A demonstration FPGA symbol generator for DSN telemetry testing

J.Masdeu INSA
Madrid Deep Space Communication Center
July 7, 2011

Abstract

A demo FPGA symbol generator (FSG) for Deep Space Network (DSN) telemetry testing is presented. Most common codes, CCSDS and DSN convolutional codes, Turbo, Cassini and Reed Solomon codes are implemented in this FSG. The realization was done using entry level, unmodified FPGA development board, and the s/w tools provided with the kit. The prototype outputs a parametrized telemetry symbol stream analyzable by the System Performance Tester (SPT). The unit underwent extensive tests and no errors were observed in the symbol stream. Operation is controlled from a serial port using a directive set similar to Test Simulator Assy (TSA) [1]

Introduction

Simple designs are often very robust and provide reliable operation in the long run. Availability of modern FPGAs has made possible the implementation of very flexible, reconfigurable data generators that can be tailored to the DSN needs without costly hardware board design. Such is the case of the demo hardware symbol generator that uses a commercial low cost FPGA development board. Fig 1 identifies the major vhdl software components.

Frame generator

It comprises two elements, the sync word generator and the data generator. It provides five types of data, all ones, all zeroes, square wave,loadable frame buffer RAM and a pseudorandom noise (PN) sequence compatible with the System Performance Test (SPT) BER analyzer.

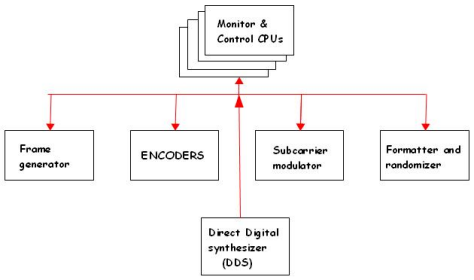


Figure 1: Major components of *FSG*.

PCM formatter

Encodes symbol streams either NRZL or Biphase

Randomizer

Implements CCSDS recommendation using generator polynomial
$X^8 + X^7 + X^5 + X^3 + 1$

Subcarrier modulator

Modulates square subcarrier with data

Encoders

Following the discussion in section , the types included in *FSG* are ..

[1] This document does not contain any ITAR sensitive information

- Convolutional DSN
- Convolutional CCSDS
- Convolutional CASSINI
- Turbo Code (all rates and sizes)
- Reed Solomon CCSDS RS (255,223) dual base

Direct Digital Synthesizer(DDS)

It is used to synthesize subcarrier and symbol rate. It should be referenced to site 10 MHZ for best results, but it can also run out of the board crystal oscillator. The frequency is set by the N increment that is input at the phase accumulator which will wrap at a desired frequency governed by the formula

$$N = \frac{f \cdot 2^{32}}{200 \cdot 10^6} = f \cdot k \qquad (1)$$

where N =increment
f= desired frequency
$k = 21.47483648$

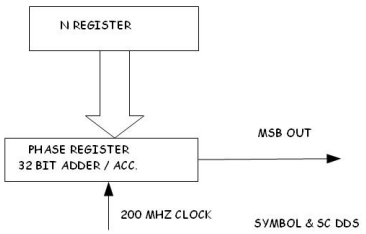
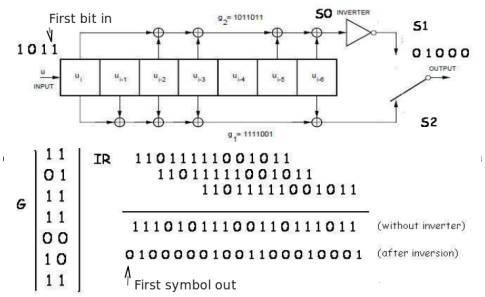


Figure 2: *DDS.*

In order to maintain the maximum 32 bit resolution , the calculation is carried out by one of the monitor and control processors using 64 bit precision by the routine "mul32x32" . The frequency resolution for this 32 bit DDS is 0.046566128730 Hz which even at Voyager spacecraft low rates represents less than .02 % symbol rate error



ht

Figure 3: *DSN schematic.*

DSN convolutional encoder

DSN encoder is a "text book" convolutional encoder. The constraint length is 7 and the rate is 2. The generator matrix and impulse response are dictated by the specification. (fig 3). shows how the impulse response is read from the rows of generator matrix which in turn,is determined by tapping. Since convolution is a linear operation, superimposition applies. Thus, response from a given input sequence, can be obtained from convolution, or weighted addition of impulse responses, as shown in fig 3. The manual procedure is carried out in two phases, first, omitting the inverter, and secondly, inverting every other symbol.Once this methodology is understood, VHDL coding poses no major problem.The core of the code that is implemented in the FSG is displayed on fig 9 on appendix A

Since the coder outputs two symbols per input bit,two different processes each running at a different clock, must be implemented. The process dsn_coder generates the output symbols.Symbol rate, must double the bit rate. S1 is generated in the first symbol clock, and S2 in the second .The process step_coder, advances the input sequence one bit.The behavior of the code can be verified by simulation tool to be as predicted as shown on fig 10. DSN is a paradigm of the convolutional coders of any rate and constraint. CCSDS convolutional is synthesized the same way. Cassini convolu-

tional is more complex, but basically,the same principles apply.

TURBO encoder

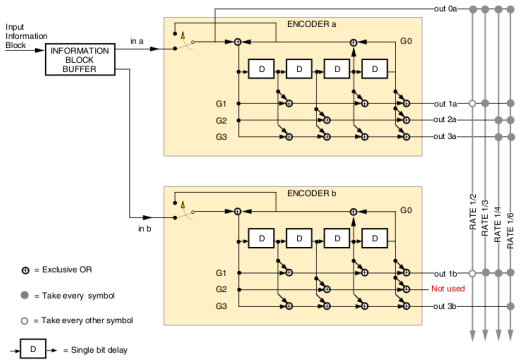


Figure 4: *Turbo encoder diagram.*

The turbo encoder comprises two parallel recursive convolutional encoders of constraint length 4. This coder differs from DSN, CCSDS etc in the fact that they have backward, recursive in addition to forward connection vectors, and therefore, they do not have a finite response. Once a single bit is entered, the response never extinguishes,even though it repeats after 15 states. Both coders have the same wiring and are fed simultaneously,but one of them receives an interleaved (at bit level) version of the data input, while the other coder receives the data unaltered. The different code rates are obtained via"puncturing"

Interleaver

The information block buffer on fig 4 ,as designed by its author Claude Berrou,provides bit interleaving achieved sequentially applying the formulas on table 1. Notice for instance the modulo k_2 operation (see table 2). Vendor VHDL does not provide modulo functions other than mod 2. Implementing via iteration would have added far too many cycles to the operation. The approach was to take advantage of the18 by 18 bit hardware multipliers, and transform division to multiplication of the inverse with appropriate scaling.
Example Compute 456 mod 223 = 10
using 1/223 scaled up by 2^{24} (75234.15247)

1 multiply $456 \times 75234 = 34306704$
2 scaled down by 2^{24} 2.0448838905
3 multiply fraction by 223 $.0448838905 \times 233 = 9.999075889$
4 round off 9.999075889 to 10
The botleneck of interleaved address generation i s alleviated by this procedure that uses 2 clocks to achieve modulo 223 via hardware.

Table 1: Formulas

$m = (s-1)\ mod\ 2$
$i = \lfloor \frac{s-1}{2k_2} \rfloor$
$j = \lfloor \frac{s-1}{2} \rfloor - ik_2$
$t = (19i+1)\ mod\ \frac{k_1}{2}$
$q = t\ mod\ 8 + 1$
$c = (p_q j + 21m)\ mod\ k_2$
$\Pi(s) = 2(t + c\frac{k_1}{2} + 1) - m$
legend $\lfloor\ \rfloor = nearest\ integer$

Table 2: Constants

information block	k_1	k_2
1784	8	223
3568	8	223×2
7136	8	223×4
8920	8	223×5

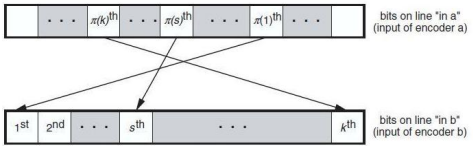


Figure 5: *Turbo interleaving scheme*

Slightly different versions of the code had to be implemented for each frame length in order

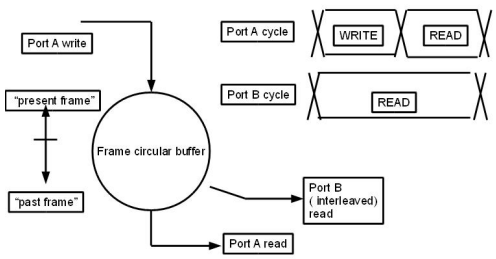


Figure 6: *Dual port RAM.*

to cope with subtle rounding errors.The frame must be available before feeding both encoders with the direct data, and the bit interleaved version of it.The frame must be stored in advance to carry out bit interleaving. This is implemented via a two port ram (fig 6) able to hold two frames, in which one port is write then read (normal path) while the other is always read (interleaved). During the first half part of the memory cycle, write data is written to port A on the "present" frame address, then in the second part, both ports are set to read "past" frame addresses, port A has the normal path data while port B has the interleaved version.

Generating a reference table with a C program proved most useful. It was used intensively while debugging the FPGA interleaver VHDL code.

Reed Solomon encoder

A Reed Solomon codeword C for data set I is defined to be the same data set I followed by the remainder R resulting from the polynomial division of the scaled up data $I \cdot X^n$ divided by a generator polynomial G of degree n. In equations:

$$\frac{I(x) \cdot x^n}{G(x)} = Q(x) + R(x) \tag{2}$$

$$C(x) = I(x) + R(x) \tag{3}$$

being $G(x)$ the "generator polynomial" of degree n The key of the encoder is to perform the division $I(x) \cdot x^n / G(x)$ and obtain the the remainder $R(x)$. Long divisions are traditionally implemented via shift registers as shown in fig 7, where the remainder of the division is held in the delay elements after all data have entered

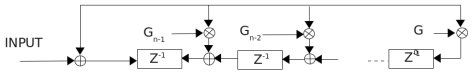


Figure 7: *RSE.* shift register implementation of long division

Here, the division is "pipelined" but very similar to the "hand" long division. The operation is carried out over a Galois Field (in CCSDS Reed Solomon, an extension 8 $GF(2^8)$). A straightforward implementation of the $I \cdot G$ products would require a table length of $256 \cdot 256 \cdot 8$. But considering that the polynomial G is fixed, then only $31 \cdot 256 \cdot 8$ bits should be required.

When this code was formulated, this was complex to implement. In 1982, E.R. Berlekamp working under a JPL contract devised a system in which the $I \cdot G$ products could be implemented serially on a "bit by bit" basis. Multiplying by G can be wired, and the encoding operation performed serially. Ref. [1] explains it in depth, and of course ref. [2], is the original work. Appendix A "bit serial product" provides the very basics.

E.R Berlekamp combined computer simulations and hand computations to find the best combination that would provide minimal gate count. He was able to implement all products with 24 two input exclusive or's with three levels of gate sharing. The parameters he choose are the $RS(255, 223)$ CCSDS encoder standard. The generator for the $GF(2^8)$ is:

$$x^8 + x^7 + x^2 + x + 1 \tag{4}$$

The element $\beta = \alpha^{117}$ to form the dual basis. The generator polynomial given by:

$$G(x) = \prod_{j=112}^{143} (x - \alpha^{11j}) = \sum_{i=0}^{32} G_i x^i \tag{5}$$

Monitor and control

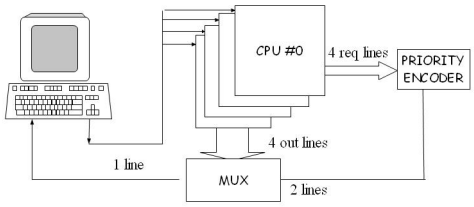


Figure 8: *M & C.* Monitor and control

CMD	DESCRIPt.	CPU
SCF	set subcarrier frequency	1
SYNC	inserted or overlay sync word in frame	1
TPA TPB	tests points signal selection	1
VFY	(dump) configuration registers	1
CNF	set configuration registers	1
TBR	set telemetry bit rate	1
TTG	time tag enable disable	1
TIME	set /read time local to *F*SG	1
IN	input port	1
RST	reset CPU	1
TURBO	frame=ffff rate=r pattern=ppp	1
RAN	randomize enable / disable	1
PTR	data pattern (pn|alt|one|zero|ram)	2
STAT	dump hardware status and config.	2
FLG	set frame length	2
FM	fill frame buffer	2
MCV	set convolutional connection vector	2
PCM	set PCM format	2
HELP	dump cmd list	2
DF	dump frame while being output	3
DM	dump frame buffer	3
FIX	select hard coded data for frame	3

Table 3: Command implementation

Four eight bit CPUs cooperate to provide human interface, parse commands and perform the necessary calculations needed to drive the generator hardware. Monitor and control is implemented in assembly language on an eight bit CPU design provided free by the vendor.Program code is limited to 1k instructions which was insufficient to implement the basic set of directives.A single CPU ocupies 6 % of real state of the target FPGA, and finally four instances of the design were used. Parts of the firmware were replicated (directive parser, arithmetic routines) but the design gained simplicity (other approaches like modifying the vendor assembly language compiler were discarded). Monitor and control appears like a single CPU to the user. All directives are routed to all CPUs but the command is targeted only to one, which decodes and configures the generator hardware according to the instruction. The acting CPU answers as required. The output RS-232 is accessed by all CPUs through a priority encoding scheme to avoid contention. Master CPU 0 has the highest priority and can reset/restart all other via a reset command. At the time of this writing, three CPUs are full, while the fourth still can be used to implement additional directives without having to recompile the whole design. The advantage of this scheme is that only the affected CPU has to be compiled, and the "bit file" can be "patched", an extremely fast procedure (compared to other compiling processes, of course). Directives listed below, are all the implemented directives. Some of them have no operational value, they were meant to assist in debugging.CPU 4 is not listed since it mostly conmtains monitor routines and the lcd display sign on.

Conclusions

Even lower end FPGAs are suitable for logic implementations of moderate complexity, such is the case of coders used in the DSN. Their best implementation is in hardware. Monitor and control was also implemented in the same FPGA to favor autonomy of the unit. The multiprocessor scheme was dictated by the necessity of keeping the code below 1K instructions (CPU design constraint). Current FPGA spartan 6 (vs spartan 3e) would have obviated this requirement (4K instructions). Data generators are tied to particular coders.This not desirable ."General purpose" are more advisable since they can be instantiated as needed. Simulation plays a very important role in system building. Getting to know well the simulation tool can spare much time and avoid many design bug headaches.

Attaching memory modules for dumping purposes reveals very useful, in particular, debugging long binary data frames. Finally, partitioning designs, improves system debugging radically.

References

[1] T.K Trung L.J. Deutsch I.S. Reed I.S. Hsu K. Wang & C.S. Yeh TDA: "The VLSI design of a Reed-solomon Encoder using Berlekamp's bit-serial multiplier algorithm". Progress Report 42-70 May and June 1982.

[2] Elwyn R. Berlekamp, Bit serial Reed-Solomon encoders Transactions on Information Theory, Vol. IT-28, No. 6, November 1982 869.

[3] Consultative Committee for Space Data Systems RECOMMENDATION FOR SPACE DATA SYSTEM STANDARDS TELEMETRY CHANNEL CODING CCSDS 101.0-B-6 BLUE BOOK October 2002

[4] KCPSM3_Manual KCPSM3 8 bit micro controller for spartan 3 Virtex II & Virtex II pro Ken Chapman Xilinx Ltd October 2003

[5] Frequency Generator for Spartan-3E Starter Kit Ken Chapman Xilinx Ltd 18th July 2006

Appendix A

```
dsn_coder: process(symclk)
begin
   -- out coded bits first
   if symclk'event and symclk='1' then
     if  bitclk ='1' then
            s1 <= not     ( coder(6) xor coder(4) xor coder(3) xor coder(1) xor coder(0) );
            coded_dsn <=not     ( coder(6) xor coder(4) xor coder(3) xor coder(1) xor coder(0) );
            coded_dsn_no_inv<= ( coder(6) xor coder(4) xor coder(3) xor coder(1) xor coder(0) );
     end if;
     if  bitclk ='0' then s2 <= ( coder(6) xor coder(5) xor coder(4) xor coder(3) xor coder(0));
            coded_dsn <=( coder(6) xor coder(5) xor coder(4) xor coder(3) xor coder(0));
            coded_dsn_no_inv<= ( coder(6) xor coder(5) xor coder(4) xor coder(3) xor coder(0));
     end if;

   end if;
end process dsn_coder;

step_coder:process (bitclk)
begin
   -- step coder on symbol clock leading edge
   if bitclk'event   and bitclk ='1' then    coder <=  uncoded & coder(6 downto 1)  ; end if;
end process step_coder;
```

Figure 9: *VHDL code.*

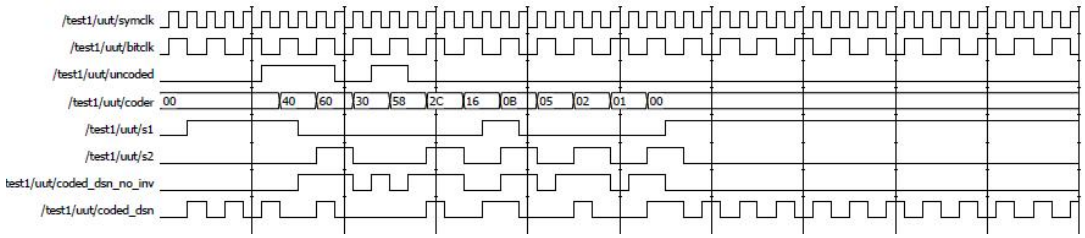


Figure 10: *CCSDS simulation.*

Appendix B

Bit serial products

Galois fields are built by listing the successive powers of a "primitive element" α that is a "root"[2] of a prime polynomial such as

$$x^3 + x + 1 = 0 \tag{6}$$

The coefficients of equation (6) belong to the "ground field" {0,1} in which addition is defined to be the same as subtraction, therefore the following can be written

$$x^3 = -x - 1 \text{ or}$$
$$x^3 = x + 1$$

If α is a "root" of eq $x^3 + x + 1 = 0$, this is equivalent to write

$$\alpha^3 = \alpha + 1 \tag{7}$$

The seven non-null elements of the finite set obtained taking into account equation (7) from the successive powers of α

Note that the set is finite as stated since "wraps around" exponent 7, $\alpha^7 = \alpha^0$.Elements can be viewed as vectors expressed by a linear combination of $\{\alpha^2, \alpha^1, \alpha^0\}$ as tabulated in Table 5. The basis is

$$A = \{\alpha^2, \alpha^1, \alpha^0\} \tag{8}$$

and any element can be expressed as

$$a = a_2\alpha^2 + a_1\alpha^1 + a_0\alpha^0 \tag{9}$$

The product of two elements can be defined simply as the sum of the exponents(taking base α logarithms, adding and then taking antilogarithms)

$$\alpha^3 \cdot \alpha^4 = \alpha^7 = \alpha^0$$

Should we associate elements to the binary number formed by their coefficients, we would have

$$011 \times 110 = 001$$

[2] Evariste Galois called these nonexistent roots, "etranges racines", not more strange that complex numbers, he argued,which do not exist, but however, we operate with them naturally

field elements
$\alpha^0 = 1$
α
α^2
$\alpha^3 = \alpha + 1$
$\alpha^4 = \alpha^2 + \alpha$
$\alpha^5 = \alpha^2 + \alpha + 1$
$\alpha^6 = \alpha^2 + 1$

Table 4: α powers

α exp	α^2	α^1	α^0
0	0	0	1
1	0	1	0
2	1	0	0
3	0	1	1
4	1	1	0
5	1	1	1
6	1	0	1

Table 5: coeffs

In binary numbering, this result is non sense. This is because these are not octal numbers, but a binary representation of (polynomial) element with a basis

$$A = \{\alpha^2, \alpha^1, \alpha^0\} \tag{10}$$

Implementing the product of two elements in logic circuitry takes either a table lookup up or reducing all possible products to combinatorial logic. The following development will show how to reduce the implementation to a mere shift register and some logic gates. Let us consider representing two numbers b and c in a different basis

$$B = \{\beta^2, \beta^1, \beta^0\} \tag{11}$$

related to A by

$$\beta^2 = \alpha \text{ , } \beta = \alpha^2 \text{ } ([3]) \text{ and } \beta^0 = \alpha^0$$

Let us develop the product of two number a expressed in base A and b expressed in base B and assume a result c also in base B, namely

$$a = a_2\alpha^2 + a_1\alpha + a_0$$

[3] Note that β is set to α^2

Table 6: Basis equivalence

exp	normal				exp	dual			
	α^2	α	α^0	octal #		α^0	α^2	α	octal #
α^0	0	0	1	1	β^0	1	0	0	4
α^1	0	1	0	2	β^2	0	0	1	1
α^2	1	0	0	4	β^1	0	1	0	2
α^3	0	1	1	3	β^3	1	0	1	5
α^4	1	1	0	6	β^4	0	1	1	3
α^5	1	1	1	7	β^5	1	1	1	7
α^6	1	0	1	5	β^6	1	1	0	6

$$b = b_2\alpha + b_1\alpha^2 + b_0$$
$$c = c_2\alpha + c_1\alpha^2 + c_0$$
$$c = a \times b$$

After evaluating the product, the equation under matrix representation would look like this:

$$\begin{vmatrix} c_0 & c_1 & c_2 \end{vmatrix} \cdot \begin{vmatrix} 1 \\ \alpha^2 \\ \alpha \end{vmatrix} = \begin{vmatrix} a_2 & a_1 & a_0 \end{vmatrix} \cdot \begin{vmatrix} b_2 & b_0+b_1 & b_2+b_1 \\ b_1 & b_2 & b_0+b_1 \\ b_0 & b_1 & b_2 \end{vmatrix} \cdot \begin{vmatrix} 1 \\ \alpha^2 \\ \alpha \end{vmatrix}$$

$$C = A \times B$$

Matrix B leads itself to hardware implementation. It can be realized with recirculating register and an adder(a shift register with feedback and and one exclusive or gate as shown in fig fig 11) The first column is equates to shift register initial load. The second and third are obtained adding the two bottom bits and recirculating the shift register. Fig fig 11 shows the scheme. The successive bits of number c are obtained by multiplying the row vector A by the columns of B. Multiplication by first column produces the least significant bit, the next column the middle bit and the last column produces the most significant digit.

Fig fig 12 provides an overall view of the serial product. First registers are preloaded with the number a and b. After three clocks, the c register contains the result product .

.

But circuitry simplifies even further when one of the multipliers is a constant as shown in fig 13 (α^3). This is the case when dividing

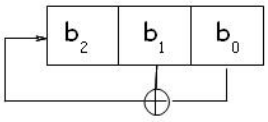


Figure 11: *Feedback register.*

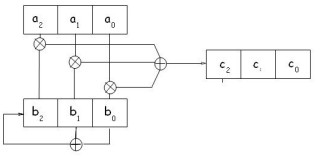


Figure 12: *Bit serial product.*

by a fixed polynomial, then, all products have one of the operands that is a constant.

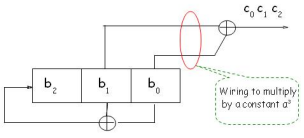


Figure 13: *Constant multiplication.*

A toy RS encoder

A RS codeword results from appending the remainder of a polynomial division to the original data. In equations

$$C(x) = I(x) \cdot x^n + R(x) \tag{12}$$
$$R(x) = \frac{I(x)x^n}{G(x)} - Q(x) \tag{13}$$

where

$$I(x) = \text{input sequence}$$
$$G(x) = \text{generating polynomial}$$
$$C(x) = \text{code word}$$

Our toy RS encoder will be of type RS(7,5,3) which encoding terminology means 7 symbols in codeword,5 symbols data word,2 check symbols and defined over a $GF(2^3)$ Galois field.

Z	Y	I	S	R	x6	x3
6	6	6	0	0	0	0
5	5	0	5	2	5	2
0	0	6	6	6	4	6
6	6	0	6	0	0	0
3	3	6	5	2	5	2
3	3	0	3	4	1	4

Table 7: *Toy RS encoder states*

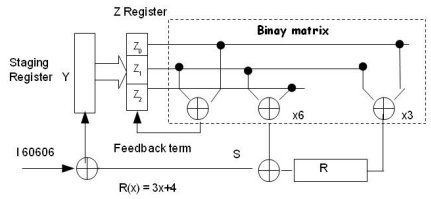


Table 8: *Toy RS encoder RS(7,5) diagram*

This code is able to correct on error but for the sake of exemplification will be enough. The input worked out in this example would be

$$5x^4 + 5x^2 + 5 \text{ (base A)}$$
$$6x^4 + 6x^2 + 6 \text{ (base B)}$$

for the generator polynomial we choose

$$(x-1)(x-2) = x^2 - 2x - x - 1 \cdot 2) = x^2 + 6x + 3$$

the remainder from this division is

$$6x + 1 \text{ (base A)}$$
$$3x + 4 \text{ (base B)}.$$

Tables 7 and 8 illustrate how the embedded multiplications in the Euclidean algorithm are transformed by the serial product procedure.Table 7 show the state od the registers every three clocks. Table 8 shows the block diagram for the dual base encoder. All combinatorial logic is housed if the shaded block labelled "binary matrix". After all data has entered, register Y and R will hold the remainder of the division e.g.

$$3x + 4 \qquad (14)$$

Remarks

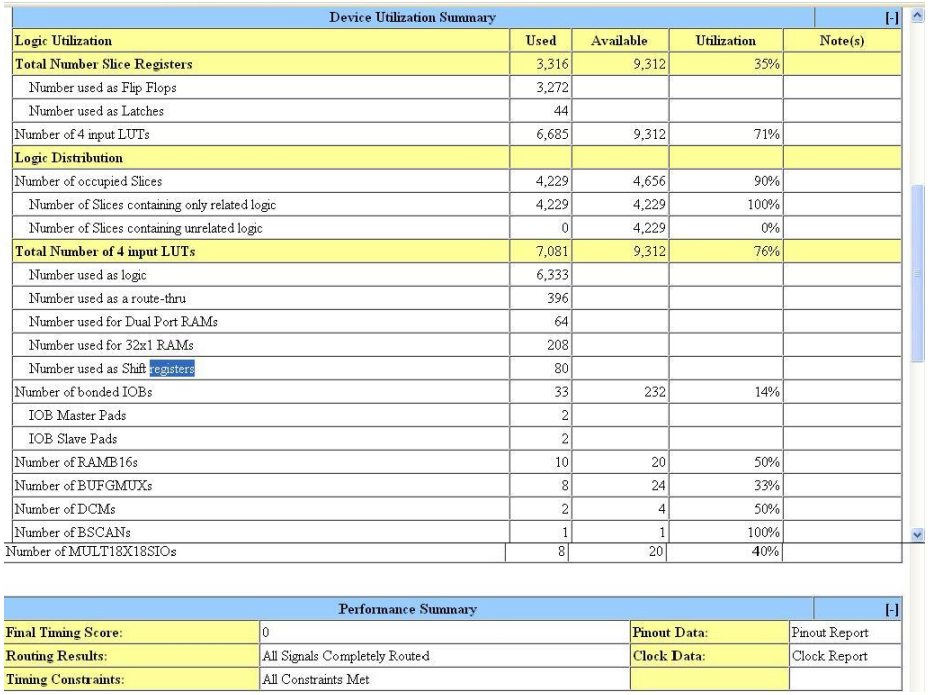
The Galois algebra and arithmetics can be associated in this case with octal number in a sort of a "short hand notation for polynomials (e.g 5 for x^2+1). Tables for addition and multiplication can be build that provide the basis for further operation like polynomial divisions of higher order field. These operations can be implemented with combinatory logic or simplified a great deal with special algorithms such as Berlekamp bit serial products.The CCSDS Reed Solomon encoder operates with GF(8), what makes this bit serial product procedure very advantageous

Appendix C

Implementation summary

Table 9: Code distribution summary

FSG8A	asm code	vhdl code
code	7807	12244
comments	584	1273
percentage	7.48%	14.07%



Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	3,316	9,312	35%	
Number used as Flip Flops	3,272			
Number used as Latches	44			
Number of 4 input LUTs	6,685	9,312	71%	
Logic Distribution				
Number of occupied Slices	4,229	4,656	90%	
Number of Slices containing only related logic	4,229	4,229	100%	
Number of Slices containing unrelated logic	0	4,229	0%	
Total Number of 4 input LUTs	7,081	9,312	76%	
Number used as logic	6,333			
Number used as a route-thru	396			
Number used for Dual Port RAMs	64			
Number used for 32x1 RAMs	208			
Number used as Shift registers	80			
Number of bonded IOBs	33	232	14%	
IOB Master Pads	2			
IOB Slave Pads	2			
Number of RAMB16s	10	20	50%	
Number of BUFGMUXs	8	24	33%	
Number of DCMs	2	4	50%	
Number of BSCANs	1	1	100%	
Number of MULT18X18SIOs	8	20	40%	

Performance Summary			[-]
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Figure 14: ISE summary