XAPP1273 (v1.0) March 14, 2016

Reed-Solomon Erasure Codec Design Using Vivado High-Level Synthesis

Author: Matt Ruan

Summary

This application note focuses on the design of an erasure codec using the Xilinx® Vivado® High-Level Synthesis (HLS) tool, which takes the source code in C programming language and generates highly efficient synthesizable Verilog or VHDL code for a Kintex® UltraScale™ FPGA. When there is a need to change erasure code parameters, for example, the generator matrix definition or erasure code rate, only slight modification to the C header files is needed. The example Reed-Solomon erasure codec has a generic architecture from which other types of erasure codecs can be obtained by modifying the C source code.

You can download the Reference Design Files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

In modern societies, people produce and consume a large amount of data every day. Data centers are constructed to process and store the data and it is never acceptable for any data to get lost despite of the error-prone nature of the electronic components in the storage devices, such as hard disks, memories, networking connectors, and so on. A common data protection practice is to keep multiple copies of the same data at various locations [Ref 1] [Ref 2] so that a loss occurs only when all the copies of the data are lost or corrupted, which has a probability that decreases quickly as the number of redundant copies increases. In order to keep the loss probability sufficiently low, at least three identical copies of the data are kept in a typical data center. In other words, the amount of redundant data is more than twice that of the original, which is not very efficient.

A similar error protection problem has been studied for telecommunication systems where the transmit signal is often impaired by severe noise and adverse channel conditions. Instead of transmitting multiple replicas of the original signal, forward error correction codes are employed to facilitate automatic error correction at the receiver side. This idea also applies to the data centers where the Reed-Solomon (RS) erasure code is one error correction scheme that can fully recover as many errors as the added coding information [Ref 1]. To explain further, this means that if you add k words of coding information to the original n words of data, then it can tolerate k error words out of the total $(n+k)$ words. If there are $(k+1)$ error words, the errors cannot be recovered. Note that this is the best an error correction code can do and requires very careful construction of the code.

For example, to tolerate possible data corruption in 4 out of every 10 data blocks, only 4 additional data blocks are required to store the coding data with an overhead as low as 40%.

However, the RS erasure codec is a computation-intensive task that can take up to 30–40% of the processor run time in a middle-range server. This is considerable overhead that calls for flexible, scalable, and resource-efficient accelerators built on programmable logic [Ref 3].

This application note explains how to describe the erasure codec in C programing language, and then use the Vivado HLS tool (see *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 4] to synthesize the C code into hardware description language (HDL) and implement the C code on programmable logic devices. Vivado HLS can generate high-performance pipelined architecture according to the given constraints, and create test benches to ensure the behaviors of HDL and C code are identical. In many cases, the Vivado HLS synthesized code has similar efficiency and performance to that of hand-coded HDL designed by an experienced logic engineer. When there is a need to change the clock rate, target logic device, or erasure codec parameters such as the generator matrix or code rate, only slight modification to the C header files is required. The C design of the erasure codec literally becomes an IP that can be easily customized for new applications by software engineers.

Theory of Operation

This section explains the basic theory of forward error correction in an erasure channel and how the erasure codec works.

In an erasure channel, define A as the alphabet of the transmit signal. (Here *alphabet* means the set of transmit signals. For example, {-1, +1} is the alphabet of the BPSK modulation signal.) For any given received symbol $S \in A$, if the erasure flag of the received symbol is not set, then the transmit symbol is certain be S; otherwise, the symbol can be anything in the alphabet with equal probability. When the latter occurs, it is as if the symbol is erased in the channel without providing any useful information to the receiver. Note that erasure flags are not available in typical communication channels where more complicated error correction schemes are required. While in data centers, the data integrity checking circuitry built in the storage devices is able to indicate erasure events.

Figure 1 illustrates how the erasure codes are constructed.

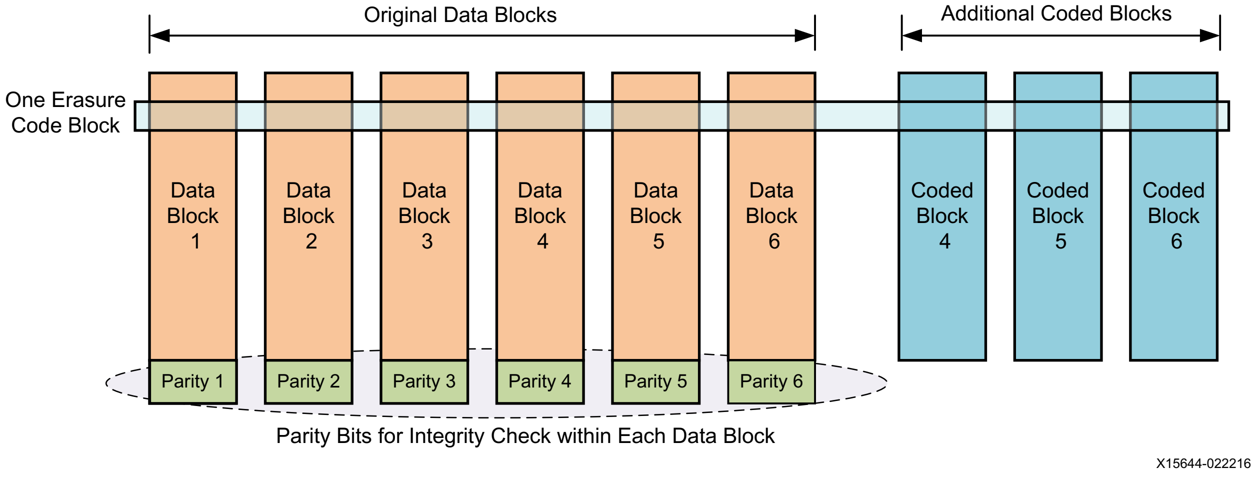
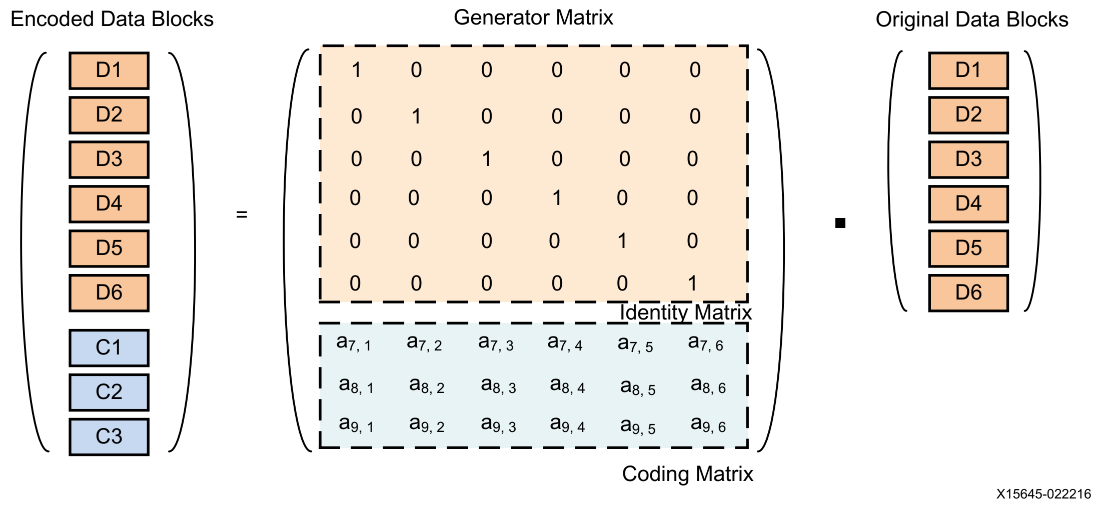


Figure 1: **Block Diagram of an Erasure Code**

The orange boxes are original data blocks, the length of which varies from 1 kilobyte to 128 megabytes. The green boxes are the parity check information appended to each data block for integrity testing. The size of the parity check information is much smaller than that of the data block, and in many cases integrity testing is a built-in function of the storage device. On the right side of the diagram is the coded data blocks generated by the erasure codec. The encoding is performed horizontally across all the data blocks. For the case shown in Figure 1, one erasure code block consists of six information symbols, one from each of the data blocks, and three coded symbols computed by the erasure encoder.

If we denote D_i as the symbol taken from the i^{th} data block and C_i as the i^{th} coded symbol, then the encoding process can be expressed by one matrix dot multiplication defined in the Galois Field of the given symbol bitwidth [Ref 1]. This is illustrated in Figure 2 where all the symbols including the generator matrix coefficients $\{a_{i,\ j}\}$ and the dot multiplication operation are defined on the same Galois Field.



$$\begin{pmatrix} D1 \\ D2 \\ D3 \\ D4 \\ D5 \\ D6 \\ C1 \\ C2 \\ C3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} \\ a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} \\ a_{9,1} & a_{9,2} & a_{9,3} & a_{9,4} & a_{9,5} & a_{9,6} \end{pmatrix} \cdot \begin{pmatrix} D1 \\ D2 \\ D3 \\ D4 \\ D5 \\ D6 \end{pmatrix}$$

Figure 2: **Encoding of Erasure Codes**

When some data blocks are lost or corrupted, as shown in Figure 3 (a), it is possible to recover the data using the survived data blocks and the pre-known information about the generator matrix. The calculation procedure is illustrated in Figure 3 (b) where the dot multiplication of the survived data blocks with the inverse of the survived rows (shown in light blue) gives the original data block, and then the dot multiplication with the erased rows (shown in gray) recovers the erased data blocks.

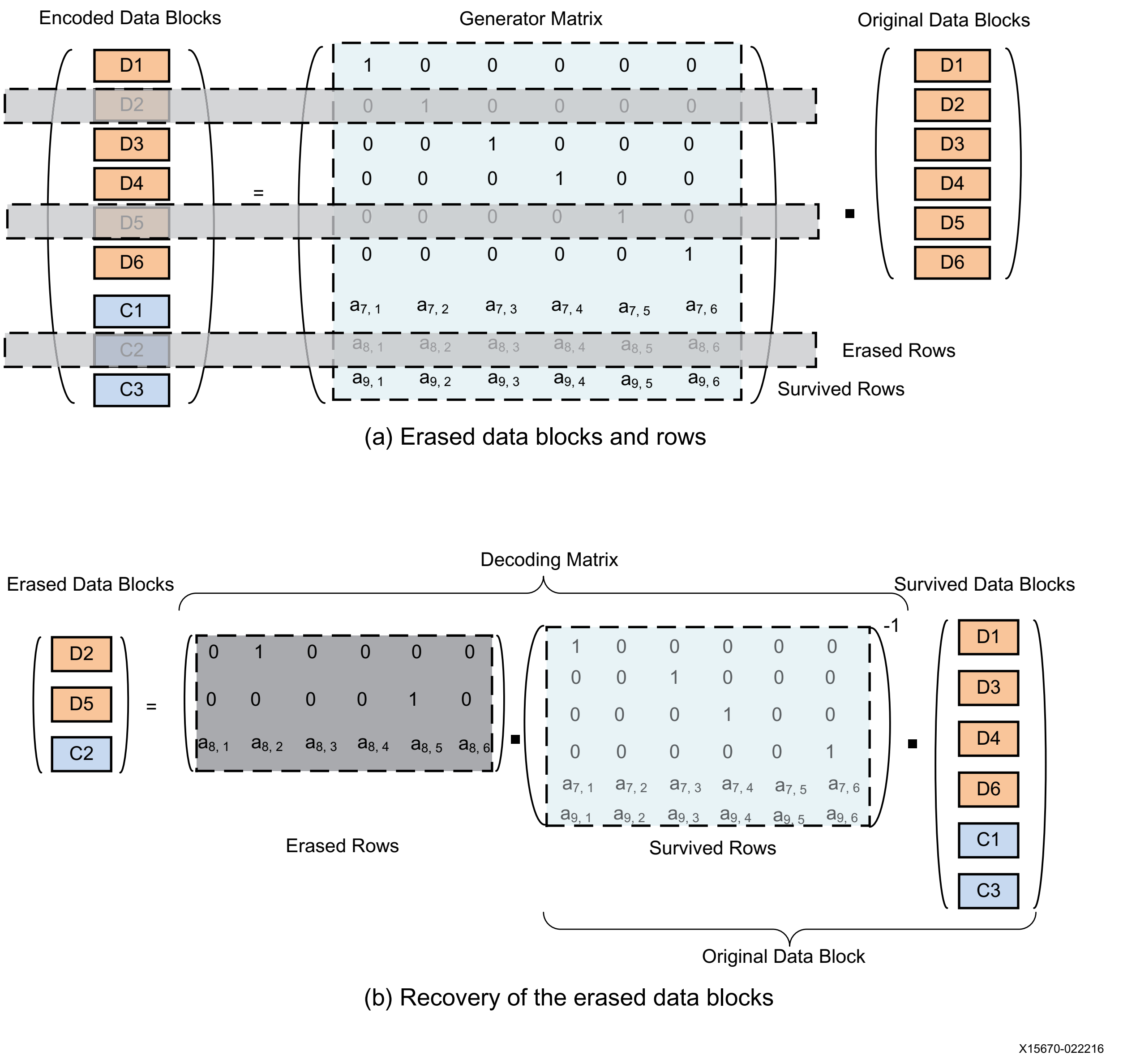


Figure 3: **Decoding of Erasure Codes**

As shown in Figure 3 (b), the decoding of erasure codes implicitly assumes the existence of the inverse of the matrix consisting of the survived rows. This assumption holds for some specially constructed generator matrices, one of which is the Reed-Solomon code [Ref 1] that is used as an example in this application note. Nevertheless, the design methodology described in this application note also applies to many other kinds of erasure codes.

Figure 3 illustrates another fact that the encoding process can be seen as a special case of decoding where all the original data blocks are survived and the coded blocks are erased. Therefore one only needs to design an erasure decoder which is also capable of encoding.

RS Erasure Codec Architecture

In data centers a number of error protection levels are needed for data blocks of different importance. This requires the codec to support a number of code rates. Also latency has to be minimized to facilitate on-the-fly coding and decoding schemes for short data blocks.

This application note gives an example of designing such a multi-rate erasure codec on Xilinx Kintex UltraScale FPGAs that typically run at 300 MHz or higher clock frequencies. The design features a streaming input interface with very short decoding latency while supporting four code rates ranging from 1/3 to 2/3 as summarized in Table 1. A wider range of code rates can be supported with minor modification to the design.

Note: This example is for Kintex FPGAs but the method or design applies to virtually any device.

Table 1: **Code Rates of the Low-latency Multi-rate Erasure Codec**

Number of Data Blocks	Number of Code Blocks	Overhead Percentage
6	4	67%
8	4	50%
10	4	40%
12	4	33%

Figure 4 shows the desirable interface timing diagram of the erasure codec where the light blue, yellow, and green colors represent three data sets with different erasure decoding parameters, that is, survival patterns and erasure code types. Because no gap insertion is required for the change of parameters, the input is simplified to a streaming interface for easy interconnection with other system components.

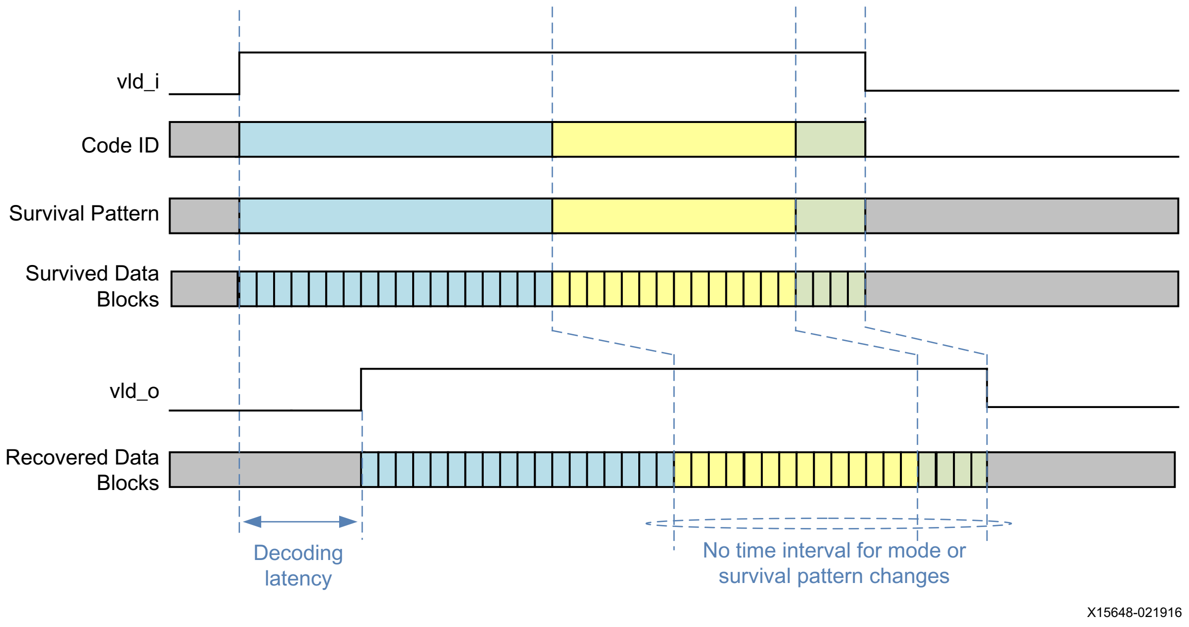


Figure 4: **Interface Timing Diagram of the Erasure Codec**

To realize the simple streaming interface, a fully pipelined architecture is selected and the matrix inversion becomes the major challenge because it takes hundreds of clock cycles to complete. One solution is to pre-calculate all the decoding matrices for all supported codes and possible survival patterns and save them in internal memories of the programmable logic device. Then the overall architecture of the erasure codec looks like Figure 5.

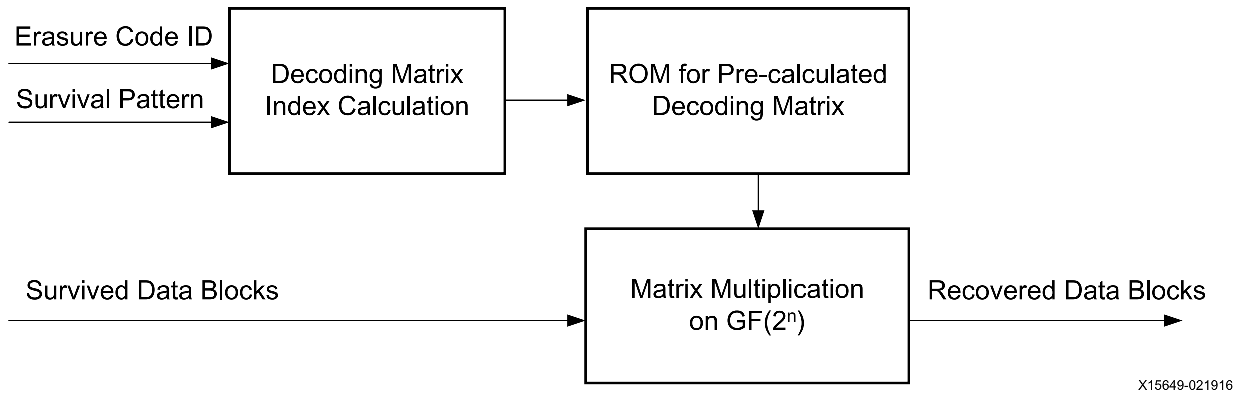

Erasure Code ID
Survival Pattern
Decoding Matrix Index Calculation
ROM for Pre-calculated Decoding Matrix
Survived Data Blocks
Matrix Multiplication on GF(2ⁿ)
Recovered Data Blocks
X15649-021916

Figure 5: **High-level Architecture of the Erasure Codec**

The size of the decoding matrix ROM needs to be feasible for programmable devices. For a given erasure code of *m* original data blocks and *k* coded blocks, the number of possible survival patterns is given by $(m+k)! / m! / k!$ where $n!$ represents the product of all positive integers not larger than *n*. Then the total number of survival patterns for all four erasure code types can be computed as shown in Equation 1:

$$10! / 4! / 6! + 12! / 4! / 8! + 14! / 4! / 10! + 16! / 4! / 12! = 3{,}526 \qquad \textit{Equation 1}$$

Each of the decoding matrices has a maximum of $12 \times 4 = 48$ elements on $GF(2^8)$, and the size of the ROM is $3{,}526 \times 48 \times 8 = 1{,}353{,}984$ bits, which is quite affordable for 20 nm UltraScale FPGAs.

Based on the high-level architecture, the algorithms are described in C programming language and the Vivado HLS tool explores various hardware implementation options and automatically selects the best one for the design.

Implementation Details

The architecture of the erasure codec (Figure 5) consists of two major calculation steps to be implemented in C programming language: one is to compute the decoding matrix index for the given survival pattern and code ID, and the other is to multiply the decoding matrix to the survived data blocks.

Decoding Matrix Index Calculation

The survival pattern can be represented by a string of bits with 1s indicating the survived data blocks and 0s indicating those being erased. For the example shown in Figure 3, the survival pattern can be written as `101101` in binary or `0x2D` in hexadecimal. To simplify the representation, the survival pattern can be translated into the erasure pattern by an XOR operation:

```
// translate the survival pattern into erasure pattern
unsigned short errpat = survival_pattern^((1<<RSCODE_LEN)-1);
```

where `RSCODE_LEN`, which equals (k+m) in this case, is the total number of data blocks including parity. There is a one-to-one mapping between the erasure pattern and decoding matrix; however, the look-up table is huge when k+m) is large. For example, suppose $m = 12$ and $k = 4$, there are (12+4)! / 12! / 4! = 1,820 decoding matrices, and the size of the look-up table is $2^{(12+4)} \times \text{ceil}(\log_2 1820)$ = 720,896 bits, occupying 40 pieces of BRAM18K. A more efficient method is required to calculate the decoding matrix index from a given erasure pattern.

Write integers in binary format as shown in Equation 2:

$$\bar{z} = \sum_{i=0}^{L-1} z_i \bullet 2^i \qquad \textit{Equation 2}$$

where $z_i \in \{0,1\}$ and L is the number of bits.

Define a set of integers $\Omega(\bar{y}_{L,p}) = \{\bar{z} \mid \bar{y}_{L,p} < \bar{z} < 2^L \text{ and } \sum_{i=0}^{L-1} z_i = p\}$

where $\bar{y}_{L,p}$ is the L-bit long integer representing the erasure patter of p erased data blocks. The construction of Ω ensures that all the L-bit integers in the set are larger than $\bar{y}_{L,p}$ and have exactly p bits being 1. For any erasure pattern $\bar{y}_{L,p}$, denote $\Phi(\bar{y}_{L,p})$ as the number of elements in $\Omega(\bar{y}_{L,p})$. The following iterative relationship is observed for $\Phi(\bar{y}_{L,p})$:

$$\Phi(\bar{y}_{L,p}) = \begin{cases} 0 & \text{if } L \le 1 \text{ or } p = 0, \\ \Phi(\bar{y}_{L-1,p}) + f(L,p) & \text{if } y_{L-1} = 0, \\ \Phi(\bar{y}_{L-1,p-1}) & \text{otherwise.} \end{cases}$$

where

$$f(L,p) = \begin{cases} 0 & \text{if } p = 0, \\ 1 & \text{if } p = 1, \\ \dfrac{(L-1)!}{(p-1)!\,(L-p)!} & \text{otherwise.} \end{cases}$$

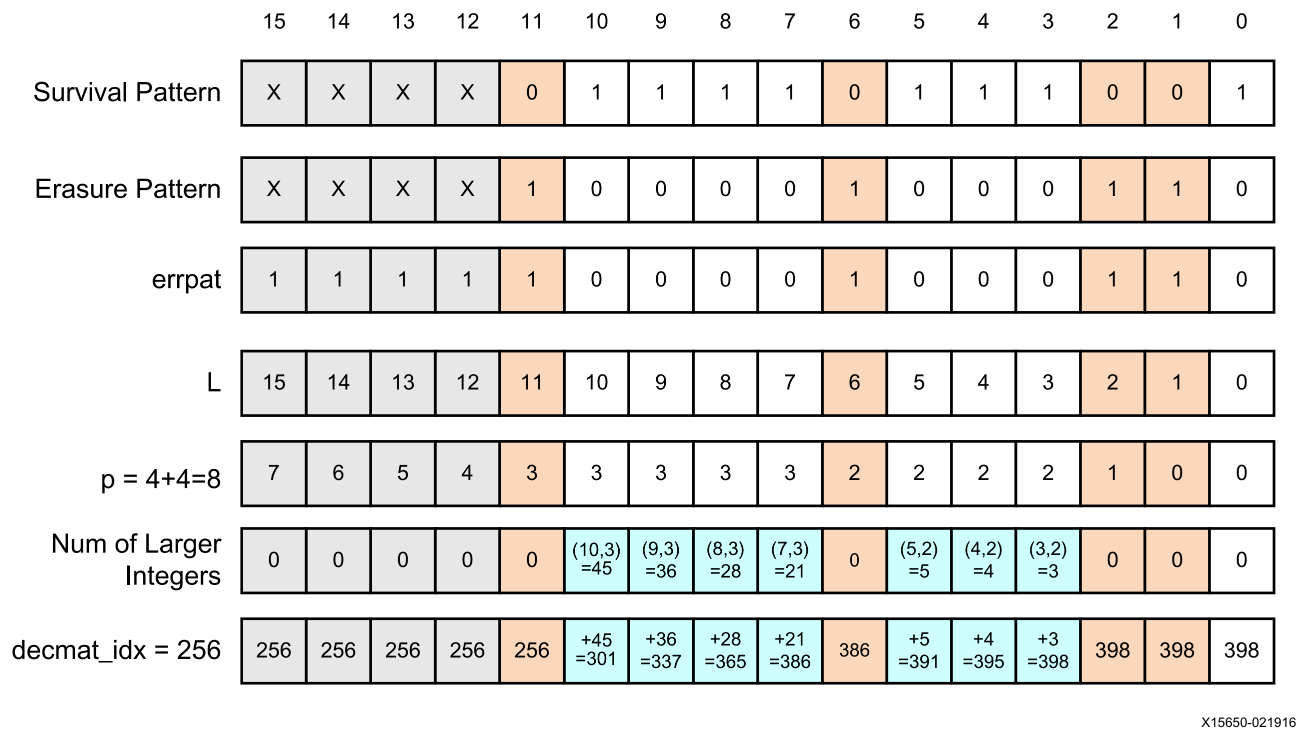
Define $\Phi(\bar{y}_{m+k,k})$ as the decoding matrix index of erasure pattern $\bar{y}_{m+k,k}$. The above iterative relationship leads to a bit-by-bit scanning algorithm described in C code as follows:

```
// pad 1's to the msb of the error pattern
unsigned char  padlen  = (3^(codeidx&3))<<1;
unsigned short padmask = ((1<<padlen)-1)<<(RSCODE_LEN-padlen);
unsigned short errpat  = erasure_pattern | padmask;

// Initialize the number of remaining ones = (p+1)
char p = NUM_EQUATION-1 + padlen;

// Scan the bits of errpat from MSB to LSB
for(k=0; k<RSCODE_LEN; k++){

   // number of remaining bits = (L+1)
   unsigned char L = (k^0xf)&0xf;

   // extract the msb of the remaining bits
   unsigned char msb = (errpat>>(RSCODE_LEN-1))&1;

   //if this bit is 0, then count the number of integers larger than it
   //if this bit is 1, decrease p by 1
   if (msb) p--;
   else if(p>=0) decmat_idx+=F_tbl[((p&3)<<4) | (L&0xf)];

   // dump the msb and move to the next bit
   errpat=(errpat<<1);
}
```

where the $f(L, p)$ function is realized by the look-up table `F_tbl`. To support multiple erasure code types, a base address is assigned to `decmat_idx` at the beginning and then $\Phi(\bar{y}_{m+k,k})$ is calculated as an offset to the base address.

Figure 6 illustrates one example of a decoding matrix index calculation. The code ID is 0x1 indicating it is RS(8, 4) code with 8 + 4 = 12 effective bits in the survival pattern, and the erasure pattern have ones padded to bits 12 to 15. The decoding matrix index `decmat_idx` is initialized with the ROM base address of the RS(8, 4) code, which is 256. Because the five most significant bits of `errpat` are all 1, the value of `decmat_idx` remains the same until bit 10, where an increment by $f(L,p) = f(10,3) = 45$ occurs. The next three bits, which are bit 9 down to

7, are all 0 so the value of `decmat_idx` increments to 386 before another 1 is reached at bit 6. The iterative process keeps going until the end of the bit string.



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Survival Pattern	X	X	X	X	0	1	1	1	1	0	1	1	1	0	0	1
Erasure Pattern	X	X	X	X	1	0	0	0	0	1	0	0	0	1	1	0
errpat	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	0
L	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p = 4+4=8	7	6	5	4	3	3	3	3	3	2	2	2	2	1	0	0
Num of Larger Integers	0	0	0	0	0	(10,3) =45	(9,3) =36	(8,3) =28	(7,3) =21	0	(5,2) =5	(4,2) =4	(3,2) =3	0	0	0
decmat_idx = 256	256	256	256	256	256	+45 =301	+36 =337	+28 =365	+21 =386	386	+5 =391	+4 =395	+3 =398	398	398	398

X15650-021916

Figure 6: **One Example of a Decoding Matrix Index Calculation (codeid = 1, survival_pattern = 0x7B9)**

Matrix Multiplication on GF(2^n)

On GF(2^n) every integer represents a polynomial with the i^{th} bit indicating the coefficient of the term x^i, where x is the root of the Galois Field generator polynomial. Then one symbol of the data block $\overline{d}$ and one element of the decoding matrix $\overline{r}$ can be written into the polynomial format:

$$\overline{d} = \sum_{i=0}^{n-1} d_i x^i, \qquad d_i \in \{0,1\}$$

$$\overline{r} = \sum_{j=0}^{n-1} r_j x^j, \qquad r_j \in \{0,1\}$$

Therefore,

$$\overline{r}\cdot\overline{d} = \overline{r}\cdot\sum_{i=0}^{n-1} d_i x^i = \sum_{i=0}^{n-1} d_i \cdot (x^i\cdot\overline{r}) = \sum_{i=0}^{n-1} d_i\cdot R_i$$

where

$$R_i \triangleq x^i\cdot\overline{r} = x\cdot x^{i-1}\cdot\overline{r} = x\cdot R_{i-1}.$$

with $R_0 = \overline{r}$.

The multiplication with x on GF(2^n) is a simple left shift of the integer followed by a reduction operation by the generator polynomial. Therefore, R_i can be calculated iteratively, and the multiplication with $d_i \in \{0,1\}$ can be computed by a multiplexer with 0.

More specifically, the matrix multiplication on GF(2^n) can be described by the following C code:

```
// initialize the decoding matrix polynomial
for(k=0;k<NUM_ELEMENT;k++) r[k] = DECMAT_ROM[decmat_idx][k];

// reset all the code bits
for(k=0; k<NUM_EQUATION; k++) c[k] = 0;

// loop for all the bits of the input data
for(i=0; i<GF_ORDER; i++)
   // loop for all survived data blocks
   for(j=0; j<NUM_TAPS; j++)
      // loop for all coding equations
      for(k=0; k<NUM_EQUATION; k++){
         unsigned char idx = k*NUM_TAPS+j;
         unsigned char  tmp = r[idx];
         // update c
         c[k] = c[k] ^  ( ((d[j]>>i)&1)? tmp : 0 );
         // update r
         r[idx] = ((tmp>>7)&1) ? ((tmp<<1)^gf_poly) : (tmp<<1);
      }
```

Directives

When converting the C code into HDL, Vivado HLS needs some side information to describe parameters like the number of clock cycles available to complete a loop, whether the module can accept new inputs before old ones are all processed, and so on. These directives, which are an integral part of the design, specify how the C code is supposed to be synthesized into HDL for the desirable behavior. When porting an existing design to a new application, sometimes only slight modifications to the directives are needed without touching the C code.

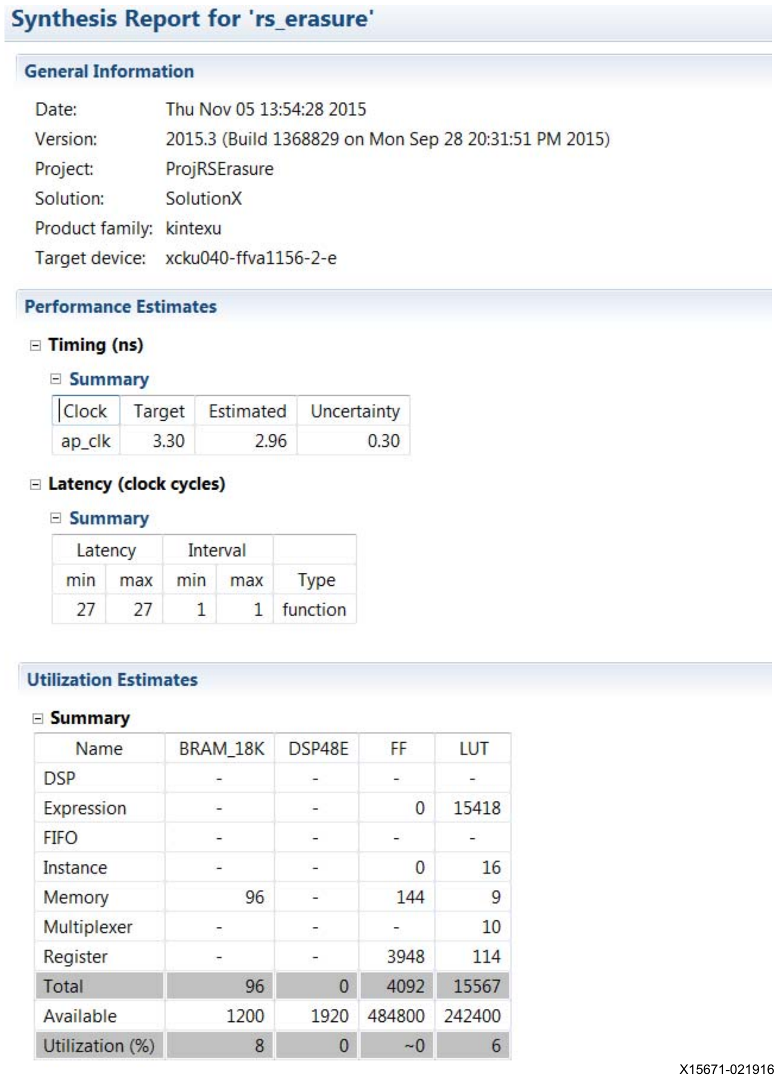
For the erasure codec, the following directives are used to realize the desirable behavior. Directives are kept to the minimum so that at a later stage, the Vivado synthesis tool can more flexibly select synthesis options given visibility to the whole design. For instance, the decoding matrix ROM can be implemented in either block or distributed RAMs. This decision is better made by the Vivado synthesis tool, which can balance resource utilization after integration with other system modules.

```
# The function takes pipelined architecture and accepts new inputs every
clock cycle
set_directive_pipeline -II 1  rs_erasure

# the arrays c and d should be partitioned completely for the access
# to all the elements of the array in one clock cycle
set_directive_array_partition -type complete rs_erasure c
set_directive_array_partition -type complete rs_erasure d
```

```
# The interface definition
set_directive_interface -mode ap_none  rs_erasure d
set_directive_interface -mode ap_vld   rs_erasure codeidx
```

Synthesis Results

Xilinx Vivado HLS analyzes all the design files and automatically selects the appropriate hardware architecture to meet the target clock frequency and data throughput specified in the form of synthesis directives. After the C compilation is completed, basic information about the synthesized HDL can be reviewed to check against the design targets. Figure 7 shows the erasure codec synthesis report generated by Vivado HLS.



Synthesis Report for 'rs_erasure'

General Information

Date:	Thu Nov 05 13:54:28 2015
Version:	2015.3 (Build 1368829 on Mon Sep 28 20:31:51 PM 2015)
Project:	ProjRSErasure
Solution:	SolutionX
Product family:	kintexu
Target device:	xcku040-ffva1156-2-e

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.30	2.96	0.30

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
27	27	1	1	function

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	15418
FIFO	-	-	-	-
Instance	-	-	0	16
Memory	96	-	144	9
Multiplexer	-	-	-	10
Register	-	-	3948	114
Total	96	0	4092	15567
Available	1200	1920	484800	242400
Utilization (%)	8	0	~0	6

X15671-021916

Figure 7: **C Synthesis Report**

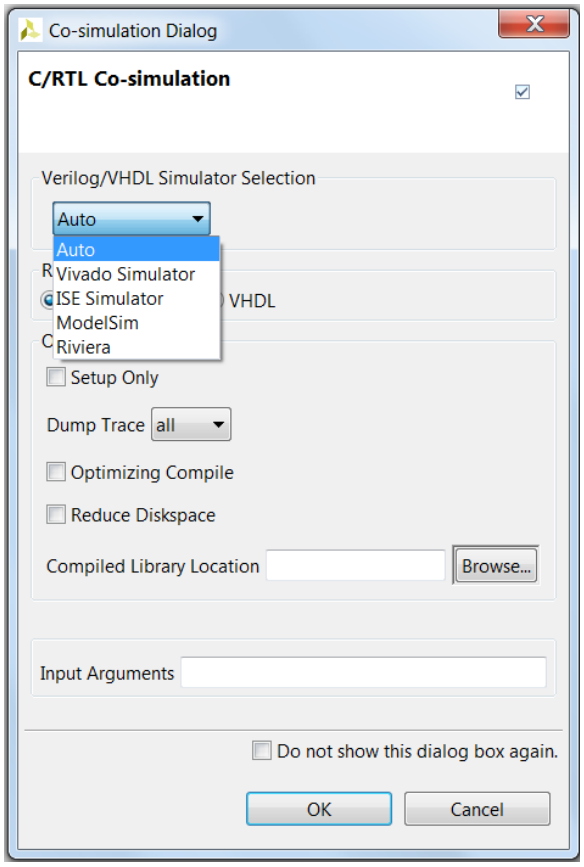
According to the report, the HLS generated RTL is estimated to run at 1/2.96 ns = 338 MHz, which meets the requirement of 300 MHz. The clock frequency estimate at the C synthesis stage can have ±15% mismatch with the final result because the logic routing latency is not fixed until the place and route process is completed. The synthesis report also shows that decoding latency of the design is 27 clock cycles, which is about 90 ns under 300 MHz clock. The pipelined input interval is 1, which means the maximum throughput of the codec is 12 data blocks $\times$ 8 bits per data block $\times$ 300 MHz = 28.8 Gb/s for the input, and 4 data blocks $\times$ 8 bits per data block $\times$ 300 MHz = 9.6 Gb/s for the output. When needed, multiple erasure codecs can be instantiated to work in parallel for higher data throughput.

Verification Results

In the Vivado HLS design flow, functional verification consists of two steps.

The first step is C functional verification that validates the C code against the golden test vectors. Because the C library provides rich file I/O functions, it is straightforward to code a C test bench based on the pre-stored input and output test vectors. The erasure codec reference design follows this approach. The golden test vector contains 1000 runs of erasure decoding covering all four erasure code rates, with 25 randomly selected survival patterns for each code rate and 10 sets of data blocks for each survival pattern.

After the C behavior has been verified and the C functions are synthesized into HDL, Vivado HLS can automatically generate an HDL test bench according to the C test code. This step is referred to as *C/RTL Co-simulation*, which ensures the HDL behavior matches C functionality. As shown in Figure 8, Vivado HLS supports a number of simulators and HDL code formats for C/RTL co-simulation.



X15672-021916

Figure 8: **C and HDL Co-simulation Dialog Window**

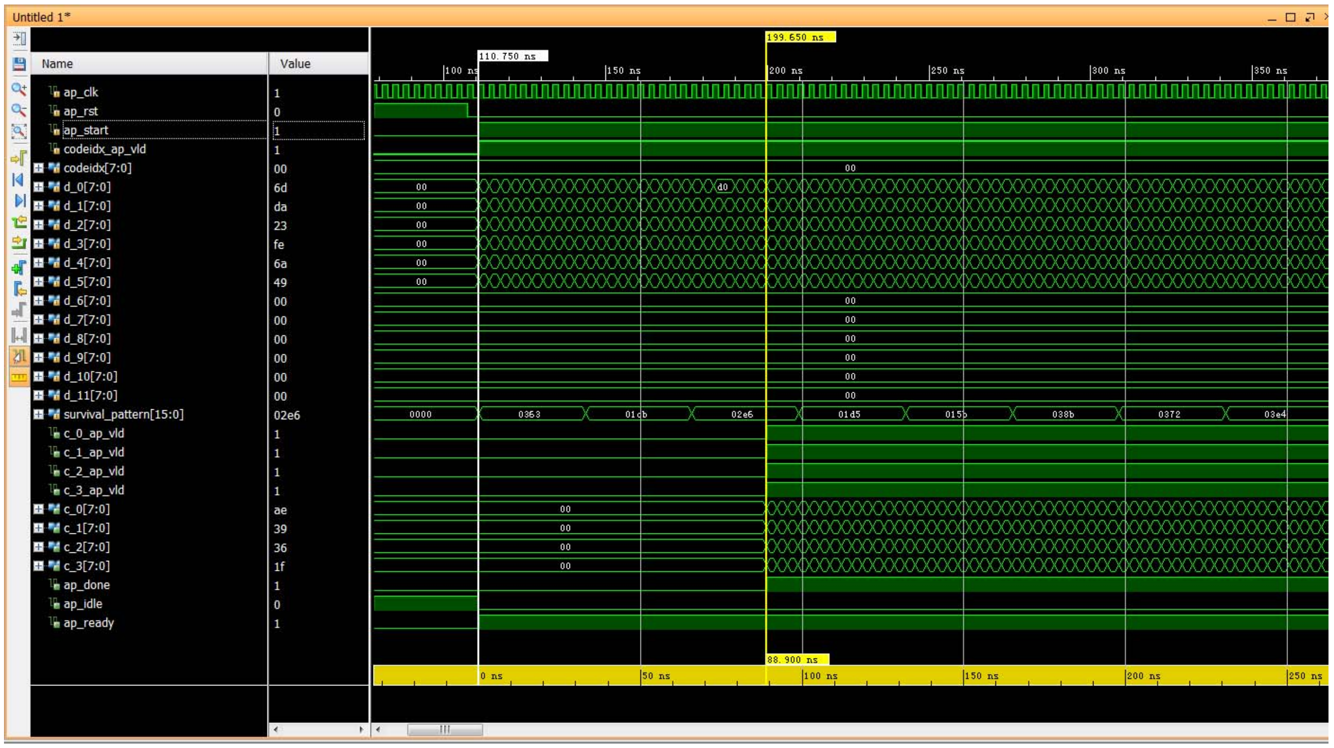
The outputs of the HDL design are compared to that of the golden test vectors to ensure the functionality is correct. At the end of simulation, Vivado HLS prints the post checking results, which look like the following:

.....

```
****** xsim v2015.3 (64-bit)
  **** SW Build 1368829 on Mon Sep 28 20:06:43 MDT 2015
  **** IP Build 1367837 on Mon Sep 28 08:56:14 MDT 2015
    ** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

source xsim.dir/rs_erasure/xsim_script.tcl
# xsim {rs_erasure} -maxdeltaid 10000 -autoloadwcfg -tclbatch
{rs_erasure.tcl}
Vivado Simulator 2015.3
Time resolution is 1 ps
source rs_erasure.tcl
## add_wave -r /
WARNING: Simulation object
/apatb_rs_erasure_top/next_trigger_ready_cnt was not traceable in the
design for the following reason:
Vivado Simulator does not yet support tracing of Verilog named events.
```

```
## save_wave_config rs_erasure.wcfg
## run all
$finish called at time : 3512850 ps : File
"C:/MattRuan/appnote/ProjRSErasure/SolutionX/sim/verilog/rs_erasure.au
totb.v" Line 1276
## quit
INFO: [Common 17-206] Exiting xsim at Thu Nov 05 13:58:56 2015...
[0]   0 out of 250 test vectors failed.
[1]   0 out of 250 test vectors failed.
[2]   0 out of 250 test vectors failed.
[3]   0 out of 250 test vectors failed.
Total 1000 Test Vectors, Err Count = 0.
Test Passed!
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
```

There is also an option to dump signal traces for manual debugging on the waveforms. Figure 9 is a simulation waveform generated by Vivado Simulator 2015.3 for the test case of various code rates and survival patterns. From the waveforms shown in Figure 9, it can be checked that the latency of the multi-mode erasure codec is 88.9 ns/3.3 ns = 27 clock cycles, which matches that in the synthesis report.

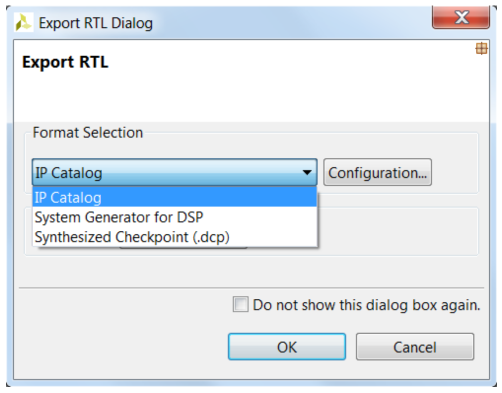


X15673-021916

Figure 9: **C and HDL Co-simulation Waveforms**

Implementation Results

Xilinx Vivado HLS not only generates the HDL code of the C functions, but also provides a number of options to package the HDL into an IP ready for integration into a larger design using the Vivado Design Suite, for example, System Generator, embedded development kit (EDK), and IP integrator. For illustration purposes, IP Catalog has been selected for the example reference design (see Figure 10).



X15674-021916

Figure 10: **HDL Export Dialog Window**

The Vivado HLS tool automatically creates a Vivado project and synthesizes all the HDL code to validate performance of the implementation. The final implementation report for the erasure codec confirms the fulfillment of design targets:

```
Implementation tool: Xilinx Vivado v.2015.3
Device target:       xcku040-ffva1156-2-e
Report date:         Thu Nov 05 14:30:47 +0800 2015
#=== Resource usage ===
CLB:               4411
LUT:              26010
FF:                5264
DSP:                  0
BRAM:                 0
SRL:                110
#=== Final timing ===
CP required:     3.300
CP achieved:     3.143
Timing met
```

The report shows that the synthesis tool selects distributed RAM instead of block RAM for the implementation of the decoding matrix ROM. This is because only RS(12, 4) utilizes the full storage space and other codes have many zeros padded in the decoding matrix ROM. The synthesis tool detects the sparseness of the ROM and decides to implement it in distributed RAM for better timing performance.

Reference Design

You can download the Reference Design Files for this application note from the Xilinx website.

Design File Hierarchy

The directory structure under the top-level folder is as follows:

```
\src
 | This folder contains C design files and header files.
 |
\tb
 | This folder contains a C design file that serves as the test bench.
 |
\tv
 | This folder contains the input and output golden test vectors for
 |   verification purposes.
 |
```

Installation and Operating Instructions

1. Install the Xilinx Vivado tools, version 2015.3 or later.
2. Unzip the design files into a clean directory.
3. In the Vivado HLS command line window:
    a. **cd** to the root of the design directory.
    b. Type **vivado_hls run.tcl**
    c. Check that the synthesized design meets expectations.

Table 2 shows the reference design matrix.

Table 2: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Matt Ruan
Target devices	Kintex® UltraScale™ FPGAs (KU040, KU060) Virtex®-7 FPGA (7V690) ***Note:*** The reference design was tested on Virtex-7 and Kintex UltraScale FPGAs, however, the design should work on other devices, such as the Zynq UltraScale+ MPSoC or even Artix®-7 and Spartan®-6 FPGAs. The achievable throughput on Spartan-6 and Artix-7 devices is expected to be lower than that of Kintex and Virtex UltraScale devices.
Source code provided	Yes

Table 2: **Reference Design Matrix *(Cont'd)***

Parameter	Description
Source code format	C, test vectors, and synthesize script
Design uses code and IP from existing Xilinx application note and reference designs or third party	No
Simulation	
Functional simulation performed	Yes
Timing simulation performed	No
Test bench used for functional and timing simulations	Yes
Test bench format	C
Simulator software/version used	Vivado Simulator 2015.3
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/versions used	Vivado High-Level Synthesis 2015.3
Implementation software tools/versions used	Vivado Design Suite 2015.3
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	No
Hardware platform used for verification	N/A

Conclusion

This application note demonstrates a method of building a low-latency multi-rate erasure codec using the Vivado HLS tool chain which takes C code as input and generates HDL code synthesizable on FPGAs. The C source code is easy to maintain and scalable to various FPGA parts, erasure code rates, and system clock frequencies. By modifying the decoding matrix ROM, it is simple to build other types of erasure codecs out of the generic pipelined GF(2^n) matrix multiplication architecture.

References

1. Burkhard, W. A. and Menon, J. (June 1993) "Disk array storage system reliability." *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, pp. 432–441

2. Dimakis, A.G.; Prabhakaran, V.; Ramchandran, K. (June 2006) "Decentralized erasure codes for distributed networked storage." *IEEE Transactions on Information Theory,* pp. 2809–2816, Volume 52, Issue 6

3. Sobe, Peter (March 2009) "Coding for Reliable Data Storage on Different Hardware Platforms." *Architecture of Computing Systems - ARCS 2009: 22nd International Conference Proceedings,* pp. 1-6

4. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/14/2016	1.0	Initial Xilinx release.

Please Read: Important Legal Notices