Reed-Solomon decoder IP core

Tom Szilagyi

Product datasheet

The Reed-Solomon decoder IP core is a highly versatile core generator for creating hardware Reed-Solomon decoder circuits suitable for high speed decoding of the ubiquitous Reed-Solomon Forward Error Correction (FEC) code. This code is used in a wide variety of data storage and transmission applications, such as the Compact Disc, DVD, various kinds of magnetic storage, digital subscriber lines and satellite communications.

The core generator is a Java application generating highly comprehensible, standard, device- and technology-independent VHDL code ready for consumption by industrial FPGA synthesis tools or ASIC processes, while also being suitable for educational use. Generated cores meet the requirements of various standards that specify Reed-Solomon codes, including ATSC, CCSDS[1], DVB, IESS-308 and Intelsat channel coding modes.

1 Features

- High speed, fully synchronous Reed-Solomon decoder using a single clock
- Implements any decoder specified via code parameters or chosen presets
- Supports continuous input data with no gaps between blocks
- One symbol in and out per clock cycle at any configuration
- Symbol size ranges from 3 to 12 bits; any primitive field polynomial usable for a given symbol size
- Supports shortened codes and erasure decoding
- Indicates errors, counts number of errors corrected and flags failures
- Support for marker bits useful for tagging input data, output in sync with output data
- User configurable generation of optional parts: VHDL for unused features is not generated, thus conserving resources
- Support for generating VHDL testbench matching the actual configuration, exercising all core features, as well as test input and reference output data

[1]Note that the converter to and from the dual-basis symbol representation required by the CCSDS standard is not generated, but has to be manually implemented as described in TM Synchronization and Channel Coding. Blue Book. Issue 1. September 2003. available at `http://public.ccsds.org/publications/archive/131x0b1.pdf`

2 Pinout

The pinout diagram of the generated Reed-Solomon core is shown in Figure 1. Common pins (present in all configurations) are shown with solid (black) arrowheads. Optional pins are shown with hollow (white) arrowheads. The pins are described in Table 1. Note that there are a few inter-dependencies between the optional pins, that is, enabling certain pins is required to get access to others. This is because certain parts of the generated logic depend on the presence of others. It is possible to select all or none of the optional pins.

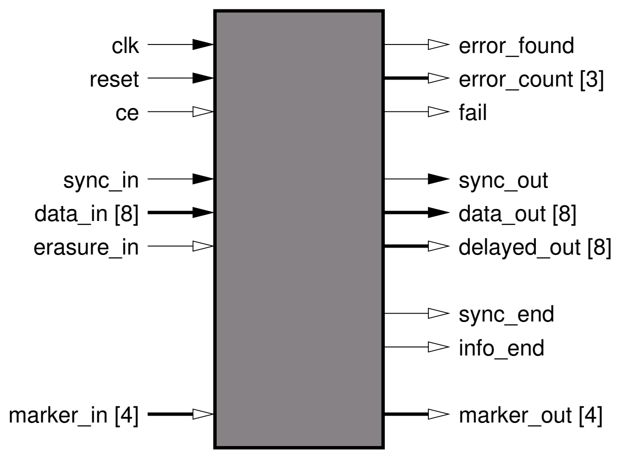

Figure 1: Core pinout diagram

The pins info_end and error_found are dependent on sync_end being enabled in the configuration. The error_count output depends on error_found (and thus also sync_end), while fail depends on error_count (and thus also error_found and sync_end).

3 Core generator parameters

The core generator is a standalone Java application requiring the Sun Java Runtime Environment (JRE) Standard Edition (SE) version 1.6 (also known as version 6)[2]. It guides the user through the task of configuring a particular Reed-Solomon decoder via an intuitive graphical user interface. The GUI enforces meaningful parameter combinations and indicates errors in the configuration that would lead to a malfunctioning core. Support is provided for saving the settings and loading them from a text file, as well as selecting predefined parameter sets

[2]The Java JRE can be freely downloaded from `http://java.com/en/download` and is available for all major computing platforms.

Signal	Direction	Optional	Description
clk	input	no	Clock, active on rising edge
reset	input	no	Active high asynchronous reset
ce	input	yes	Clock enable
sync_in	input	no	Codeword sync input. Must be driven high when first symbol of a codeword is passed to data_in, low otherwise
data_in	input	no	Input data
erasure_in	input	yes	Flag an input symbol as an erasure
marker_in	input	yes	Marker data input
sync_out	output	no	Codeword sync output, indicates the first symbol of a codeword being output
data_out	output	no	Corrected output data
delayed_out	output	yes	Original input data delayed with the core latency (in sync with data_out)
sync_end	output	yes	Codeword end sync, indicates the last symbol of a codeword being output
info_end	output	yes	Payload end sync, indicates the last symbol of the payload being output
error_found	output	yes	Indicates that there was at least one error in the last codeword
error_count	output	yes	Outputs the number of errors corrected in the last codeword
fail	output	yes	Indicates whether there was a decoding failure in the last codeword
marker_out	output	yes	Marker data output: data passed to marker_in delayed with core latency

Table 1: Core I/O signals

according to some popular standards. The input parameters are described below.

3.1 Code parameters

Symbol	Description	Range
M	Symbol width	3 ... 12 bits
n/a	Field polynomial	Any valid value
N	Codeword length	4 / 6 ... $2^M - 1$
K	Payload length	2 / 4 ... $N - 2$
m_0	Generator start index	0 ... $2^M - 2$
h	Scaling factor	1 ... $2^M - 2$

The lower limits of N and K depend on whether erasure decoding is supported by the core. If erasure decoding is not supported, $K \geqq 2$ and as a result $N \geqq 4$. If erasure decoding capability is requested, the lower limits for K and N are increased to 4 and 6, respectively.

The field polynomial is entered in decimal notation, for example at $M = 8$ bits per symbol a valid (and popular) field polynomial is 391 which is the decimal notation for $\alpha^8 + \alpha^7 + \alpha^2 + \alpha + 1$ where α commonly denotes the primitive element of the Galois Field[3].

The generator polynomial of the code that the Reed-Solomon decoder "understands" is given by the formula

$$G(z) = \prod_{i=0}^{R-1} \left(z - \alpha^{h(m_0+i)}\right)$$

where $R = N - K$ is the number of parity symbols. The formula shows the effect m_0 and h have on the result. The generator polynomial is conventionally the product of the first R terms of the form $\left(z - \alpha^i\right)$, where $0 \leqq i < R$. In our case m_0 allows to set the first root index (the lower limit of i) to differ from 0. Similarly h allows to have a spacing other than 1 between the root indices. It is important to note that h and the number of elements in GF$\left(2^M\right)$ have to be relative primes, i.e. $\gcd\left(h,\, 2^M - 1\right) = 1$ must be true, otherwise an incorrect core would be generated. This condition is enforced by the core generator GUI. A further restriction on the available code parameters is

$$T \geqq 2\,, \quad T = \begin{cases} R & \text{(erasures on)} \\ \left\lfloor \frac{1}{2} R \right\rfloor & \text{(erasures off)} \end{cases}$$

also enforced by the core generator application.

3.2 Optional features

The GUI lets the user select which optional features are desired while enforcing the aforementioned interdependencies. The width of the marker word (if used) can also be set in the range of 1–16 bits.

3.3 Output setup

The software lets setting the name of the Reed-Solomon decoder's top VHDL entity to avoid namespace collisions when multiple decoder cores are used in the same hardware project. This VHDL entity also acts as a prefix for the name of all sub-entities.

The core generator can optionally produce a VHDL testbench which reads input vectors from file, feeds them to the core, and compares the output from the core to

[3]For all valid field polynomials refer to the document Table of GF(2^M) field polynomials available as a separate download.

reference output also read from file. A configurable number of input testvectors are generated along with reference output. This is useful for verifying the functionality of the generated core.

The last step before the decoder's VHDL sources can be generated is specifying a target directory. A new directory named after the VHDL entity will be created here, containing the decoder's VHDL source files along with the testbench input and reference output data (if selected).

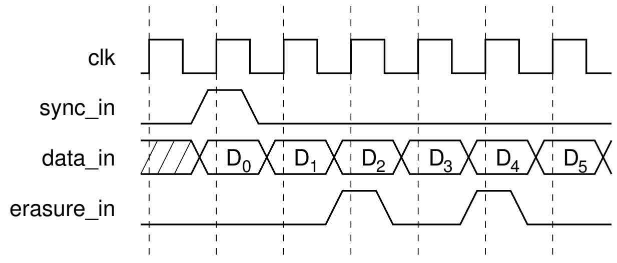

Figure 2: General input timing

4 Signal timing and behaviour

4.1 Core latency

The core has a fixed latency for any configuration parameter set. The core reads a new symbol at each clock cycle (enabled with ce, if selected) while outputting a corrected symbol at the same time. Thus, no extra buffering or flow control is necessary. The core latency equation is

$$L = N + R + T + 2\ , \quad T = \begin{cases} R & \text{(erasures on)} \\ \left\lfloor \frac{1}{2} R \right\rfloor & \text{(erasures off)} \end{cases}$$

Unlike some other implementations, the generated core never "runs out of time" for processing input data, despite the fact that the number of clock cycles needed to process a codeword (L) is greater than the length of a codeword (N). This capability is due to the pipelined design of the core.

4.2 General signal timing

Apart from the asynchronous reset signal, ce, and the error statistics outputs error_found, error_count and fail, each pin is sampled or set on each rising edge of clk, provided that ce is high at the time. ce is a true clock enable; deasserting it completely freezes the core's operation. Asserting reset clears internal counters of the design; however, it does not clear all internal data registers. After reset has been deasserted, output is undefined until the first symbol is output L clock cycles after it has been input.

If there are gaps between input codewords (sync_in is not high every N clock cycles) then there are corresponding gaps in the output during which the core acts as a dumb pipe of length L. That is, data_out and delayed_out mirror data_in, and marker_out mirrors marker_in with a delay of L clock cycles. The error statistics outputs hold their previously acquired values, while sync_out, sync_end and info_end are held zero.

Figure 2. illustrates the general input timing showing the use of sync_in and erasure_in. The figure shows that the beginning of a new codeword has been indicated via sync_in. Of the codeword symbols being input to data_in, D_2 and D_4 have been flagged as erasures via erasure_in.

4.3 Error statistics

The error statistics outputs error_found, error_count and fail are set after each codeword has been output. The values corresponding to a codeword are set on the rising edge of clk just after the last symbol has been output. If there are no gaps between codewords, this rising edge of clk also outputs the first symbol of the next codeword (sync_out is high). The above are demonstrated by Figure 3. which shows the timing of the fail output. The other error statistics outputs work in the same way.

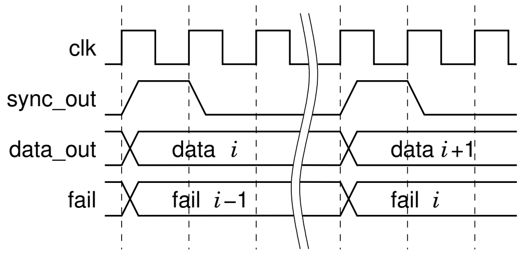

Figure 3: Statistics output timing

All error statistics outputs are held valid while the next codeword is being output, after which they are updated on the succeeding rising edge of clk. After a reset they are undefined until after the first codeword ends.

The core sets fail high if, after decoding a codeword, it is determined that not all errors in the input codeword were corrected. If the number of errors in the input codeword exceeds the error correcting capacity of the Reed-Solomon code, it is usually possible to detect this condition and assert fail. However, there may be cases when this is impossible. Note that this is a theoretical limitation and not a flaw in the decoder core. The probability of missing a decoding failure diminishes as R increases. Enabling erasure decoding support considerably increases this probability. For typical configurations ($8 \leqq R \leqq 20$) and erasure decoding off, the probability of missing a decoding failure is on the order of $10^{-4} \ldots 10^{-8}$ or even lower. With erasure decoding support enabled, this may rise to as high as $10^{-1} \ldots 10^{-4}$. These figures have been obtained via simulation over an artificial noisy channel where – in case of an uncorrectable codeword – the probability of exactly k symbol errors in the codeword is $P(k) = \frac{1}{2^{k-T}}$ for $T < k \leqq N$.

In case of a decoding failure (detected or not) the corresponding values on data_out are undefined.

Property	ATSC	DVB 1	DVB 2	CCSDS
M	8	8	8	8
N	207	204	204	255
K	187	188	188	223
R	20	16	16	32
m_0	0	0	0	112
h	1	1	1	11
Field Polynomial	285	285	285	391
Erasure decoding	no	no	**yes**	no
Latency	239	230	238	305
Slice utilisation [%]	23	19	39	40
LUT/FF pairs	907	744	1486	1439
LUTs	2477	2021	4444	4208
FFs	165	135	185	212
18k BRAMs	2	2	3	3
Max clock freq [MHz]	150	150	130	135

Table 2: Example decoders

5 Performance and resource usage

The area of the core increases with M, R, N (in this order of significance) and the optional features that are selected. Note that erasure decoding support is particularly resource-hungry: enabling it roughly doubles the overall resource demand! There is also a notable decrease in achievable clock speeds as the complexity of the decoder increases. However, this is less significant than the variance of the area requirement, since speed only varies according to the number of levels of logic, which is a logarithmic function of the total gate complexity.

When the selected parameters satisfy $R + T > N$, a slightly less area-efficient version of the Key Equation Solver module is generated because of pipeline limitations. However, such parameter sets are not used in practice.

Although the core generator supports symbol sizes up to 12 bits and the number of parity symbols can be as high as $N - 2$, there are practical limits on the size of a synthesizable core. In particular, symbol sizes above $M = 10$ and parity symbols above $T \approx 20$ yield cores that are too large to be implemented on current devices.

Table 2. shows performance and usage information for a few decoders implemented as examples with the freely available Xilinx ISE toolset. Optional pins were not used unless otherwise stated. The Xilinx FPGA XC5VLX (speed grade -3) was chosen for implementation. PAR effort was set to high with optimisation target set to speed. Note that resource usage and especially the maximum achievable clock frequency may vary with the settings of the ISE project and new versions of the Xilinx implementation tools. The aim of the figures is to give a hint about the achievable results (speed values have been slightly rounded downwards for safety), and are believed to be accurate within 10%.

To gain an overview of basic parameter dependencies, a representative set of decoders were implemented at a symbol width of $M = 8$ on the same FPGA device. No optional pins were used. PAR settings were left at Xilinx ISE application defaults (standard effort, area optimisation). The number of parity symbols were $R = 4$, 8, 12, 16 and 20. Codeword lengths were $N = 16$, 32, 64, 128, 192 and 255. All valid combinations of these parameters were formed.

Figure 4. indicates implementation results. The combined number[4] of LUTs and flip-flops as well as the maximum clock frequency are shown as a function of R. Since the results depend only slightly on N, the differences that stem from different values of N for a given R have been merged into error bars. Values shown are averages for all N while the errors show the greatest difference from the average value within the result set.

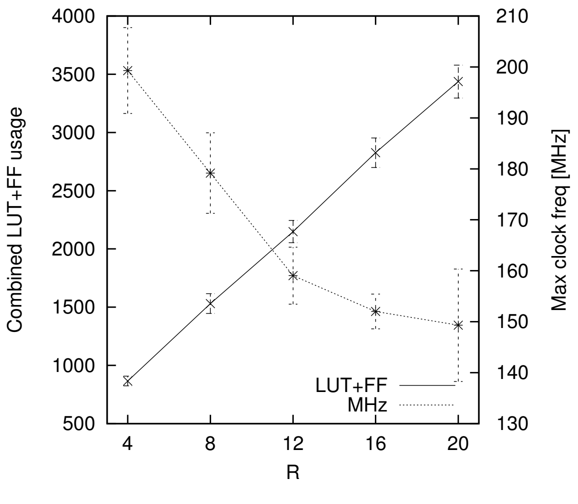

Figure 4: Performance and resource usage

Further information

The Reed-Solomon decoder IP core has been developed by Tom Szilagyi. Visit the product website at: `http://tomszilagyi.github.io/prod/reed-solomon` for updates and supplemental information.
Inquiries should be sent to tomszilagyi@gmail.com.

[4]This number includes slices with an unused flip-flop, slices with an unused LUT, and slices with fully used LUT-FF pairs.