Kodowanie Reed–Solomona i VHDL, część 1
Wprowadzenie

W EP 4/2005 opisano sposób zabezpieczenia w transmisji danych za pomocą dodatkowych słów kodowych CRC. Takie dodatkowe słowa dodane w nadajniku pozwalały odbiornikowi zweryfikować poprawność odebranych danych. Niestety, dodatkowe informacje zawarte w CRC nie pozwalają na korekcję błędów w przypadku ich wystąpienia. Tą niedogodność eliminuje, stosowane obecnie prawie zawsze w urządzeniach profesjonalnych w transmisji cyfrowej kodowanie Reed–Solomona (w skrócie RS).* Kodowanie Reed–Solomona zostało opisane w 1960 roku w piśmie „Journal of the Society for Industrial and Applied Mathematics" przez Irvinga Reeda i Gusa Solomona.

Przypuśćmy, że transmitujemy na odległość paczki danych o wielkości 188 bajtów. Na każde dodane w nadajniku do takiej paczki 2 bajty kodu RS, odbiornik jest w stanie poprawić jeden uszkodzony bajt w przesyłanej paczce. Jeśli dodamy 16 bajtów w kodzie RS to będziemy mogli poprawić dowolne, uszkodzone 8 bajtów, spośród wszystkich 204 przesłanych. Właśnie w taki sposób zabezpieczane są dane przesyłane w transmisji telewizji cyfrowej w Europie w standardzie DVB.

Pierwszy raz w praktyce kodowania RS użyto podczas misji Voyager II. Obecnie z użyciem kodowania RS są zabezpieczane dane w telewizji cyfrowej, na nośnikach CD, DVD i dyskach twardych a także przy cyfrowej obróbce obrazów.

Niestety do kodowania RS są wymagane dosyć skomplikowane obliczenia matematyczne. W niniejszym artykule chciałbym przedstawić, w jak najprostszy sposób, możliwości opisu enkodera RS w języku VHDL, wraz ze sprawdzaniem poprawności odebranych danych. Implementacja w języku VHDL pełnego dekodera RS, który jest w stanie poprawić błędy w transmisji jest bardzo skomplikowana i wykracza poza ramy tego artykułu.

Wszystkie przedstawione implementacje kodu VHDL zostały sprawdzone i przesymulowane z użyciem oprogramowania dostępnego na stronie WWW, firmy Xilinx: *www.xilinx.com*.
1. Do symulacji wykorzystano oprogramowanie Modelsim XE III Starter 6.0a.
2. Do implementacji wykorzystano oprogramowanie XILINX ISE 7.1i.

Arytmetyka Galois – pola

Aby poprawnie zrozumieć jak działa kodowanie i dekodowanie kodów typu RS jest niezbędne zrozumienie matematyki pól skończonych, znanych pod nazwą pól Galois. W artykule nie będę omawiał wszystkich zagadnień matematycznych związanych z polami Galois. Przedstawię jedynie niezbędne elementy, których znajomość pozwoli samodzielnie zaimplementować w kodzie VHDL koder i dekoder kodu RS.

Do konstrukcji kodów Reed–Solomona używa się symboli z rozszerzonych pól Galois GF(2^m), gdzie:

GF(2^m) oznacza zbiór elementów $\{0, 1, \alpha, \alpha^2, ..., \alpha^{(2^m-2)}\}$

Jak widać ilość elementów w zbiorze wynosi 2^m. W podanym zbiorze obowiązuje arytmetyka binarna, tzn.:
- $1=-1$;
- $\alpha+\alpha=0$
- podstawową operację sumowania wykonujemy za pomocą funkcji XOR,
- podstawową operację mnożenia wykonujemy za pomocą funkcji AND

Każdy kolejny element zbioru powstaje przez pomnożenie ostatniego elementu przez α, przy czym zbiór

Tab. 1. Wybrane wielomiany pierwotne

m	wielomian
3	$1 + x + x^3$
4	$1 + x + x^4$
5	$1 + x^2 + x^5$
6	$1 + x + x^6$
7	$1 + x^3 + x^7$
8	$1 + x^2 + x^3 + x^4 + x^8$
9	$1 + x^4 + x^9$
10	$1 + x^3 + x^{10}$
11	$1 + x^2 + x^{11}$
12	$1 + x + x^4 + x^6 + x^{12}$
13	$1 + x + x^3 + x^4 + x^{13}$
14	$1 + x + x^6 + x^{10} + x^{14}$
15	$1 + x + x^{15}$
16	$1 + x + x^3 + x^{12} + x^{16}$
21	$1 + x^2 + x^{21}$
23	$1 + x^5 + x^{23}$

*)*Istnieją również inne sposoby zabezpieczania danych w transmisji cyfrowej, np. kodowanie Trellis'a.*

Tab. 2. Elementy rozszerzonych pól Galois GF(2³)=1+x+x³

	0	1	2	stdlogic_vector(2 downto 0)
0	0	0	0	”000”
α^0	1	0	0	”001”
α^1	0	1	0	”010”
α^2	0	0	1	”100”
α^3	1	1	0	”011”
α^4	0	1	1	”110”
α^5	1	1	1	”111”
α^6	1	0	1	”101”

tych elementów jest ograniczony, dzięki wzorowi redukującemu:

$$\alpha^{(2^m-1)}=1=\alpha^0$$
$$\alpha^{(2^m+n)}=\alpha^{(2^m-1)}\alpha^{(n+1)}=\alpha^{(n+1)}$$

Dzięki wzorom upraszczającym, otrzymujemy jedną z podstawowych własności pól Galois: wyniki operacji matematycznych (+, –, ·, /) przeprowadzanych na elementach zbioru należą do tego zbioru.

Tworzenie rozszerzonych pól Galois

Aby utworzyć pola Galois potrzebny jest wielomian pierwotny. Przy opisie kodu RS, zawsze jest podawany wielomian pierwotny, tworzący dany zbiór pól Galois. W **tab. 1** przedstawiono wybrane wielomiany pierwotne.

Dla przykładu weźmy najprostszy wielomian pierwotny $f(x)=1+x+x^3$, który definiuje rozszerzone pola Galois GF(2^3). Zbiór ten składa się z 8 elementów:

- Zerowy element to 0.
- Pierwszy element to $\alpha^0=1$
- Drugi element to α^1, podobnie trzeci element to α^2.
- Czwarty element wyprowadzamy, korzystając z wielomianu pierwotnego:

$$F(\alpha)=0$$
$$1+\alpha+\alpha^3=0$$
$$\alpha^3=-1-\alpha$$

Ponieważ operujemy na arytmetyce binarnej to $-1=1$ oraz $\alpha+\alpha=0$, stąd element α^3, może być reprezentowany jako:

$$\alpha^3=1+\alpha$$
$$\alpha^4=\alpha*\alpha^3=\alpha*(1+\alpha)=\alpha+\alpha^2$$
$$\alpha^5=\alpha*\alpha^4=\alpha*(\alpha+\alpha^2)=\alpha^2+\alpha^3=\alpha^2+1+\alpha$$
$$\alpha^6=\alpha*\alpha^5=\alpha*(1+\alpha+\alpha^2)=\alpha+\alpha^2+\alpha^3=\alpha+\alpha^2+1+\alpha=1+\alpha^2$$
$$\alpha^7=\alpha*\alpha^6=\alpha*(1+\alpha^2)=\alpha+\alpha^3=\alpha+1+\alpha=1$$

– element pierwszy

Jednym ze sposobów przedstawienia zbioru rozszerzonego pól Galois jest tablica dwuelementowa, gdzie jeden wymiar oznacza kolejny element, a drugi wymiar potęgę elementu α. W samej tablicy jedynka oznacza istnienie danej potęgi α, a 0 jej brak. Przykład przedstawiono w **tab. 2**. Dodatkowo w tabeli umieszczono przykładową reprezentację danego elementu w języku VHDL jako wektor *m*–elementowy.

Tab. 3. Wyniki sumowania rozszerzonych pól Galois GF(2³)=1+x+x³

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

Tab. 4. Wyniki mnożenia rozszerzonych pól Galois GF(2³)=1+x+x³

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

Realizacja dodawania rozszerzonych pól Galois w języku VHDL

W języku VHDL dowolny element zbioru GF(2^m) można reprezentować przez wektor *m*–elementowy, np.:

```
signal alfa : std_logic_
vector(2 downto 0);
```

Operację dodawania rozszerzonych pól Galois realizuje się przez funkcję XOR na poszczególnych pozycjach dwóch elementów. Operacja ta jest oznaczana symbolem ⊕. Na **list. 1** pokazano przykładową implementację sumatora elementów pól Galois. W tej implementacji dokonuje się operacji XOR dla każdego bitu sygnału wyjściowego osobno:

List. 1. Implementacja sumatora oraz układu mnożącego dla symboli kodu RS GF(2³)=1+x+x³

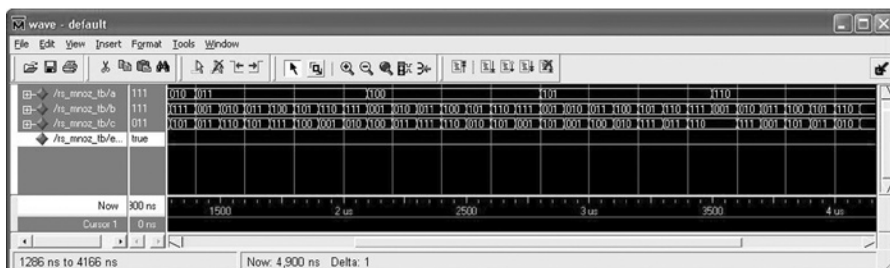
```
--sumator pól Galois,

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity rs_sum is
  port(
   A : in STD_LOGIC_VECTOR (2 downto 0);
   B : in STD_LOGIC_VECTOR (2 downto 0);
   C : out STD_LOGIC_VECTOR (2 downto 0));
end rs_sum;

architecture rs_sum_arch of rs_sum is
begin
  C(0) <= A(0) xor B(0);
  C(1) <= A(1) xor B(1);
  C(2) <= A(2) xor B(2);
end architecture rs_sum_arch;


-- mnozenie pól Galois, wielomian
pierwotny: 1 + x + x3

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity rs_mnoz is
  port(
    A : in STD_LOGIC_VECTOR (2 downto
0);
    B : in STD_LOGIC_VECTOR (2 downto
0);
    C : out STD_LOGIC_VECTOR (2 downto
0));
end rs_mnoz;

architecture rs_mnoz_arch of rs_mnoz
is
begin
  C(0) <= (A(0) and B(0)) xor (A(1)
and B(2)) xor (A(2) and B(1));
  C(1) <= (A(0) and B(1)) xor (A(1)
and B(0)) xor (A(1) and B(2)) xor
         (A(2) and B(1)) xor (A(2)
and B(2));
  C(2) <= (A(0) and B(2)) xor (A(1)
and B(1)) xor (A(2) and B(0)) xor
         (A(2) and B(2));
end architecture rs_mnoz_arch;
```

```
C(0)  <=  A(0)  xor  B(0);
C(1)  <=  A(1)  xor  B(1);
C(2)  <=  A(2)  xor  B(2);
```

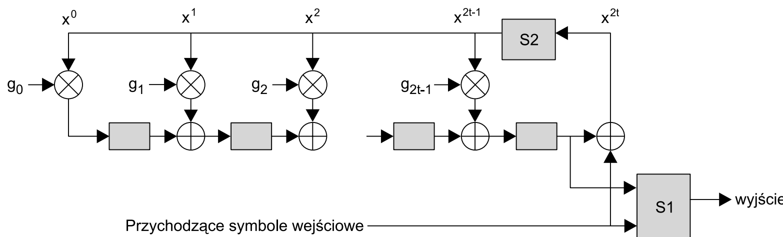
Po symulacji implementacji sumatora z **list. 1**, można utworzyć tabelkę wyników sumowania dla wszystkich wartości sygnałów wejściowych a i b (**tab. 3**).

Mnożenie i pola Galois

Zasady mnożenia pól Galois, podlegają podstawowym zasadom mnożenia liczb podniesionych do potęgi w matematyce, czyli na sumowaniu potęg elementów zbioru pól modulo (2^m–1). Mnożenie oznaczamy symbolem ⊗. Odpowiedni wzór jest taki:



Rys. 1. Wyniki symulacji układu mnożącego dla symboli kodu RS GF(2³)=1+x+x³ w programie Modelsim XE III Starter 6.0a



Rys. 2. Schemat blokowy układu LFSR

$\alpha^x \otimes \alpha^y = \alpha^{(x+y)}$ modulo (2^m-1)

Korzystając z tej zasady można wyznaczyć tabelkę dla naszego przykładowego wielomianu prymitywnego $GF(2^3)=1+x+x^3$, jaką otrzymano dla wyników sumowania (**tab. 4**).

Aby wykonać mnożenie w układach cyfrowych łatwiej korzysta się z trochę innego podejścia. Wynika to z tego, że o ile łatwo byłoby zaimplementować mnożenie według powyższej tabelki dla tylko 7 symbolów $GF(2^3)$, to dla 255 symbolów $GF(2^8)$ byłoby to już o wiele bardziej skomplikowane. Jak już wcześniej mówiłem w języku VHDL poszczególne elementy pól Galois można zaprezentować jako wektor m–elementowy, w postaci wielomianu:

$$a_1*\alpha^0+a_2*\alpha^1+...a_m*\alpha^{m-1}$$

Jeśli chcemy pomnożyć dwa takie elementy korzystamy z zasad mnożenia wielomianów. Najlepiej będzie zobrazować to przykładem mnożenia rozszerzonych pól Galois, z wielomianu prymitywnego $GF(2^3)=1+x+x^3$:

$$(a_1\alpha^0+a_2\alpha^1+a_3\alpha^2)\otimes(b_1\alpha^0+b_2\alpha^1+b_3\alpha^2)=$$
$$a_1b_1\alpha^0+a_1b_2\alpha^1+a_1b_3\alpha^2+b_1a_2\alpha^1+a_2b_2\alpha^2+a_2b_3\alpha^3+b_1a_3\alpha^2+a_3b_2\alpha^3+a_3b_3\alpha^4=$$
$$a_1b_1\alpha^0+a_1b_2\alpha^1+a_1b_3\alpha^2+b_1a_2\alpha^1+a_2b_2\alpha^2+a_2b_3\alpha^0+a_2b_3\alpha^1+b_1a_3\alpha^2+a_3b_2\alpha^0+a_3b_2\alpha^1+a_3b_3\alpha^1+a_3b_3\alpha^2=$$
$$(a_1b_1+a_2b_3+a_3b_2)\alpha^0+(a_1b_2+b_1a_2+a_2b_3+a_3b_2+a_3b_3)\alpha^1+(a_1b_3+a_2b_2+b_1a_3+a_3b_3)\alpha^2$$

Operacje mnożenia wykonuje się za pomocą funkcji AND i XOR. Na list. 1 przedstawiono implementację jednostki mnożącej i sumującej w języku VHDL dla podanego przykładu. Do weryfikacji poprawności działania układu mnożącego posłużono się wcześniej przygotowaną tabelką z wynikami mnożenia dla wielomianu pierwotnego $GF(2^3)=1+x+x^3$.

Irving S. Reed oraz Gustave Solomon

Enkoder

Kody Reed–Solomona oznaczamy symbolicznie jako parę liczb (n, k). Liczba *n* oznacza liczbę słów kodowych w danym pakiecie danych i nie może być ona większa od wartości 2^m-1. Liczba *m* oznacza w tym przypadku ilość bitów słowa kodowego. Liczba *k* oznacza liczbę słów, które nie są słowami dodatkowymi. Stąd, liczba *n–k* oznacza ilość dodatkowych słów kodowych. Dodatkowe słowa kodowe oznacza się również wartością *2t*, gdzie *t* oznacza ile błędnych słów kodowych w danym kodzie można poprawić.

Do generacji kodu RS wykorzystuje się tzw. wielomian generujący g(x) w postaci:

$$g(x)=g_0+g_1x+g_2x^2+...+g_{2t-1}x^{2t-1}+x^{2t}$$

Stopień tego wielomianu jest równy ilości dodatkowych słów kodowych.

Wielomian g(x) zapisuje się z reguły w trochę innej postaci:

$$g(x)=(x-\alpha)(x-\alpha^2)...(x-\alpha^{2t-1})(x-\alpha^{2t})$$

Wiedząc, jaką liczbę słów kodowych chcemy mieć możliwość poprawienia, możemy przygotować odpowiedni wielomian generujący. Skorzystajmy z naszego przykładu dla $GF(2^3)=1+x+x^3$. Wyliczymy wielomian generujący dla kodu RS (7, 3). Oznacza to, że chcemy transmitować 3 słowa i dodatkowo 4 słowa korekcji. W takiej transmisji na każde przesłane 7 słów jesteśmy w stanie poprawić 2 błędne (n=7, k=3, t=2).

Chcąc wygenerować kod RS w postaci (7, 3) potrzebujemy następującego wielomianu generującego:

$$g(x)=(x-\alpha)(x-\alpha^2)(x-\alpha^3)(x-\alpha^4)$$

Dokonując odpowiednich przekształceń oraz korzystając z przekształceń omówionych wcześniej otrzymamy:

$$g(x)=\alpha^3+\alpha^1x+\alpha^0x^2+\alpha^3x^3+x^4$$

Do zbudowania elementu generującego, wykorzystuje się, podobnie jak w przypadku generowania CRC, element techniki cyfrowej zwany LFSR (*Linear Feedback Shift Register*). Na **rys. 2** przedstawiono jego schemat.

Kodowanie RS z wykorzystaniem LFSR przebiega następująco:

1. Najpierw przełącznik S2 jest załączony a przełącznik S1 przekazuje symbole pochodzące z wejścia.
2. Po *k* taktach zegarowych przełącznik S2 jest zamykany (przekazuje od tej pory wartości 0) a przełącznik S1 przekazuje dane z rejestru.
3. Po kolejnych n–k taktach zegarowych kodowanie jednej paczki danych się kończy.
4. Przed kodowaniem kolejnej paczki danych wszystkie rejestry w układzie LFSR są zerowane.

Na **list. 2** przedstawiono implementację LFSR w języku VHDL dla przykładowego kodu RS (7, 3). Poniżej podaję szczegółową opis implementacji kodu.

W implementacji najpierw opisano multiplekser S2. Sygnał na wyjściu multipleksera nazwano Mux_G. Jego stan jest sterowany linią wejściową *RS_switch*. Jeśli *RS_switch* jest ustawiony na '0', na wyjście multipleksera podawany jest rezultat sumowania S3. W kodowaniu RS(3,7) wartość '0' powinna być ustawiona przez 3 pierwsze takty sygnału taktującego CLK, a później powinna być przestawiona na '0' na 4 następne takty. Wtedy na wyjście multipleksera podawane są zera:

```
Mux_G <= (others => ,0')
when (RS_switch = ,1') else
S3;
```

Wszystkie wartości sumatorów reprezentowane są przez sygnały od S0 do S3:

```
S0  <=  R0  xor  M1;
S1  <=  R1  xor  M2;
S2  <=  R2  xor  M3;
S3  <=  R3  xor  RS_in;
```

Sygnały od R1 do R3 są wyjściami przerzutników, zaś sygnały od M0 do M3 reprezentują wyniki mnożeń. Odpowiednio układy mnożące zostały zainstancjonowane jako komponenty:

```
RS_multi_0:  RS_mnoz  port
map(Mux_G,  G0,  M0);
RS_multi_1:  RS_mnoz  port
map(Mux_G,  G1,  M1);
RS_multi_2:  RS_mnoz  port
map(Mux_G,  G2,  M2);
```

Jednym z wejść wszystkich układów mnożących jest wyjście mul-

tipleksera S2 – Mux_G. Drugim czynnikiem jest stała G0 do G3. Ponieważ wartości stałych G0 i G3 są jednakowe to wynik mnożenia M0 i M3 zawsze będą jednakowe. Stąd nie trzeba było instancjonować czwartego komponentu mnożącego, zaś wartość sygnału M0 przypisano bezpośrednio do wartości sygnału M3.

Multiplekser S1 opisano następująco:

```
RS_out <= RS_in when (RS_switch = '0') else R3;
```

RS_out jest sygnałem wyjściowym układu enkodera. Jest to jednocześnie sygnał wyjściowy multipleksera S1. Przez pierwsze trzy takty zegarowe *RS_switch* jest ustawiony na '0' i na wyjście przekazywane są dane wejściowe *RS_in*. Potem przez cztery takty zegarowe na wyjście przekazywana jest zawartość rejestru R3.

Zachowanie rejestrów jest opisane w procesie:

```
process (CLK, RS_Reset)
begin
  if (RS_Reset = ,1') then
    R0 <= „000";
    R1 <= „000";
    R2 <= „000";
    R3 <= „000";
  elsif (CLK'event and CLK = ,1')
then
    if (EN1 = ,1') then
      R0 <= M0;
      R1 <= S0;
      R2 <= S1;
      R3 <= S2;
    end if;
  end if;
end process;
```

Do układu doprowadzony jest sygnał wejściowy *RS_Reset*, który trzeba uaktywnić wartością '1', aby ustawić na zero wszystkie rejestry układu enkodera, przed każdym kodowaniem trzech kolejnych transmitowanych słów. Rejestry przepisują wejście (odpowiednio są to sygnały: M0, S0, S1 i S2) na wyjścia (R0 do R3), przy narastającym zboczu sygnału zegarowego CLK. W całym enkoderze występują 4 rejestry 3–bitowe.

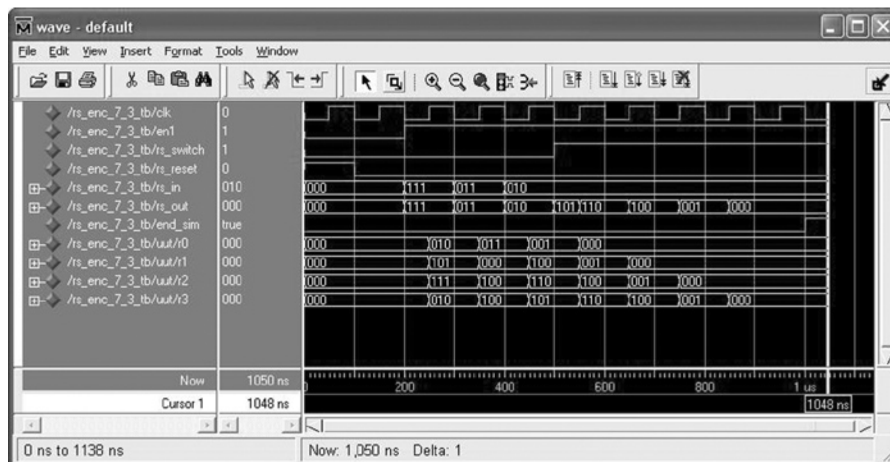
Przykładowy testbench (kod używany do testowania układów opisanych w językach opisu sprzętu), umieszczony na CD-EP12/2005B wyniki symulacji układu przedstawiono odpowiednio na **rys. 3**.

Marcin Nowakowski

Zalecana literatura

[1] L.H. Charles Lee: „Error–Control Block Codes for Communications Engineers", Artech House, inc, 2000, Boston – London

[2] Bernard Sklar "Introduction to Reed–Solomon Codes": article is provided courtesy of Prentice Hall, 12.04.2002



Rys. 3. Wyniki symulacji układu enkodera dla symboli kodu RS $GF(2^3)=1+x+x^3$ w programie Modelsim XE III Starter 6.0a

List. 2. Implementacja enkodera kodu RS (7, 3), dla $GF(2^3)=1+x+x^3$ w języku VHDL

```
---------------------------------------------------------------------------
---- RS ENCODER RS(7,3)
---------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity RS_enc_7_3 is
  port(
     CLK : in STD_LOGIC;
     EN1 : in STD_LOGIC;
     RS_switch : in STD_LOGIC;
     RS_Reset : in STD_LOGIC;
     RS_in : in STD_LOGIC_VECTOR (2 downto 0);
     RS_out : out STD_LOGIC_VECTOR (2 downto 0));
end RS_enc_7_3;
architecture RS_enc_7_3_arch of RS_enc_7_3 is
  component rs_mnoz
  port(
    A : in std_logic_vector(2 downto 0);
    B : in std_logic_vector(2 downto 0);
    C : out std_logic_vector(2 downto 0));
  end component;
-- g(x) = a3 + a1x + a0x2 + a3x3 + x4
  constant G0: std_logic_vector(2 downto 0) := "011";
  constant G1: std_logic_vector(2 downto 0) := "010";
  constant G2: std_logic_vector(2 downto 0) := "001";
  constant G3: std_logic_vector(2 downto 0) := "011";
-- wyniki sumowania
  signal S0 : std_logic_vector(2 downto 0);
  signal S1 : std_logic_vector(2 downto 0);
  signal S2 : std_logic_vector(2 downto 0);
  signal S3 : std_logic_vector(2 downto 0);
-- wyniki mnozenia
  signal M0 : std_logic_vector(2 downto 0);
  signal M1 : std_logic_vector(2 downto 0);
  signal M2 : std_logic_vector(2 downto 0);
  signal M3 : std_logic_vector(2 downto 0);
-- wartosci rejestrow
  signal R0 : std_logic_vector(2 downto 0);
  signal R1 : std_logic_vector(2 downto 0);
  signal R2 : std_logic_vector(2 downto 0);
  signal R3 : std_logic_vector(2 downto 0);
-- przelacznik S1, jako multiplexer
  signal MUX_G : STD_LOGIC_VECTOR(2 downto 0);
begin
  Mux_G <= (others => ,0') when (RS_switch = ,1') else S3;
  S0 <= R0 xor M1;
  S1 <= R1 xor M2;
  S2 <= R2 xor M3;
  S3 <= R3 xor RS_in;
  RS_multi_0: RS_mnoz port map(Mux_G, G0, M0);
  RS_multi_1: RS_mnoz port map(Mux_G, G1, M1);
  RS_multi_2: RS_mnoz port map(Mux_G, G2, M2);
  -- poniewaz G3 = G0 to zamiast M3 mozemy uzywac M0
  -- RS_multi_3: RS_mnoz port map(Mux_G, G3, M3);
  M3 <= M0;
  RS_out <= RS_in when (RS_switch = '0') else R3;
  process (CLK, RS_Reset)
  begin
    if (RS_Reset = '1') then
      R0 <= "000";
      R1 <= "000";
      R2 <= "000";
      R3 <= "000";
    elsif (CLK'event and CLK = '1') then
      if (EN1 = '1') then
        R0 <= M0;
        R1 <= S0;
        R2 <= S1;
        R3 <= S2;
      end if;
    end if;
  end process;

end RS_enc_7_3_arch;
```