symbol. A specific Galois field is defined by a polynomial, which is user-defined for the RS Compiler. IP Toolbench lets you select only valid field polynomials.

The maximum number of symbols in a codeword is limited by the size of the finite field to $2^m - 1$. For example, a code based on 10-bit symbols can have up to 1,023 symbols per codeword. The RS Compiler supports shortened codewords.

The following equation represents the generator polynomial of the code:

$$g(x) = \prod_{i=0}^{R-1} (x - \alpha^{a.i + i_0})$$

where:

$i0$ is the first root of the generator polynomial
a is the rootspace
R is the number of check symbols
α is a root of the polynomial.

For example, for the following information:

$$g(x) = \prod_{i=0}^{3} (x - \alpha^{i + i_0})$$

a is a root of the binary primitive polynomial $x^8 + x^7 + x^2 + x + 1$
$i0 = 120$

You can calculate the following parameters:

- $R - 1 = 3$
- $a = 1$ (α is to the power 1 times i)

The field polynomial can be obtained by replacing x with 2, thus:
$2^8 + 2^7 + 2^2 + 2 + 1 = 391$

Erasures

In normal operation, the RS decoder detects and corrects symbol errors.

The number of symbol errors that can be corrected, C, depends on the number of check symbols, R and is given by $C \leq R/2$.

If the location of the symbol errors is marked as an erasure, the RS decoder can correct twice as many errors, so $C \leq R$.

Erasures are symbol errors with a known location.

External circuitry identifies which symbols have errors and passes this information to the decoder using the `eras_sym` signal. The `eras_sym` input indicates an erasure (when the erasures-supporting decoder option is selected).

The RS decoder can work with a mixture of erasures and errors.

A codeword is correctly decoded if $(2e + E) \leq R$

where:

e = errors with unknown locations
E = erasures
R = number of check symbols.

For example, with ten check symbols the decoder can correct ten erasures, or five symbol errors, or four erasures and three symbol errors.

If the number of erasures marked approaches the number of check symbols, the ability to detect errors without correction (`decfail` asserted) diminishes. Refer to Table 3–1 on page 3–4.

Shortened Codewords

A shortened codeword contains fewer symbols than the maximum value of N, which is $2^m - 1$. A shortened codeword is mathematically equivalent to a maximum-length code with the extra data symbols at the start of the codeword set to 0.

For example, (204,188) is a shortened codeword of (255,239). Both of these codewords use the same number of check symbols, 16.

To use shortened codewords with the Altera RS encoder and decoder, you use IP Toolbench to set the codeword length to the correct value, in the example, 204.

Variable Encoding and Decoding

Under normal circumstances, the encoder and decoder allow variable encoding and decoding—you can change the number of symbols per codeword (*N*) using `sink_eop`, but not the number of check symbols while decoding.

However, you cannot change the length of the codeword, if you turn on the erasure-supporting option.

If you turn on the variable option, you can vary the number of symbols per codeword (using the `numn` signal) and the number of check symbols (using the `numcheck` signal), in real time, from their minimum allowable values up to their selected values, even with the erasures-supporting option turned on. Table 3–7 on page 3–10 shows the variable option signals.

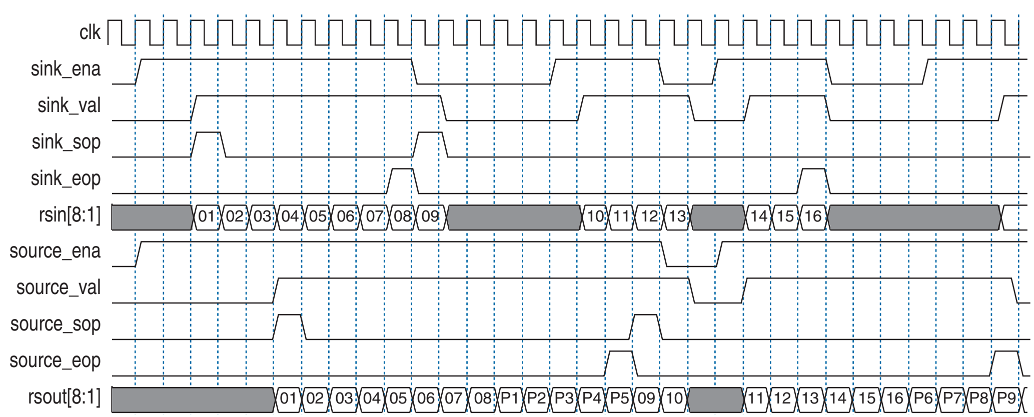
RS Encoder

The `sink_sop` signal starts a codeword; `sink_eop` signals its termination. An asserted `sink_val` indicates valid data. The `sink_sop` is only valid when `sink_val` is asserted.

Only assert `sink_val` one clock cycle after the encoder asserts `sink_ena`.

By de-asserting `sink_ena`, the encoder signals that it cannot sink more incoming symbols after `sink_eop` is signalled at the input. During this time it is generating the check symbols for the current codeword. Figure 3–2 shows the operation of the RS encoder. The example shows a codeword with eight information symbols and five check symbols.

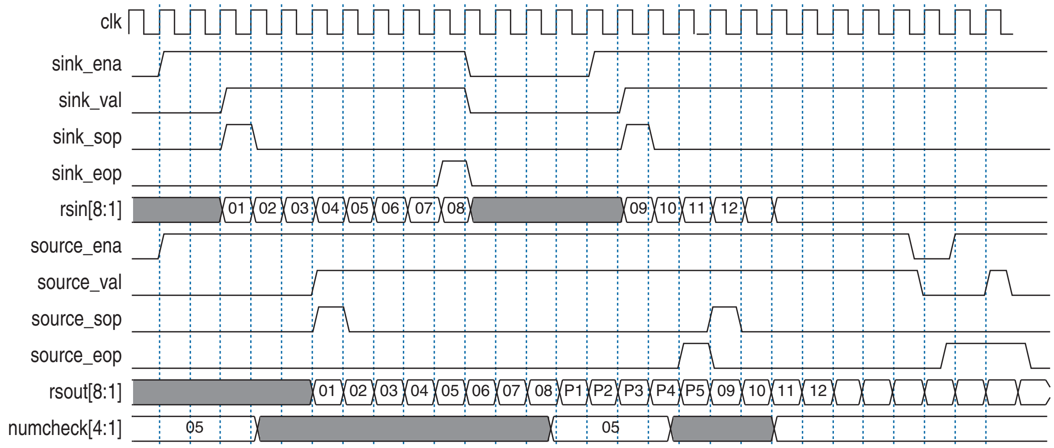
Figure 3–2. Encoder Timing



The `numcheck` input is latched inside the encoder when `sink_sop` is asserted.

You can change the number of symbols in a codeword at run-time without resetting the encoder. You must make the changes between complete codewords; you cannot change `numcheck` during encoding. Figure 3–3 shows variable encoding.

Figure 3–3. Variable Encoding



RS Decoder

The decoder implements an Avalon-ST-based pipelined three-codeword-depth architecture. However, if the parameters are in the continuous range (refer to Table 3–3 on page 3–8), the decoder shows continuous behavior and can accept a new symbol every clock cycle.

The decoder is self-flushing—it processes and delivers a codeword without needing a new codeword to be fed in. Therefore, latency between the input and output does not depend on the availability of input data. The throughput latency is approximately three codewords

The reset is active high and can be asserted asynchronously. However, it has to be de-asserted synchronously with `clk`.

The RS decoder always tries to detect and correct errors in the codeword. However, as the number of errors increases, the decoder gets to a stage where it can no longer correct but only detect errors, at which point the decoder asserts the `decfail` signal. As the number of errors increases still further, the results become unpredictable. Table 3–1 shows how the decoder corrects and detects errors depending on *R*.

Table 3–1. Decoder Detection and Correction

Number of Errors	Decoder Behavior
Errors ≤ *R*/2	Decoder detects and corrects errors.
R/2 ≤ errors ≤ *R*	Decoder asserts `decfail` and can only detect errors. *(1)*
Errors > *R*	Unpredictable results.

Note to Table 3–1:

(1) The decoder may fail to assert `decfail`, for low values of *R* (4,5, or 6), or when using erasures and the differences between the number of erasures and *R* is small (4, 5 or 6).

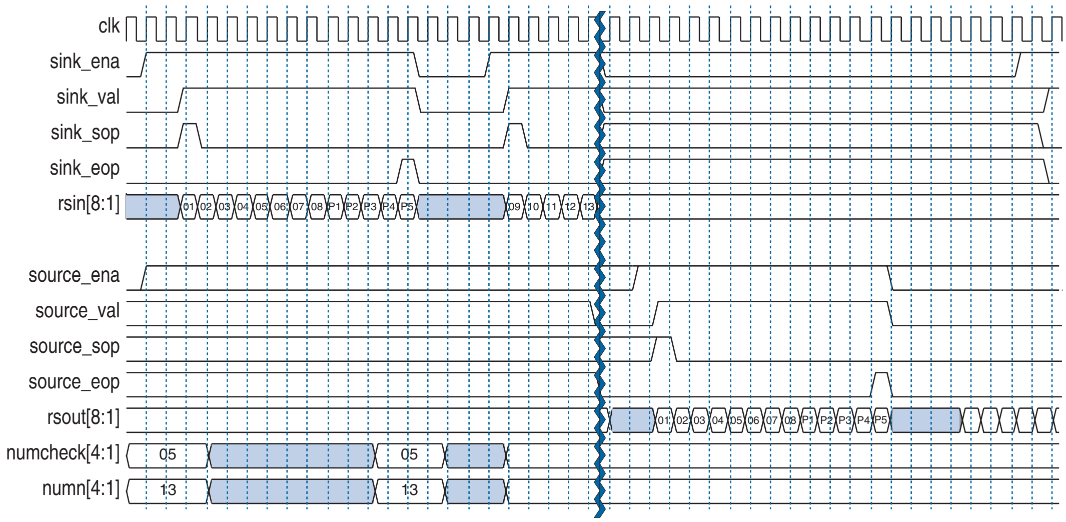
The RS decoder observes Avalon-ST interface standard for input and output data. One clock cycle after the decoder asserts `sink_ena`, you can assert `sink_val`. The decoder accepts the data at `rsin` as valid data. The codeword is started with `sink_sop`. The `numcheck` and `numn` signals are latched to `sink_sop`.

The codeword is finished when `sink_eop` is asserted. If `sink_ena` is de-asserted, from one clock cycle onwards the decoder cannot process any more data until `sink_ena` is asserted again.

At the output the operation is identical. If you assert `source_ena`, the decoder asserts `source_val` and provides valid data on `rsout` if available. Also, it indicates the start and end of the codeword with `source_sop` and `source_eop` respectively.

Figure 3–4 shows the operation of the RS decoder.

Figure 3–4. Decoder Timing



The decoder has the following optional outputs, which you turn on in IP Toolbench:

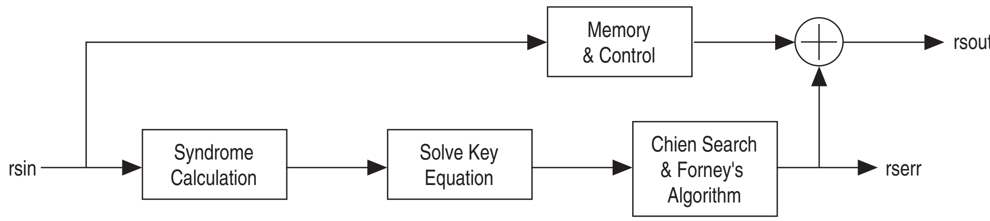
- Error symbol
- Bit error count

Error Symbol Output

The error symbol output, `rserr` is the Galois field error correction value. The RS decoder finds the error values and location and adds these values in the Galois field to the input value. Galois field addition and subtraction is the same operation. An `XOR` operation performs this operation between bits of the two values.

Figure 3–5 on page 3–6 shows the error symbol output.

Figure 3–5. Error Symbol Output



Whenever `rserr` is not 0 (while `decfail` is 0), an error correction successfully takes place. The `rsout` is the `rserr` `XOR`ed with the corresponding `rsin`, where `XOR` is done for each bit, so you know that the respective symbol has been corrected. The value of `rserr` shows which bits of the symbol have been corrected. For each bit of `rserr` that is 1, the corresponding bit of `rsout` is corrected.

The `rsout` and the corresponding `rserr` value appear at the output at the same clock cycle.

Bit Error Count

The decoder can provide the bit error count found in the correction process. The bit error count has the following options:

- Full count. The output `num_err_bit` is connected, which shows the valid value.
- Split count. The outputs `num_err_bit0` and `num_err_bit1` are connected, which show the valid values

For information about these outputs, refer to Table 3–8 on page 3–10.

Interfaces

The RS encoder and decoder use the Avalon® Streaming (Avalon-ST) interface for data input and output. The input is an Avalon-ST sink and the output is an Avalon-ST source. The Avalon-ST interface `READY_LATENCY` parameter is set to 1. The Avalon-ST interfaces allow for flow control.

The Avalon-ST interface is an evolution of the Atlantic™ interface. The Avalon-ST interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals.

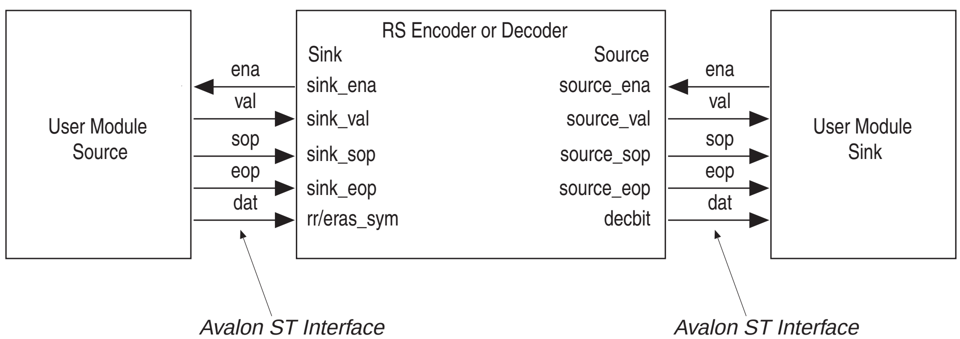
The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism, where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output. When designing a datapath, which includes the RS MegaCore function, you may not need backpressure if you know the downstream components can always receive data. You may achieve a higher clock rate by driving the source ready signal `source_ena` of the RS high, and not connecting the sink ready signal `sink_ena`.

For more information about the Avalon-ST interface, refer to the *Avalon Interface Specifications*.

Figure 3–6 shows the RS encoder and decoder Avalon-ST interfaces.

Figure 3–6. Avalon ST Interface



Parameters

Table 3–2 shows the implementation parameters.

Table 3–2. Implementation Parameters

Parameter	Value	Description
Function	Encoder or Decoder	Specifies an encoder or a decoder. Refer to "Functional Description" on page 3–1.
Variable	On or Off	Specifies the variable option. Refer to "Variable Encoding and Decoding" on page 3–3.
Erasures-supporting decoder *(1)*	On or Off	Specifies the erasures-supporting decoder option. This option substantially increases the logic resources used. Refer to "Erasures" on page 3–2.
Error symbol *(1)*	On or Off	Specifies the error symbol output. Refer to "RS Decoder" on page 3–4 and Table 3–8 on page 3–10.
Bit error *(1)*	On or Off	You can set the bit error output to be either **Split count** or **Full count**. Refer to "RS Decoder" on page 3–4 and Table 3–8 on page 3–10.
Keysize *(1)*	Half or Full.	The keysize parameter allows you to trade off the amount of logic resources against the supported throughput. Full has twice as many Galois field multipliers as half. A full decoder uses more logic and is probably slightly slower in frequency, but supports a higher throughput. If both full and half give you the required throughput for your parameters, always select half.

Note to Table 3–2:

(1) This parameter applies to the decoder only.

Table 3–3 shows the RS codeword parameters.

Table 3–3. RS Codeword Parameters

Parameter	Range	Range (Continuous)	Description
Number of bits per symbol	3 to 12	6 to 12	Specifies the number of bits per symbol (m).
Number of symbols per codeword	5 to $(2^m - 1)$	$7(R + 1)$ to $2^m - 1$	Specifies the total number of symbols per codeword (N).
Number of check symbols per codeword	2 to min(128, $N - 1$)	4 to $N/7 - 1$	Specifies the number of check symbols per codeword (R)
Field polynomial	Any valid polynomial *(1)*		Specifies the primitive polynomial defining the Galois field.
First root of generator polynomial	0 to $(2^m - 2)$		Specifies the first root of the generator polynomial (i_0).
Root spacing in generator polynomial	Any valid root space *(1)*		Specifies the minimum distance between roots in the generator polynomial (a).

Notes to Table 3–3:

(1) IP Toolbench allows you to select only legal values. For $m > 8$, not all legal values of the field polynomials and rootspace are present in IP Toolbench. If you cannot find your intended field polynomial or rootspace in the IP Toolbench list, contact Altera MySupport.

Signals

Table 3–4 shows the global signals.

Table 3–4. Global Signals

Name	Description
`clk`	`clk` is the main system clock. The whole MegaCore function operates on the rising edge of `clk`.
`reset`	Reset. The entire decoder is asynchronously reset when reset is asserted high. The `reset` signal resets the entire system. The `reset` signal must be de-asserted synchronously with respect to the rising edge of `clk`.

Table 3–5 shows the Avalon-ST sink (data input) interface.

Table 3–5. Avalon-ST Sink Interface (Part 1 of 2)

Name	Avalon-ST Type	Direction	Description
`sink_ena`	`ena`	Output	Data transfer enable signal. `sink_ena` is driven by the sink interface and controls the flow of data across the interface. `sink_ena` behaves as a read enable from sink to source. When the source observes `sink_ena` asserted on the `clk` rising edge it drives, on the following `clk` rising edge, the Avalon-ST data interface signals and asserts `val`, if data is available. The sink interface captures the data interface signals on the following `clk` rising edge. If the source is unable to provide new data, it de-asserts `val` for one or more clock cycles until it is prepared to drive valid data interface signals.

Table 3–5. Avalon-ST Sink Interface (Part 2 of 2)

Name	Avalon-ST Type	Direction	Description
`sink_val`	`val`	Input	Data valid signal. `sink_val` indicates the validity of the data signals. `sink_val` is updated on every clock edge where `sink_ena` is asserted. `sink_val` and the `dat` bus hold their current value if `sink_ena` is de-asserted. When `sink_val` is asserted, the Avalon-ST data interface signals are valid. When `sink_val` is de-asserted, the Avalon-ST data interface signals are invalid and must be disregarded. To determine whether new data has been received, the sink interface qualifies the `sink_val` signal with the previous state of the `sink_ena` signal.
`sink_sop`	`sop`	Input	Start of packet (codeword) signal. `sop` delineates the codeword boundaries on the `rsin` bus. When sink_`sop` is high, the start of the packet is present on the `rsin` bus. sink_`sop` is asserted on the first transfer of every codeword.
`sink_eop`	`eop`	Input	End of packet (codeword) signal. sink_`eop` delineates the packet boundaries on the `rsin` bus. When sink_`eop` is high, the end of the packet is present on the `dat` bus. sink_`eop` is asserted on the last transfer of every packet.
`rsin[m:1]`	`data`	Input	Data input for each codeword, symbol by symbol. Valid only when `sink_val` is asserted.
`eras_sym`	`data`	Input	When asserted, the symbol in `rsin[]` is marked as an erasure. Valid only for the decoder with **Erasures-supporting decoder** option.

Table 3–6 shows the Avalon-ST source (data output) interface.

Table 3–6. Avalon-ST Source Interface

Name	Avalon-ST Type	Direction	Description
`source_ena`	`ena`	Input	Data transfer enable signal. `source_ena` is driven by the sink interface and controls the flow of data across the interface. `ena` behaves as a read enable from sink to source. When the source interface observes `source_ena` asserted on the `clk` rising edge it drives, on the following `clk` rising edge, the Avalon-ST data interface signals and asserts `source_val` when data from sink interface is available. The sink interface captures the data interface signals on the following `clk` rising edge. If this source is unable to provide new data, it de-asserts `source_val` for one or more clock cycles until it is prepared to drive valid data interface signals.
`source_val`	`val`	Output	Data valid signal. source_val is asserted high, whenever there is a valid output on `rsout`; it is de-asserted when there is no valid output on `rsout`.
`source_sop`	`sop`	Output	Start of packet (codeword) signal.
`source_eop`	`eop`	Output	End of packet (codeword) signal.
`rsout`	`data`	Output	The `rsout` signal contains decoded output when source_val is asserted. The corrected symbols are in the same order that they were entered.
`rserr`	`data`	Output	Error correction value (decoder only, optional). Refer to “Error Symbol Output” on page 3–5.

Table 3–7 shows the configuration signals.

Table 3–7. Configuration Signals

Name	Description
bypass	A one-bit signal that sets if the codewords are bypassed or not (decoder only). The decoder continuously samples bypass.
numcheck	Sets the variable number of check symbols up to a maximum value set by the parameter *R* (variable option only). The decoder samples numcheck only when sink_sop is asserted.
numn	Variable value of *N*. Can be any value from the minimum allowable value of *N* up to the selected value of *N* (variable and erasures-supporting option only). The decoder samples numn only when sink_sop is asserted.

Table 3–8 shows the status signals (decoder only).

Table 3–8. Status Signals

Name	Description
decfail	Indicates non-correctable codeword. Valid when source_sop is asserted. Avalon-ST type err.
num_err_sym	Number of symbols errors. Valid when source_sop is asserted; invalid when decfail is asserted.
num_err_bit	Number of bits errors corrected in the codeword. Valid when source_sop is asserted; invalid when decfail is asserted. Connected only when the **Bit error** (**Full count**) option is turned on. Refer to "RS Decoder" on page 3–4.
num_err_bit0	Number of bit errors for the corrections from bit 1 to bit 0. The latest is the correct bit. Valid when sop_source is asserted; invalid when decfail is asserted. The decoder presents these values at the next source_sop assertion (at the next codeword). Connected only when the **Bit error** (**Split count**) option is turned on.
num_err_bit1	Number of bit errors for the corrections from bit 0 to bit 1. The latest is the correct bit. Valid when sop_source is asserted; invalid when decfail is asserted. The decoder presents these values at the next source_sop assertion (at the next codeword). Connected only when the **Bit error** (**Split count**) option is turned on.

Throughput Calculator

The IP Toolbench throughput calculator (decoder only) uses the following equation:

$$\text{Throughput in megasymbols per second} = N \times \text{frequency (MHz)}/N_C$$

For Mbps, multiply by m, the number of bits per symbol.

Table 3–9 shows the value of N_C.

Table 3–9. Calculate N_C

Erasures	Keysize	N_C
No	Half	Max (N, $10 \times R + 4$)
No	Full	Max (N, $7 \times R + 5$)
Yes	Half	Max (N, $10 \times R + 6$)
Yes	Full	Max (N, $8 \times R + 4$)

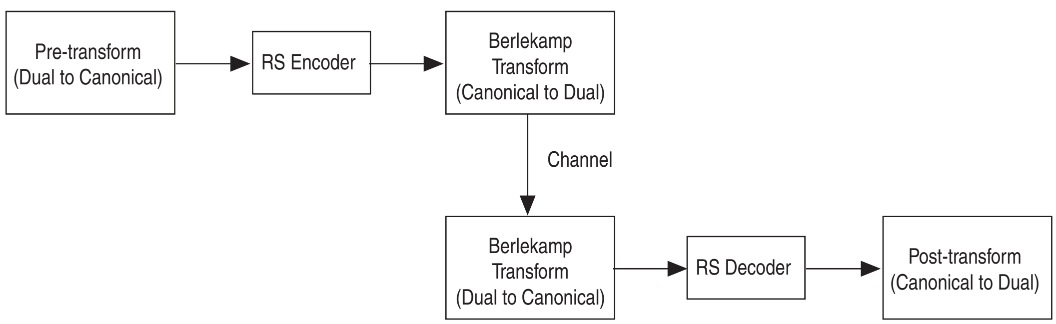
A. Using the RS Encoder or Decoder in a CCSDS System

Introduction

The Reed-Solomon (RS) encoder or decoder MegaCore functions work in canonical base (otherwise known as conventional base). This base can cause confusion when trying to implement the RS encoder or decoder directly into a dual-base system, for example, when working with the Consultative Committee for Space Data Systems (CCSDS) standard.

To transfer from a canonical-base to a dual-base system, a Berlekamp transform is used, which you need to implement in logic. Figure A–1 shows an example use of the Berlekamp transform.

Figure A–1. Using the Berlekamp Transform



Test Patterns

If you are working with a dual-base system, for example, CCSDS, and wish to supply the RS encoder or decoder with some test patterns from the dual-base system, follow these steps:

1. Apply the Berlekamp transform (dual to canonical) to the test pattern.
2. Apply the test pattern to RS encoder or decoder.
3. Apply the Berlekamp transform (canonical to dual) to the encoder output.
4. Check the test pattern.

For more information about implementing the transformation function, refer to *Annex B* of the standard specification document *CCSDS-101.0-B-5* at **www.ccsds.org**.

Revision History

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

Date	Version	Changes Made
May 2011	11.0	■ Updated support level to final support for Arria® II GX, Arria II GZ, Cyclone® III LS, and Cyclone IV GX devices. ■ Updated support level to HardCopy Compilation for HardCopy III, HardCopy IV E, and HardCopy IV GX devices.
December 2010	10.1	■ Added preliminary support for Arria II GZ devices. ■ Updated support level to final support for Stratix® IV GT devices.
July 2010	10.0	■ Added prelminary support for Stratix V devices
November 2009	9.1	■ Maintenance update ■ Reorganized to clarify two design flows. ■ Added preliminary support for Cyclone III LS, Cyclone IV, and HardCopy IV GX devices
March 2009	9.0	Added Arria II GX device support
November 2008	8.1	No changes
May 2008	8.0	Added device support for Stratix IV devices
October 2007	7.2	No changes
May 2007	7.1	Updated `rserr` signal
December 2006	7.0	Added support for Cyclone III devices
December 2006	6.1	Updated format

How to Contact Altera

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

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Technical documentation	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example: *AN 519: Stratix IV Design Guidelines.*
Italic type	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input.` Active-low signals are denoted by suffix `n`. Example: `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	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 your work.
WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
↵	The angled arrow instructs you to press the enter key.
	The feet direct you to more information about a particular topic.