

## 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
- Use with Xilinx CORE Generator™ software and Xilinx System Generator for DSP v13.1
- Available under terms of the [SignOnce IP Site License](#)

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family <sup>(1)</sup>	Virtex-7 and Kintex™-7, Virtex-6, Virtex-5, Virtex-4, Spartan®-6, Spartan-3, Spartan-3E, Spartan-3A/3AN/3A DSP				
Supported User Interfaces	Not Applicable				
	Resources <sup>(2)</sup>				Frequency
Configuration	LUTs	FFs	DSP Slices	Block RAMs <sup>(3)</sup>	Max. Freq. <sup>(4)</sup>
DVB	768	783	0	2/0	379
G.709	766	783	0	2/0	371
CCSDS	1364	1350	0	3/0	345
Provided with Core					
Documentation	Product Specification				
Design Files	Netlist				
Example Design	Not Provided				
Test Bench	Not Provided				
Constraints File	Not Applicable				
Simulation Model	VHDL, Verilog				
Tested Design Tools					
Design Entry Tools	CORE Generator tool 13.1 System Generator for DSP 13.1				
Simulation	Mentor Graphics ModelSim 6.6d Cadence Incisive Enterprise Simulator (IES) 10.2 Synopsys VCS and VCS MX 2010.06 ISIM 13.1				
Synthesis Tools	N/A				
Support					
Provided by Xilinx, Inc.					

1. For a complete listing of supported devices, see the [release notes](#) for this core.
2. Resources listed here are for Virtex-6 (-3) devices. For more complete device performance numbers, see [Table 9](#).
3. Based on 18K/36K block RAMs.
4. Performance numbers listed are for Virtex-6 (-3) FPGAs. For more complete performance data, see [Performance Characteristics, page 28](#).

## 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, and so on.

## Pinout

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 1](#) and described in [Table 1](#).

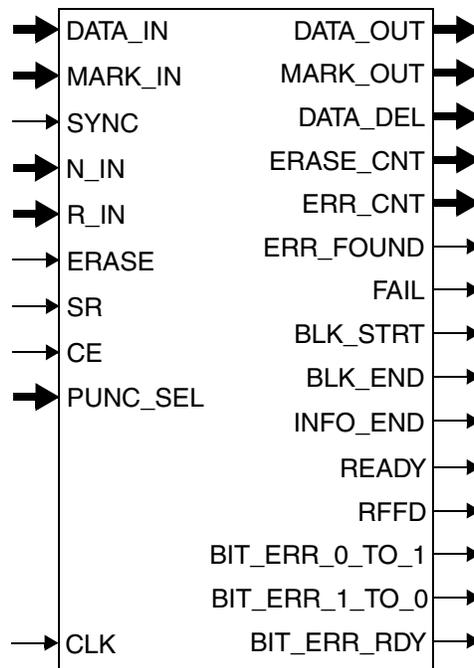


Figure 1: Core Schematic Symbol

Table 1: Core Signal Pinout

Signal	Direction	Description
DATA_IN	INPUT	Input data
MARK_IN	INPUT	Marker bits for tagging DATA_IN
N_IN	INPUT	Allows block length to be varied on a block-by-block basis (optional)
R_IN	INPUT	Allows number of check symbols to be varied on a block-by-block basis (optional)
PUNC_SEL	INPUT	Select one of a number of predefined puncture patterns (optional)
SYNC	INPUT	Timing control input
ERASE	INPUT	Flag an input symbol as an erasure (optional)

Table 1: Core Signal Pinout (Cont'd)

Signal	Direction	Description
CE	INPUT	Clock enable (optional)
SR	INPUT	Synchronous reset (optional)
CLK	INPUT	Clock-Active on rising edge
DATA_OUT	OUTPUT	Corrected data output
MARK_OUT	OUTPUT	MARK_IN delayed by core latency
DATA_DEL	OUTPUT	Uncorrected data output (optional)
ERASE_CNT	OUTPUT	Number of punctured symbols or erasures flagged in a block; present if erasures or puncturing used (optional)
ERR_CNT	OUTPUT	Number of errors corrected in a block
ERR_FOUND	OUTPUT	High if decoder found any errors in the block
FAIL	OUTPUT	High if decoder failed to correct the block
BLK_STRT	OUTPUT	High to signal the start of a block on DATA_OUT
BLK_END	OUTPUT	High to signal the end of a block on DATA_OUT
INFO_END	OUTPUT	High to signal the last information symbol of a block on DATA_OUT (optional)
READY	OUTPUT	High when the decoder is ready to accept symbols
RFFD	OUTPUT	High when the decoder is ready to accept the start of a new codeword
BIT_ERR_0_TO_1	OUTPUT	Number of bits received as 1 but corrected to 0 (optional)
BIT_ERR_1_TO_0	OUTPUT	Number of bits received as 0 but corrected to 1 (optional)
BIT_ERR_RDY	OUTPUT	High to signal BIT_ERR_0_TO_1 and BIT_ERR_1_TO_0 valid (optional)

## Functional Description

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.

The Reed-Solomon decoder samples the  $n$  symbols on the DATA\_IN port and attempts to correct any errors. The corrected symbols are output on the DATA\_OUT port after a fixed latency.

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 may 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,  $n = 2^{(Symbol\ Width)} - 1$ . 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 CORE Generator GUI or supplied on the N\_IN port. 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.

## Synchronous Reset Input

The Synchronous Reset (SR) input is an optional pin. It can be used to re-initialize the decoder at any time. SR needs to be asserted high for at least one symbol period to initialize the circuit. The decoder becomes ready for normal operation as soon as SR goes low. This pin should be selected with caution, as it increases the size of the core and may reduce performance.

The timing for the SR input is illustrated in Figure 2. Note that the DATA\_OUT output is not reset by SR. The decoder symbol buffer is not cleared, so any symbols sampled prior to the synchronous reset continue to be shifted out after the normal latency, with no error correction.

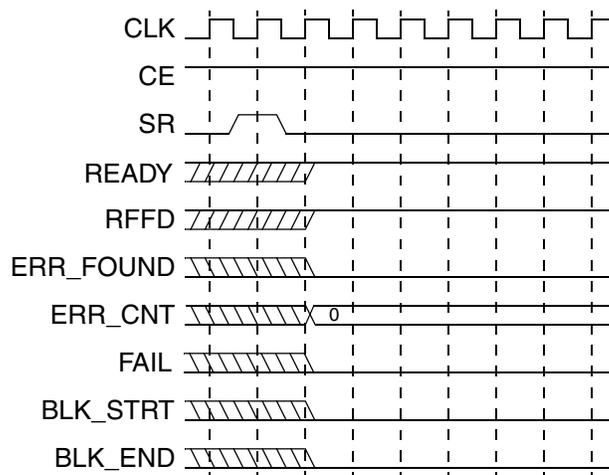


Figure 2: Synchronous Reset Timing

## DATA\_IN Input

This is the input bus for the incoming Reed-Solomon coded data. The width of the bus is set by the symbol width parameter. For the fixed  $n$  decoder, prior to a code block start being signaled with the SYNC input, DATA\_IN is passed through to DATA\_OUT with no error correction and latency as described in Latency, page 19. In between code blocks, DATA\_IN is just passed through to DATA\_OUT. If  $n$  is variable then, between code blocks, DATA\_OUT remains at the last DATA\_IN value of the most recently output block.

## MARK\_IN Input

This optional input is used to tag each symbol sampled on DATA\_IN with marker bits. The number of marker bits is parameterizable. The marker bits are delayed with the same latency as DATA\_IN to DATA\_OUT and output on MARK\_OUT. For example, if “5” is sampled on MARK\_IN at the same time as the first symbol on DATA\_IN, then “5” is output on MARK\_OUT at the same time the first symbol is output on DATA\_OUT.

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.

### CE Input

The Clock Enable input is another optional pin. It can be used to tell the decoder to ignore some of the symbols coming in on DATA\_IN. When CE is deasserted (low), all the other synchronous inputs are ignored 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.

CE is a *true* clock enable and causes the entire core to freeze state when it is low.

An example of CE operation is shown in Figure 3. In this case, the decoder ignores symbol  $D_{i4}$  as input to the block, and the current DATA\_OUT value remains unchanged. (The decoder still samples  $n$  symbols.) As  $D_{i4}$  is not included in the code block, the output sequence ... $D_{i0}, D_{i1}, D_{i2}, D_{i3}, D_{i5}$ ... appears on DATA\_OUT during the output stage of this block.

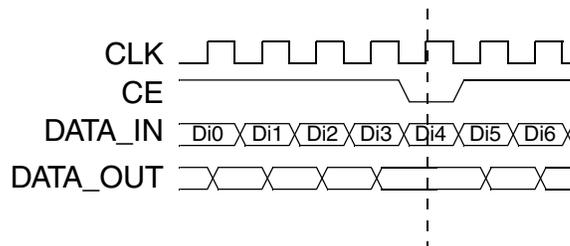


Figure 3: Clock Enable Timing

In Figure 3, the symbol period is the same as the clock period, with one symbol per clock cycle. It is possible to have more than one clock cycle per symbol period. This is explained in Processing Delay, page 16 of this document. In this case CE is the only synchronous input that does *not* need to be synchronized to symbol periods. The core state freezes whenever CE is low, regardless of when this occurs.

### SYNC Input

The timing of the core is controlled via the SYNC input. This is asserted high for one, and only one, symbol period when the first symbol of a code block is on DATA\_IN; otherwise it is low. This is illustrated in Figure 4.  $D_0$  is the first symbol of a code block. Once started, the remaining symbols in the code block are sampled on consecutive symbol periods. It is impossible to start another code block before the first one has been completely sampled. SYNC is ignored if the RFFD output is low.

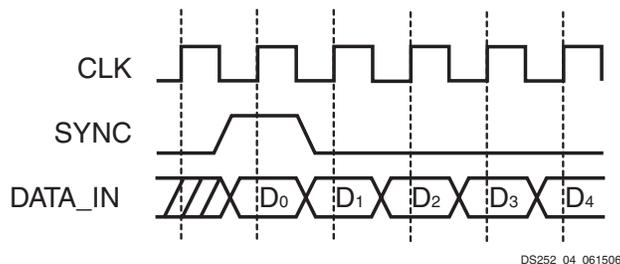


Figure 4: SYNC Timing

DS252\_04\_061506

For a fixed block length decoder, the SYNC pulses can be applied at most once every  $n$  symbol periods. If the Processing Delay is greater than  $n$ , then a SYNC pulse can only be applied once every Processing Delay symbol period. This is explained in [Processing Delay, page 16](#).

If RFFD is fed back to the SYNC input, then the core operates in a continuous mode, starting a new block as soon as the previous one completes. It is still important to align the symbols so that the first symbol of the block aligns with the RFFD pulse in this case.

In earlier versions of this core, another SYNC timing mode was available, called *Data Symbol Enable* mode. In this mode, the SYNC input was held high while the  $k$  data symbols were on DATA\_IN; otherwise it was low. This behavior can be replicated, if required, by a simple logic circuit outside the core. This should assert the core SYNC input when the old SYNC signal goes high, keep the SYNC input asserted for one symbol period, and then deassert it.

## N\_IN input

This allows the block length to be changed every block. N\_IN is sampled at the same time as the first symbol of the new block. The timing is described in [Variable Block Length, page 22](#). Selecting this input significantly increases the size of the core. Unless there is an R\_IN input, the number of check symbols is fixed, so varying  $n$  automatically varies  $k$ .

If a value less than the minimum  $n$  defined in [Table 3](#) is sampled, then the SYNC pulse for that block is ignored.

For example, in [Figure 23](#), if N\_IN is set to 255 and R\_IN is set to 16, they are sampled on the rising clock edge when SYNC is high. The core begins sampling symbols assuming a ( $n=255, k=239$ ) code block. After the last (255<sup>th</sup>) symbol of the block has been sampled on DATA\_IN, SYNC may be pulsed high again with different N\_IN and R\_IN values. For example, N\_IN may be 64 and R\_IN equal to 8, beginning a ( $n=64, k=56$ ) code block. 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 port width of 5 bits.

## R\_IN input

This allows the number of check symbols to be changed every block. R\_IN is sampled at the same time as the first symbol of a new block. The timing is identical to the case when N\_IN is used: see [Variable Block Length, page 22](#).

The width of the R\_IN port is the minimum number of bits required to represent the maximum  $n$  value minus the minimum  $k$  value.

If a value less than the minimum  $r$  defined in [Table 3](#) is sampled, then the SYNC pulse for that block is ignored. Blocks with  $r$  values greater than ( $\max n - \min k$ ) are also ignored.

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 input should not be used in these cases. The PUNC\_SEL input is provided to handle this.

## PUNC\_SEL input

This selects a puncture pattern to be applied to the new code block. Puncturing is explained in [Puncturing, page 20](#). PUNC\_SEL is sampled at the same time as the first symbol of each new block. PUNC\_SEL is provided only if the number of puncture patterns is greater than one.

## ERASE Input

This optional input is available only when erasure support is required. Erasure handling is described later in this document.

## DATA\_OUT Output

This is the output bus for the corrected symbols. This bus always has the same width as DATA\_IN.

Corrected symbols start to appear at a predefined number of symbol periods after the first symbol is sampled on DATA\_IN. This delay is termed the LATENCY of the decoder and is fully explained in [Latency, page 19](#). The timing for DATA\_OUT is different for the variable  $n$  decoder, as the latency can vary if the block size is dynamically varied with the N\_IN input.

## MARK\_OUT Output

This optional output is simply MARK\_IN, delayed by the same latency as DATA\_IN to DATA\_OUT.

## DATA\_DEL Output

This optional output bus is an uncorrected version of DATA\_OUT. It is simply DATA\_IN delayed by the latency of the core. DATA\_DEL is useful for making comparisons of corrected and uncorrected data. This bus always has the same width as DATA\_IN. With the fixed  $n$  decoder, DATA\_DEL continues to follow DATA\_IN between code blocks. With the variable block length decoder, DATA\_DEL changes only when DATA\_OUT changes, so in between blocks it remains at the last input symbol value of the previous block.

This output 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.

## BLK\_STRT Output

BLK\_STRT is pulsed high for one symbol period to indicate that the first symbol of a block is currently on DATA\_OUT. The timing for BLK\_STRT is illustrated in [Figure 5](#).

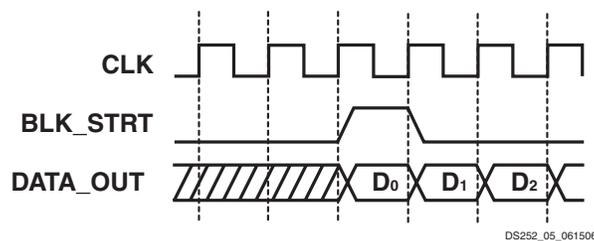


Figure 5: BLK\_STRT Timing

### BLK\_END Output

BLK\_END is pulsed high for one symbol period to indicate that the last symbol of a block is currently on DATA\_OUT. The timing for BLK\_END is illustrated in Figure 6.

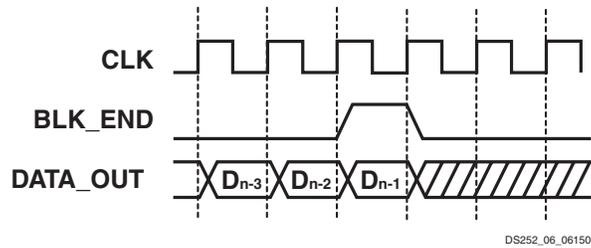


Figure 6: BLK\_END Timing

### INFO\_END Output

INFO\_END is pulsed high for one symbol period to indicate that the last *information* symbol of a block is currently on DATA\_OUT. This is the  $k$ th data symbol. The  $n-k$  check symbols are output on DATA\_OUT after this. The timing for INFO\_END is illustrated in Figure 7. This assumes the first symbol of the block is  $D_0$ .

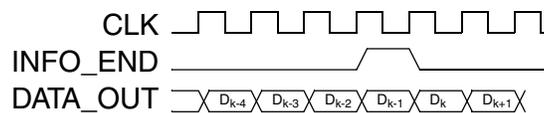


Figure 7: INFO\_END Timing

### ERR\_FOUND Output

This is one of a number of status outputs, which are set as the last symbol of a block is output on DATA\_OUT. At this time, if the decoder detected any errors, erasures, or punctures in the code block, ERR\_FOUND goes high. If no errors, erasures, or punctures are found, ERR\_FOUND goes low. The status outputs retain their state until the end of the next code block or until the core is reset. The timing for all the status outputs is illustrated in Figure 8.

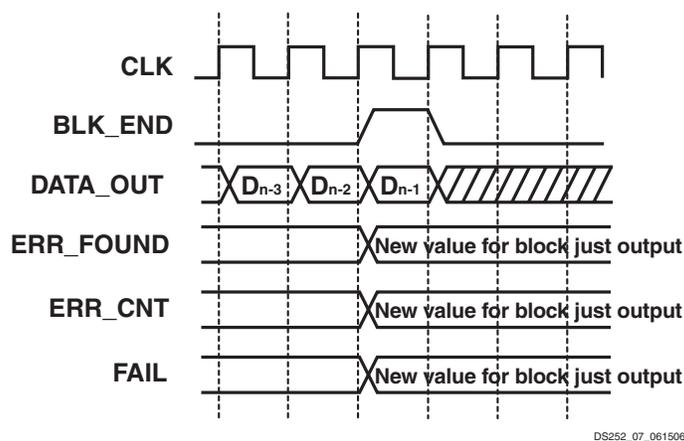


Figure 8: Status Output Timing

## ERR\_CNT Output

This is another of the status outputs. The ERR\_CNT bus gives the number of errors, erasures, and punctures that were corrected in the most recent output block. The width of the bus depends on the input parameters  $n$  and  $k$ . The bus width is equal to the number of binary bits required to represent  $(n-k)$ . If  $n-k = 16$ , for example, the ERR\_CNT bus is five bits wide.

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

## ERASE\_CNT Output

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

## FAIL Output

FAIL is also a status output. 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 may 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.

## READY Output

The READY output is high when the decoder is ready to sample symbols on DATA\_IN. If the processing delay (see [Processing Delay, page 16](#)) is greater than  $n$ , READY goes low after the last symbol of a block is sampled. This is the case in [Figure 9](#). It remains low until the processing delay is over, as shown in [Figure 10](#). If the processing delay is less than or equal to  $n$ , READY is always high.

When the processing delay is greater than  $n$ , READY consistently goes low one symbol period after the second-to-last symbol has been sampled.

The timing for READY is different if the block length is variable. See [Variable Block Length, page 22](#).

READY may not be registered inside the core. It should be registered outside the core if necessary.

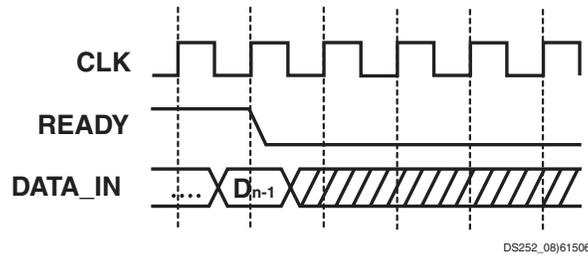


Figure 9: READY Timing

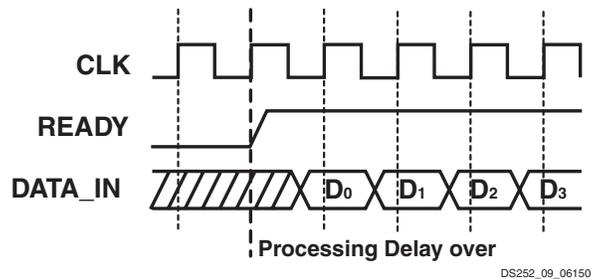


Figure 10: READY Timing

### RFFD Output

RFFD is the Ready for First Data output. The RFFD output is high when the decoder is ready for a new SYNC pulse. RFFD goes low as soon as SYNC has been sampled high and remains low until it is safe to reassert SYNC. Any SYNC pulses that occur while RFFD is low are ignored. The timing for RFFD is shown in Figure 23.

**Note:** RFFD is not registered inside the core. It should be registered outside the core if necessary.

### BIT\_ERR\_0\_TO\_1 Output

This optional output gives the number of bits that were received as 1 but corrected to 0 in the block just output. 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. BIT\_ERR\_0\_TO\_1 is accumulated as the codeword is being output on DATA\_OUT. The value changes as data is output. The final value is valid for a single symbol period when BIT\_ERR\_RDY goes high. This occurs four symbol periods after the BLK\_END pulse.

### BIT\_ERR\_1\_TO\_0 Output

This optional output has the same functionality as BIT\_ERR\_0\_TO\_1, except it counts the number of bits received as 0 but corrected to 1.

### BIT\_ERR\_RDY Output

This optional output is high for one symbol period when BIT\_ERR\_0\_TO\_1 and BIT\_ERR\_1\_TO\_0 should be sampled. It is simply a delayed version of BLK\_END.

## Erasure 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 11, D<sub>2</sub> 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.

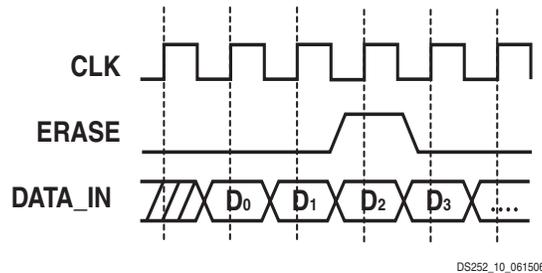


Figure 11: ERASE Timing

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, and is recommended only for codes where  $n - k$  is less than eight. The core does support erasures for larger values of  $n - k$ , but there is a large area overhead compared to the same core without erasure support. See the example implementations toward the end of this data sheet.

## Parameters

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:

### GeneratorStart

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

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

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

### *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  $2^{(\text{Symbol\_Width})-1}$  is 1, that is,  $h$  and  $2^{(\text{Symbol\_Width})-1}$  must be relative primes.

***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.

***n***

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

**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 2: Default Polynomials**

Symbol Width	Default Polynomial	Decimal Representation
3	$x^2+x+1$	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

**Symbol Width**

This is the bus-width of DATA\_IN and DATA\_OUT.

**Clock Enable**

This is selected when a clock enable input is required.

**Synchronous Reset**

This is selected when a synchronous reset input is required.

**Info End**

This is selected when the INFO\_END output is required.

**Variable Block Length**

This is selected when the N\_IN input is required.

## Variable Number of Check Symbols

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

## Delayed Data

This is selected when the `DATA_DEL` output is required.

## Marker Bits

This is selected when `MARK_IN` and `MARK_OUT` are required.

## Number of Marker Bits

This sets the width of `MARK_IN` and `MARK_OUT`.

## Erase

This is selected when erasure support is required. See the explanation in [Erasure Decoding, page 11](#).

## Clock Periods Per Symbol

Normally, there is only one clock period per symbol. This may be increased to reduce the processing delay, which is described in the next section. The symbol period must always be a whole number of clock periods.

## 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 20](#). This parameter is not available if erasures are selected. The puncturing can be handled externally by asserting the `ERASE` pin in this case.

## Number of Channels

This parameter defines how many channels the core should support. Multi-channel operation is described in [Multiple Channels](#), page 26.

## 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. Note that 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 may 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.

## 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 illustrated in [Figure 12](#). 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.

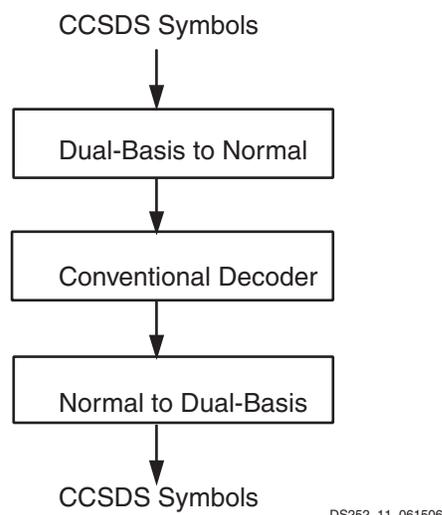


Figure 12: CCSDS Decoder

## Optimization

For Virtex<sup>®</sup>-5, the optimization of the core can be chosen from the CORE Generator GUI. With *area* optimization, the core uses the smallest possible area within the FPGA. With *speed* optimization, additional area is traded to provide an increased maximum clock rate.

## 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 may 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 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.

## Parameter Ranges

Valid ranges for the parameters are given in [Table 3](#).

Table 3: Parameter Ranges

Parameter	Min	Max	Notes
$n$	4	$2^{(\text{Symbol\_Width})}-1$	[5]
$k$	1	$2^{(\text{Symbol\_Width})}-3$	[1]
$h$	1	$2^{(16)}-1$	
Polynomial	0	$2^{(13)}-1$	
$r=n-k$	2	256	[2] [3]
Symbol Width	3	12	
Gen Start	0	1023	
Clock Periods Per Symbol	1	$2^{(16)}-1$	[4]
Number of Puncture Patterns	0	128	
Number of Channels	1	128	
Number of Marker Bits	1	16	[6]

### Notes:

1.  $\text{Max}=n-r$
2. 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.
3. For CCSDS the minimum value of  $r=3$ .
4. The practical limit for this parameter is usually caused by the core maximum clock frequency being reached.
5. The lower limit for the variable  $n$  decoder is  $\text{Maximum}(5, r+1)$ .
6. Only used if Marker Bits option is selected in the GUI.

## Processing Delay

For some parameter selections, the decoder might not be ready to accept one code block immediately after another. This is because it is still processing the first block. The processing delay for a given  $t$ , assuming one clock cycle per symbol period, is shown in Figure 13.

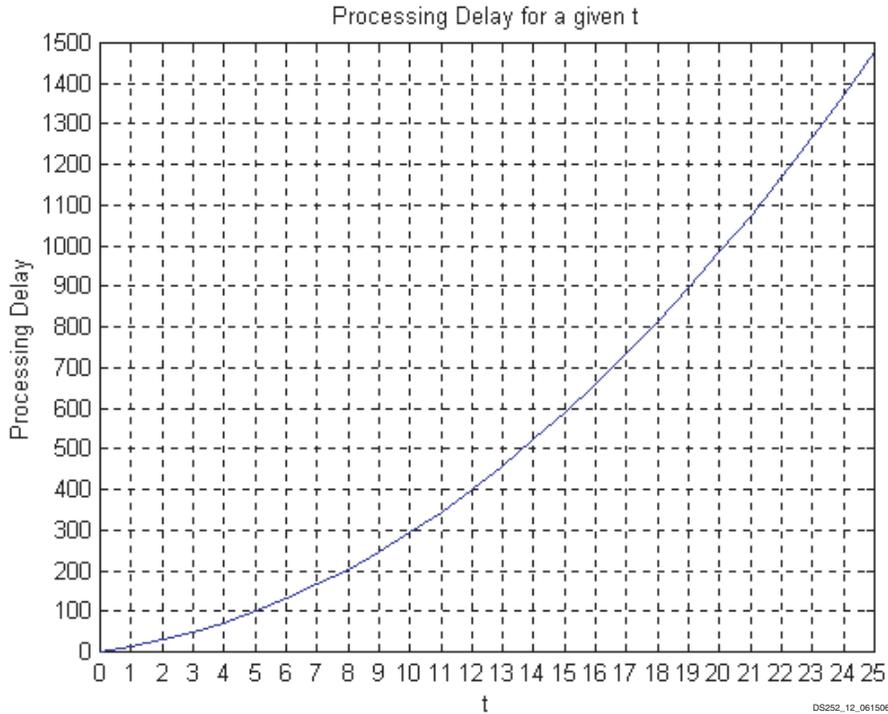


Figure 13: Processing Delay against t Where  $t = (n-k)/2$

In the case of the fixed  $n$  decoder, this is the minimum number of symbol periods from the start of the first symbol period of a code block before another code block can be started. If the processing delay is greater than  $n$ , it is not possible to follow one code block immediately with another.

The variable  $n$  decoder can still accept a new code block, 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  $n$  greater than the processing delay, at some point it is unable to accept a new code block. The timing for the variable  $n$  decoder is described in Variable Block Length, page 22.

The number of symbol periods can be calculated using the following equation:

$$ProcessingDelay = t^2 + 2 \left( 3t + \sum_{i=1}^{t+1} i \right) + 2$$

If erasure decoding or puncturing is enabled, the following equation should be used:

$$ProcessingDelay = 2 \left( \sum_{i=1}^{n-k+1} i \right) + 3(n-k) + 3$$

**Note:** Measures can be taken to reduce the processing delay.

If necessary, the processing delay can be reduced by increasing the number of clock periods per symbol. In this case, the processing delay (in symbol periods) can be calculated using the following expression:

$$1 + \text{RoundDown}(P/\text{ClksPerSym})$$

where  $P$  is the result from the processing delay calculation.  $\text{ClksPerSym}$  is the number of clock cycles per symbol. This can be increased until the maximum input clock frequency of the core is reached. The processing delay is always rounded down to the nearest whole number. To calculate the processing delay in terms of clock cycles, multiply the preceding result by the number of clock periods per symbol.

If the number of clock periods per symbol is greater than one, all synchronous inputs must be synchronized to symbol periods. This is illustrated for four clock periods per symbol in Figure 14.

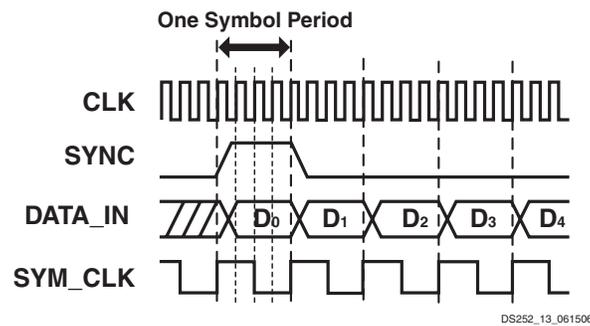


Figure 14: Input Timing, Four Periods Per Symbol

The decoder samples its inputs on rising edges of CLK coincident with rising edges of the imaginary SYM\_CLK signal. The outputs are also all synchronized to the symbol period. This is illustrated for three clock periods per symbol in Figure 15.

Figure 15 also shows the timing for SR when the number of clock periods per symbol is greater than one. SR must be pulsed for a whole number of symbol periods, just like all the other synchronous inputs. It is sampled high and the outputs are reset one symbol period later.

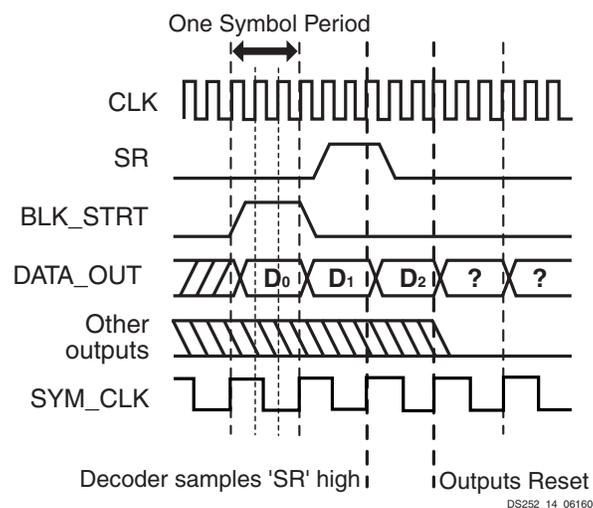


Figure 15: Output Timing, Three Periods per Symbol

The decoder always resynchronizes itself when it detects a rising edge on SYNC. If the rising edge of SYNC occurs less than two CLK periods before the rising edge of SYM\_CLK, the decoder samples the first symbol twice. This situation is illustrated in Figure 16. The decoder still operates correctly, with  $D_0$  as the first symbol of the block and  $D_1$  as the second.  $D_0$  appears on DATA\_OUT LATENCY symbol periods after  $t_1$ . The corrected value of  $D_0$  appears LATENCY symbol periods after  $t_2$ .

This may occur when SYNC goes high at the start of the first code block. It does not reoccur if the gaps between blocks are always a whole number of symbol periods.

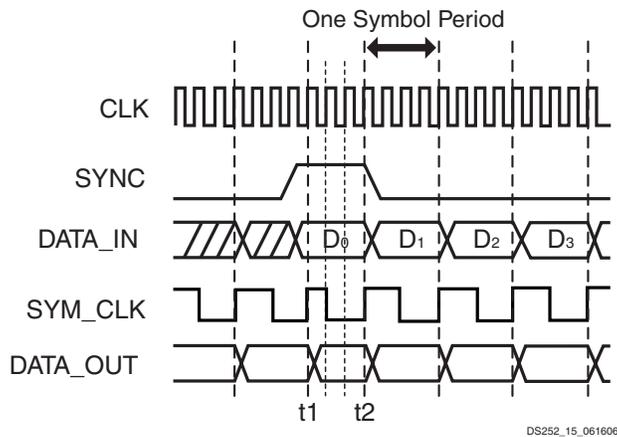


Figure 16: Resynchronization Using SYNC

**IMPORTANT:** If a code block is started prior to corrected symbols from the previous block appearing on DATA\_OUT, there must be an integer number of symbol periods between the last symbol of the previous block and the first symbol of the new block. This is illustrated in Figure 17. Figure 18 shows the case where there is a non-integer number of symbol periods between blocks A and B. This may cause the decoder to lose track of block A.

It is safe to start another block with SYNC misaligned, relative to the symbol periods from the previous block, only after the corrected symbols from all the previous blocks have started to appear on DATA\_OUT.

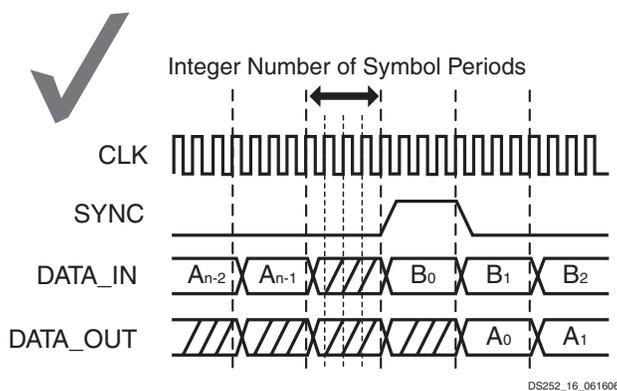


Figure 17: Integer Number of Symbols between Blocks

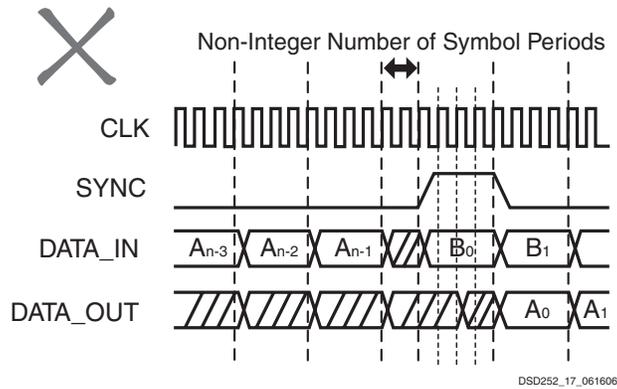


Figure 18: Resynchronization before DATA\_OUT for Previous Block

### Latency

The latency is the number of sampled symbols from a symbol being sampled on DATA\_IN to the corrected version of that symbol appearing on DATA\_OUT. This should not be confused with the processing delay; a subsequent code block may be started before the latency delay is over.

An example, with a latency of three symbol periods and one clock period per symbol, is shown in Figure 19. 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) and  $t$  (the number of correctable errors). The total latency can be determined from the following equation:

$$\text{Latency} = n + m + \text{Processing Delay} - s + c$$

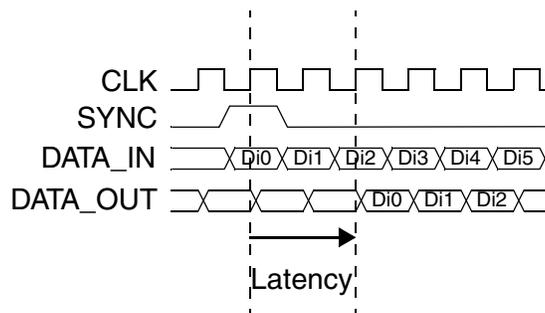


Figure 19: Latency = 3

This gives the latency in symbol periods. The variables,  $m$ ,  $s$ , and  $c$ , are defined in Table 4, Table 5, and Table 6, respectively. To calculate the latency in clock periods, multiply the result from the table by the number of clock periods per symbol.

Table 4: Latency Calculation  $m$

Symbol Width	M
=8	7
Not Equal To 8	6

Table 5: Latency Calculation  $c$

Code Specification	c
CCSDS	3
Any other value	0

Table 6: Latency Calculation  $s$

Clock Periods Per Symbol	s
=1	0
>1	1

## 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 may 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 input 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 the PUNC\_SEL input 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 7](#).

**Table 7: 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 20](#), [Figure 21](#), and [Figure 22](#). `PUNC_SEL` is sampled as "1" at the start of the block. 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). This is shown in [Figure 21](#). The core still samples `DATA_IN` for the four punctured symbols. The value on `DATA_IN` is irrelevant at this time. `Dn-5` is the last real symbol received.

[Figure 22](#) shows the block being output. The decoder actually determines the values of the punctured symbols and outputs them in the correct sequence. The `ERASE_CNT` output shows how many symbols were punctured. The `ERROR_CNT` output shows the number of errors plus the number of punctures. So if there were no errors in the block, `ERROR_CNT` is 4. The number of true errors is `ERROR_CNT-ERASE_CNT`.

`SYNC` cannot be reasserted until all the symbols in a block are sampled, including the dummy symbols in the puncture positions.

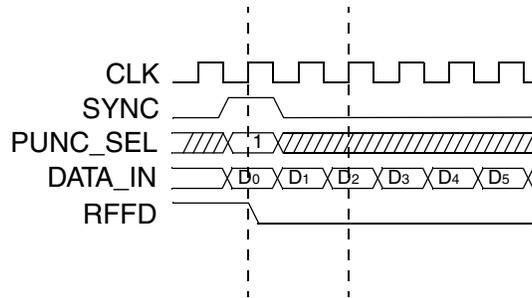


Figure 20: Puncture Timing – Start of Input Block

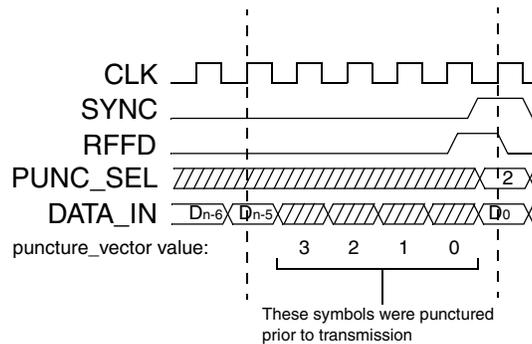


Figure 21: Puncture Timing – End of Input Block

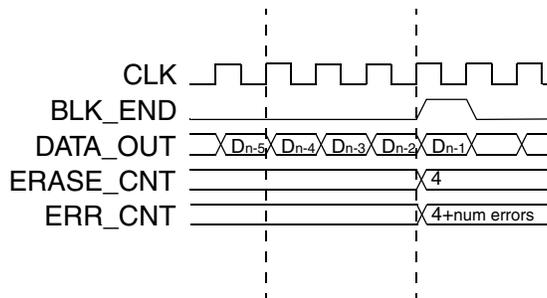


Figure 22: Puncture Timing – End of Output Block

If puncturing is used, the latency and processing delay are derived from the same equation as if erasures were enabled. See [Processing Delay](#), page 16.

## Variable Block Length

If the  $N\_IN$  input is used, the block length can be different for every new block.  $N\_IN$  can vary from 5 up to  $2^{(\text{Symbol\_Width})}-1$ . It must also be greater than the number of check symbols,  $r$ .  $N\_IN$  is sampled at the same time as the  $SYNC$  pulse that starts the block.  $N\_IN$  is the  $n$  value for the block. The number of check symbols,  $r$ , can be fixed or variable.

If the  $N\_IN$  input is used, the core cannot pass non-Reed-Solomon data through in between code blocks like the fixed  $n$  decoder. No input data is sampled until a block is started with a  $SYNC$  pulse. See [Figure 23](#) for  $N\_IN$  timing.

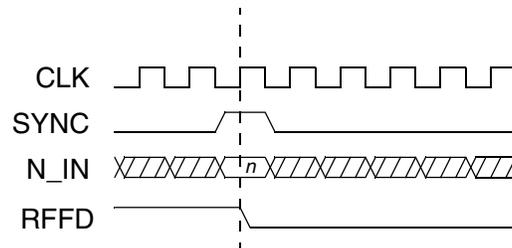


Figure 23: N\_IN Timing

When variable block length is used, the timing of the core is slightly different compared to the fixed block length core. The latency and processing delay for each individual block are the same as for the fixed block core and can be computed using the equations in the previous sections. The values sampled on N\_IN and R\_IN may be used in the equations to compute the processing delay and best case latency for each block sampled. Note that the Processing Delay depends only on the number of check symbols.

A block may actually take longer to appear on DATA\_OUT than the calculated latency, as an earlier, larger block may still be being processed. This is illustrated in Figure 24.

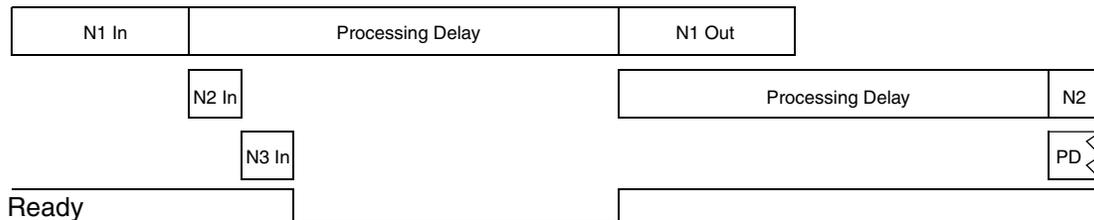


Figure 24: Variable n – Ready Operation

The core always samples data and outputs results as soon as possible. It is possible for the READY output to go low, even if the Processing Delay is not greater than all the sampled block sizes (see Figure 25 for example). However, if the Processing Delay is not greater than any possible sampled block size, it can be guaranteed that, **on average**, no gaps are required between code blocks. Thus, the input data may need to be temporarily held up, as in Figure 25, because a large block was followed by many small ones. A small FIFO could be placed in front of the core to smooth out these effects if necessary. The input symbols to the FIFO would never need to have any gaps, and there would never be any gaps between output symbols from the core.

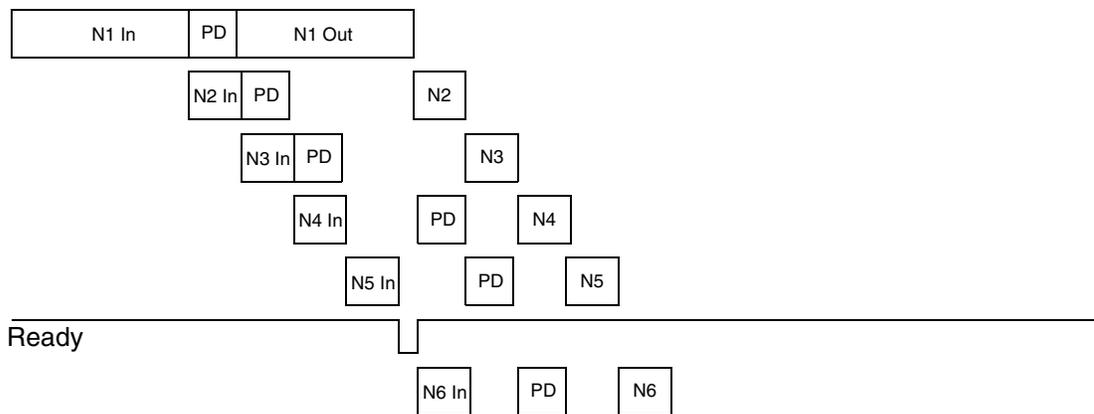


Figure 25: Processing Delay Buffer

Figure 24 shows an example where the Processing Delay is greater than the block length. In the fixed  $n$  decoder, `READY` would have gone low as soon as the last symbol of block N1 had been sampled. The variable  $n$  decoder is able to buffer intermediate results. As soon as block N1 has been sampled, the values required by the processing section are passed in and processing begins. Block N2 is then sampled. The processing section is still busy with block N1, but the N2 values are buffered, allowing block N3 to be loaded. The N3 values are also buffered; however, the buffer is now full, and no new symbols can be sampled until the N2 values have been loaded into the processing section. As soon as this happens, the core is ready for another new block.

If a large block is followed immediately by a small block, then the small block has to wait until the large block has been completely output before it can be output. This is illustrated in an example in Figure 26. In this example, the inputs are not held up because each block can be passed into the processing section before the two-block input buffer has been filled. Notice that the overall latency for blocks N2 to N4 is larger than predicted by the latency equation. This is because they had to wait for N1 to be output. `READY` is high all the time in this example, and there is still no gap between symbols as they are input or output.

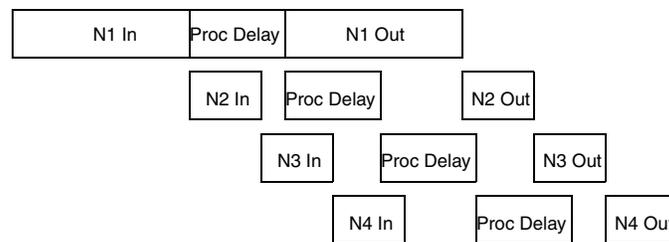


Figure 26: Large Block Followed by Small Blocks – Effect on Latency

The processing section can also buffer up to two blocks worth of results. This is illustrated in the example in Figure 25. 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 section buffer is now full, and PD4 cannot begin until the PD3 results have been unloaded. This occurs as the N2 values begin to be output. PD4 then begins and completes while the core is still outputting values from earlier blocks. Therefore, the PD4 results are buffered, once more filling the processing buffer. PD5 cannot begin until the PD4 results are unloaded. This occurs as the N3 values begin to be output. `READY` goes low after N5 because the input buffer is now full. It frees up again as the N4 values are fed into the processing section at the start of PD4.

Block N6 can begin as soon as `READY` is high again. Notice that there is no gap between N5 Out and N6 Out, even though there was a gap between them at the input side. This is because the core always outputs results as soon as possible.

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

## 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 multi-channel configurations, all channels have the same settings for  $n\_block$ ,  $k\_block$ , and  $r\_block$ . See [Table 8](#).

Table 8: Block Code Settings – Value and Range

Block Code Settings	Value [1]
<b>Fixed Block Length</b>	
$n\_block$	$n$
$k\_block$	$k$
$r\_block$	$(n-k)$
<b>Variable Block Length. Fixed Number of Check Symbols</b>	
$n\_block$	N_IN
$k\_block$	N_IN - $(n-k)$
$r\_block$	$(n-k)$
<b>Variable Number of Check Symbols</b>	
$n\_block$	N_IN
$k\_block$ [2]	N_IN - R_IN
$r\_block$	R_IN

### Notes:

1. The minimum and maximum values are defined in [Table 3](#).
2. 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 set at the sync pulse of each new block to the value sampled on N\_IN.

### $k\_block$

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 at the sync pulse of each new block, to the value sampled on N\_IN minus the parameter  $(n-k)$ .
- When a variable number of check symbols is required,  $k\_block$  is set at the start pulse of each new block to the value sampled on N\_IN 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 at the start pulse of each new block to the value sampled on R\_IN.

## 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 multi-channel core generally runs at a higher clock frequency than a single-channel core. Using one multi-channel core in a high-speed application can be more efficient than instantiating several single-channel RS decoder cores. Multi-channel is available only for fixed  $n$  and  $r$  decoders.

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 sampled when SYNC is high.

With multiple channels, there is still only one DATA\_IN port. 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. Symbols (both information and check) are output on DATA\_OUT in the same sequence. An example with three channels is shown in Figure 27.

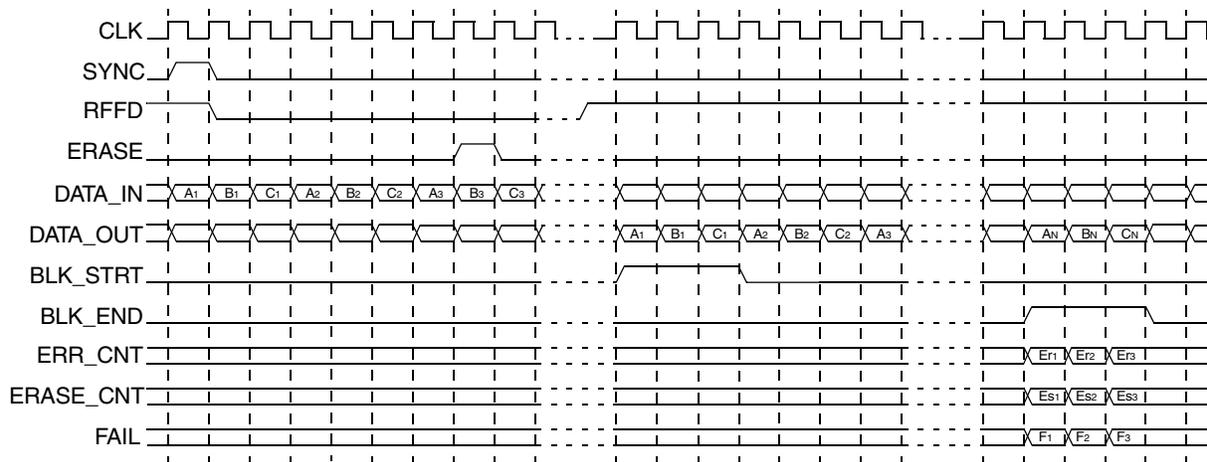


Figure 27: Multi-Channel Operation

Notice that SYNC is asserted only for a single symbol period. This starts a new block for all three channels. A1, B1 and C1 are the first symbols of the new block for channels A, B and C. RFFD goes high as soon as it is safe to begin a new block with another SYNC pulse.

If erasures are enabled, then ERASE can be asserted at any time for each channel independently. In this example, symbol B3 is marked as an erasure.

Symbols on DATA\_OUT are interlaced in the same way as symbols on DATA\_IN.

BLK\_STRT is driven high while the first symbol for each channel is on DATA\_OUT. BLK\_END is driven high while the last symbol for each channel is on DATA\_OUT. The status outputs (ERR\_CNT, etc.) are also output for each channel in sequence at this time. After BLK\_END goes low, they retain the value for the last channel until a new BLK\_END pulse occurs. INFO\_END is driven high while the  $k$ th symbol for each channel is on DATA\_OUT.

The processing delay (in clock cycles) is approximately the single-channel processing delay multiplied by the number of channels. See the core GUI for the exact value. The number of cycles allowed for continuous code blocks is also multiplied by the number of channels, so multi-channel operation does not significantly affect whether or not there needs to be a gap between code blocks.

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:

RS(255,239) is a configuration of 255 symbols and 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 204 cycles, which is less than 255, so this configuration is capable of continuous processing.

Symbol Width = 8

Symbols per Block ( $n$ ) = 255

Data Symbols ( $k$ ) = 239

### Example 2:

RS(255,229) is a configuration of 255 symbols and 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 459 cycles, and is therefore greater than 255, continuous processing cannot be done.

Symbol Width = 8

Symbols per Block ( $n$ ) = 255

Data Symbols ( $k$ ) = 229

If continuous processing is required, then the number of clock periods per symbol can be set to 2. By setting *Clocks per Symbols* to 2, the processing delay is now 230 symbol periods, but each symbol period is two clock periods. Therefore, for a symbol rate of 100 Msymbols/sec, the core would need to be clocked at 200 MHz.

### Example 3:

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 400, so continuous code blocks are possible.

Symbol Width = 12

Symbols per Block ( $n$ ) = 400

Data Symbols ( $k$ ) = 376

### Example 4:

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. The largest expected  $R\_IN$  value is 16, so  $k$  must be set to  $255-16=239$ . This gives an  $R\_IN$  port width of 5 bits.

So the  $N\_IN$  can have a value up to 255 and  $R\_IN$  can have a value up to 16. Lower limits are defined in [Table 3](#).

Symbol Width = 8

Symbols per Block ( $n$ ) = 255

Data Symbols ( $k$ ) = 239

Variable Block Length Checked

Variable Number of Check Symbols Checked

## 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 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 7.1.” See the System Generator User Manual for more information.

The controls in the System Generator GUI work identically to those in the CORE Generator GUI, although the layout has changed slightly. See [Parameters, page 11](#), for detailed information about all other parameters.

## Core Resource Utilization

The area of the core increases with  $n$ ,  $n-k$ , and the symbol width. Some example configurations are shown in [Table 9](#). In this table, optional pins were not used, unless otherwise stated. Memory style was always set to automatic. The option to map primary I/O registers into IOBs during placement should be selected if the core I/Os are to be connected directly onto a PCB via the FPGA package pins. This gives lower output clock-to-out times and predictable setup and hold times. In this case, it is especially important to register outputs that may not be registered inside the core, such as RFFD and READY.

## Performance Characteristics

It is important to set a maximum period constraint on the core clock input. The figures in [Table 9](#) show clock speeds that can be achieved when this is done. Apart from high map and par effort, default implementation tools options were used. It may be possible to improve slightly on these values by trying different options for the place and route software. Performance increases as  $n$ ,  $n-k$ , and the symbol width decrease.

[Table 9](#) provides resource and performance data for Virtex-6 FPGAs. For other devices, the user should generate a core and consult a map report to determine device utilization. The Xilinx SmartXplorer utility may be used to determine the maximum achievable frequency for the configuration.

**Table 9: Example Decoder Implementations**

	ATSC 1	DVB1	DVB2	CCSDS	G.709	G.709 Two-Channel	ETSI-BRAN	IEEE-802.16d
Generator Start	0	0	0	112	0	0	0	0
$h$	1	1	1	11	1	1	1	1
$k$	187	188	188	223	239	239	239	239
$n$	207	204	204	255	255	255	255	255
Polynomial	285	285	285	391	285	285	285	285
Symbol Width	8	8	8	8	8	8	8	8
Erasur Decoding	No	No	Yes	No	No	No	No	No
Clock Periods Per Symbol	1	1	2 <sup>[3]</sup>	1	1	1	1	1
Variable Block Length	No	No	No	No	No	No	Yes	Yes
Number of Channels	1	1	1	1	1	2	1	1

Table 9: Example Decoder Implementations (Cont'd)

	ATSC 1	DVB1	DVB2	CCSDS	G.709	G.709 Two-Channel	ETSI-BRAN	IEEE-802.16d
Puncture Patterns	0	0	0	0	0	0	0	4
Processing Delay <sup>[1]</sup>	294	204	179	660	204	406	204	357
Latency <sup>[1]</sup>	508	415	389	925	466	926	Variable	Variable
Xilinx Part	XC6VLX130T	XC6VLX130T	XC6VLX130T	XC6VLX130T	XC6VLX130T	XC6VLX130T	XC6VLX130T	XC6VLX130T
LUT/FF Pairs	995	780	1845	1355	762	1022	999	1333
LUTs <sup>[4]</sup>	962	763	1813	1340	754	992	969	1306
FFs	919	781	1369	1347	781	1517	845	1139
Block RAMs (36k)	0	0	0	0	0	0	0	1
Block RAMs (18k)	2	2	2	3	2	2	3	2
Maximum Clock Frequency <sup>[2]</sup>	281/376	281/379	273/349	276/345	287/371	334/423	281/391	274/380

**Notes:**

1. Measured in symbol periods.
2. Maximum clock frequencies are shown in MHz for -1/-3 parts with Speed Optimization selected. Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification. ISE speed file version used for -1 speed grade was "PRODUCTION 1.07a 2010-05-25." ISE speed file version used for -3 speed grade was "PRELIMINARY 1.07a 2010-05-25."
3. Set to 2 to keep Processing Delay  $\leq n$ .
4. LUT count includes route-thrus and may vary when the core is packed with other logic. Resource information is for -1 speed grade.

## 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. If a full license is installed for the core to run on hardware, delete the old XCO file and recreate the core from new.

## Support

Xilinx provides technical support at [www.xilinx.com/support](http://www.xilinx.com/support) for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled *DO NOT MODIFY*.

Refer to the IP Release Notes Guide ([XTP025](#)) for further information on this core. On the first page there is a link to "All DSP IP." The relevant core can then be selected from the displayed list.

For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Bug Fixes
- Known Issues

## Ordering Information

This Xilinx LogiCORE IP product is provided under the terms of the [SignOnce IP Site License](#).

To evaluate this core in hardware, generate an evaluation license, which can be accessed from the Xilinx [IP Evaluation](#) page.

After purchasing the core, you will receive instructions for registering and generating a full core license. The full license can be requested and installed from the Xilinx [IP Center](#) for use with the Xilinx CORE Generator software v13.1. The CORE Generator software is bundled with the ISE<sup>®</sup> Design Suite software v13.1 at no additional charge.

Contact your local Xilinx [sales representative](#) for pricing and availability on Xilinx LogiCORE products and software.

## Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
03/28/03	1.0	Revision History added to document.
03/16/04	2.0	Updated to version 5.0 standards.
08/25/04	3.0	Updated to version 5.1 standards.
09/28/06	4.0	Updated to version 6.0 standards.
05/17/07	4.1	Updated to version 6.1 standards.
06/24/09	5.0	Updated to version 7.0 standards.
07/23/10	5.1	Updated to version 7.1 standards.
03/01/11	5.2	Support added for Virtex-7 and Kintex-7. ISE Design Suite 13.1 Reset last document version from 7.1 to 5.1 because document version is independent of core version.

## Notice of Disclaimer

Xilinx is providing this product documentation, hereinafter "Information," to you "AS IS" with no warranty of any kind, express or implied. Xilinx makes no representation that the Information, or any particular implementation thereof, is free from any claims of infringement. You are responsible for obtaining any rights you may require for any implementation based on the Information. All specifications are subject to change without notice. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE INFORMATION OR ANY IMPLEMENTATION BASED THEREON, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Except as stated herein, none of the Information may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx.