

Reed-Solomon Encoder v9.0

LogiCORE IP Product Guide

Vivado Design Suite

PG025 November 18, 2015

Table of Contents

IP Facts

Chapter 1: Overview

Feature Summary	6
Applications	6
Unsupported Features	7
Licensing and Ordering Information	7

Chapter 2: Product Specification

Standards	9
Performance	9
Resource Utilization	10
Port Descriptions	10

Chapter 3: Designing with the Core

Functional Description	19
Block Code Settings	20

Chapter 4: Design Flow Steps

Customizing and Generating the Core	23
Constraining the Core	30
Simulation	31
Synthesis and Implementation	31

Chapter 5: Test Bench

Demonstration Test Bench	32
Simulation	33

Appendix A: Migrating and Upgrading

Migrating to the Vivado Design Suite	34
Upgrading in the Vivado Design Suite	34

Appendix B: Debugging

Finding Help on Xilinx.com	35
--------------------------------------	----

Debug Tools	37
Simulation Debug	38
AXI4-Stream Interface Debug	39

Appendix C: Additional Resources and Legal Notices

References	40
Revision History	41
Please Read: Important Legal Notices	41

Introduction

The Xilinx® LogiCORE™ IP Reed-Solomon Encoder core is used in many Forward Error Correction (FEC) applications and in systems such as communications systems and disk drives where data is transmitted and subject to errors before reception.

Features

- Implements many different Reed-Solomon coding standards, including all ITU-T J.83 and CCSDS codes
- Automatically configured by user-entered parameters
- Efficiently handles multiple channels
- Fully synchronous design using a single clock
- Supports continuous output data with no gap between code blocks
- Symbol width from 3 bits to 12 bits
- Code block length variable up to 4095 symbols with up to 256 check symbols
- Block length can be changed in real time
- The number of check symbols can be changed in real time
- Supports shortened codes
- Supports any primitive field polynomial for a given symbol width
- User-configured generator polynomial
- AXI4-Stream compliant interfaces
- Use with Xilinx Vivado® Design Suite and Xilinx System Generator for DSP

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family ⁽¹⁾	UltraScale+™ Families, UltraScale™ Architecture, Zynq®-7000 All Programmable SoC, 7 Series
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Applicable
Simulation Model	Verilog, VHDL VHDL Behavioral, VHDL or Verilog Structural
Supported S/W Driver	N/A
Tested Design Flows ⁽²⁾	
Design Entry	Vivado® Design Suite Vivado System Generator for DSP
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide .
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx at the Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog .
2. For the supported versions of the tools, see the [Xilinx Design Tools: Release Notes Guide](#).

Overview

Reed-Solomon codes are usually referred to as (n,k) codes, where n is the total number of symbols in a code block and k is the number of information or data symbols. This core generates systematic code blocks where the complete code block is formed from the k information symbols, followed by the $(n-k)$ check symbols. The maximum number of symbol errors in a block that can be guaranteed to be correct is $t = (n-k)/2$. A symbol error can contain any number of bit errors.

Normally $n = 2^{(\text{Symbol_Width})} - 1$. If n is less than this, the code is referred to as a “shortened code.” The encoder core handles both full-length and shortened codes.

The parameters n , k , and $(n-k)$ are optionally variable from block to block. The current code block settings for n , k , and $(n-k)$ are referred to as n_block , k_block , and r_block , respectively.

A Reed-Solomon code is also characterized by two polynomials: the field polynomial and the generator polynomial. The field polynomial defines the Galois field, of which the symbols are members. The generator polynomial defines how the check symbols are generated. Both of these polynomials are usually defined in the specification for any particular Reed-Solomon code. The core Vivado Integrated Design Environment (IDE) allows both of these polynomials to be configured.

The core synchronous input control signals are not registered inside the core. It is assumed these are registered external to the core if required.

Feature Summary

The core configuration Vivado IDE allows several pre-configured standards to be selected. Some of these standards result in additional standard-specific hardware being inserted around the core, such as CCSDS and ITU J.83. After the appropriate standard has been selected it is still possible to edit some parameter settings, depending on the standard. If this does not give the required configuration then "Custom" can be selected and any parameter values can be chosen. See [Customizing and Generating the Core in Chapter 4](#).

Multiple channels can be supported by selecting the number of channels required in the Vivado IDE. This results in time-division multiplexing of channels on the input and output. If this does not give sufficient throughput, multiple channels can be handled by using multiple instances of the core.

Most commonly used Reed-Solomon codes have an 8-bit symbol width. This gives a good balance of error correction capability, throughput, and implementation cost. The core does support any symbol width from 3 to 12.

Some standards required the block length or the number of check symbols or both to be run time variables. The core supports all of these options. Changing the number of check symbols automatically changes the generator polynomial, although the core assumes the scaling factor (h) remains at the value configured in the Vivado IDE.

The core uses standard AXI4-Stream interfaces with full handshaking and there is only one clock input, making it straightforward to use in a system.

Applications

Reed-Solomon forward error correction is often used where speed and latency are important and reasonable error correction performance is required. For example, the core can be used in high-speed wired networking, satellite and space communications, digital video broadcast, and data storage.

Each code block is independent, so multiple encoders and decoders can be used to process blocks in parallel to achieve arbitrarily high data rates. Another property is good burst error correction performance, as it does not matter how many bits are received in error within a single input symbol. RS codes are sometimes used in concatenation with convolution codes. The RS code can correct bursts of errors that sometimes occur when the convolution encoder fails.

Unsupported Features

The core cannot dynamically change field polynomial or symbol width.

Licensing and Ordering Information

License Checkers

If the IP requires a license key, the key must be verified. The Vivado design tools have several license checkpoints for gating licensed IP through the flow. If the license check succeeds, the IP can continue generation. Otherwise, generation halts with error. License checkpoints are enforced by the following Vivado design flow:

- Vivado Synthesis
- Vivado Implementation
- write_bitstream (Tcl Console)



IMPORTANT: IP license level is ignored at checkpoints. The test confirms a valid license exists. It does not check IP license level.

License Type

This Xilinx LogiCORE™ IP module is provided under the terms of the [Xilinx Core License Agreement](#). The module is shipped as part of the Vivado® Design Suite. For full access to all core functionalities in simulation and in hardware, you must purchase a license for the core. Contact your [local Xilinx sales representative for information](#) about pricing and availability.

For more information, visit the Reed-Solomon Encoder [product web page](#).

Information about other Xilinx LogiCORE IP modules is available at the [Xilinx Intellectual Property](#) page. For information on pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your [local Xilinx sales representative](#).

An [evaluation license](#) is available for this core. The evaluation version of the core operates in the same way as the full version for several hours, dependent on clock frequency. Operation is then disabled and the data output does not change. If you notice this behavior in hardware, it probably means you are using an evaluation version of the core. The Xilinx tools warn that an evaluation license is being used during netlist implementation. If a full license is installed for the core to run on hardware, delete the old XCO file and recreate the core from new.

Product Specification

The Reed-Solomon Encoder inputs k information symbols and appends $n-k$ check symbols. This is illustrated in the [Figure 2-1](#), which also shows how the symbols can be interpreted as polynomial coefficients. The check symbols are generated to form $c(x)$ such that $c(x)$ is divisible by the generator polynomial, $g(x)$. If the received code word is not divisible by $g(x)$, the code word contains errors. Fig 2 illustrates how the input data is passed through the core while check symbols are computed. When k symbols have been output, the multiplexer is switched and the $(n-k)$ check symbols are output.

$$\begin{aligned}
 \mathbf{d(x)} &= d_0x^{k-1} + d_1x^{k-2} + \dots \dots \dots + d_{k-2}x^1 + d_{k-1}x^0 \\
 &\quad \boxed{\begin{array}{|c|c|} \hline d_0 & k \text{ Information Symbols} \\ \hline \end{array}} \quad \boxed{d_{k-1}}
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{c(x)} &= d_0x^{n-1} + d_{k-1}x^{n-k} + \dots \dots \dots + d_{k-2}x^{n-k+1} + d_{k-1}x^{n-k} \\
 &\quad \boxed{\begin{array}{|c|c|} \hline k \text{ Information Symbols} & (n-k) \text{ Check Symbols} \\ \hline \end{array}} \\
 &\quad \mathbf{x^{n-k}d(x)} \quad + \quad \mathbf{check(x)}
 \end{aligned}$$

X13182

Figure 2-1: Reed-Solomon Codeword Format

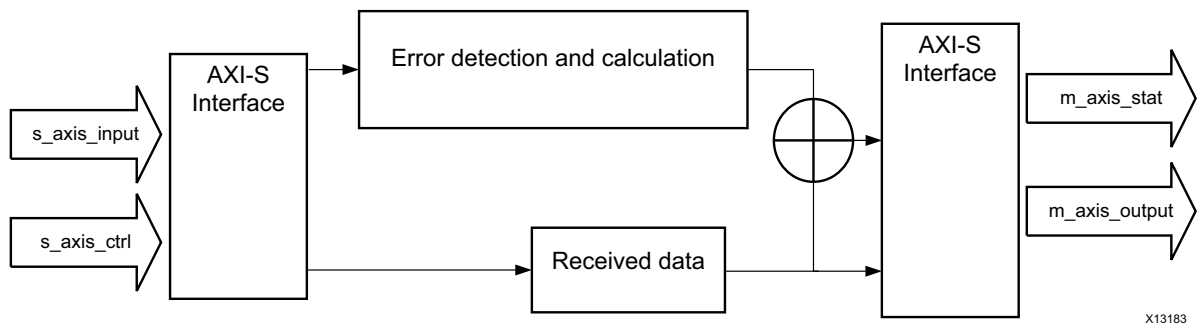


Figure 2-2: Block Diagram

Standards

The Reed-Solomon IP core adheres to the AMBA® AXI4-Stream standard [Ref 1].

Performance

This section contains the following subsections.

- [Latency](#)
- [Throughput](#)

Latency

For this core, *latency* is defined as the number of rising clock edges from sampling `s_axis_input_tdata` to the sampled value appearing on `m_axis_output_tdata`. The basic latency of the core is (2 + number of channels). For example, the latency in [Figure 2-3](#) is 3.

- Selecting *CCSDS* increases the latency of the encoder by 2.
- Selecting *ITU J.83 Annex B* increases the latency by 1.
- Selecting `m_axis_output_tready` increases the latency by a further 2, but also makes latency variable due to the presence of a FIFO to accommodate backpressure inherent in the AXI4-Stream protocol.

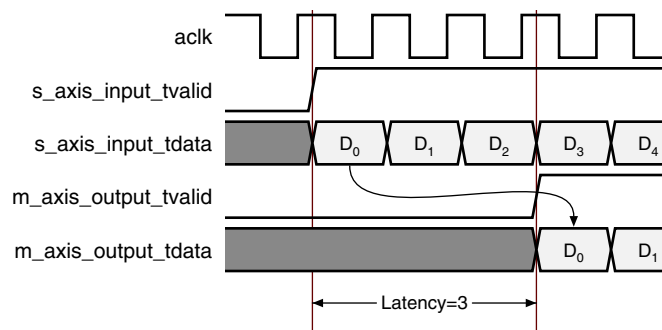


Figure 2-3: Latency

Throughput

The maximum raw data input rate in Mb/s can be calculated as:

$$F_{\max} \text{ (MHz)} * \text{Symbol_Width (bits)} * k/n$$

Resource Utilization

For full details about performance and resource utilization, visit the [Performance and Resource Utilization web page](#).

The area of the core increases with $(n-k)$ and Symbol_Width.

When a variable number of check symbols is not required, the check symbol generator is implemented efficiently as a fixed architecture. When a variable number of check symbols is required, the check symbol generator must be either *optimized for area*, where the implementation area is increased by a factor of approximately 3, or *optimized for flexibility*, where the implementation area is increased by a factor of approximately 5.

The option to map primary I/O registers into IOB flip-flops should be selected if the core I/Os are to be connected directly onto a PCB using the FPGA package pins. This gives lower output clock-to-out times and predictable setup and hold times. Remember the control signal inputs are used unregistered inside the core, so these should be registered external to the core.

Port Descriptions

Pinout

Port names for the core module are shown in [Figure 2-4](#) and described in [Table 2-1](#).

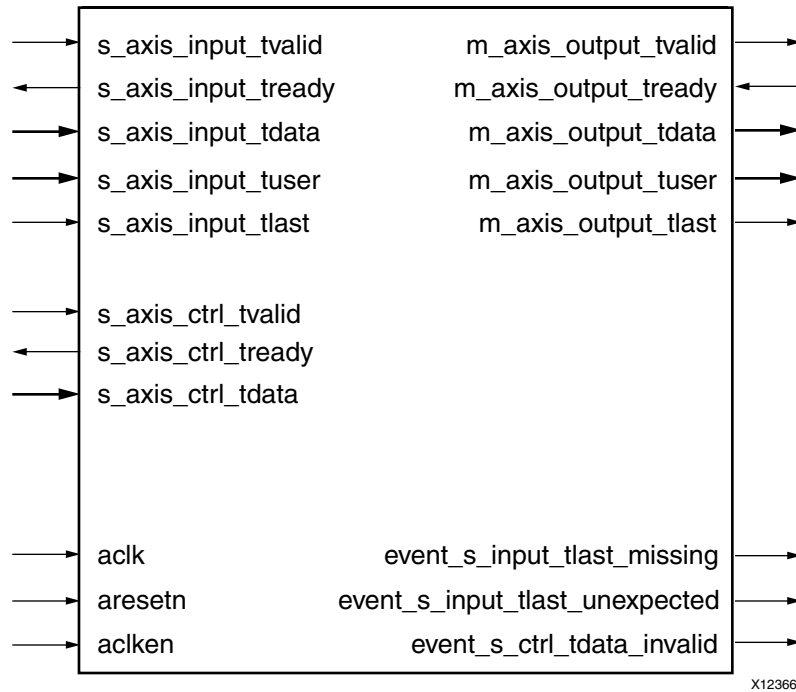


Figure 2-4: Core Schematic Symbol

X12366

Table 2-1: Core Signal Pinout

Signal	Direction	Optional	Description
aclk	INPUT	No	Rising edge clock
aclken	INPUT	Yes	Active-High clock enable
aresetn	INPUT	Yes	Active-Low synchronous clear (overrides aclken). aresetn must be asserted for at least 2 clock cycles.
s_axis_input_tvalid	INPUT	No	TVALID for S_AXIS_INPUT channel. See AXI4-Stream Protocol [Ref 1] for protocol.
s_axis_input_tready	OUTPUT	No	TREADY for S_AXIS_INPUT
s_axis_input_tdata	INPUT	No	Input data and erase flag, if applicable
s_axis_input_tuser	INPUT	Yes	User bits, passed through core unmodified, with same latency as s_axis_input_tdata
s_axis_input_tlast	INPUT	No	Marks last symbol of input block. Only used to generate event outputs. Can be tied Low or High if event outputs not used.
s_axis_ctrl_tvalid	INPUT	Yes	TVALID for S_AXIS_CTRL channel. This channel is only present if core has variable block length or number of check symbols
s_axis_ctrl_tready	OUTPUT	Yes	TREADY for S_AXIS_CTRL channel
s_axis_ctrl_tdata	INPUT	Yes	Block length and number of check symbols, if applicable
m_axis_output_tvalid	OUTPUT	No	TVALID for M_AXIS_OUTPUT channel
m_axis_output_tready	INPUT	Yes	TREADY for M_AXIS_OUTPUT channel. Tie High if downstream slave is always able to accept data from M_AXIS_OUTPUT
m_axis_output_tdata	OUTPUT	No	Corrected data output
m_axis_output_tuser	OUTPUT	Yes	s_axis_input_tuser delayed by core latency
m_axis_output_tlast	OUTPUT	No	High when last symbol of last channel is on m_axis_output_tdata
event_s_input_tlast_missing	OUTPUT	No	Flags that s_axis_input_tlast was not asserted when expected. Leave unconnected if not required.
event_s_input_tlast_unexpected	OUTPUT	No	Flags that s_axis_input_tlast was asserted when not expected. Leave unconnected if not required.
event_s_ctrl_tdata_invalid	OUTPUT	No	Flags that values provided on s_axis_ctrl_tdata were illegal. Core must be reset if this is asserted. Leave unconnected if not required.

aclken

The clock enable input (`aclken`) is an optional pin. When `aclken` is deasserted (Low), all the other synchronous inputs are ignored, except `aresetn`, and the core remains in its current state. This pin should be used only if it is genuinely required because it has a high fan out within the core and can result in lower performance.

`aclken` is a *true* clock enable and causes the entire core to freeze state when it is Low.

An example of `aclken` operation is shown in Figure 2-5. In this case, the core ignores symbol `D4` as input to the block, and the current `m_axis_output_tdata` value remains unchanged. (The encoder still samples *n* symbols.) As `D4` is not included in the code block, the output sequence ...`D0`,`D1`,`D2`,`D3`,`D5`... appears on `m_axis_output_tdata` during the output stage of this block.

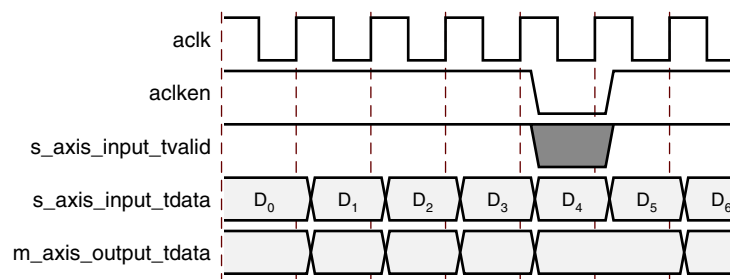


Figure 2-5: Clock Enable Timing

aresetn

The synchronous reset (`aresetn`) input is an optional pin. It can be used to re-initialize the core at any time, regardless of the state of `aclken`. `aresetn` needs to be asserted Low for at least two clock cycles to initialize the circuit. The core becomes ready for normal operation two cycles after `aresetn` goes High.



CAUTION! This pin should be selected with caution, as it increases the size of the core and can reduce performance.

The timing for the `aresetn` input is illustrated in Figure 2-6. Note that some outputs are not reset by `aresetn`.

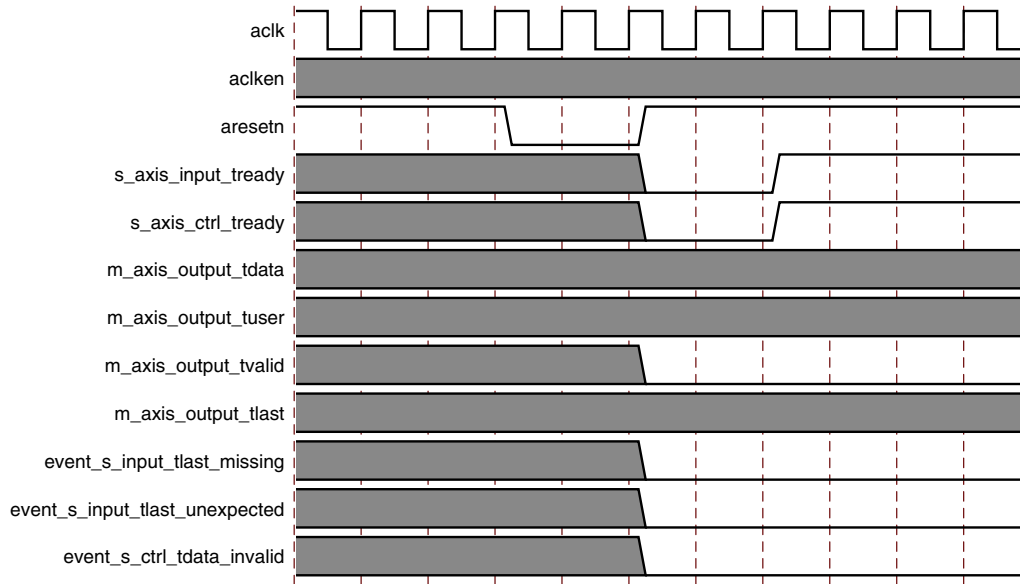
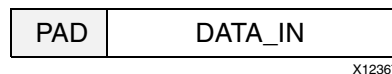


Figure 2-6: Synchronous Reset Timing

S_AXIS_INPUT Channel

s_axis_input_tdata

Data to be processed is passed into the core on this port. To ease interoperability with byte-oriented buses, TDATA is padded with zeros, if necessary, to fit a bit field which is a multiple of 8 bits. The padding bits are ignored by the core and do not result in additional resource use. The structure is shown in Figure 2-7.



X12367

Figure 2-7: Input Channel TDATA Structure

The DATA_IN field is the input bus for the incoming uncoded data. The width of the DATA_IN portion of the field is set by the Symbol Width parameter in the Vivado IDE.

s_axis_input_tuser

This optional input is used to pass information through the core with exactly the same latency as *s_axis_input_tdata*. This could be used to tag each symbol sampled on DATA_IN with marker bits, for example. The number of TUSER bits is parameterizable and set by the Number of Marker Bits parameter in the Vivado IDE. The TUSER bits are delayed with the same latency as DATA_IN to DATA_OUT and output on *m_axis_output_tuser*.

For example, if "5" is sampled on `s_axis_input_tuser` at the same time as the first symbol on `s_axis_input_tdata`, then "5" is output on `m_axis_output_tuser` at the same time the first symbol is output on `m_axis_output_tdata`.

This feature can be used to mark special symbols within a frame or to tag data from different blocks with block identification numbers.

s_axis_input_tlast

This input can be tied Low or High if the event outputs (`event_s_input_tlast_missing` and `event_s_input_tlast_unexpected`) are not used. It is present purely to provide a check that the system and core are in sync with block sizes. If the event outputs are used then `s_axis_input_tlast` must be asserted High when the last symbol of a block is sampled on `s_axis_input_tdata`. In the multichannel case it must be asserted when the last symbol of the last channel of the block is sampled on `s_axis_input_tdata`. The core maintains its own internal count of the symbols, so it knows when the last symbol is being sampled. If `s_axis_input_tlast` is not sampled High when the last input symbol is sampled then `event_s_input_tlast_missing` is asserted until the next input sample is taken. Similarly, if `s_axis_input_tlast` is sampled High when the core is not expecting it, `event_s_input_tlast_unexpected` is asserted until the next input sample is taken. If either of these events occurs then the system and the core are out of sync and the core, and possibly the system, should be reset.

S_AXIS_CTRL Channel

s_axis_ctrl_tdata

If the S_AXIS_CTRL channel is present, control data for each block is passed into the core on this port. The port is composed of several subfields, depending on parameter settings. Each subfield is padded to make it a multiple of 8 bits. The padding bits are ignored by the core and do not result in additional resource use. The structure is shown in Figure 2-8. Care should be taken to ensure only valid combinations of `N_IN` and `R_IN` are provided, as the core might need to be reset if invalid values are written.

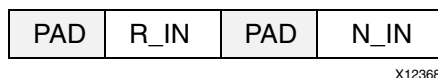


Figure 2-8: Control Channel TDATA Structure

N_IN Field

This field is only present if “Variable Block Length” is selected in the Vivado IDE. This allows the block length to be changed every block. Unless there is an R_IN field, the number of check symbols is fixed, so varying n automatically varies k .

For example, if N_IN is set to 255 and R_IN is set to 16 in the control word C_1 in Figure 2-10, the next input block (starting D_1) is treated as a ($n=255$, $k=239$) codeword. If C_2 has N_IN equal to 64 and R_IN is equal to 8, then the next input block (starting D_N) is treated as a ($n=64$, $k=56$) codeword. For this example, n should be set to 255 and k to 239 in the Vivado IDE, as the largest expected R_IN value is 16. This would give an R_IN field width of 5 bits (plus 3 padding bits).

R_IN Field

This field is only present if “Variable Number of Check Symbols” is selected in the Vivado IDE. It allows the number of check symbols to be changed every block. Selecting this input significantly increases the size of the core.

The width of the R_IN field is the minimum number of bits required to represent the maximum n value minus the minimum k value, padded with unused inputs to round up to the nearest multiple of 8.

M_AXIS_OUTPUT Channel

m_axis_output_tdata

Uncoded data sampled on `s_axis_input_tdata` is encoded and output from the core on this port. The port is composed of several subfields, depending on parameter settings. All output fields are padded with zeroes to fit a bit field which is a multiple of 8 bits. The structure is shown in Figure 2-9.



Figure 2-9: Output Channel TDATA Structure

DATA_OUT Field

This is the output field for the corrected symbols. This field always has the same width as DATA_IN.

Corrected symbols start to appear several clock cycles after the first symbol is sampled on DATA_IN. This delay is termed the latency of the encoder and is explained in [Latency, page 9](#). Latency can vary if the block size is dynamically varied with the N_IN field or if the output is stalled by deassertion of a TREADY input.

INFO Field

This optional output field contains a single information bit, INFO, which is High when data symbols are on DATA_OUT and Low when check symbols are on DATA_OUT (that is, the last $n-k$ symbols of the block).

$m_axis_output_tuser$

This optional output is $s_axis_input_tuser$ delayed by the same latency as $s_axis_input_tdata$ to $m_axis_output_tdata$. The width is the same as $s_axis_input_tuser$. As only k values are sampled on the input, only k values can be output.

$m_axis_output_tlast$

This output is High when the last symbol of a block is on $m_axis_output_tdata$ (the n th symbol). In the multichannel case, $m_axis_output_tlast$ is only asserted High when the last symbol of the last channel is present on $m_axis_output_tdata$. This is shown in Figure 2-11.

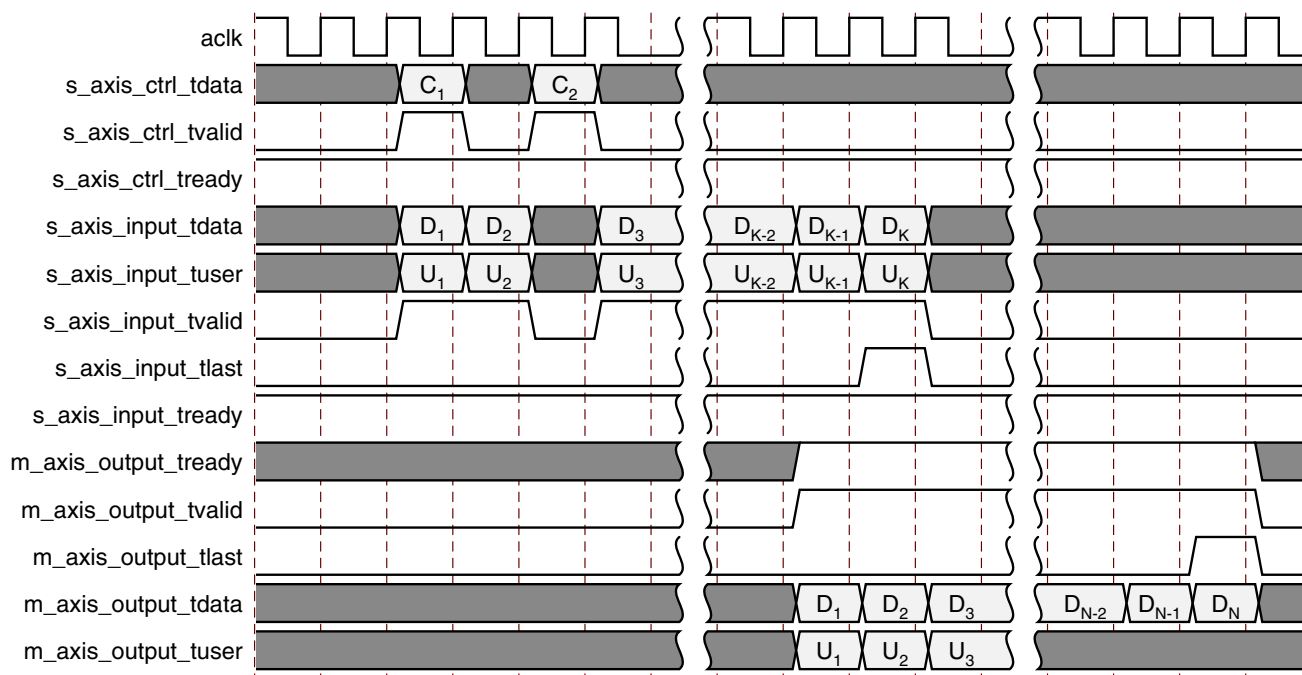


Figure 2-10: Block Input to Output Timing

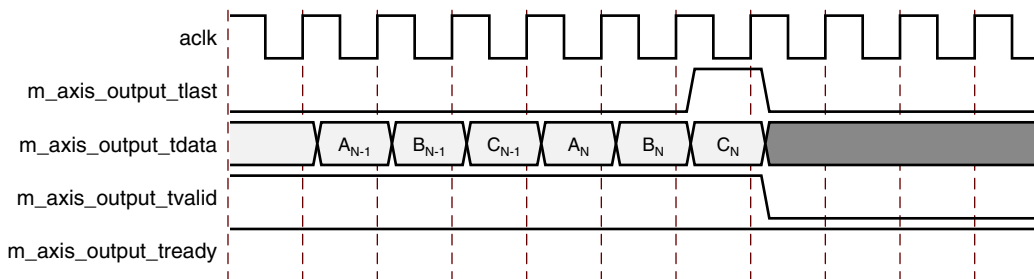


Figure 2-11: TLAST Output Timing for 3 Channel Example

event_s_input_tlast_missing

This output is asserted High if `s_axis_input_tlast` is not sampled High when the last symbol of a block is sampled. It should be left unconnected if not required and the logic used to generate it is optimized away.



IMPORTANT: This output is only asserted until the next input sample starts to be processed inside the core, so care must be taken not to miss a pulse on this output. This output can be used to interrupt the system and possibly instigate a reset sequence.

event_s_input_tlast_unexpected

This output is asserted High if `s_axis_input_tlast` is sampled High when an input symbol that is not the last symbol of a block is sampled. Its timing and operation are the same as `event_s_input_tlast_missing`.

event_s_ctrl_tdata_invalid

This output is asserted High if the core has an `S_AXIS_CTRL` channel and values are sampled on `N_IN` or `R_IN` that are outside the absolute limits the core can handle. The limits are computed at core generation time, based on the parameters selected. When asserted, this output remains asserted until the core is reset. The core must be reset if this output is asserted, as invalid `N_IN` or `R_IN` values can cause the core to malfunction for subsequent blocks and not recover. Control values should be within the limits defined in [Table 3-1](#).

Designing with the Core

This chapter includes guidelines and additional information to make designing with the core easier.

Functional Description

AXI4-Stream Protocol

The use of AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE™ IP solutions. Other than general control signals such as `ac1k`, `ac1ken` and `aresetn`, and event outputs, all inputs and outputs to the core are conveyed using AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports and fields. In the RS Encoder core, the additional ports used are TREADY, TLAST and TUSER. Together, TVALID and TREADY perform a handshake to transfer a value, where the payload is TDATA, TUSER and TLAST. The payload is indeterminate when TVALID is deasserted.

The RS Encoder core operates on the values contained in the S_AXIS_INPUT channel TDATA fields and outputs the results in the TDATA fields of the M_AXIS_OUTPUT channel. The RS Encoder core does not use inputs TUSER and TLAST as such, but the core provides the facility to convey TUSER with the same latency as TDATA. This facility of passing TUSER from input to output is intended to ease use of the core in a system. TLAST is provided purely as a check that the core is in sync with the system and its use is optional.

For further details on AXI4-Stream Interfaces see [\[Ref 1\]](#) and [\[Ref 2\]](#).

Basic Handshake

[Figure 3-1](#) shows the transfer of data in an AXI4-Stream channel. TVALID is driven by the source (master) side of the channel and TREADY is driven by the receiver (slave). TVALID indicates that the value in the payload fields (TDATA, TUSER and TLAST) is valid. TREADY indicates that the slave is ready to receive data. When both TVALID and TREADY are TRUE in a cycle, a transfer occurs. The master and slave set TVALID and TREADY respectively for the next transfer appropriately.

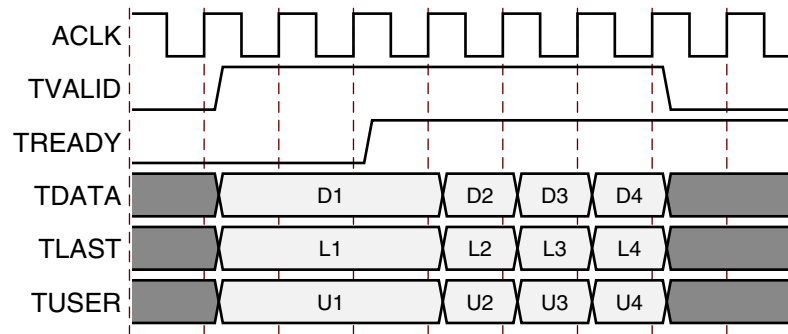


Figure 3-1: Data Transfer in an AXI-Stream Channel

The full flow control of AXI4-Stream aids system design because the flow of data is self-regulating. Data loss is prevented by the presence of back pressure (TREADY), so that data is only propagated when the downstream datapath is ready to process it.

The core has two input channels: S_AXIS_INPUT and S_AXIS_CTRL. If any of the block parameters, such as block length, have been selected to be run time configurable then a block cannot be processed until the control values for that block have been loaded on S_AXIS_CTRL. A new control value must be loaded for every new block or the core will stall the S_AXIS_INPUT channel by deasserting `s_axis_input_tready`. Some data can be input without a control value until the input FIFO fills. It is recommended to write control values before the data is supplied. To guarantee that the input channel is not stalled due to lack of control information, the control value should be written no later than one clock cycle before the first data symbol is sampled. Control values are stored in a FIFO inside the core and used when a new input block is started. Up to 16 control values can be stored before any input data is provided. After the control FIFO fills, `s_axis_ctrl_tready` is deasserted.

The core has one output channel: M_AXIS_OUTPUT. If the output is prevented from off-loading data because `m_axis_output_tready` is Low then data accumulates in the core. When the internal buffers of the core are full the core stops further operations. This prevents the input buffers from off-loading data for new operations so the input buffers fill as new data is input. When the input buffers fill, their respective TREADYs (`s_axis_input_tready` and `s_axis_ctrl_tready`) are deasserted to prevent further input. This is the normal action of back pressure.

Block Code Settings

The RS Encoder generates a systematic (n_block, k_block) block code, where the output block is n_block symbols long, comprised from k_block data symbols followed by r_block check symbols. The block code settings n_block , k_block and r_block are optionally variable on a block-by-block basis. For multichannel configurations, all channels have the same settings for n_block , k_block and r_block . See [Table 3-1](#).

Table 3-1: Block Code Settings – Value and Range

Block Code Settings	Value	Range Min	Range Max
Fixed Block Length			
n_block	n	4	$2^{(Symbol_Width)-1}$
k_block	k	2	$2^{(Symbol_Width)-3}$
r_block	$(n-k)$	2	$\min(n-k, 256)$
Variable Block Length. Fixed Number of Check Symbols			
n_block	N_IN	4	$2^{(Symbol_Width)-1}$
k_block	$N_IN - (n-k)$	2	$2^{(Symbol_Width)-3}$
r_block	$(n-k)$	2	$\min(n-k, 256)$
Variable Number of Check Symbols (optimized for flexibility)			
n_block	N_IN	5	$2^{(Symbol_Width)-1}$
k_block	$N_IN - R_IN$	3	$2^{(Symbol_Width)-3}$
r_block	R_IN	2	$\min(n-k, 128)$
Variable Number of Check Symbols (optimized for area)			
n_block	N_IN	$2^{*(n-k)}$	$2^{(Symbol_Width)-1}$
k_block	$N_IN - R_IN$	3	$2^{(Symbol_Width)-3}$
r_block	R_IN	2	$\min(n-k, 128)$

n_block

The block code setting n_block specifies the total number of symbols in the current code block.

- When a variable block length is not required, n_block is set to the parameter n for every code block.
- When a variable block length is required, n_block is set to the value written for the current block on the CTRL channel N_IN field.

k_block

The block code setting k_block specifies the number of data symbols in the current code block.

- When a variable block length is not required, k_block is set to the parameter k for every block.
- When a variable block length is required and a variable number of check symbols is not required, k_block is set to the value written for the current block on the CTRL channel N_IN field minus the parameter $(n-k)$.

- When a variable number of check symbols is required, k_block is set to the value written for the current block on the CTRL channel N_IN field minus the value sampled on R_IN .

r_block

The block code setting r_block specifies the number of check symbols in the current code block.

- When a variable number of check symbols is not required, r_block is set to parameter $(n-k)$ for every block.
- When a variable number of check symbols is required, r_block is set to the value written for the current block on the CTRL channel R_IN field.

Design Flow Steps

This chapter describes customizing and generating the core, constraining the core, and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [\[Ref 3\]](#)
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [\[Ref 4\]](#)
- *Vivado Design Suite User Guide: Getting Started* (UG910) [\[Ref 5\]](#)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [\[Ref 7\]](#)

Customizing and Generating the Core

This section includes information about using Xilinx tools to customize and generate the core in the Vivado Design Suite.

If you are customizing and generating the core in the Vivado IP Integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [\[Ref 3\]](#) for detailed information. IP Integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [\[Ref 4\]](#) and the *Vivado Design Suite User Guide: Getting Started* ((UG910) [\[Ref 5\]](#).

Note: Figures in this chapter are illustrations of the Vivado IDE. This layout might vary from the current version.

Code Block Specification Parameters

Code Specification

The Vivado IDE aids the creation of cores for a number of common Reed-Solomon specifications. After a particular specification has been chosen, the Vivado IDE automatically selects the values necessary to meet the specification.

Most of the standards listed just result in particular values being set and then greyed out for most of the parameters in the Vivado IDE. However, some of the standards result in additional logic being added to the core. These are described in the following sections.

CCSDS

When implementing the *CCSDS* specification, the core automatically implements the dual-basis conversions defined in the *CCSDS* specification. This is illustrated in [Figure 4-1](#). If the dual-basis conversions are not required, select *custom* specification instead of *CCSDS* and enter all the code parameters manually. Selecting *CCSDS* increases the latency of the encoder by 2.

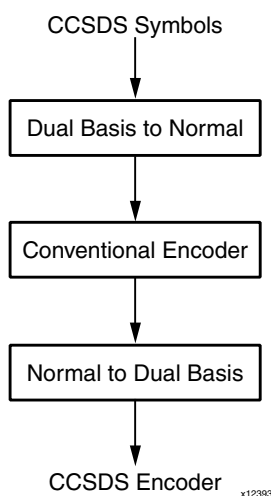


Figure 4-1: CCSDS Encoder

ITU J.83 Annex B

This standard is unusual in that it calls for a (128,122) code. This suggests that n is greater than $2^{(Symbol_Width)-1}$, as the Symbol Width is only 7 bits. However, the standard specifies that the first 127 symbols are generated as a normal RS code. A special 128th symbol is then appended to the end of the block. If *ITU J.83 Annex B* is selected, then the core includes the logic required to generate this 128th symbol.

Selecting *ITU J.83 Annex B* increases the latency of the encoder by 1. The other RS codes specified in the ITU J.83 standard do not require this additional symbol, and the *custom* code specification should be selected for them.

Variable Number of Check Symbols

The `R_IN` field is added when a variable number of check symbols is required.

- Whenever a block is started, the new number of check symbols, r_block , is read from the internal CTRL data FIFO.
- The number of check symbols in the new block is independent of the block length, so varying `R_IN` also changes the number of data symbols in the block, k_block , by negative the same amount.
- The n parameter must be set to $2^{(Symbol_Width)}-1$. The k parameter should be set such that $(n-k)$ equals the maximum number of check symbols required. The width of `R_IN` port is the number of bits required to represent $(n-k)$ in unsigned binary format.
- A multichannel implementation is not available if a variable number of check symbols is required.
- The value sampled on `R_IN` must be in the range given for r_block in [Table 3-1, page 21](#).

For full details on the variable block code settings k_block and r_block , see [Block Code Settings, page 20](#).

Variable Block Length

The `N_IN` field is added when a variable block length is required.

- Whenever a block is started, the new block length, n_block , is read from the internal CTRL data FIFO.
- The number of check symbols in the new block is independent of the block length, so varying n_block also changes the block number of data symbols, k_block , by the same amount.
- The n parameter must be set to $2^{(Symbol_Width)}-1$, and the k parameter should be set such that $(n-k)$ equals the number of check symbols required.
- For multichannel implementations, n_block is the same for all channels.
- The value sampled on `N_IN` must be in the range given for n_block in [Table 3-1, page 21](#).

For full details on the variable block code settings n_block and k_block , see [Block Code Settings, page 20](#).

Symbol Width

This is the width of the `N_IN`, `DATA_IN` and `DATA_OUT` fields.

Field Polynomial

This is used to generate the Galois field for the code. It is entered as a decimal number where the bits of the binary equivalent correspond to the polynomial coefficients. For example,

$$x^8 + x^4 + x^3 + x^2 + 1 \Rightarrow 100011101 \Rightarrow 285$$

A value of zero causes the default polynomial for the given Symbol Width to be selected. If Field Polynomial is not primitive, the core Vivado IDE highlights it in red. Table 4-1 shows the default field polynomial.

Table 4-1: Default Polynomials

Symbol Width	Default Polynomial	Decimal Representation
3	x^3+x+1	11
4	x^4+x+1	19
5	x^5+x^2+1	37
6	x^6+x+1	67
7	x^7+x^3+1	137
8	$x^8+x^4+x^3+x^2+1$	285
9	x^9+x^4+1	529
10	$x^{10}+x^3+1$	1033
11	$x^{11}+x^2+1$	2053
12	$x^{12}+x^6+x^4+x+1$	4179

Scaling Factor (h)

The scaling factor for the generator polynomial root index. Normally, h is 1.

To ensure correct operation of the Reed-Solomon encoder, the value of h must be chosen so that the greatest common divisor of h and $2^{(\text{Symbol_Width})}-1$ is 1, that is, h and $2^{(\text{Symbol_Width})}-1$ must be relative primes.

GeneratorStart

This is the Galois field logarithm of the first root of the generator polynomial.

Normally, GeneratorStart is 0 or 1; however, it can be any positive integer up to 1023.

$$g(x) = \prod_{i=0}^{n-k-1} (x - \alpha^{h \times (\text{GeneratorStart} + i)})$$

Symbols per Block (n)

The number of symbols in a fixed length code block. If this is a shortened code, n should be the shortened number.

When a variable block length is required, the n parameter is defaulted to $2^{(Symbol_Width)-1}$ and the block length, n_block , is set to the value sampled on the N_IN field.

Data symbols (k)

The number of information or data symbols in a fixed length code block.

When a variable block length and a fixed number of check symbols are required, the block number of data symbols, k_block , is set to the value sampled on N_IN minus parameter $(n-k)$.

When a variable number of check symbols is required, the block number of data symbols, k_block , is set to the value sampled on N_IN minus the value sampled on the R_IN field.

Check Symbol Generator Optimization

If a variable number of check symbols is not required, then Check Symbol Generator Optimization must be set to *Fixed Architecture*.

- *Fixed Architecture* – The check symbol generator is implemented using a highly efficient fixed architecture.

If a variable number of check symbols is required, the Check Symbol Generator Optimization must be set to one of the following:

- *Optimized for Flexibility* – The check symbol generator implementation is optimized to maximize the range of input field, N_IN .
- *Optimized for Area* – The check symbol generator implementation is optimized for area and speed efficiency. The range of input field, N_IN , is reduced.

Memory Style

If the target device architecture supports block memory, the following options are available:

- *Distributed* – Core should not use any block memories if possible. This is useful if they are required elsewhere in the design.
- *Block* – Core should use block memories wherever possible. This keeps the number of CLBs used to a minimum, but might use block memory wastefully.
- *Automatic* – This allows the core to use the most appropriate style of memory for each case, based on required memory depth.

Number of Channels

The core can process multiple input channels simultaneously with only a small increase in area. This is much more efficient than instantiating multiple RS Encoder cores.

When a new block is started for one channel, a new block is started for all the other channels as well. The code settings n_block , k_block and r_block are the same for all channels. The multichannel configuration is not available when a variable number of check symbols is required.

The latency is increased by 1 for each additional channel.

With multiple channels, there is still only one `DATA_IN` port. Incoming symbols for the channels are interlaced so the core samples the first symbol of channel 1 on the first rising clock edge, then the first symbol of channel 2 on the second rising clock edge, and so on. Symbols (both information and check) are output on `DATA_OUT` in the same sequence. An example with three channels is shown in Figure 4-2.

A new block is started for all three channels when `s_axis_input_tvalid` is asserted. A1, B1 and C1 are the first symbols of the new block for channels A, B and C.

`s_axis_input_tvalid` can be deasserted at any time. For example, no value is sampled at the start of clock cycle 8.

Symbols on `m_axis_output_tdata` are interlaced in the same way as symbols on `s_axis_input_tdata`.

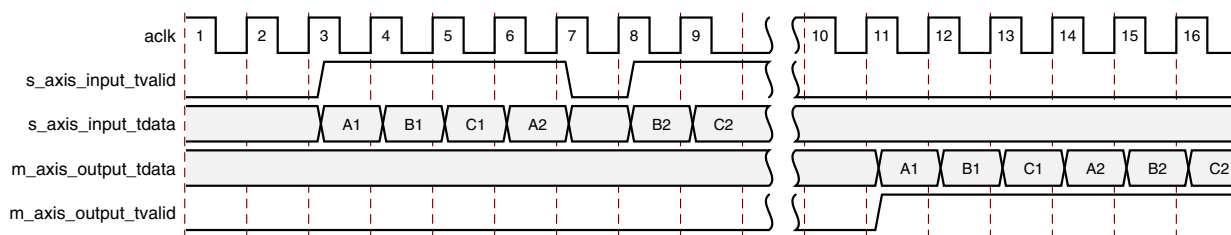


Figure 4-2: Multi-Channel Operation

User Parameters

Table 4-2 shows the relationship between the GUI fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl console). All parameters are available in the IP integrator and none are set automatically.

Table 4-2: GUI Parameter to User Parameter Relationship

GUI Parameter/Value ⁽¹⁾	User Parameter/Value ⁽¹⁾	Default Value
Code Specification	code_specification	Custom
Variable Number of Check Symbols	variable_number_of_check_symbols	False
Variable Block Length	variable_block_length	False
Symbol Width	symbol_width	8
Field Polynomial	field_polynomial	0
Scaling Factor	scaling_factor	1
Generator Start	generator_start	0
Symbols Per Block (n)	symbol_per_block	255
Data Symbols (k)	data_symbols	239
Check Symbols Generator Optimization	check_symbol_generator	Fixed_Architecture
Area	Optimized_For_Area	
Fixed_Architecture	Fixed_Architecture	
Flexibility	Optimized_For_Flexibility	
Memory Style	memory_style	Automatic
Number of Channels	number_of_channels	1
Clock Enable	aclken	False
Synchronous Reset	aresetn	False
M_axis_output_tready	output_has_tready	False
Info Bit	info	False
Marker Bits	marker_bits	False
Number of Marker Bits	number_of_marker_bits	1

Notes:

- Parameter values are listed in the table where the GUI parameter value differs from the user parameter value. Such values are shown in this table as indented below the associated parameter.

System Generator for DSP Graphical User Interface

The Reed-Solomon Encoder core is available through Xilinx System Generator, a DSP design tool that enables the use of The Mathworks model-based design environment Simulink® for FPGA design. The Reed-Solomon Encoder core is one of the DSP building blocks provided in the Xilinx blockset for Simulink. The core can be found in the Xilinx Blockset in the Communication section. The block is called 'Reed-Solomon Encoder 9.0.' See the *System Generator for DSP User Guide* (UG640) [Ref 6] for more information.

The controls in the System Generator GUI work identically to those in the Vivado Integrated Design Environment, although the layout has changed slightly.

Output Generation

Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the .veo and .vho files respectively.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [\[Ref 4\]](#).

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Required Constraint

There are no required constraints.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

It is recommended that the core `ac1k` input is constrained to the required operating frequency.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 7].

Synthesis and Implementation

For details about synthesis and implementation, see “Synthesizing IP” and “Implementing IP” in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].

Test Bench

This chapter contains information about the provided test bench in the Vivado® Design Suite.

Demonstration Test Bench

When the core is generated using the Vivado IDE, a demonstration test bench is created. This is a simple VHDL test bench that exercises the core.

The demonstration test bench source code is one VHDL file:

`demo_tb/ tb_<component_name>.vhd` in the Vivado output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

Compile the netlist and the demonstration test bench into the work library (see your simulator documentation for more information on how to do this). Then simulate the demonstration test bench. View the test bench signals in your simulator waveform viewer to see the operations of the test bench.

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates an input codeblock consisting of a sinusoid
- Generates a clock signal
- Drives the core input signals to demonstrate core features
- Checks that the core output signals obey AXI protocol rules (data values are not checked to keep the test bench simple)
- Provides signals showing the separate fields of AXI TDATA and TUSER signals

The demonstration test bench drives the core input signals to demonstrate the features and modes of operation of the core. The operations performed by the demonstration test bench are appropriate for the configuration of the generated core and are a subset of the following operations:

1. An initial phase where the core is initialized and no operations are performed.
2. Encode a codeblock.
3. Use a different codeblock configuration, with fewer symbols and fewer check symbols, as appropriate to the core.
4. Encode 20 codeblocks, streaming data continuously as fast as the core can process it.
5. Encode 10 more codeblocks which demonstrating the AXI control signals' use and effects.
6. If clock enable is present: Demonstrate the effect of toggling acklen.
7. If reset is present: Demonstrate the effect of asserting aresetn.

Customizing the Demonstration Test Bench

It is possible to modify the demonstration test bench to use different codeblock data or different control information.

Input data is pre-generated in the `create_ip_table` function and stored in the `IP_DATA` constant. Data from this constant is driven into the core by the `drive_input_codeblock` procedure.

For cores with an `S_AXIS_CTRL` control channel, control information is generated and driven into the core by the `ctrl_stimuli` process. Ensure that control information is provided for each data codeblock to prevent the core stalling.

The clock frequency of the core can be modified by changing the `CLOCK_PERIOD` constant.

Simulation

To simulate the core, generate the core simulation model and demonstration test bench. Ensure that the demonstration test bench is the top level entity in the simulation options. Then select 'Run Simulation' in the Vivado IDE.

For full instructions on simulating your core, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 7].

Migrating and Upgrading

This appendix contains information about upgrading to a more recent version of the IP core.

Migrating to the Vivado Design Suite

For information about migrating to the Vivado Design Suite, see the *ISE to Vivado Design Suite Migration Guide* (UG911) [\[Ref 8\]](#).

Upgrading in the Vivado Design Suite

There are no changes compared to the previous Vivado version, because v9.0 was the first Vivado version.

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools.

TIP: *If the IP generation halts with an error, there might be a license issue. See [License Checkers in Chapter 1](#) for more details.*

Finding Help on Xilinx.com

To help in the design and debug process when using the Reed-Solomon Encoder core, the [Xilinx Support web page](#) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Reed-Solomon Encoder core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the [Downloads page](#). For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main [Xilinx support web page](#). To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Reed-Solomon Encoder core

AR: [54510](#)

Technical Support

Xilinx provides technical support in the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the [Xilinx Support web page](#).

Debug Tools

There are many tools available to address Reed-Solomon Encoder core design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature represents the functionality in the Vivado IDE that is used for logic debugging and validation of a design running in Xilinx devices in hardware.

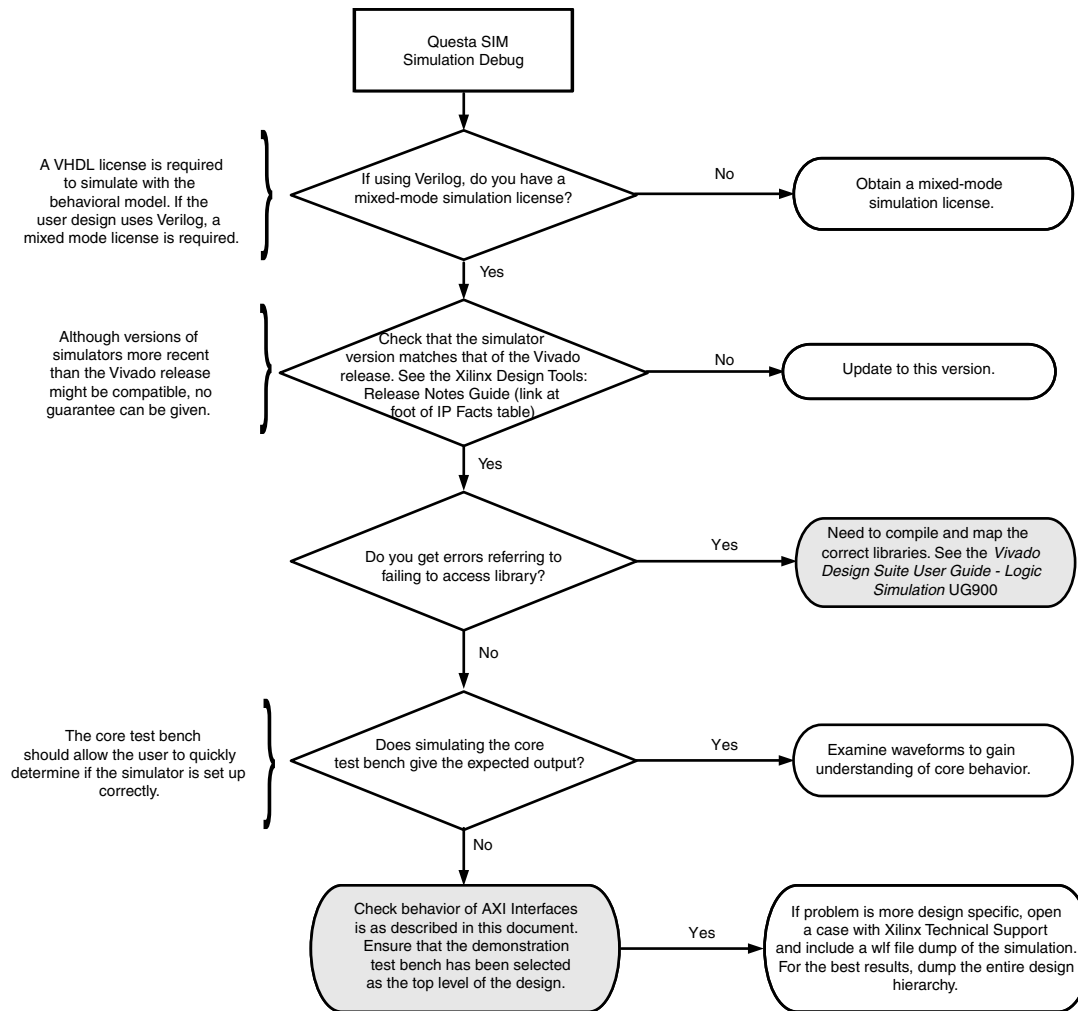
The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 9].

Simulation Debug

The simulation debug flow for Questa® SIM is illustrated in [Figure B-1](#). A similar approach can be used with other simulators.



AXI4-Stream Interface Debug

If data is not being transmitted or received, check the following conditions:

- If transmit `m_axi_output_tready` is stuck Low following the `<interface_name>_tvalid` input being asserted, the core cannot send data.
- If the receive `s_axi_input_tvalid` or `s_axi_ctrl_tvalid` is stuck Low, the core is not receiving data.
- Check that the `ACLK` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed as shown in the diagrams in [Chapter 2, Product Specification](#) and [Chapter 3, Designing with the Core](#).
- Check core configuration.
- Check that none of the event outputs have been asserted. These indicate incorrect input signal timing or invalid control data.

Additional Resources and Legal Notices

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

References

Unless otherwise noted, IP references are for the product documentation page.

1. *AMBA® AXI4-Stream Protocol Specification* ([ARM® IHI 0051A](#))
2. *Xilinx Vivado AXI Reference Guide* ([UG1037](#))
3. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* ([UG994](#))
4. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
5. *Vivado Design Suite User Guide: Getting Started* ([UG910](#))
6. *System Generator for DSP User Guide* ([UG640](#))
7. *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#))
8. *ISE to Vivado Design Suite Migration Guide* ([UG911](#))
9. *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#))
10. *Synthesis and Simulation Design Guide* ([UG626](#))

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/18/2015	9.0	<ul style="list-style-type: none"> Added support for UltraScale+ families. Updated link to resource utilization data.
04/02/2014	9.0	<ul style="list-style-type: none"> Added link to resource utilization figures Updated template
12/18/2013	9.0	Beginning in this release, the document revision number has been changed to match the core version number. Added UltraScale™ architecture support.
03/20/2013	2.0	Removed all ISE® and CORE Generator™ tool material. Also removed material related to devices not supported in Vivado.
01/18/2012	1.0	Initial Xilinx release. Previous data sheet for this core (non-AXI) is DS251.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>.

© Copyright 2012–2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. Simulink is a registered trademark of The MathWorks, Inc. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, and PrimeCell are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.