

Systemy wbudowane

Lista 7

Magdalena Ośka
Nr indeksu: 221492

15 maja 2017

1 Kod Hamminga

Kod ten pozwala wykrywać i skorygować pojedyncze przekłamanie bitów w odebranym słowie binarnym. W zadaniu 2, należy napisać kod kodera i dekodera z 4 na 7 bitów.

1.1 Kodowanie

1. Mamy słowo składające się z 4 bitów: $b_3b_2b_1b_0$.
2. Umieszczamy je w słowie koda Hamminga:

$$\begin{array}{c|c|c|c|c|c|c} 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ b_3 & b_2 & b_1 & x_2 & b_0 & x_0 & x_0 \end{array}$$

Gdzie wszystkie pozycje będące potęgami 2 (czyli: 1, 2, 4) są bitami parzystości, a pozostałe to bity informacyjne.

3. Zasada kodowania bitów:

$$\begin{array}{c|c|c|c|c|c|c|c} \text{pozycja} & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ \text{bity} & b_3 & b_2 & b_1 & x_2 & b_0 & x_1 & x_0 \\ x_2 & x & x & x & x & & & \\ x_1 & x & x & & & x & x & \\ x_0 & x & & x & & x & & x \end{array}$$

Każdy bit ma unikalną kombinację sprawdzających go bitów parzystości.

4. Niech nasze słowo ma na przykład wartość 1011, wtedy:

$$\begin{array}{c|c|c|c|c|c|c} 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 0 & 1 & x_2 & 1 & x_1 & x_0 \end{array}$$

Obliczamy bity parzystości:

$$x_0 = 1 \oplus 1 \oplus 1 = 1$$

$$x_1 = 1 \oplus 0 \oplus 1 = 0$$

$$x_2 = 1 \oplus 0 \oplus 1 = 0$$

Po obliczeniu bitów parzystości podajemy je w odpowiednie miejsca w słowie:

$$\begin{array}{c|c|c|c|c|c|c} 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

1.2 Dekodowanie

1. Załóżmy, że nastąpiło przekłamanie b_2 : 1010101 – \rightarrow 1**1**10101”
2. Według tabeli z pkt 1.1.3 obliczamy bit parzystości:
 $c_0 = b_3 + b_1 + b_0 + x_0$, czyli $c_0 = 1 \oplus 1 \oplus 1 \oplus 1 = 0$
 $c_1 = b_3 + b_2 + b_0 + x_1$, czyli $c_1 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$
 $c_2 = b_3 + b_2 + b_1 + x_2$, czyli $c_2 = 1 \oplus 1 \oplus 1 \oplus 0 = 1$
3. Otrzymaliśmy 110 czyli dziesiętnie jest to liczba 6, która oznacza pozycję w słowie kodowym, na której wystąpiło przekłamanie.
4. Aby naprawić wiadomość należy zanegować otrzymany w ten sposób bit: 1**1**10101” – \rightarrow 1**0**10101

2 Implementacja

2.1 encoder

```
ENTITY encoder IS
PORT(
data_in: in std_logic_vector(3 downto 0) := (others => '0');
data_out: out std_logic_vector(6 downto 0) := (others => '0')
);
END encoder;

ARCHITECTURE Behavioral OF encoder IS
BEGIN
PROCESS(data_in)
BEGIN
data_out(0) <= data_in(0) xor data_in(1) xor data_in(3);
data_out(1) <= data_in(0) xor data_in(2) xor data_in(3);
data_out(2) <= data_in(0);
data_out(3) <= data_in(1) xor data_in(2) xor data_in(3);
data_out(4) <= data_in(1);
data_out(5) <= data_in(2);
data_out(6) <= data_in(3);
END PROCESS;
END;
```

2.2 decoder

```
ENTITY decoder IS
PORT(
data_out: out std_logic_vector(3 downto 0) := (others => '0');
data_in: in std_logic_vector(6 downto 0) := (others => '0');
error_out: out std_logic_vector(2 downto 0) := (others => '0')
);
END decoder;

ARCHITECTURE Behavioral OF decoder IS
signal checker : std_logic_vector(2 downto 0);
BEGIN
PROCESS(data_in)
```

```

BEGIN
checker(0) <= data_in(6) xor data_in(4) xor data_in(2) xor data_in(0);
checker(1) <= data_in(6) xor data_in(5) xor data_in(2) xor data_in(1);
checker(2) <= data_in(6) xor data_in(5) xor data_in(4) xor data_in(3);

error_out <= "000";

if checker = "011" then
data_out(0) <= not data_in(2);
error_out <= checker;
else data_out(0) <= data_in(2);
end if;

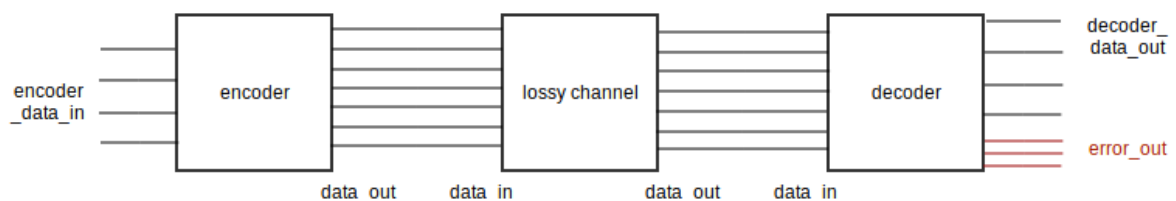
if checker = "101" then
data_out(1) <= not data_in(4);
error_out <= checker;
else data_out(1) <= data_in(4);
end if;

if checker = "110" then
data_out(2) <= not data_in(5);
error_out <= checker;
else data_out(2) <= data_in(5);
end if;

if checker = "111" then
data_out(3) <= not data_in(6);
error_out <= checker;
else data_out(3) <= data_in(6);
end if;
END PROCESS;
END;

```

2.3 Schemat połączenia komponentów



2.4 test bench

Deklaracja komponentów do testu:

```

COMPONENT lossy_channel
GENERIC (N : positive);
PORT(
data_in : IN std_logic_vector(N-1 downto 0);
clk : IN std_logic;
data_out : OUT std_logic_vector(N-1 downto 0)

```

```

);
END COMPONENT;

COMPONENT encoder IS
PORT(
data_in : IN  std_logic_vector(3 downto 0);
data_out : OUT std_logic_vector(6 downto 0)
);
END COMPONENT;

COMPONENT decoder IS
PORT(
data_out: out std_logic_vector(3 downto 0);
data_in: in  std_logic_vector(6 downto 0);
error_out: out std_logic_vector(2 downto 0)
);
END COMPONENT;

```

Zestawienie encode i decode po dwóch stronach kanału.

```

 uut: lossy_channel
GENERIC MAP ( N => WIDTH )
PORT MAP (
data_in => data_in,
clk => clk,
data_out => data_out
);

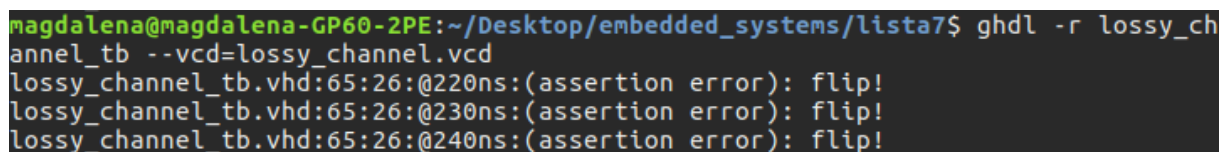
eencoder: encoder
PORT MAP(
data_in => encoder_data_in,
data_out => data_in
);

ddecoder: decoder
PORT MAP(
data_in => data_out,
error_out => error_out,
data_out => decoder_data_out
);

```

3 Sprawdzenie poprawności działania

3.1 Kanał stratny



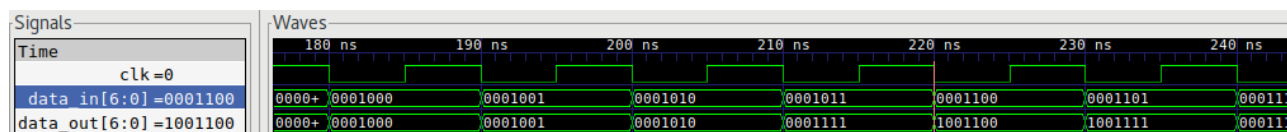
```

magdalena@magdalena-GP60-2PE:~/Desktop/embedded_systems/lista7$ ghdl -r lossy_ch
annel_tb --vcd=lossy_channel.vcd
lossy_channel_tb.vhd:65:26:@220ns:(assertion error): flip!
lossy_channel_tb.vhd:65:26:@230ns:(assertion error): flip!
lossy_channel_tb.vhd:65:26:@240ns:(assertion error): flip!

```

Rysunek 1: Kanał stratny bez kodowania Hamminga

Przesłanie przez kanał stratny kolejno danych liczbowych od 0 do 15¹ zakłamanie zostaje 3 krotnie. Błędy zgłaszane są przez asercję w momentach 220ns, 230ns i 240ns.



Rysunek 2: Wykres poziomów sygnałów kanału stratnego

Na wykresie widać, że w momencie:

- 220ns zgłoszona została zmiana bitu 3,
- 230ns zgłoszona została zmiana bitu 7,
- 240ns zgłoszone zostały zmiany bitów 2 i 7.

W dwóch przypadkach zmiany dotyczą tylko jednego bitu. Możemy się zabezpieczyć przed utratą danych stosując kodowanie Hamminga.

3.2 Kanał stratny z kodowaniem Hamminga

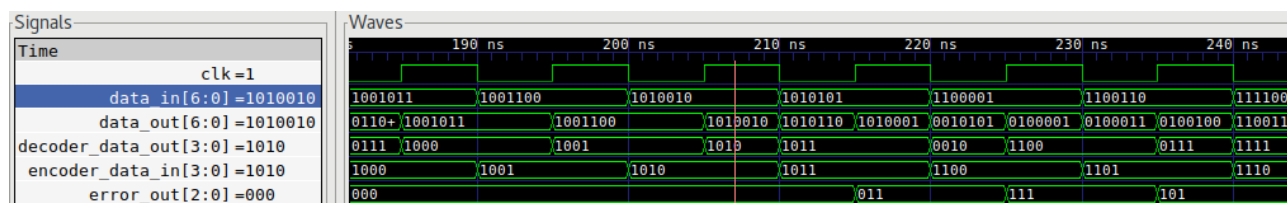
Oczekujemy, że zakłamania jednego bitu danych w przesyśle zostaną naprawione dzięki użyciu kodowania Hamminga.

```
magdalena@magdalena-GP60-2PE:~/Desktop/embedded_systems/lista7/zadanie2$ ghdl -r
lossy_channel_tb --vcd=lossy_channel_tb.vcd
lossy_channel_tb.vhd:95:13:@220ns:(assertion error): 3
lossy_channel_tb.vhd:95:13:@230ns:(assertion error): 7
lossy_channel_tb.vhd:95:13:@240ns:(assertion error): 5
lossy_channel_tb.vhd:96:24:@240ns:(assertion error): flip!
```

Rysunek 3: Kanał stratny zabezpieczony przez kodowanie Hamminga

Ponownie zostały przesłane dane liczbowe od 0 do 15, tym razem uzupełnione przez bity parzystości. Wykorzystany generator liczb pseudolosowych LFSR pozwala przy każdym uruchomieniu uzyskać takie samo zachowanie kanału stratnego.

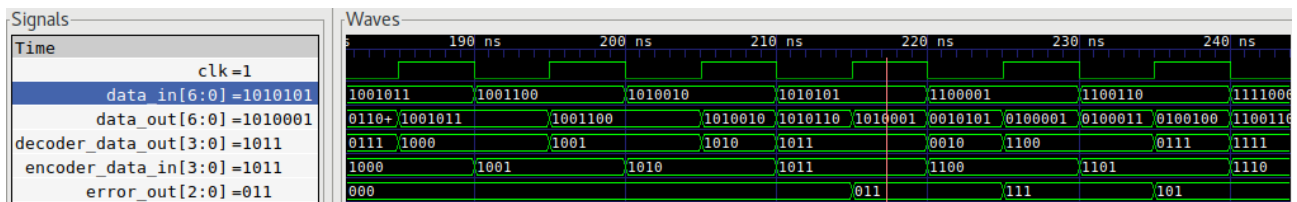
W tym wypadku zakłamanie znów wystąpiło 3 krotnie, jednak błąd w danych wystąpił jedynie raz dzięki zastosowaniu kodowania Hamminga.



Rysunek 4: Moment poprawnego przesyłu danych bez zakłamania przez kanał stratny

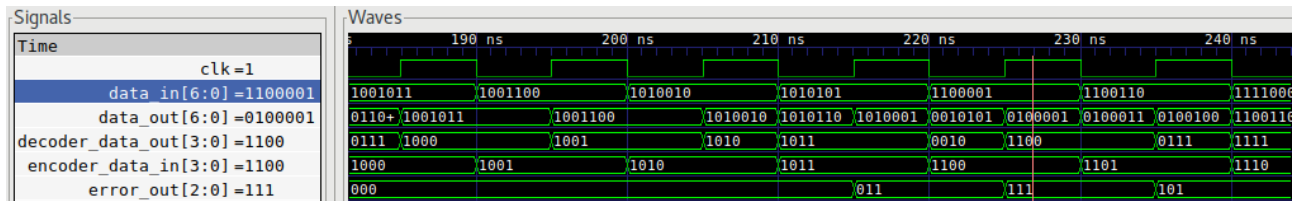
Gdy kanał stratny nie spowoduje zakłamania, wszystko przebiega poprawnie, **error_out** przyjmuje wartość "000".

¹Stosujemy szerokość kanału taką jakiej wymaga kodowanie Hamminga w celu umożliwienia porównania. Dlatego dane liczbowe 0-15, które wymagają maksymalnie 4 bitów dopełniane są przez '0' od lewej do szerokości 7 bitów.



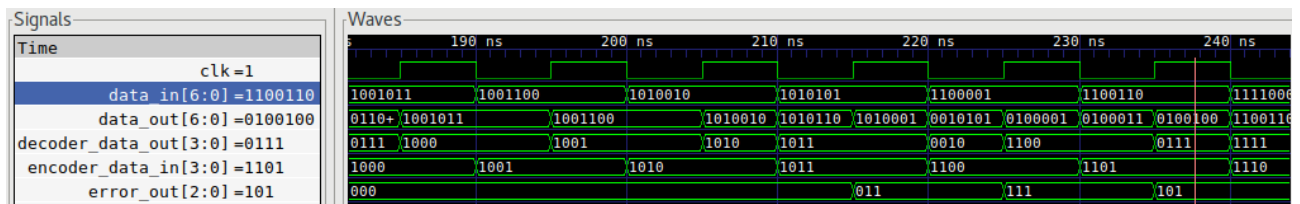
Rysunek 5: Moment przesyłu danych z zakłamaniem 3 bitu przez kanał stratny

`error_out` w 220ns wskazuje, że bit 3 został zakłamaný w trakcie przesyłu, jednak dane zwrócone przez dekodery są takie same jak te, które zostały zakodowane przed ich wysłaniem. Dzięki kodowaniu Hamminga udało się odzyskać zakłamaný bit.



Rysunek 6: Moment przesyłu danych z zakłamaniem 7 bitu przez kanał stratny

`error_out` w 230ns wskazuje, że bit 7 został zakłamaný w trakcie przesyłu, jednak dane zwrócone przez dekodery są takie same jak te, które zostały zakodowane przed ich wysłaniem. Dzięki kodowaniu Hamminga udało się odzyskać zakłamaný bit.



Rysunek 7: Moment przesyłu danych z zakłamaniem 2 i 7 bitu przez kanał stratny

`error_out` w 240ns wskazuje, że bit 5 został zakłamaný w trakcie przesyłu, jednak po analizie wartości `data_in` oraz `data_out` widać, że zakłamaniu uległy bity 2 i 7. Ponieważ zmiana dotyczy dwóch bitów, kodowanie Hamminga nie było w stanie naprawić błędu, a dane zwrócone przez dekodery są inne niż zakodowane przed wysłaniem.