Reed-Solomon Compiler

User Guide

CAUTION The Reed-Solomon IP Core is scheduled for product obsolescence and discontinued support as described in PDN1410. Therefore, Altera does not recommend use of this IP in new designs. For more information about Altera's current IP offering, refer to Altera's Intellectual Property website.

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

UG-RSCOMPILER-14.1

Feedback Subscribe

Contents

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
- 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	14.1
Release Date	December 2014
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*.

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* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release.

Device Family Support

Altera offers the following device support levels for Altera IP cores:

- Preliminary support—Altera verifies the IP core with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. You can use it in production designs with caution.
- Final support—Altera verifies the IP core 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.

Table 1–2 shows the level of support offered by the RS Compiler to each of the Altera device families.

Table 1–2. Device Family Support (Part 1 of 2)

Device Family	Support
Arria® II GX	Final
Arria II GZ	Final
Arria V	final
Cyclone® IV GX	Final
Stratix® IV GT	Final
Stratix IV GX/E	Final
Stratix V	Final

Table 1–2. Device Family Support (Part 2 of 2)

Device Family	Support
Stratix GX	Final
Other device families	No support

Performance and Resource Utilization

Table 1–3 shows the typical performance using the Quartus II software for Stratix IV (EP4SGX70DF29C2X) devices.

Table 1–3. 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 1–3:

(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 3–11 and maximum frequency at which the design can operate.

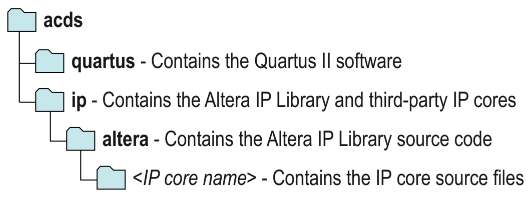
2. Getting Started

Installing and Licensing IP Cores

The Quartus II software includes the Altera IP Library. The library provides many useful IP core functions for production use without additional license. You can fully evaluate any licensed Altera IP core in simulation and in hardware until you are satisfied with its functionality and performance.

Some Altera IP cores, such as MegaCore® functions, require that you purchase a separate license for production use. After you purchase a license, visit the Self Service Licensing Center to obtain a license number for any Altera product. For additional information, refer to *Altera Software Installation and Licensing*.

Figure 2–1. IP core Installation Path


acds
quartus - Contains the Quartus II software
ip - Contains the Altera IP Library and third-party IP cores
altera - Contains the Altera IP Library source code
<IP core name> - Contains the IP core source files

☞ The default installation directory on Windows is **<*drive*>:\altera\<*version number*>**; on Linux it is **<*home directory*>/altera/<*version number*>**.

OpenCore Plus IP Evaluation

Altera's free OpenCore Plus feature allows you to evaluate licensed MegaCore IP cores in simulation and hardware before purchase. You need only purchase a license for MegaCore IP cores if you decide to take your design to production. OpenCore Plus supports the following evaluations:

- Simulate the behavior of a licensed IP core in your system.
- Verify the functionality, size, and speed of the IP core quickly and easily.
- Generate time-limited device programming files for designs that include IP cores.
- Program a device with your IP core and verify your design in hardware.

OpenCore Plus evaluation supports the following two operation modes:

- Untethered—run the design containing the licensed IP for a limited time.
- Tethered—run the design containing the licensed IP for a longer time or indefinitely. This requires a connection between your board and the host computer.

All IP cores using OpenCore Plus in a design time out simultaneously when any IP core times out.

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.

Specifying IP Core Parameters and Options

The parameter editor GUI allows you to quickly configure your custom IP variation. Use the following steps to specify IP core options and parameters in the Quartus II software. Refer to Specifying IP Core Parameters and Options (Legacy Parameter Editors) for configuration of IP cores using the legacy parameter editor.

1. In the IP Catalog (**Tools > IP Catalog**), locate and double-click the name of the IP core to customize. The parameter editor appears.
2. Specify a top-level name for your custom IP variation. The parameter editor saves the IP variation settings in a file named .<*your_ip*>**qsys**. Click **OK**.
3. Specify the parameters and options for your IP variation in the parameter editor, including one or more of the following. Refer to your IP core user guide for information about specific IP core parameters.
   - Optionally select preset parameter values if provided for your IP core. Presets specify initial parameter values for specific applications.
   - Specify parameters defining the IP core functionality, port configurations, and device-specific features.
   - Specify options for processing the IP core files in other EDA tools.
4. Click **Generate HDL**, the **Generation** dialog box appears.
5. Specify output file generation options, and then click **Generate**. The IP variation files generate according to your specifications.
6. To generate a simulation testbench, click **Generate > Generate Testbench System**.
7. To generate an HDL instantiation template that you can copy and paste into your text editor, click **Generate > HDL Example**.
8. Click **Finish**. The parameter editor adds the top-level **.qsys** file to the current project automatically. If you are prompted to manually add the **.qsys** file to the project, click **Project > Add/Remove Files in Project** to add the file.

9. After generating and instantiating your IP variation, make appropriate pin assignments to connect ports.

For information about using a legacy parameter editor, refer to "Specifying IP Core Parameters and Options (Legacy Parameter Editors)" in the *Introduction to Altera IP Cores*.

Parameterizing the Reed Solomon MegaCore Function

To parameterize your MegaCore function, follow these steps:

1. Select **Encoder** or **Decoder**.
2. 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.
3. 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.
4. Click **Next**.
5. Select the parameters that define the specific RS codeword that you wish to implement (Figure 2–5).

   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.
6. Click **Next**.
7. 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.

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

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

Generated Files (For Arria V, Cyclone V, MAX 10, and Stratix V Devices)

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.

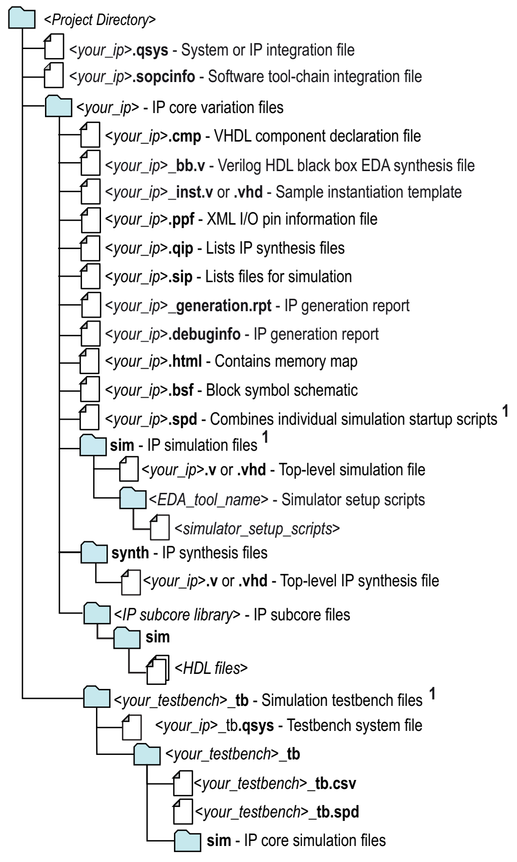
Table 2–1. Generated Files (1)

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
<*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.

Files Generated for Altera IP Cores

The Quartus II software version 14.0 Arria 10 Edition and later generates the following output file structure for Altera IP cores:

Figure 2–2. IP Core Generated Files


<Project Directory>
<your_ip>.qsys - System or IP integration file
<your_ip>.sopcinfo - Software tool-chain integration file
<your_ip> - IP core variation files
<your_ip>.cmp - VHDL component declaration file
<your_ip>_bb.v - Verilog HDL black box EDA synthesis file
<your_ip>_inst.v or .vhd - Sample instantiation template
<your_ip>.ppf - XML I/O pin information file
<your_ip>.qip - Lists IP synthesis files
<your_ip>.sip - Lists files for simulation
<your_ip>_generation.rpt - IP generation report
<your_ip>.debuginfo - IP generation report
<your_ip>.html - Contains memory map
<your_ip>.bsf - Block symbol schematic
<your_ip>.spd - Combines individual simulation startup scripts 1
sim - IP simulation files 1
<your_ip>.v or .vhd - Top-level simulation file
<EDA_tool_name> - Simulator setup scripts
<simulator_setup_scripts>
synth - IP synthesis files
<your_ip>.v or .vhd - Top-level IP synthesis file
<IP subcore library> - IP subcore files
sim
<HDL files>
<your_testbench>_tb - Simulation testbench files 1
<your_ip>_tb.qsys - Testbench system file
<your_testbench>_tb
<your_testbench>_tb.csv
<your_testbench>_tb.spd
sim - IP core simulation files

1. If supported and enabled for your IP variation

Simulating IP Cores

The Quartus II software supports RTL- and gate-level design simulation of Altera IP cores in supported EDA simulators. Simulation involves setting up your simulator working environment, compiling simulation model libraries, and running your simulation.

You can use the functional simulation model and the testbench or example design generated with your IP core for simulation. The functional simulation model and testbench files are generated in a project subdirectory. This directory may also include scripts to compile and run the testbench. For a complete list of models or libraries required to simulate your IP core, refer to the scripts generated with the testbench. You can use the Quartus II NativeLink feature to automatically generate simulation files and scripts. NativeLink launches your preferred simulator from within the Quartus II software.

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

Adding IP Cores to IP Catalog

The IP Catalog automatically displays Altera IP cores found in the project directory, in the Altera installation directory, and in the defined IP search path. The IP Catalog can include Altera-provided IP components, third-party IP components, custom IP components that you provide, and previously generated Qsys systems.

You can use the IP Search Path option (**Tools > Options**) to include custom and third-party IP components in the IP Catalog. The IP Catalog displays all IP cores in the IP search path. The Quartus II software searches the directories listed in the IP search path for the following IP core files:

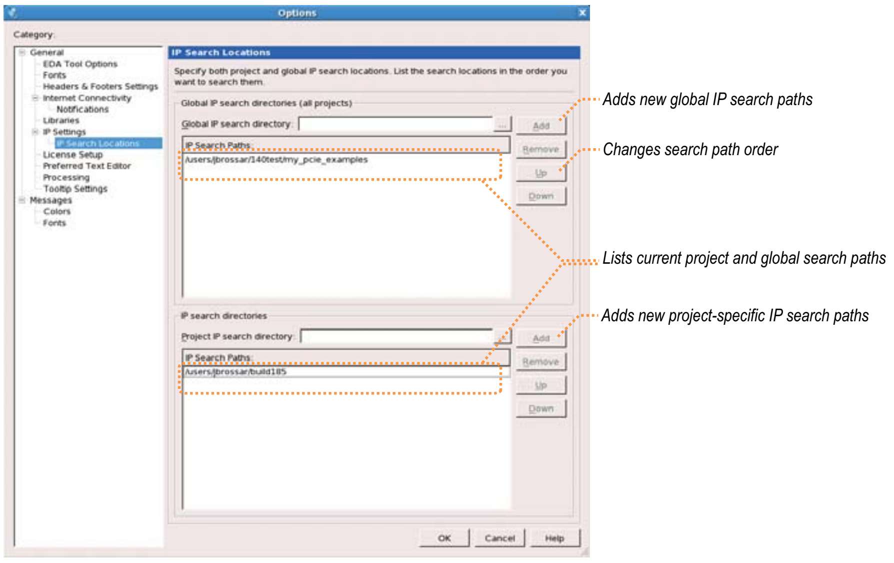
- Component Description File (**_hw.tcl**)—Defines a single IP core.
- IP Index File (**.ipx**)—Each **.ipx** file indexes a collection of available IP cores, or a reference to other directories to search. In general, **.ipx** files facilitate faster searches.

The Quartus II software searches some directories recursively and other directories only to a specific depth. When the search is recursive, the search stops at any directory that contains an **_hw.tcl** or **.ipx** file. In the following list of search locations, a recursive descent is annotated by **. A single * signifies any file.

Table 2–2. IP Search Locations

Location	Description
PROJECT_DIR/*	Finds IP components and index files in the Quartus II project directory.
PROJECT_DIR/ip//***	Finds IP components and index files in any subdirectory of the **/ip** subdirectory of the Quartus project directory.

Figure 2–3. Specifying IP Search Locations



Specify both project and global IP search locations. List the search locations in the order you want to search them.

/users/jbrossar/140test/my_pcie_examples

/users/jbrossar/build185

If the Quartus II software recognizes two IP cores with the same name, the following search path precedence rules determine the resolution of files:

1. Project directory files.
2. Project database directory files.
3. Project libraries specified in **IP Search Locations**, or with the `SEARCH_PATH` assignment in the Quartus II Settings File (**.qsf**).
4. Global libraries specified in **IP Search Locations**, or with the `SEARCH_PATH` assignment in the Quartus II Settings File (**.qsf**).
5. Quartus II software libraries directory, such as <*Quartus II Installation*>**\libraries**.

If you add a component to the search path, you must refresh your system by clicking **File > Refresh** to update the IP Catalog.

Upgrading Outdated IP Cores

IP cores generated with a previous version of the Quartus II software may require upgrade before use in the current version of the Quartus II software. Click **Project > Upgrade IP Components** to identify and upgrade outdated IP cores.

The **Upgrade IP Components** dialog box provides instructions when IP upgrade is required, optional, or unsupported for specific IP cores in your design. Most Altera IP cores support one-click, automatic simultaneous upgrade. You can individually migrate IP cores unsupported by auto-upgrade.

The **Upgrade IP Components** dialog box also reports legacy Altera IP cores that support compilation-only (without modification), as well as IP cores that do not support migration. Replace unsupported IP cores in your project with an equivalent Altera IP core or design logic.Upgrading IP cores changes your original design files.

Before you begin

- Migrate your Quartus II project containing outdated IP cores to the latest version of the Quartus II software. In a previous version of the Quartus II software, click **Project > Archive Project** to save the project. This archive preserves your original design source and project files after migration. le paths in the archive must be relative to the project directory. File paths in the archive must reference the IP variation **.v** or **.vhd** file or **.qsys** file, not the **.qip** file.
- Restore the project in the latest version of the Quartus II software. Click **Project > Restore Archived Project**. Click **Ok** if prompted to change to a supported device or overwrite the project database.

To upgrade outdated IP cores, follow these steps:

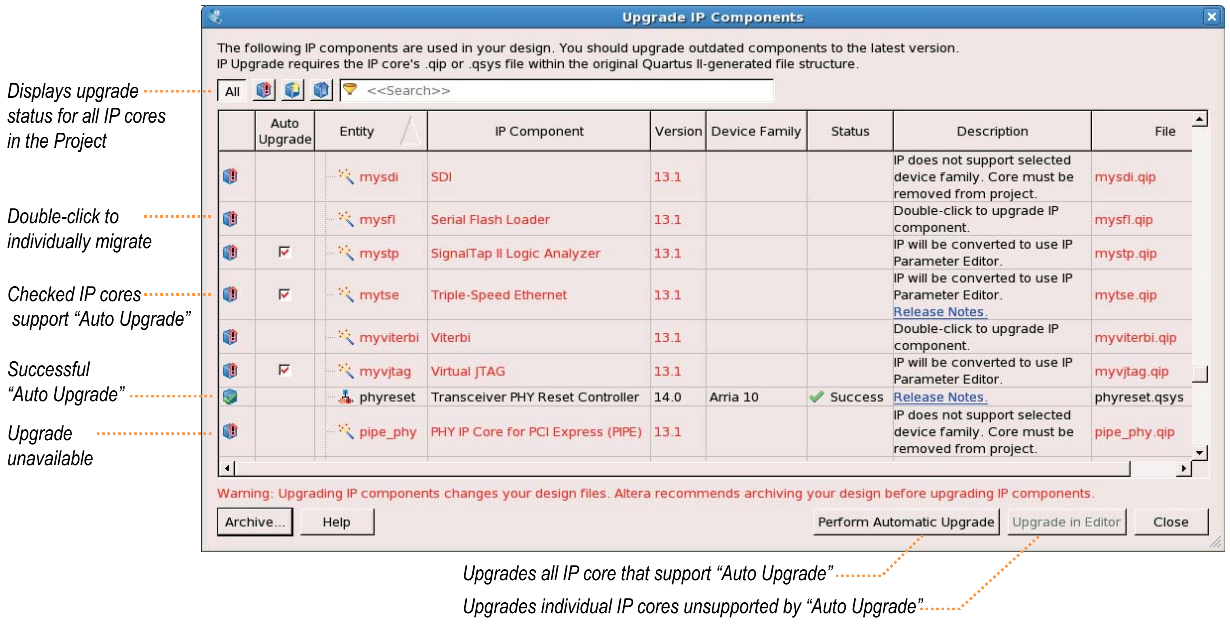
1. In the latest version of the Quartus II software, open the Quartus II project containing an outdated IP core variation.

   File paths in a restored project archive must be relative to the project directory and you must reference the IP variation **.v** or .**vhd** file or **.qsys** file, not the **.qip** file.

2. Click **Project > Upgrade IP Components**. The **Upgrade IP Components** dialog box displays all outdated IP cores in your project, along with basic instructions for upgrading each core.

3. To simultaneously upgrade all IP cores that support automatic upgrade, click **Perform Automatic Upgrade**. The IP cores upgrade to the latest version. The **Status** and **Version** columns reflect the update.

Figure 2–4. Upgrading IP Cores


Displays upgrade status for all IP cores in the Project
Double-click to individually migrate
Checked IP cores support “Auto Upgrade”
Successful “Auto Upgrade”
Upgrade unavailable
Upgrades all IP core that support “Auto Upgrade”
Upgrades individual IP cores unsupported by “Auto Upgrade”
Upgrade IP Components
All
<<Search>>
Archive...
Help
Perform Automatic Upgrade
Upgrade in Editor
Close

The following IP components are used in your design. You should upgrade outdated components to the latest version.
IP Upgrade requires the IP core's .qip or .qsys file within the original Quartus II-generated file structure.

	Auto Upgrade	Entity	IP Component	Version	Device Family	Status	Description	File
		mysdi	SDI	13.1			IP does not support selected device family. Core must be removed from project.	mysdi.qip
		mysfl	Serial Flash Loader	13.1			Double-click to upgrade IP component.	mysfl.qip
	✓	mystp	SignalTap II Logic Analyzer	13.1			IP will be converted to use IP Parameter Editor.	mystp.qip
	✓	mytse	Triple-Speed Ethernet	13.1			IP will be converted to use IP Parameter Editor. Release Notes.	mytse.qip
		myviterbi	Viterbi	13.1			Double-click to upgrade IP component.	myviterbi.qip
	✓	myvjtag	Virtual JTAG	13.1			IP will be converted to use IP Parameter Editor.	myvjtag.qip
		phyreset	Transceiver PHY Reset Controller	14.0	Arria 10	✓ Success	Release Notes.	phyreset.qsys
		pipe_phy	PHY IP Core for PCI Express (PIPE)	13.1			IP does not support selected device family. Core must be removed from project.	pipe_phy.qip

Warning: Upgrading IP components changes your design files. Altera recommends archiving your design before upgrading IP components.

Upgrading IP Cores at the Command Line

Alternatively, you can upgrade IP cores at the command line. To upgrade a single IP core, type the following command:

```
quartus_sh --ip_upgrade -variation_files <my_ip_path> <project>
```

To upgrade a list of IP cores, type the following command:

```
quartus_sh --ip_upgrade -variation_files
"<my_ip>.qsys;<my_ip>.<hdl>; <project>"
```

☞ IP cores older than Quartus II software version 12.0 do not support upgrade. Altera verifies that the current version of the Quartus II software compiles the previous version of each IP core. The *MegaCore IP Library Release Notes* reports any verification exceptions for MegaCore IP. The *Quartus II Software and Device Support Release Notes* reports any verification exceptions for other IP cores. Altera does not verify compilation for IP cores older than the previous two releases.

DSP Builder Design Flow

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

This IP core supports DSP Builder. Use the DSP Builder flow if you want to create a DSP Builder model that includes an IP core variation; use IP Catalog if you want to create an IP core variation that you can instantiate manually in your design.

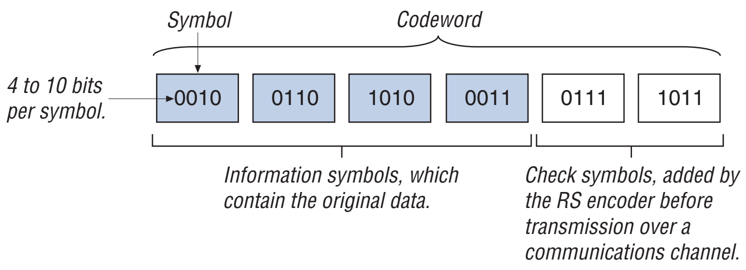
For more information about the DSP Builder flow, refer to the *Using MegaCore Functions* chapter in the *DSP Builder Handbook*.

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


Symbol
Codeword
4 to 10 bits per symbol.
0010
0110
1010
0011
0111
1011
Information symbols, which contain the original data.
Check symbols, added by the RS encoder before transmission over a communications channel.

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.

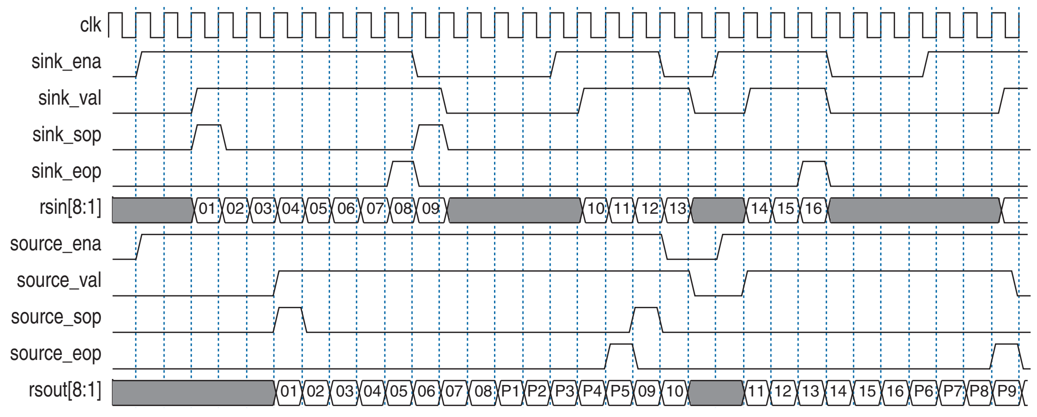
RS Encoder

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.

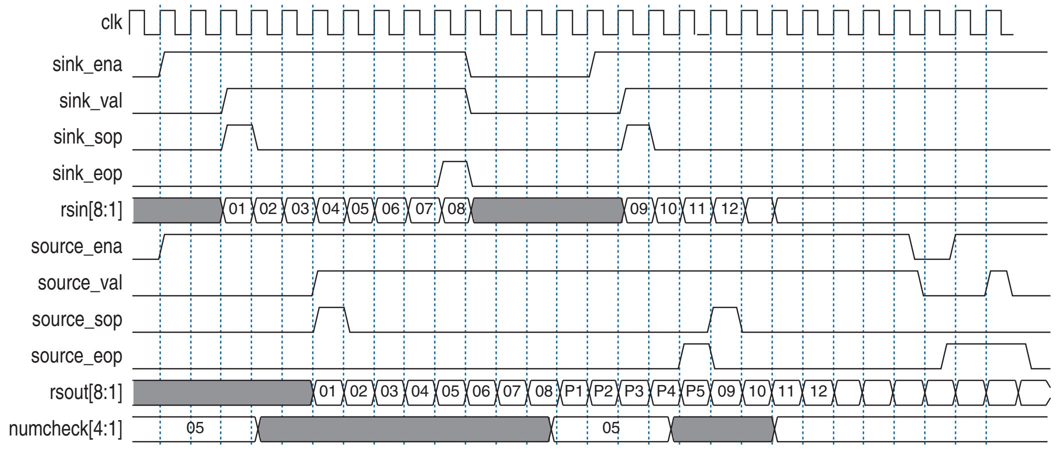
Figure 3–2. Encoder Timing



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 $\leq$ *R*/2	Decoder detects and corrects errors.
R/2 $\leq$ errors $\leq$ *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).

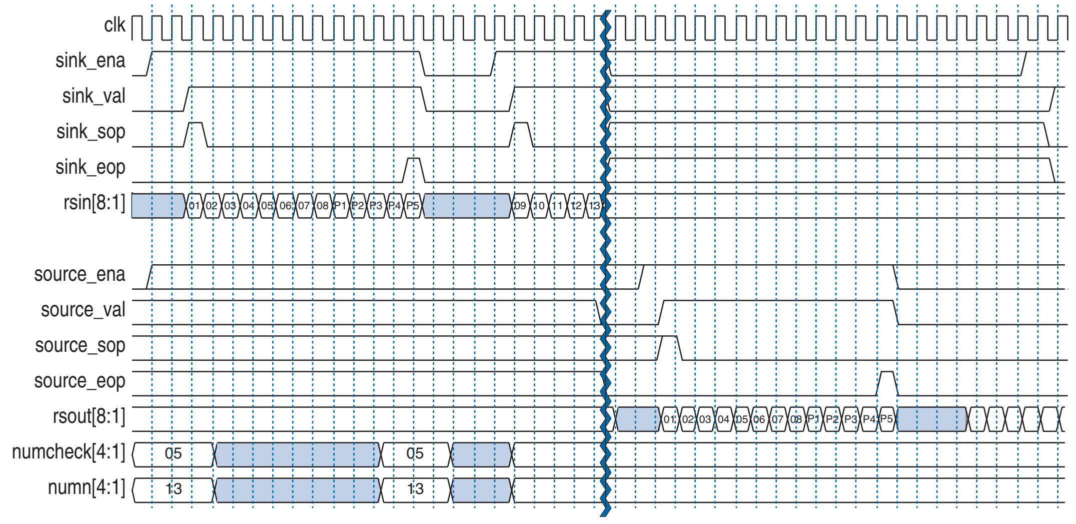
The RS decoder observes Avalon-ST interface standard for input and output data. One clock cycle after the decoder asserts `sink_ena`, you can assert `sink_val`. The decoder accepts the data at `rsin` as valid data. The codeword is started with `sink_sop`. The `numcheck` and `numn` signals are latched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

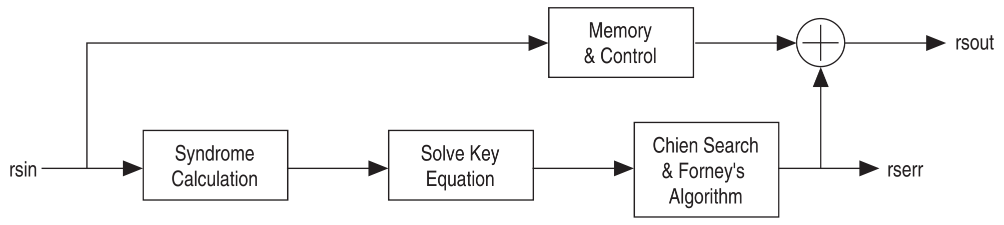
- Error symbol
- Bit error count

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.

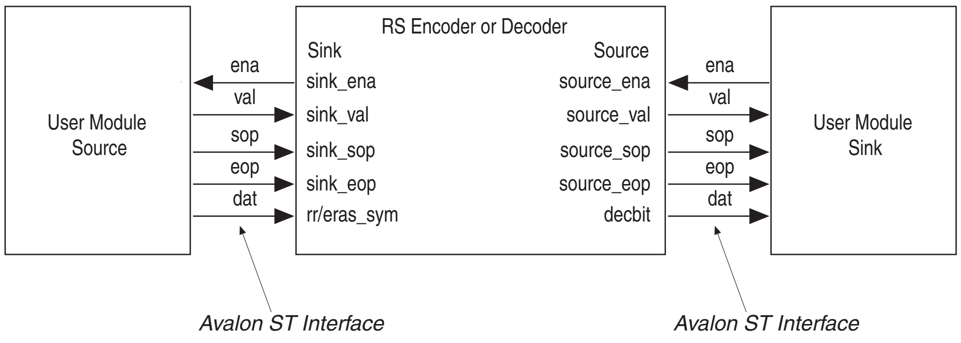
The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having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

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.
`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 (Part 1 of 2)

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.

Table 3–6. Avalon-ST Source Interface (Part 2 of 2)

Name	Avalon-ST Type	Direction	Description
`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

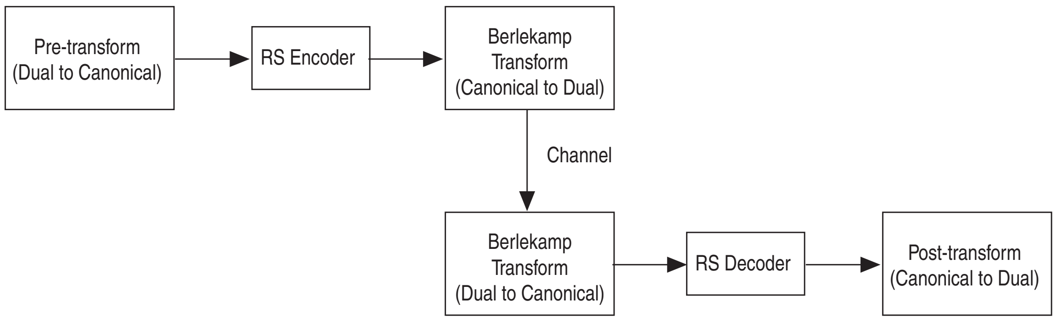
Erasures	Keysize	N_C
No	Half	Max (N, 10 × R + 4)
No	Full	Max (N, 7 × R + 5)
Yes	Half	Max (N, 10 × R + 6)
Yes	Full	Max (N, 8 × R + 4)

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

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


Pre-transform (Dual to Canonical)
RS Encoder
Berlekamp Transform (Canonical to Dual)
Channel
Berlekamp Transform (Dual to Canonical)
RS Decoder
Post-transform (Canonical to Dual)

Supplying 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**.

Additional Information

This chapter provides additional information about the document and Altera.

Revision History

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

Date	Version	Changes Made
2014.12.15	14.1	Added obsolescence notice.
June 2014	14.0	■ Removed support for Cyclone III and Stratix III devices ■ Added instructions for using IP Catalog
November 2013	13.1	■ Removed support for the following devices: ■ Arria ■ Cyclone II ■ HardCopy II, HardCopy III, and HardCopy IV ■ Stratix, Stratix II, Stratix GX, and Stratix II GX ■ Added final support for the following devices: ■ Arria V ■ Stratix V
November 2012	12.1	Added support for Arria V GZ devices.
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

To locate the most up-to-date information about Altera products, refer to the following table.

Contact [1]	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com
(software licensing)	Email	authorization@altera.com

Note to Table:

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

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *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	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in 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`. The suffix `n` denotes an active-low signal. For 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`).
↵	An angled arrow instructs you to press the Enter key.
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.

Visual Cue	Meaning
?	The question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
🎥	The multimedia icon directs you to a related multimedia presentation.
⚠ 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 envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.