

Reed-Solomon Decoder v9.0

LogiCORE IP Product Guide

Vivado Design Suite

PG107 February 4, 2021



Table of Contents

IP Facts

Chapter 1: Overview

Navigating Content by Design Process	5
Core Overview	5
Functional Description	6
Feature Summary	6
Applications	7
Unsupported Features	7
Licensing and Ordering	7

Chapter 2: Product Specification

Performance	10
Resource Utilization	10
Port Descriptions	10

Chapter 3: Designing with the Core

Clocking	13
Resets	14
AXI4-Stream Protocol	14
Erasur e Decoding	23
Processing Delay	24
Latency	25
Puncturing	26
Variable Block Length	29
Block Code Settings	31
Multiple Channels	32
Examples	33

Chapter 4: Design Flow Steps

Customizing and Generating the Core	35
Constraining the Core	42
Simulation	43
Synthesis and Implementation	44

Chapter 5: Test Bench

Demonstration Test Bench	45
--------------------------------	----

Appendix A: Upgrading

Migrating to the Vivado Design Suite	47
Upgrading in the Vivado Design Suite	47

Appendix B: Debugging

Finding Help on Xilinx.com	48
Debug Tools	49
Simulation Debug	50
Interface Debug	51

Appendix C: Additional Resources and Legal Notices

Xilinx Resources	52
Documentation Navigator and Design Hubs	52
References	53
Revision History	53
Please Read: Important Legal Notices	54

Introduction

The Reed-Solomon decoder is used for Forward Error Correction (FEC) in systems where data are transmitted and subject to errors before reception.

Features

- High speed, compact Reed-Solomon Decoder
- Implements many different Reed-Solomon (RS) coding standards
- Fully synchronous design using a single clock
- Supports continuous input data with no gap between code blocks
- Symbol size from 3 to 12 bits
- Code block length variable up to 4095 symbols
- Code block length and number of check symbols can be dynamically varied on a block-by-block basis
- Supports shortened codes
- Supports error and erasure decoding
- Supports puncturing (as in IEEE 802.16d standard)
- Supports multiple channels
- Parameterizable number of errors corrected
- Supports any primitive field polynomial for a given symbol size
- Counts number of errors corrected and flags failures
- Marker bits provided with same latency as input data
- User-selectable control signal behavior

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family ⁽¹⁾	Versal™ ACAP UltraScale+™ Families UltraScale™ architecture Zynq®-7000 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 Provided
Simulation Model	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	
Release Notes and Known Issues	Master Answer Record: 54510
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
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

Navigating Content by Design Process

Xilinx[®] documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado timing, resource and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
 - [Port Descriptions](#)
 - [Clocking](#)
 - [Resets](#)
 - [Customizing and Generating the Core](#)
-

Core 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. In a systematic code, the complete code block is formed from the k data symbols, followed by the $n-k$ check symbols.

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 GUI allows both of these polynomials to be user-defined.

Functional Description

The Reed-Solomon decoder samples the n symbols on the S_AXIS_INPUT channel and attempts to correct any errors. The corrected symbols are output on the M_AXIS_OUTPUT channel.

The maximum number of symbol errors in a block that can be guaranteed to be corrected by the Reed-Solomon algorithm is $t = (n-k)/2$. (Each symbol error can contain any number of bit errors). This is always rounded down to the nearest whole number. The decoder core implements the Reed-Solomon algorithm in full, but if a block is received with more than t errors the decoder will fail. The Reed-Solomon decoder algorithm can generally detect that an excess of errors has occurred and can therefore indicate a failure to decode a block. However, it is possible for excessive errors to produce a codeword that the decoder algorithm recognizes as a legitimate lower number of errors, in which case the failure is not detected. This is a function of the Reed-Solomon algorithm and not a limitation of the core.

Shortened Codes

Normally, $n = 2^{(\text{Symbol Width})} - 1$. If n is less than this, the code is referred to as a "shortened code." The decoder core handles both full-length and shortened codes. Only n symbols are input and output, where n is the value entered in the Vivado® Integrated Design Environment (IDE) or supplied on the S_AXIS_CTRL channel. This is the case even if the code is shortened. Shortening does not affect k or the number of check symbols or the number of errors that can be corrected.

Feature Summary

The core configuration GUI allows a number of pre-configured standards to be selected. The CCSDS standard results in additional standard-specific hardware being inserted around the core. Once the appropriate standard has been selected it is still possible to edit some parameter settings, depending on the standard. If this doesn't give the required configuration then 'Custom' can be selected and any parameter values can be chosen.

Multiple channels can be supported by selecting the number of channels required in the GUI. This will result in time-division multiplexing of channels on the input and output. If this does not give sufficient throughput then 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 require the block length or the number of check symbols or both to be run-time variable. 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 GUI.

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. The core comes with some optional pins, such as clock enable. These allow extra functionality at the expense of larger size or slower speed.

Applications

The Reed-Solomon decoder (with the Reed-Solomon algorithm) is used for Forward Error Correction (FEC) in systems where data are transmitted and subject to errors before reception, for example, communications systems, disk drives, and so on.

The core meets the requirements of most standards that employ RS codes, such as CCSDS, DVB, ETSI-BRAN, IEEE802.16, G.709, IESS-308.

Unsupported Features

The field polynomial or symbol width cannot be changed during the operation of the core.

Licensing and Ordering

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. To generate a full license, visit the [product licensing web page](#). Evaluation licenses and hardware timeout licenses might be available for this core or subsystem. Contact your [local Xilinx sales representative](#) for information about pricing and availability.

For more information, visit the Reed-Solomon Decoder [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](#).



TIP: To verify that you need a license, check the “License” column of the IP Catalog. “Included” means that a license is included with the Vivado® Design Suite; “Purchase” means that you have to purchase a license to use the core or subsystem.

Evaluation

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.

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 tools:

- Vivado design tools: Vivado Synthesis
- Vivado Implementation
- write_bitstream (Tcl command)

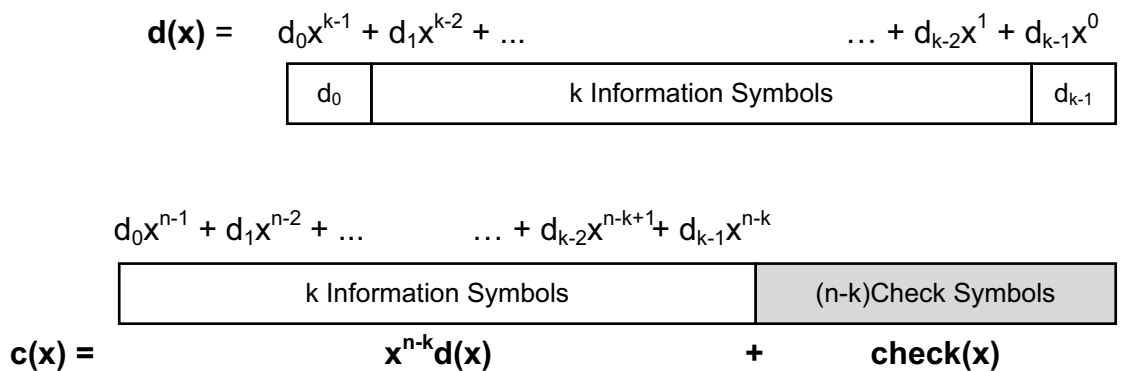


IMPORTANT: IP license level is ignored at checkpoints. The test confirms a valid license exists. It does not check IP license level.

Product Specification

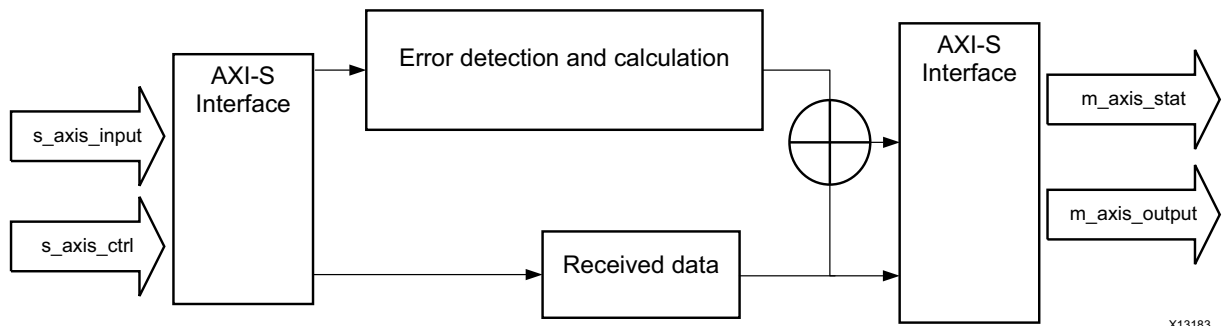
The Reed-Solomon Decoder inputs n symbols, comprising k information symbols and $n-k$ check symbols. This is shown in Figure 2-1, which also shows how the symbols can be interpreted as polynomial coefficients. The check symbols are generated to form $c(x)$ such that $c(x)$ is divisible by the generator polynomial, $g(x)$. If the received code word is not divisible by $g(x)$ then the code word contains errors. The core corrects the errors and outputs n symbols (or just k symbols if the output check symbol option is not selected).

Figure 2-2 shows how the input data is passed through the core while error values are computed. The error values are added to the received data to reconstruct the original transmitted data. The received data with errors still present can also be optionally output.



X13182

Figure 2-1: Reed Solomon Codeword Structure



X13183

Figure 2-2: Reed-Solomon Decoder Block Diagram

Performance

Latency

See [Latency, page 25](#).

Throughput

See [Processing Delay, page 24](#).

Resource Utilization

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

Port Descriptions

Some of the pins are optional. The outputs that are not required should be left unconnected. The Xilinx mapping software removes the logic driving them, ensuring that FPGA resources are not wasted.

A representative symbol, with the signal names, is shown in [Figure 2-3](#) and described in [Table 2-1](#). The AXI slave channel is indicated by s_* and the AXI master channel by m_*.

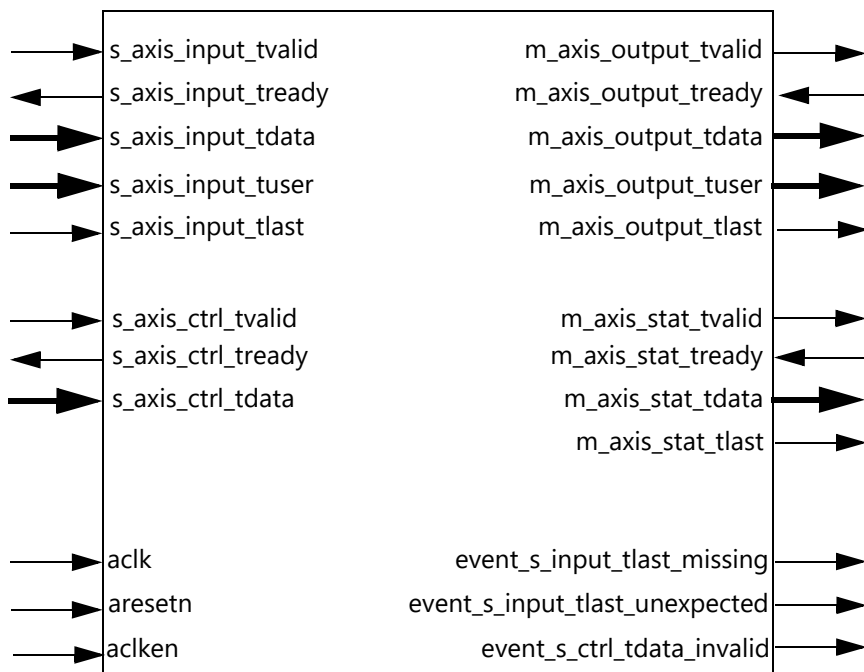


Figure 2-3: Core Schematic Symbol

Table 2-1: Core Signal Pinout

Signal	I/O	Optional	Description
aclk	I	No	Rising edge clock
aclken	I	Yes	Active-High clock enable
aresetn	I	Yes	Active-Low synchronous clear (overrides aclken)
s_axis_input_tvalid	I	No	TVALID for S_AXIS_INPUT channel. See AXI4-Stream Protocol for protocol.
s_axis_input_tready	O	No	TREADY for S_AXIS_INPUT
s_axis_input_tdata	I	No	Input data and erase flag, if applicable
s_axis_input_tuser	I	Yes	User bits, passed through core unmodified, with same latency as s_axis_input_tdata
s_axis_input_tlast	I	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	I	Yes	TVALID for S_AXIS_CTRL channel. This channel is only present if core has variable block length, number of check symbols or variable puncturing
s_axis_ctrl_tready	O	Yes	TREADY for s_axis_ctrl_channel
s_axis_ctrl_tdata	I	Yes	Block length, number of check symbols and puncture select, if applicable
m_axis_output_tvalid	O	No	TVALID for M_AXIS_OUTPUT channel

Table 2-1: Core Signal Pinout (Cont'd)

Signal	I/O	Optional	Description
m_axis_output_tready	I	No	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	O	No	Corrected data output
m_axis_output_tuser	O	Yes	s_axis_input_tuser delayed by core latency
m_axis_output_tlast	O	No	High when last symbol of last channel is on m_axis_output_tdata
m_axis_stat_tvalid	O	No	TVALID for M_AXIS_STAT channel
m_axis_stat_tready	I	No	TREADY for M_AXIS_STAT channel. Tie High if downstream slave is always able to accept data from M_AXIS_STAT, or if stat channel is not used
m_axis_stat_tdata	O	No	Status information for the last block processed
m_axis_stat_tlast	O	Yes	High when status information for the last channel is on m_axis_output_tdata. This output is only present in multichannel mode.
event_s_input_tlast_missing	O	No	Flags that s_axis_input_tlast was not asserted when expected. Leave unconnected if not required.
event_s_input_tlast_unexpected	O	No	Flags that s_axis_input_tlast was asserted when not expected. Leave unconnected if not required.
event_s_ctrl_tdata_invalid	O	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.

Designing with the Core

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

Clocking

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 3-1](#). In this case, the decoder ignores symbol `D4` as input to the block, and the current `m_axis_output_tdata` value remains unchanged. (The decoder 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.

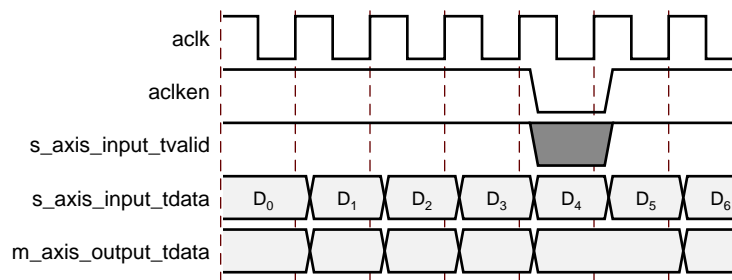


Figure 3-1: Clock Enable Timing

Resets

aresetn

The synchronous reset (`aresetn`) input is an optional pin. It can be used to re-initialize the decoder at any time, regardless of the state of `aclk`. `aresetn` needs to be asserted Low for at least two clock cycles to initialize the circuit. The decoder becomes ready for normal operation two cycles after `aresetn` goes High. 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 shown in Figure 3-2. Note that some outputs are not reset by `aresetn`.

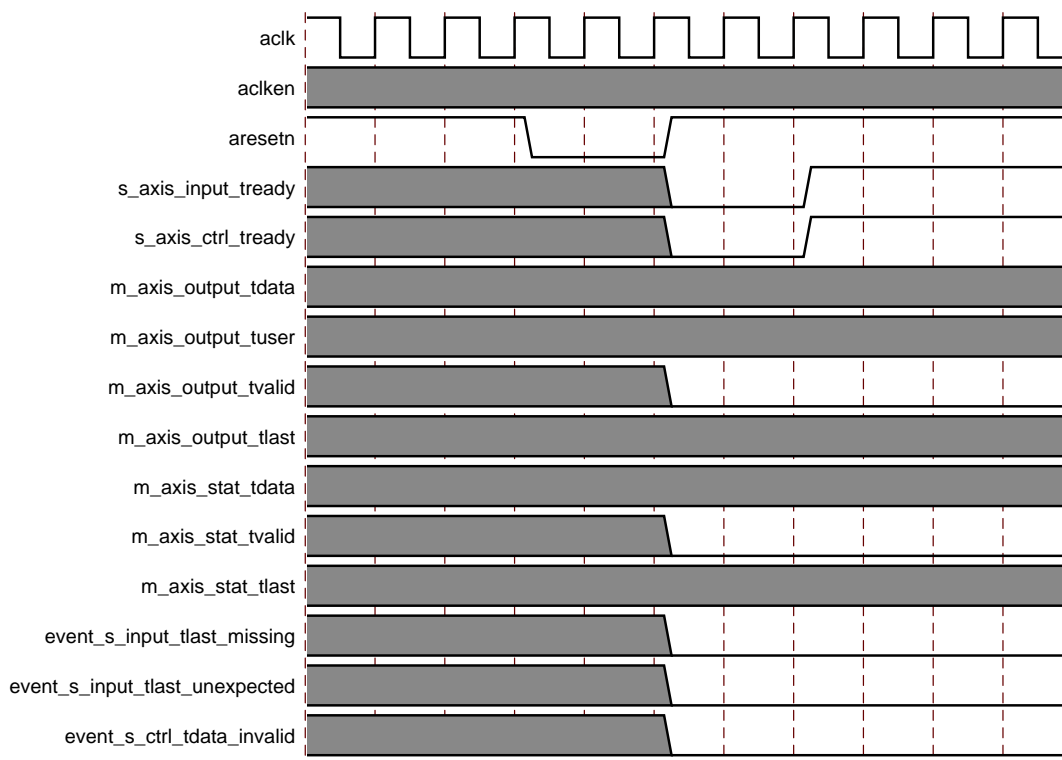


Figure 3-2: Synchronous Reset Timing

AXI4-Stream Protocol

The use of AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE™ solutions. Other than general control signals such as `aclk`, `aclk` and `aresetn`, and event outputs, all inputs and outputs to the core are conveyed via AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports and fields. In the RS Decoder 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 Decoder 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 Decoder 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 3-3 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.

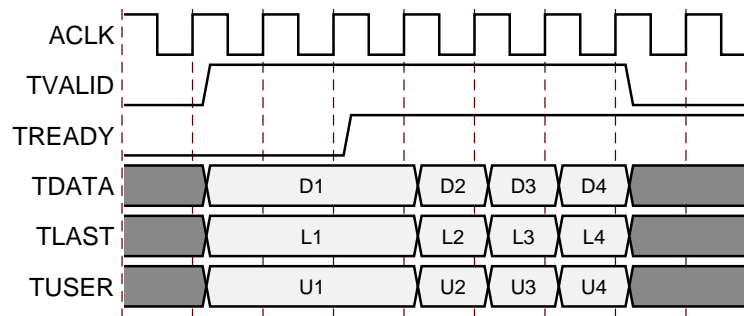


Figure 3-3: Data Transfer in an AXI-Stream Channel

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 two output channels: `M_AXIS_OUTPUT` and `M_AXIS_STAT`. If the output is prevented from off-loading data because `m_axis_output_tready` is Low then data accumulates in the core. When the core's internal buffers 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 de-asserted to prevent further input. This is the normal action of back pressure. One status value is output on `M_AXIS_STAT` for each block output on `M_AXIS_OUTPUT`. In multichannel mode a separate status value is output for each channel, with `m_axis_stat_tlast` indicating the last channel. If `m_axis_stat_tready` is Low and this status information is not read then the status information is buffered inside the core. When this buffer fills and the core needs to output more status information, the input channel is eventually blocked and `s_axis_input_tready` is deasserted. To prevent the output channel stalling, it is recommended to read the status information for a block before the status information for the next block is output.

S_AXIS_INPUT Channel

s_axis_input_tdata

Data to be processed is passed into the core on this port. The port is composed of a number of subfields, depending on parameter settings. To ease interoperability with byte-oriented buses, each subfield within 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 3-4](#).

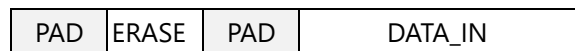


Figure 3-4: Input Channel TDATA Structure

DATA_IN Field

This is the input bus for the incoming Reed-Solomon coded data. The width of the DATA_IN portion of the field is set by the Symbol Width parameter in the GUI.

ERASE Field

This field is only present when erasure support is required. It only contains a single bit of information: the ERASE input. Erasure handling is described later in this document.

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 GUI. 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.

In general, using a small number of marker bits makes very little difference to the core size. However, a point is reached where extra marker bits cause more memory to be used. This point is dependent on the symbol width and latency.

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 a number of 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 3-5](#). 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.

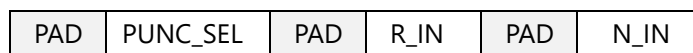


Figure 3-5: Control Channel TDATA Structure

N_IN Field

This field is only present if “Variable Block Length” is selected in the GUI. This allows the block length to be changed every block. Selecting this input significantly increases the size of the core. 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 3-7, 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 GUI, 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 GUI. It allows the number of check symbols to be changed every block.

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.

The value input on R_IN must correspond to the generator polynomial (and, hence, number of check symbols) used to encode the codeword. Some specifications appear to vary the number of check symbols, but in reality the codewords are all generated by the same generator polynomial, and the number of check symbols is varied by deleting some of them. The R_IN field should not be used in these cases. The PUNC_SEL field is provided to handle this.

PUNC_SEL Field

This field is only present if the number of puncture patterns is greater than one. It selects a puncture pattern to be applied to the code block. Puncturing is explained in [Puncturing](#), page 26.

M_AXIS_OUTPUT Channel

m_axis_output_tdata

Raw data with errors sampled on `s_axis_input_tdata` is corrected and output from the core on this port. The port is composed of a number of 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 3-6](#).



Figure 3-6: 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 at a number of clock cycles after the first symbol is sampled on `DATA_IN`. This delay is termed the latency of the decoder and is explained in [Latency](#). 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.

DATA_DEL Field

This optional output field is an uncorrected version of `DATA_OUT`. It is `DATA_IN` delayed by the latency of the core. `DATA_DEL` is useful for making comparisons of corrected and uncorrected data. This field always has the same width as `DATA_IN`.

This field can be compared to `DATA_OUT` to gather error statistics and examine the position of error bits. The positions of individual bit errors can be obtained by XORing `DATA_OUT` and `DATA_DEL`.

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`.

m_axis_output_tlast

This output is High when the last symbol of a block is on *m_axis_output_tdata*. This is either the *k*th symbol (if the “Output Check Symbols” option is not selected in the GUI) or the *n*th symbol (if the “Output Check Symbols” option is selected in the GUI) of the code word block. 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*.

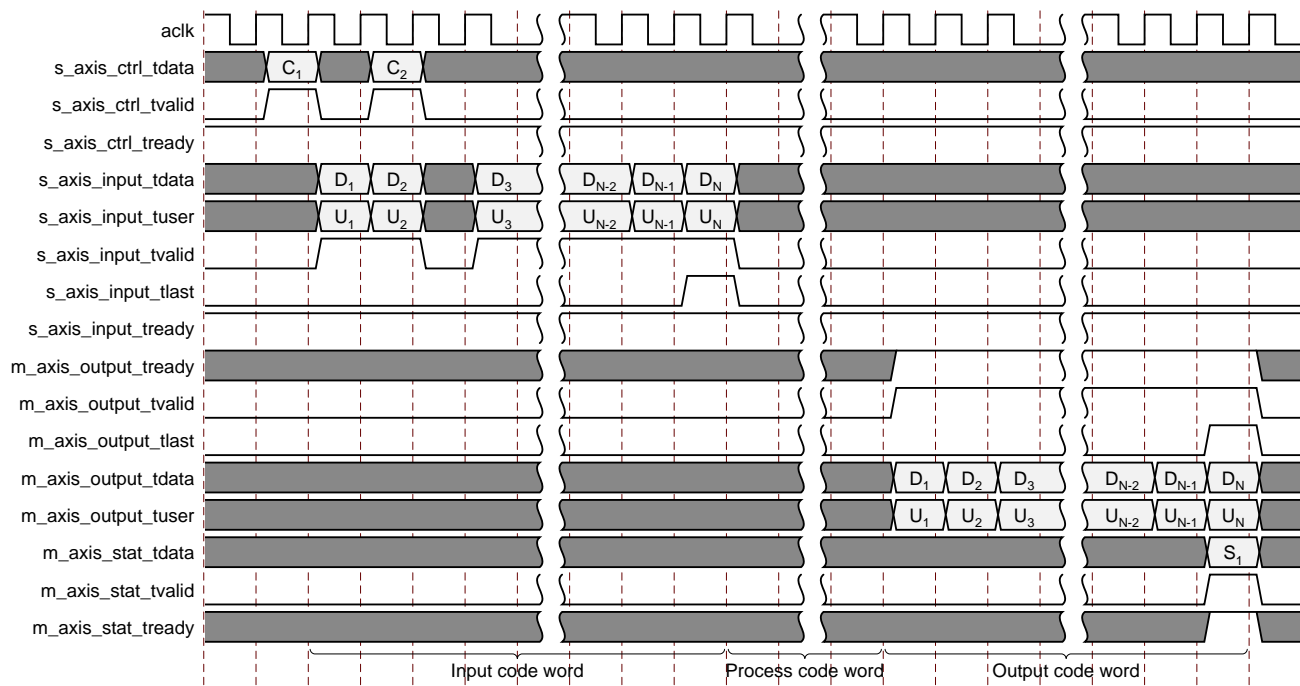


Figure 3-7: Block Input to Output Timing

M_AXIS_STAT Channel

m_axis_stat_tdata

Status information for the block just output is provided on this port. One status word is provided for each output block, one word for each channel in the multichannel case. The status word is output after the last symbol has been processed *inside* the core. The status word(s) must be read before the core needs to write more status information to its internal buffer or the input channel is eventually blocked. If the status channel is not required then *m_axis_stat_tready* should be tied High.

There is a parameter-dependent latency on both the output data and the status output. These latencies are different. For some parameter combinations the status output can be output before the last output *tdata*. They are independent AXI4-Stream channels, so their timing is decoupled and the status information for the previous block might not even have

been read when the last value is output on `m_axis_output_tdata`. There is always one set of status information for each block.

The port is composed of a number of elements, depending on parameter settings. The port is padded with zeroes to be a multiple of 8 bits. The elements are always packed into the least significant bits. For example, if erasures are not required there is no `ERASE_CNT` element and `BIT_ERR_0_TO_1` abuts `ERR_CNT`, assuming 'Error Statistics' is selected in the GUI. The structure is shown in [Figure 3-8](#).

PAD	BIT_ERR_1_TO_0	BIT_ERR_0_TO_1	ERASE_CNT	ERR_CNT	ERR_FOUND	FAIL
-----	----------------	----------------	-----------	---------	-----------	------

Figure 3-8: Stat Channel TDATA Structure

FAIL Element

The decoder sets `FAIL` High if it determines that there were more errors in the code block than it could correct. In this case, `ERR_FOUND`, `ERR_CNT`, `ERASE_CNT`, `BIT_ERR_0_TO_1` and `BIT_ERR_1_TO_0` status outputs are now undefined and should not be relied upon until `FAIL` goes Low again.

With Reed-Solomon codes, if the error correcting capacity of the code is exceeded, it is usually possible to detect this and assert `FAIL`. However, there might be some cases where it is impossible. For example, consider a (5,1) code. This code can correct up to two symbol errors. Any more than two symbol errors should result in a failure. Assume the transmitted codeword symbol sequence was [a, b, c, d, e]. Also assume that [g, h, i, j, k] is another legitimate codeword. Suppose the received codeword is [a, b, i, j, k]. This contains three symbol errors; however, this is the same as [g, h, i, j, k] with two symbol errors.

The decoder corrects this to yield [g, h, i, j, k], and `FAIL` is not asserted. This is a function of the codes themselves and not the decoder implementation. As the block sizes become larger, it is extremely unlikely that one codeword will be converted into another, and `FAIL` generally detects that the correction capacity of the code has been exceeded.

If the error correction capacity of the code is exceeded in a particular code block, then the values on `DATA_OUT` when that block is output are undefined.

ERR_FOUND Element

If the decoder detected any errors, erasures, or punctures in the code block, `ERR_FOUND` is High. If no errors, erasures, or punctures are found, `ERR_FOUND` is Low.

ERR_CNT Element

The `ERR_CNT` element gives the number of errors, erasures, and punctures that were corrected. The width of the element depends on the input parameters n and k . The width is equal to the number of binary bits required to represent $(n-k)$. If $n-k = 16$, for example, the `ERR_CNT` element is five bits wide.

If decoding fails, then FAIL is asserted and the ERR_CNT value cannot be relied upon.

ERASE_CNT Element

This element is only included when erasure or puncture support is required. The element width is equal to the number of binary bits required to represent n . Erasure handling is described later in this document.

BIT_ERR_0_TO_1 Element

This element is only included when 'Error Statistics' is selected in the GUI. It gives the number of bits that were received as 1 but corrected to 0 in the block. As long as the error correction capability of the code has not been exceeded, this is the same as the number of 0 bits that were corrupted to 1 during transmission. The element width is the number of binary bits required to represent $((n-k) * Symbol_Width)$.

BIT_ERR_1_TO_0 Output

This element is included when BIT_ERR_0_TO_1 is included. It has the same functionality and width as BIT_ERR_0_TO_1, except it counts the number of bits received as 0 but corrected to 1.

m_axis_stat_tlast

This output is only driven in the multichannel case. It is asserted High when *m_axis_stat_tdata* holds the information for the last channel. This is illustrated in Figure 3-9.

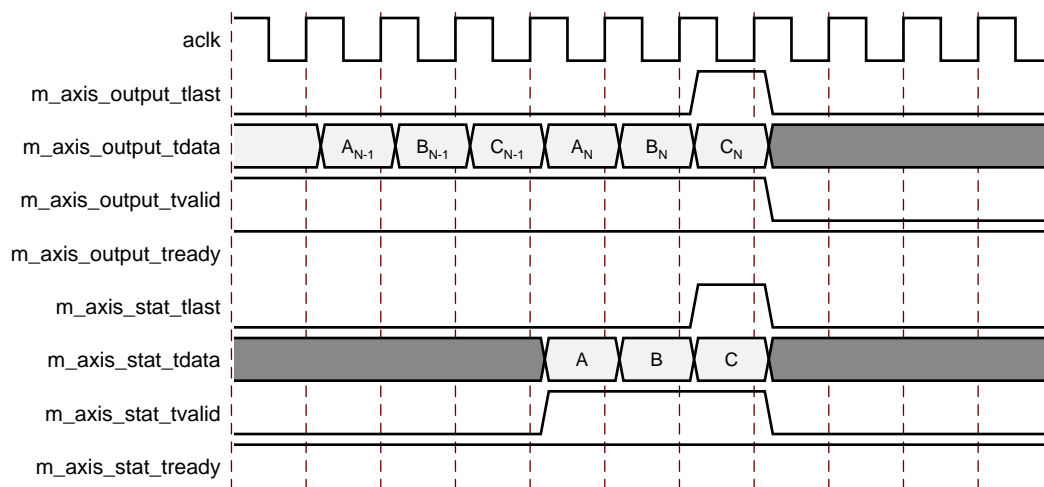


Figure 3-9: TLAST Timing for 3 Channel Example

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. 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 4-2](#).

Erase Decoding

An erased symbol is an input symbol that is known to be wrong. The symbol is flagged as being erased by asserting the `ERASE` input High while the symbol is being sampled. In the example shown in [Figure 3-10](#), `D2` is flagged as an erasure.

The decoder corrects the code block if $2e + E \leq n - k$, where e is the number of errors and E is the number of erasures.

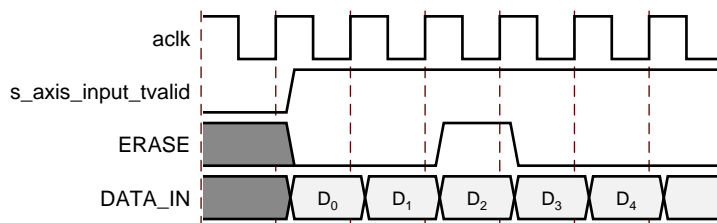


Figure 3-10: ERASE Timing

The `ERASE_CNT` output provides a count of the number of erasures that were flagged for the block just output. It is updated at the same time as `ERR_CNT` and the other status outputs. If erasure decoding is selected, `ERR_CNT` provides a count of the number of erasures plus errors that were corrected.

Erasure decoding increases the size of the core considerably. It should be selected only if it is essential as there is a large area overhead compared to the same core without erasure support. See [Examples](#).

Processing Delay

The core inputs a block, processes it and outputs the corrected block. The times to input and output the block are dependent on the block length. The time to process the block is dependent on the number of check symbols, $(n-k)$. The Processing Delay (PD) in clock cycles, for a given t , is shown in [Figure 3-11](#). The Processing Delay should not be confused with latency. It is a component of the latency. Processing delay is important because it determines if blocks can be indefinitely input without pause.

The core can still accept a new code block immediately after the previous one has been sampled, even if the Processing Delay is greater than n , due to its internal buffering. However, if new blocks are continually fed to the decoder with n less than PD, at some point it is unable to accept a new code block and `s_axis_input_tready` is deasserted. If PD is less than or equal to n then blocks can be input continuously, without pause, providing the output is not stalled by deasserting one of the output channel TREADY inputs. The timing is described in [Variable Block Length, page 29](#).

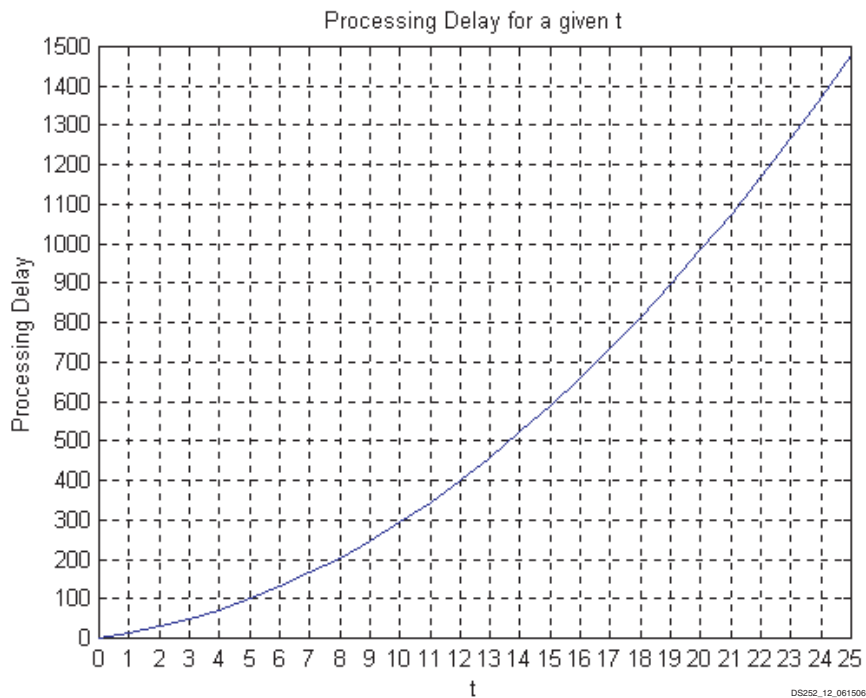


Figure 3-11: Processing Delay against t, where $t = (n-k)/2$

The number of clock cycles can be calculated using Equation 3-1:

$$PD = 2t^2 + 9t + 3 \tag{Equation 3-1}$$

If erasure decoding or puncturing is enabled, Equation 3-2 should be used:

$$PD = (n - k)^2 + 6(n - k) + 4 \tag{Equation 3-2}$$

If $PD \leq n$ then the maximum throughput is equal to the clock frequency * symbol width Mb/s. If $PD > n$ then maximum throughput is approximately (n/PD) * clock frequency * symbol width Mb/s.

Latency

The latency is the number of clock edges from a symbol being sampled on DATA_IN to the corrected version of that symbol appearing on DATA_OUT. An example, with a latency of three, is shown in Figure 3-12. In reality, the latency is usually much greater than this.

The latency is dependent on the values of n (the number of symbols in a code block), t (the number of correctable errors), whether erasures or puncturing are selected, symbol width, number of channels and code specification. The GUI computes the actual latency based on the entered parameters and displays the value on the last page.

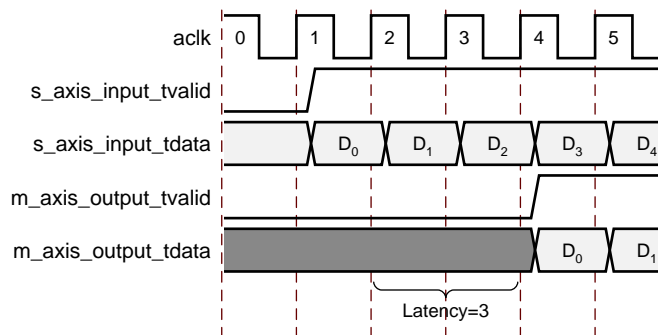


Figure 3-12: Latency = 3

Latency can vary if the processing delay is greater than the block length. The core can buffer data for the next blocks while still processing previous blocks. Internally it has to wait for the previous block to be processed, increasing latency on the new block. If n is always greater than the processing delay or the number of cycles taken from the start of one block to the start of the next is greater than the processing delay then the latency calculation in Table 3-5 is valid.

The following equations are used to calculate the latency, where $r=n-k$ and $t=r/2$.

If erasure decoding or puncturing is enabled use [Equation 3-3](#):

$$L = (r^2 + 6r + 5 + n) \times \text{num_channels} + 10 \quad \text{Equation 3-3}$$

otherwise use [Equation 3-4](#):

$$L = (2t^2 + 9t + 4 + n) \times \text{num_channels} + 10 \quad \text{Equation 3-4}$$

The latency is then defined in [Equation 3-5](#):

$$\text{Latency} = L + m + nc + c + ev \quad \text{Equation 3-5}$$

where m , nc , c , and ev are defined in [Table 3-1](#).

Table 3-1: Latency Variables

		m	nc	c	ev
Symbol Width	= 8	1			
	≠ 8	0			
Number Channels	> 1		1		
	≤ 1		0		
Code Specification	= CCSDS			2	
	≠ CCSDS			0	
(Erasures AND Variable Block Length) OR (CCSDS selected)	≠ 0				1
	= 0				0

Puncturing

Puncturing can be thought of as erasure decoding where the erasure positions are known prior to the block being received. For example, in the IEEE802.16d standard, the RS codeword always has 16 check symbols; however, some of those symbols might not be transmitted. If only the first 12 check symbols are transmitted, the number of errors that can be corrected is reduced from 8 to 6. The decoder still decodes as if there were 16 check symbols. The last 4 check symbols are sampled, but ignored. One way of handling this is to flag the last 4 symbols of the block as erasures; however, the complexity of the full erasure decoding logic is not required. It is possible to define the known erasure positions in a file when generating the core. The core then automatically compensates for the deleted symbols. Erasure decoding must be unselected if puncturing is required. If both puncturing and erasure decoding are required, then the puncturing must be handled externally by asserting the ERASE input at the appropriate time.

As far as the core is concerned, the length of the block (n) still includes the punctured symbols. So for variable N codes, the value sampled on N_IN must include the number of

punctured symbols. For example, IEEE802.16d specifies a (120,108,6) code, that is, $n=120$, $k=108$, and $t=(n-k)/2=6$. It would seem this code has only 12 check symbols, but it is actually a 16 check symbol code with 4 punctured check symbols. Therefore, the real value of n is 124 and N_IN must be set to 124 to allow for the 4 dummy symbols that are sampled after the 120 real symbols.

The PUNC_SEL field can be used to select between a number of predefined puncture patterns. The number of puncture patterns is set in the core GUI. If this is fewer than two, then PUNC_SEL is not required. If it is greater than zero, then a puncture definition file must be supplied to define the puncture patterns. For example, the file for IEEE802.16d is as follows:

```
radix=10;
puncture_select_vector= 0,4,8,12;
puncture_vector=       0,1,2,3,
                       0,1,2,3,4,5,6,7,
                       0,1,2,3,4,5,6,7,8,9,10,11;
```

In this example, there are four possible puncture patterns. The number of symbols to be punctured from a block is defined in the `puncture_select_vector`. The number of symbols punctured for each PUNC_SEL value in this example is shown in [Table 3-2](#).

Table 3-2: puncture_select_vector Example

PUNC_SEL	Number of Symbols Punctured
0	0
1	4
2	8
3	12

The `puncture_select_vector` entries can be in the range 0 to $n-k$. This is because the maximum number of punctured symbols that can be recovered is $n-k$.

The `puncture_vector` defines which symbols are punctured for each of the `puncture_select_vector` entries. In the previous example, there are no entries for PUNC_SEL=0, as the `puncture_select_vector` has defined 0 symbols to be punctured in this case. If PUNC_SEL=1, then the `puncture_select_vector` has defined that four symbols are to be punctured. The first four entries of the `puncture_vector` define the symbol positions. The entries count back from the last symbol in a block, with 0 being the last symbol. Thus if PUNC_SEL=1, symbols 0, 1, 2 and 3 are all punctured, that is, the last four symbols in the block. If PUNC_SEL=2, then the last eight symbols in the block are punctured. If PUNC_SEL=3, then the last twelve symbols in the block are punctured.

The number of entries in the `puncture_vector` must equal the sum of the entries in the `puncture_select_vector`.

Each `puncture_vector` entry must be less than n . If n is variable, then the selected `puncture_vector` entry for a given block must be less than the value sampled on `N_IN`.

If the number of puncture patterns is not a power of two and an *illegal* `PUNC_SEL` value is sampled, then the punctured pattern applied by the core is not defined. For example, if the number of puncture patterns was set to 3, then only 0, 1, and 2 are legal values for `PUNC_SEL`. A value of 3 should never be sampled on `PUNC_SEL`.

Timing for the IEEE802.16 example is shown in [Figure 3-13](#). `PUNC_SEL` is set to 1 using the control channel. The `puncture_select_vector` defines this as four punctured symbols. The `puncture_vector` specifies that the last four symbols of the block are to be punctured (symbols 0, 1, 2, and 3). The core still samples `DATA_IN` for the four punctured symbols. Dummy symbols must be provided to the core in the puncture positions, as shown in [Figure 3-13](#). The value on `DATA_IN` is irrelevant at this time. D_{N-4} is the last real symbol received.

The decoder actually determines the values of the punctured symbols and outputs them in the correct sequence. The `ERASE_CNT` element of S_1 shows how many symbols were punctured. The `ERROR_CNT` element shows the number of errors plus the number of punctures. So, if there were no errors in the block, `ERROR_CNT` would be 4 and `ERASE_CNT` would be 4 as well. The number of true errors is `ERROR_CNT - ERASE_CNT`.

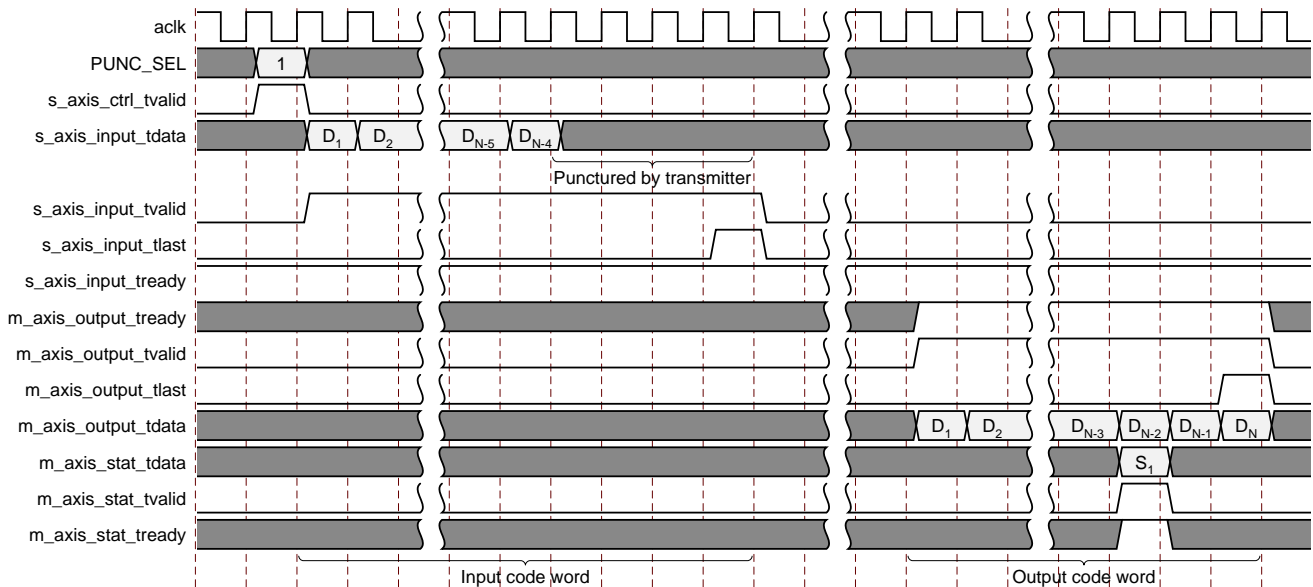


Figure 3-13: Puncture Timing

If puncturing is used, the latency and Processing Delay are derived from the same equation as if erasures were enabled. See [Processing Delay](#).

Variable Block Length

If the `N_IN` field of `s_axis_ctrl_tdata` is used, the block length can be different for every new block. `N_IN` can vary within the ranges shown in Table 4-2. It must also be greater than the number of check symbols, r . The number of check symbols can be fixed or variable, depending on whether “Variable Number of Check Symbols” is selected in the GUI.

When variable block length is used the latency and Processing Delay for each individual block are the same as for the fixed block core and can be checked in the core GUI. The values sampled on `N_IN` and `R_IN` can be used in the equations to compute the Processing Delay for each block sampled. The Processing Delay depends only on the number of check symbols.

A block might actually take longer to appear on `DATA_OUT` than the calculated latency, as an earlier, larger block might still be being processed. Figure 3-14 shows an example where the Processing Delay is greater than the block length. In this example block `N2` is input while block `N1` is still being processed. Block `N2` cannot be processed until processing has completed for block `N1`. It is buffered until the Processing Delay for `N1` completes. `N3` is also input and buffered. The start of `N4` is also input but the input FIFO fills at this point and `s_axis_input_tready` is deasserted. When the processing of `N1` has completed, processing of `N2` begins and `N3` is prepared for processing. This preparation takes r cycles. When this is complete the rest of `N4` can be loaded.

Note that if all the block lengths had been greater than or equal to the Processing Delay then `s_axis_input_tready` would not have been deasserted.

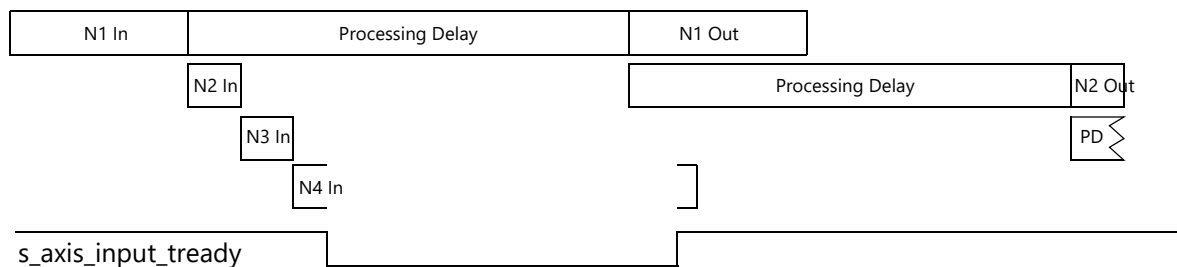


Figure 3-14: Variable n – TREADY Operation

The core always samples data and outputs results as soon as possible. If the Processing Delay is not greater than any sampled block size there are never gaps between output blocks. However, it is possible for `s_axis_input_tready` to go Low, even if the Processing Delay is not greater than all the sampled block sizes. This can happen if a large block is followed by many relatively small blocks: see Figure 3-15 for example. Because the large block (`N1`) takes a long time to output, the small input blocks start to back up inside the core. Thus, the input data might need to be temporarily held up, as in Figure 3-15, because a large block was followed by many small ones. An additional FIFO could be placed

in front of the core to smooth out these effects if necessary. The input symbols to the FIFO would not need to have any gaps, and there would never be any gaps between output symbols from the core.

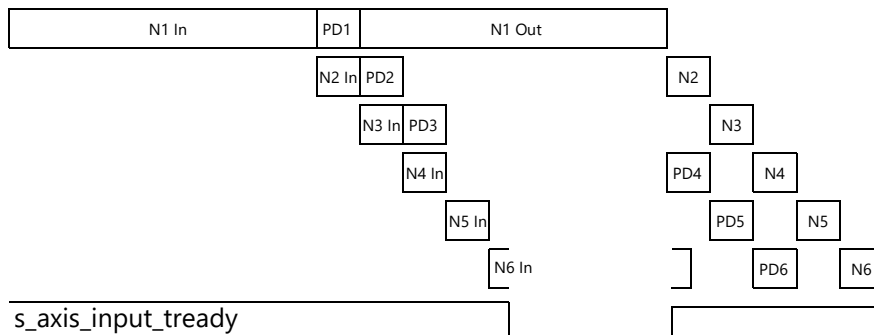


Figure 3-15: Processing Delay Buffer

Note that the overall latency for block N2 is larger than predicted by the latency equation. This is because it had to wait for N1 to be output.

The processing section can also buffer results for up to two blocks. This is shown in the example in Figure 3-15. As the processing for N2 (that is, PD2) completes, the core is still outputting N1, so it stores the PD2 results in a buffer. These results are then used when outputting N2. PD3 completes before the PD2 results have been used, so the PD3 results are also buffered. The processing buffer can hold two blocks, so it is now full, and PD4 cannot begin until the PD3 results have been unloaded. N5 can still be input as the core can hold just over two complete blocks of data (or more if the blocks are extremely small) in its input stage. PD4 can begin when the N2 values begin to be output, as this is when the PD2 values are unloaded from the processing buffer. PD4 then begins and completes while the core is still outputting values from earlier blocks so its results are buffered until N3 has been output. PD5 can begin immediately after PD4, because the processing buffer can hold two blocks. The first few samples of block N6 can be loaded into the core input stage but the input stage already contains N4 and N5, so the input FIFO soon fills up and `s_axis_input_tready` goes Low. The input buffer frees up again as the N4 values are fed into the processing section at the start of PD4 and the rest of N6 can be read in.

Note that there is no gap between the output blocks, even though there were gaps at the input side. This is because the core always outputs results as soon as possible.

Figure 3-14 and Figure 3-15 are a slight simplification of what actually happens, but they serve to illustrate the core behavior. For example, there are some small fixed latencies between the input section, the processing section, and the output section.

Block Code Settings

The core decodes a systematic (n_block , k_block) block code, where the input 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 3-3](#).

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

Block Code Settings	Value ⁽¹⁾
Fixed Block Length, Fixed Number of Check Symbols	
n_block	$n^{(2)}$
k_block	$k^{(2)}$
r_block	$(n-k)$
Fixed Block Length, Variable Number of Check Symbols	
n_block	n
k_block	$n - R_IN$
r_block	R_IN
Variable Block Length, Fixed Number of Check Symbols	
n_block	N_IN
k_block	$N_IN - (n-k)$
r_block	$(n-k)$
Variable Block Length, Variable Number of Check Symbols	
n_block	N_IN
$k_block^{(3)}$	$N_IN - R_IN$
r_block	R_IN

Notes:

1. The minimum and maximum values are defined in [Table 4-2](#).
2. n and k are the values set in the GUI.
3. Set k in GUI so that $(n-k)$ equals the largest value the core needs to handle on R_IN . For example, if $n=255$ and the largest legal R_IN value is 20, then set k to 235.

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 written on the S_AXIS_CTRL channel prior to each new block.

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 and a variable number of check symbols is not required, *k_block* is set to the parameter *k* for every block.
- When a variable block length is not required and a variable number of check symbols is required, *k_block* is set to the parameter *n* minus the value written on R_IN.
- 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 on N_IN minus the parameter (*n-k*).
- When a variable number of check symbols is required, *k_block* is set to the value written on N_IN minus the value written 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 written on R_IN prior to each new block.

Multiple Channels

The core can process multiple input channels simultaneously with a relatively small increase in the number of LUTs used. There is a larger increase in the number of registers used. A multichannel core generally runs at a higher clock frequency than a single-channel core. Using one multichannel core in a high-speed application can be more efficient than instantiating several single-channel RS decoder cores. Multichannel is available only for fixed *n* and *r* decoders. All channels have the same code parameters.

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*, *k*, etc.) are the same for all channels. If puncturing is used, then a single PUNC_SEL value that applies to all channels is written on S_AXIS_CTRL.

With multiple channels, there is still only one S_AXIS_INPUT channel. Incoming symbols for the channels are interlaced, so that 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, assuming s_axis_input_tvalid and s_axis_input_tready are asserted. Symbols (both information and check) are output on m_axis_output_tdata in the same sequence. An example with three channels is shown in [Figure 3-16](#).

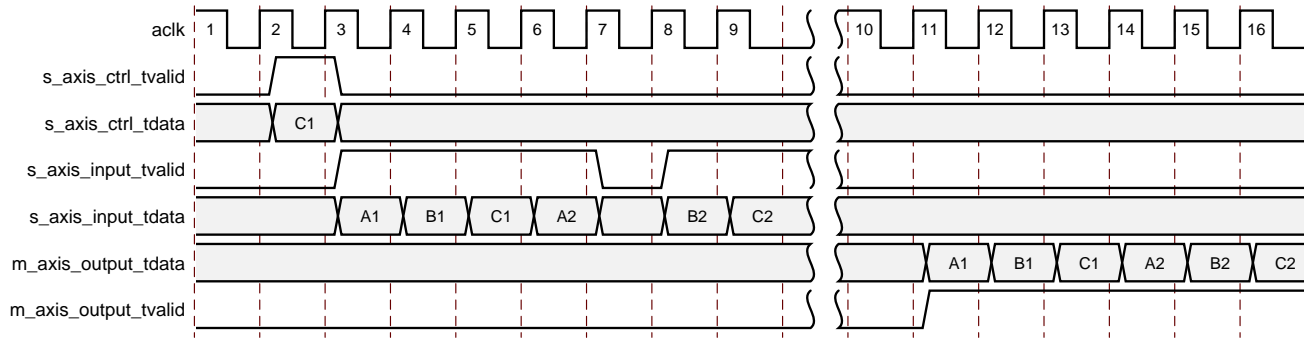


Figure 3-16: Multi-Channel Operation

A single control value (C1) is written. This is only required if there is more than one puncture pattern and sets PUNC_SEL for all three channels. 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.

If erasures are enabled, then the ERASE field can be asserted at any time for each channel independently. Symbols on m_axis_output_tdata are interlaced in the same way as symbols on s_axis_input_tdata. The timing for the output of the end of the block and the status channel is shown in Figure 3-9, page 22.

The Processing Delay is the single-channel Processing Delay multiplied by the number of channels. This must be less than or equal to n multiplied by the number of channels for continuous input of code blocks with no input stalling.

The latency is multiplied by an amount roughly proportional to the number of channels. See the GUI for the exact latency value for a given set of parameters.

Examples

Example 1:

Symbol Width = 8
 Symbols per Block (n) = 255
 Data Symbols (k) = 239

RS(255,239) is a configuration of 255 symbols, including 239 8-bit data symbols. This code is capable of correcting 8 symbol errors, that is, up to 64 bit errors. The Processing Delay is 203 cycles, which is less than 255, so this configuration is capable of continuous processing and the throughput in Mb/s is 8 times the clock frequency (MHz).

Example 2:

Symbol Width = 8
 Symbols per Block (n) = 255
 Data Symbols (k) = 229

RS(255,229) is a configuration of 255 symbols, including 229 8-bit data symbols. This has a greater error correcting capability than Example 1, in that 13 symbols, or 104 bits of data, can be corrected. However, as the Processing Delay is 458 cycles, and is therefore greater than 255, continuous processing cannot be done.

Maximum throughput is approximately $(255/458) * 8 * \text{clock frequency}$.

Example 3:

Symbol Width = 12
 Symbols per Block (n) = 400
 Data Symbols (k) = 376

The requirement is to be able to detect and correct a minimum of 3% of the symbols in a block of 12-bit data and have continuous operation. As this is 12-bit data, the maximum number of symbols in the block is 4095, and to meet the correction criteria the configuration would be RS(4095,3849). The Processing Delay (31369 symbol periods) would be prohibitive due to the $n-k$ value of 246.

The solution could be to use a shortened code. If RS(400,376) was used, this would correct 3% within the 400 symbols block. The Processing Delay is 399, so continuous code blocks are possible.

Example 4:

Symbol Width = 8
 Symbols per Block (n) = 255
 Data Symbols (k) = 239
 Variable Block Length Checked
 Variable Number of Check Symbols Checked

In this case there is a requirement to vary the number of symbols and the number of check symbols in the block. The symbol width is 8 bits, so n must be set to 255, or less. The largest expected R_IN value is 16, so k must be set to $n-16=239$. This gives an R_IN field width of 5 bits, plus 3 padding bits.

So N_IN can have a value up to 255 and R_IN can have a value up to 16. Lower limits are defined in [Table 4-2](#).

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) [\[Ref 4\]](#)
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [\[Ref 6\]](#)
- *Vivado Design Suite User Guide: Getting Started* (UG910) [\[Ref 7\]](#)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [\[Ref 8\]](#)

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) [\[Ref 4\]](#) for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value you can 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) [\[Ref 6\]](#) and the *Vivado Design Suite User Guide: Getting Started* (UG910) [\[Ref 7\]](#).

The core GUI provides a number of preset parameter values for several common Reed-Solomon standards. It also allows the user to define the following parameters.

Code Block Specification Parameters

Code Specification (including CCSDS)

The GUI aids creation of cores for a number of common Reed-Solomon specifications. Upon selecting a particular specification, the GUI automatically selects the parameter values necessary to meet the specification.

When implementing the CCSDS specification, the core automatically implements the dual-basis conversions defined in the CCSDS specification. This is shown in [Figure 4-1](#). If the dual-basis conversions are not wanted, select *custom* specification instead of *CCSDS* and enter all the code parameters manually. Short CCSDS codes are also supported by selecting the appropriate values of n and k from the GUI. If *IEEE 802.16d* is selected, then the GUI uses a predefined COE file to define the required puncture patterns. This file can be modified if required.

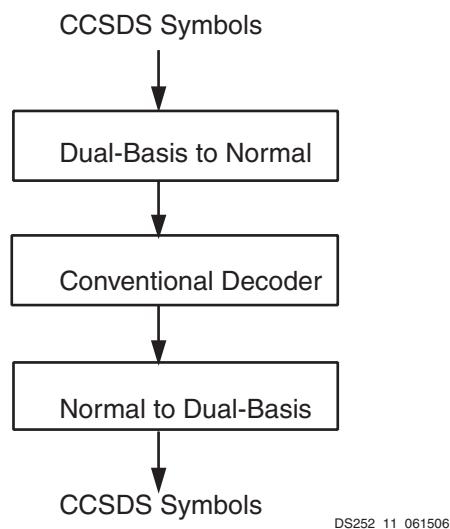


Figure 4-1: **CCSDS Decoder**

Symbol Width

This is the width of `DATA_IN` and `DATA_OUT`.

Field Polynomial

This is the Galois Field polynomial, used to generate the Galois Field for the code. Polynomials are entered as decimal numbers. The bits of the binary equivalent correspond to the polynomial coefficients. For example,

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

A value of zero causes the default polynomial for the given symbol width to be selected.

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)

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

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

Generator Start

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

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

Normally, Generator Start is 0 or 1; however, the core accepts other values.

Variable Block Length

This is selected when the N_IN field is required in `s_axis_ctrl_tdata`.

Symbols Per Block (n)

This is the number of symbols in an entire code block. If this is a shortened code, n should be the shortened number.

Data Symbols (k)

This is the number of information or data symbols in a code block. If the core has an `N_IN` or `R_IN` input, then k is used to specify the maximum number of check symbols supported. For example, if $n=255$ and $k=239$, then there can be a maximum of 16 check symbols.

Variable Number of Check Symbols

This is selected when the `R_IN` field is required in `s_axis_ctrl_tdata`. Take care that this is actually required, and variable check symbols are not to be implemented using puncture patterns.

Define Supported R_IN Values

If only a subset of the possible values that could be sampled on `R_IN` is actually required, then it is possible to reduce the size of the core slightly. For example, for the Intelsat standard, the `R_IN` input is 5 bits wide but it only requires r values of 14, 16, 18, and 20. The core size can be slightly reduced by defining only these four values to be supported. If any other value is sampled on `R_IN`, the core does not decode the data correctly.

Number of Supported R_IN Values

If “Define Supported `R_IN` Values” has been selected, then the number of supported `R_IN` values must be entered.

Supported R_IN Definition File

This is a COE file that defines the R values to be supported. It has the following format:

```
radix=10;

legal_r_vector=14,16,18,20;
```

The number of elements in the `legal_r_vector` must equal the “Number of Supported `R_IN` Values” set in the GUI.

Implementation Parameters

Self-Recovering

Selecting this option causes extra logic to be generated in the core to detect if the controlling state machine has entered an illegal state. This should never happen; however, in some systems illegal timing conditions can be generated by switching clocks outside of the core, for example. If the core is not reset after a violation like this, then it might end up in an illegal state. If this is detected, then the core automatically synchronously resets itself.

Selecting this option means that all the logic to handle synchronous reset is included in the core.

Memory Style

The following options are available:

- *Distributed* – The core should not use any block memories if possible. This is useful if they are required elsewhere in the design. For symbol widths of 8 and under, this option results in no block memories being used. For symbol widths greater than 8, some block memories are used, but their use is kept to a minimum.
- *Block* – The 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 option allows the core to use the most appropriate style of memory for each case, based on required memory depth.

Number of Channels

This parameter defines how many channels the core should support. Multichannel operation is described in [Multiple Channels, page 32](#).

Output Check Symbols

If selected, then the entire n symbols of each block are output on the M_AXIS_OUTPUT channel. If not selected, then only the k information symbols are output.

Puncture Options

Number of Puncture Patterns

This defines how many puncture patterns the core needs to handle. It is set to 0 if puncturing is not required, which is explained in [Puncturing, page 26](#). This parameter is not available if erasures are selected. The puncturing can be handled externally by asserting the ERASE input in this case.

Puncture Definition File

This is the .coe file that defines the punctured symbol positions within a block for each PUNC_SEL value. This is explained in [Puncturing, page 26](#).

Optional Pins

Clock Enable

This is selected when the `aclken` input is required.

Synchronous Reset

This is selected when the `aresetn` input is required.

Erase

This is selected when erasure support is required. See the explanation in [Erasure Decoding, page 23](#).

Info

This is selected when the INFO field is required in `m_axis_output_tdata`. This option is not available if "Output Check Symbols" is de-selected, as it is redundant in that case.

Original Delayed Data

This is selected when the `DATA_DEL` field is required in `m_axis_output_tdata`.

Error Statistics

This is selected when the `BIT_ERR_0_TO_1` and `BIT_ERR_1_TO_0` elements are required in `m_axis_stat_tdata`.

Marker Bits

This is selected when `s_axis_input_tuser` and `m_axis_output_tuser` are required.

Number of Marker Bits

This sets the width of `s_axis_input_tuser` and `m_axis_output_tuser`.

Parameter Ranges

Valid ranges for the parameters are given in [Table 4-2](#).

Table 4-2: Parameter Ranges

Parameter	Min	Max	Notes
n	5	$2^{(\text{Symbol_Width})}-1$	[1]
k	1	$2^{(\text{Symbol_Width})}-3$	[2]
h	1	$2^{(16)}-1$	
Polynomial	0	$2^{(13)}-1$	
$r=n-k$	2	256	[3] [4]
Symbol Width	3	12	
Gen Start	0	1023	
Number of Puncture Patterns	0	128	
Number of Channels	1	128	
Number of Marker Bits	1	16	[5]

Notes:

1. The lower limit for the variable n decoder is Maximum(5, $r+1$).
2. Max = $n-r$
3. In reality, r is limited by the maximum size of the device available. If the core exceeds the device size because r is so large, and a larger FPGA cannot be selected, the size of the core can be reduced by increasing the number of clock periods per symbol.
4. For CCSDS the minimum value of r is 3.
5. Only used if Marker Bits option is selected in the GUI.

User Parameters

Table 4-3 shows the relationship between the fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl Console).

All parameters are visible in the IP integrator. No parameters are set automatically.

Table 4-3: Vivado IDE Parameter to User Parameter Relationship

Vivado IDE Parameter	User Parameter	Default Value
Code Specification	code_specification	Custom
Variable Block Length	variable_block_length	False
Variable Number of Check Symbols	variable_number_of_check_symbols	False
Symbol Width	symbol_width	8
Field Polynomial	field_polynomial	0
Scaling Factor	scaling_factor	1
Generator Start	generator_start	0
Symbols Per Block	symbols_per_block	255
Data Symbols	data_symbols	239
Define Supported R_IN Values	define_supported_r_in_values	False

Table 4-3: Vivado IDE Parameter to User Parameter Relationship (Cont'd)

Vivado IDE Parameter	User Parameter	Default Value
Number of Supported R_IN Values	number_of_supported_r_in_values	2
Supported R_IN Definition File	supported_r_in_definition_file	No_coe_file_loaded
Memory Style	memory_style	Automatic
Number of Channels	number_of_channels	1
Output Check Symbols	output_check_symbols	True
Self Recovering	self_recovering	False
Number of Puncture Patterns	number_of_puncture_patterns	0
Puncture Definition File	puncture_definition_file	No_coe_file_loaded
Clock Enable	aclken	false
Synchronous Reset	aresetn	False
Info	info	False
Erase	erase	False
Error Statistics	error_statistics	False
Original Delayed Data	original_delayed_data	False
Marker Bits	marker_bits	False
Number of Marker Bits	number_of_marker_bits	1

System Generator for DSP Graphical User Interface

The Reed-Solomon Decoder 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 Decoder core is one of the DSP slice 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 Decoder 9.0.' See the *System Generator for DSP User Guide* [Ref 3] for more information.

The controls in the System Generator GUI work identically to those in the Vivado IDE, although the layout has changed slightly. See [Customizing and Generating the Core, page 35](#), for detailed information about all other parameters.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Constraining the Core

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

Required Constraints

This section is not applicable for this IP core.

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 8].



IMPORTANT: For cores targeting 7 series or Zynq-7000 devices, UNIFAST libraries are not supported. Xilinx IP is tested and qualified with UNISIM libraries only.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [\[Ref 6\]](#).

Test Bench

This chapter contains information about the test bench provided 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.

Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates a source codeblock consisting of a sinusoid
- RS encodes the source codeblock to create input codeblocks for the RS Decoder core
- Generates a clock signal
- Drives the core input signals to demonstrate core features
- Checks that the core output signals obey AXI protocol rules
- Checks that the core output corrected data values match the source data values
- 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. Decode a codeblock containing no errors.
3. Decode and correct a codeblock containing the maximum number of errors the core can correct.
4. Try and fail to decode and correct a codeblock containing more errors than the core can correct.
5. Decode and correct a codeblock containing errors and erasures.
6. Use a different codeblock configuration, with fewer symbols, fewer check symbols, and a different puncture pattern, as appropriate to the core: decode and correct a codeblock containing errors.
7. Decode and correct 20 codeblocks, streaming data continuously as fast as the core can process it.
8. Decode and correct 10 more codeblocks which demonstrating the AXI control signals' use and effects.
9. If `aclken` is present: Demonstrate the effect of toggling `aclken`.
10. If `aresetn` 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.

Source data is pre-generated in the `create_src_table` function and stored in the `SRC_DATA` constant. Data from this constant is RS encoded and driven into the core by the `drive_input_codeblock` procedure. The RS encoding is performed by the `rs_encoder` function from the `XilinxCoreLib` library: this function is obfuscated and its source code is not available. It is recommended to use the `drive_input_codeblock` procedure to drive a codeblock into the core.

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.

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) [\[Ref 9\]](#).

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

Parameter Changes

No changes.

Port Changes

No changes.

Other Changes

No changes.

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 Decoder, the [Xilinx Support web page](#) (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 Decoder. 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 are listed below, and 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.

Answer Records for the Reed-Solomon Decoder

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 Decoder 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 in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx devices.

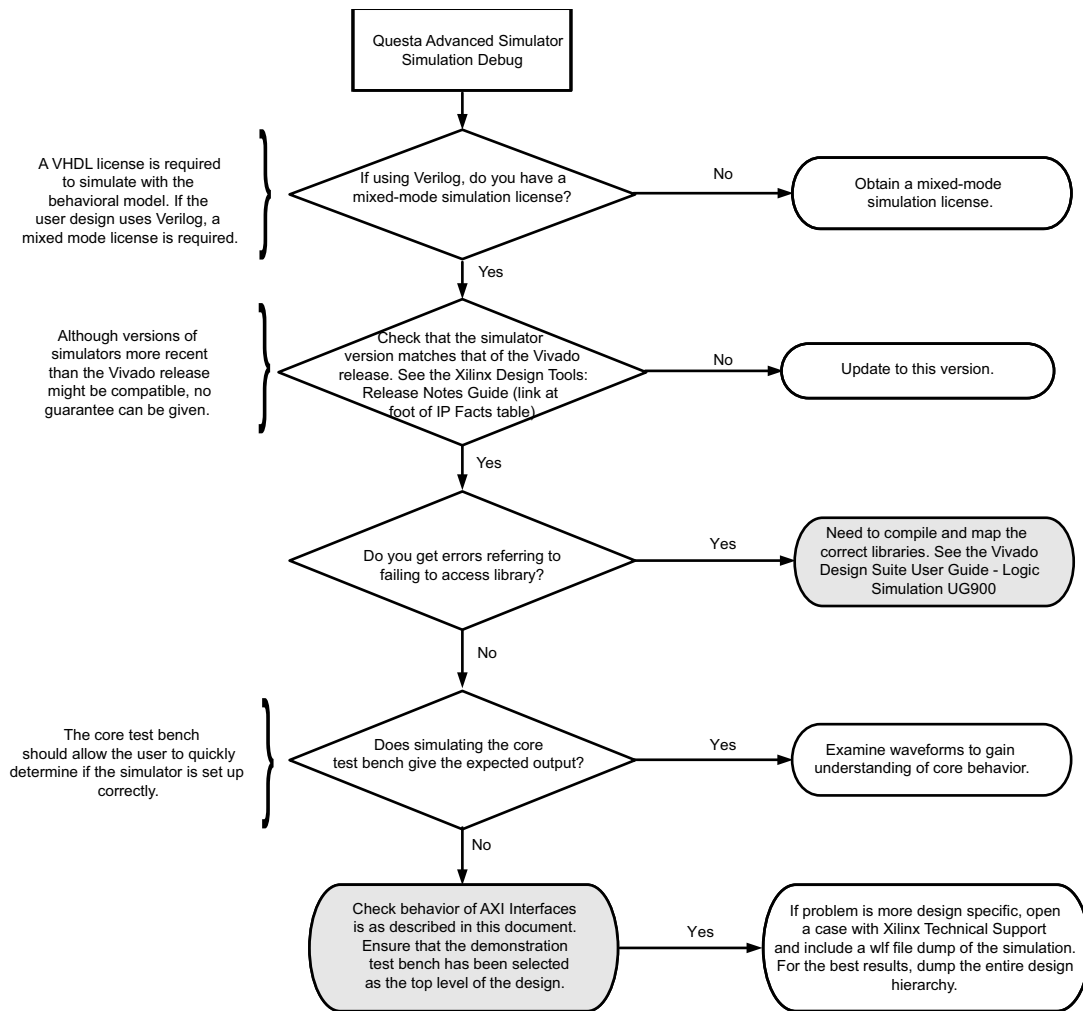
The Vivado logic analyzer is used with the logic debug IP cores, including:

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

See *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [[Ref 10](#)].

Simulation Debug

The simulation debug flow for Mentor Graphics Questa® Advanced Simulator is illustrated in Figure B-1. A similar approach can be used with other simulators.



Interface Debug

AXI4-Stream Interfaces

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

- If transmit `m_axis_output_tready` or `m_axis_stat_tready` is stuck Low following the `m_axis_output_tvalid` or `m_axis_stat_tvalid` input being asserted, the core cannot send data.
- If the receive `<interface_name>_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. See [Figure 3-3](#), [Figure 3-7](#), [Figure 3-9](#), and [Figure 3-10](#).
- Check core configuration.
- Check none of the event outputs have been asserted. These indicate incorrect input signal timing or invalid control data.

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on Documentation Navigator, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this product guide:

1. AMBA® AXI4-Stream Protocol Specification ([ARM IHI 0051A](#))
2. *Vivado® Design Suite AXI Reference Guide* ([UG1037](#))
3. *System Generator for DSP User Guide* ([UG640](#))
4. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* ([UG994](#))
5. *Vivado Design Suite User Guide: Implementation* ([UG904](#))
6. *Vivado Design Suite User Guide: Designing with IP* ([UG896](#))
7. *Vivado Design Suite User Guide: Getting Started* ([UG910](#))
8. *Vivado Design Suite User Guide: Logic Simulation* ([UG900](#))
9. *ISE® to Vivado Design Suite Migration Guide* ([UG911](#))
10. *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#))

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
02/04/2021	9.0	<ul style="list-style-type: none"> • Added Versal ACAP support.
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	<ul style="list-style-type: none"> • Revision number advanced to 9.0 to align with core version number. • Added UltraScale™ architecture support. • Template updated.
03/20/2013	1.0	Initial release as a Product Guide; replaces DS862. Latency calculation added. Appendix B, Debugging added.