POLITECHNIKA WROCŁAWSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

Układy cyfrowe i systemy wbudowane

Układy wielobitowych wejść i wyjść

Laboratorium 4

Termin zajęć: Czwartek TP, 7:30

Autorzy:

Daria Jeżowska, 252731

Kacper Śleziak, 252703

Prowadzący zajęcia:

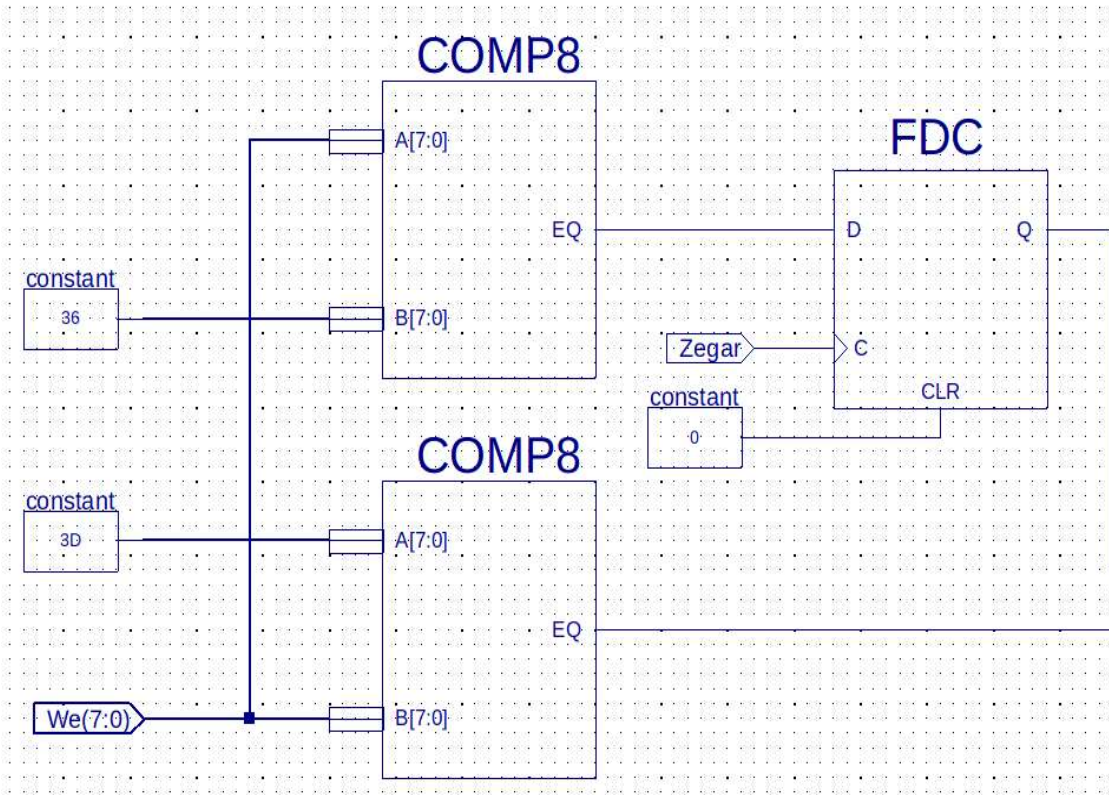
dr inż. Jacek Mazurkiewicz

Zadanie 1

Detektor 2-znakowej sekwencji słów 8-bitowych: wejścia 2 znaków 8-bitowych, 1 wyjście 1-bitowe – sekwencja rozpoznana / sekwencja błędna.

Schemat

Do tego celu użyliśmy dwóch funktorów z biblioteki dostępnej w programie Xilinx. Komparator COMP8 posiada dwa wejścia przyjmujące 8-bitowe dane oraz jedno wyjście. Są to odpowiednio A, B i EQ. Porównywane są kolejne bity obu słów (A(7) z B(7), A(6) z B(6), ..., A(0) z B(0)). Na jedno wejście wprowadzamy liczbę, której stałą wartość nadajemy jej na samym początku. Natomiast drugie wejście jest wejściem od użytkownika. Jeśli dane wejściowe są takie same to na wyjściu EQ otrzymujemy 1. Użyliśmy także dodatkowo przerzutnika D, który umożliwia opóźnienie sygnału, aby można było porównać dane z obu komparatorów. Do tego natomiast została użyta bramka AND2. Gdy z obu komparatory na wyjściu będzie 1 to nasze wyjście, potwierdzające wykrycie wybranej sekwencji, także będzie jedynką. Szukaliśmy sekwencji składającej się po kolei z liczb 35 oraz 36.



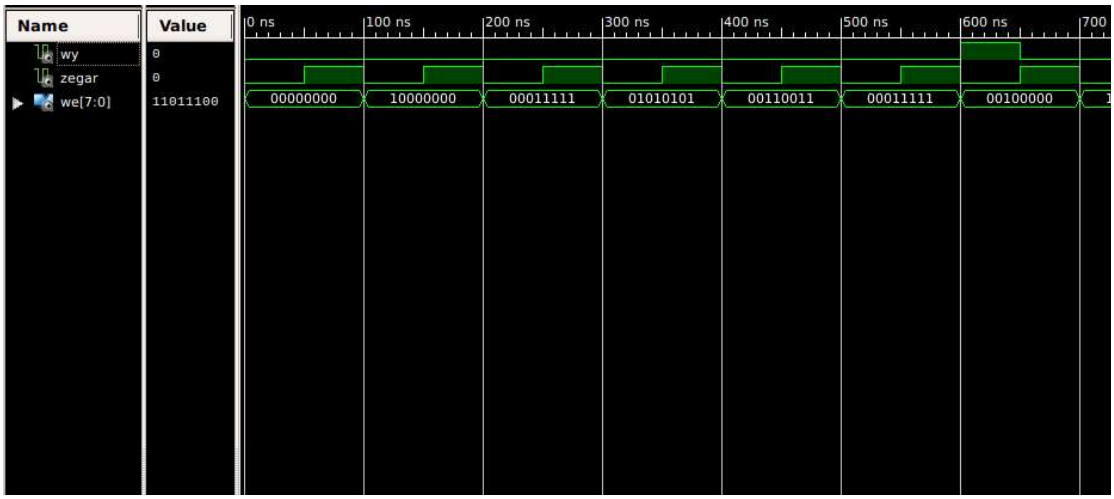
Kod w języku VHDL

Zdecydowanie większa część kodu została wygenerowana automatycznie na podstawie powyższego schematu. Przypisaliśmy dla wejścia We kolejne słowa ośmiobitowe, które po kolei są porównywane z liczbami wyżej podanymi. Jest także ustawiony zegar, abyśmy mogli na jeden jego takt przechować wyjście z komparatora pierwszego.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
ENTITY schemat_schemat_sch_tb IS
END schemat_schemat_sch_tb;
ARCHITECTURE behavioral OF schemat_schemat_sch_tb IS

   COMPONENT schemat
   PORT( Wy       :  OUT   STD_LOGIC;
         Zegar    :  IN STD_LOGIC;
         We       :  IN STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;

   SIGNAL Wy      :  STD_LOGIC;
   SIGNAL Zegar   :  STD_LOGIC:='0';
   SIGNAL We      :  STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

   UUT: schemat PORT MAP(
      Wy => Wy,
      Zegar => Zegar,
      We => We
   );

   We <= "00000000", "10000000" after 100ns, "00011111" after 200ns, "01010101" after 300ns, "00110011" after
         "00011111" after 500ns, "00100000" after 600ns, "11101110" after 700ns, "11110000" after 800ns, "1101

   Zegar <= not Zegar after 50ns;

   tb : PROCESS
   BEGIN
      WAIT; -- will wait forever
   END PROCESS;
END;
```

Symulacja

Na symulacji widać kolejne słowa z wejścia We, takty zegara oraz stany wyjścia Wy. W 500ns pojawia się na wejściu 00011111, a następnie 00100000 – stany, dla których wyjście powinno mieć stan wysoki i jak widać na schemacie dokładnie tak jest.



Testowanie układu na płytce

Aby przetestować program na płytce musieliśmy najpierw wprowadzić zmiany w pliku UCF. W pliku UCF uaktywniliśmy zegar zmieniający się co 5ms. Nasze kolejne bity słowa ośmiobitowego są odpowiednio na KEY od 7 do 0. Naszej wyjście umieściliśmy na diodzie LED<1>. Po uruchomieniu układu na płytce działał on prawidłowo. Przykładowo – dla kolejnych słów 00001111 i 00110011 dioda led nie wskazywała na wykrycie sekwencji. Dopiero po wprowadzeniu 00011111, a następnie 00100000 dioda led wskazała, że sekwencja została odnaleziona.

```
1  #----------------------------------------------
2  #  ZL-9572 CPLD board,  J.Sugier 2009
3  #----------------------------------------------
4
5  # Clocks
6  NET "Zegar" LOC = "P7" | BUFG = CLK | PERIOD = 5m
7
8  #NET "Clk_XT" LOC = "P5" | BUFG = CLK | PERIOD =
9
10 # Keys
11 NET "We(0)" LOC = "P42"; #wejscie
12 NET "We(1)" LOC = "P40";
13 NET "We(2)" LOC = "P43";
14 NET "We(3)" LOC = "P38";
15 NET "We(4)" LOC = "P37";
16 NET "We(5)" LOC = "P36";  # shared with ROT_A
17 NET "We(6)" LOC = "P24";  # shared with ROT_B
18 NET "We(7)" LOC = "P39";  # GSR
19
20 # LEDS
21 NET "Wy"  LOC = "P35";
22 #NET "LED<1>"  LOC = "P29";
23 #NET "LED<2>"  LOC = "P33";
24 #NET "LED<3>"  LOC = "P34";
25 #NET "LED<4>"  LOC = "P28";
26 #NET "LED<5>"  LOC = "P27";
27 #NET "LED<6>"  LOC = "P26";
28 #NET "LED<7>"  LOC = "P25";
```

Zadanie 2

Sumator pracujący na dwóch argumentach 4-bitowych wyrażonych w odzie Aikena i generujący wynik w tym samym kodzie.

Schemat

Najbardziej intuicyjnym rozwiązaniem wyżej wymienionego problemu było zastosowanie zwykłego sumatora ADD4. ADD4 jako argumenty przyjmuje dane w kodzie NKB więc pierwszym krokiem do wykonania zadania była zamiana liczb z kodu Aikena na NKB. Wykorzystując fakt, że liczby od 0 do 4 w NKB pokrywają się w obu kodach należało wyłącznie zamienić liczby z przedziału 5 do 9. Udało nam się zauważyć,że liczby w kodzie Aikena są przesunięte o wartość 0110 względem ich odpowiedników w NKB, wystarczyło więc odjąć wartość 6 od liczby w kodzie Aikena aby otrzymać jej odpowiednik w kodzie NKB. Wykorzystując wiedzę zdobytą na kursie AK ustaliliśmy, że wartość ,,-6" w kodzie U2 jest równa kodowi 1010. Dodanie tej liczby z przedziału 5 do 9 pozwoli zamienić je na kod Aikena np. 1111(15) + 1010(-6) = 1001(9). Według założeń zadania wynik również powinien zostać zapisany w kodzie Aikena w związku z czym należało dokonać następnej konwersji, tym razem z kodu NKB na kod Aikena. Z faktu, że wynik musi być zapisany na 4 bitach pozwala nam to na zapisanie wyniku nie większego niż 9. W takim przypadku jedynym słusznym rozwiązaniem rozwiązania kwestii zapisanego wyniku było generowanie informacji o przepełnieniu dla wyników większych niż 9. Ostatnim krokiem było dodanie wartości 0110(6), w przypadku gdy suma uzyskana suma należała do przedziału <5;9>.

Wykrywanie czy dane wejściowe mają wartość większą niż 4

1- 4 – Takie same w kodzie Aikena i kodzie NKB

6 -10 – Nie występują w kodzie Aikena

11-15 – Trzeba obniżyć o 6, 11

Wartość	x3	x2	x1	x0	y
1	0	0	0	0	0
2	0	0	0	1	0
3	0	0	1	0	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	-
7	0	1	1	1	-
8	1	0	0	0	-
9	1	0	0	1	-
10	1	0	1	0	-
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

x3x2/x1x0	00	01	11	10
00	0	0	1	-

01	0	-	1	-
11	0	-	1	1
10	0	-	1	-

Z tabeli powyżej można zauważyć, że gdy najbardziej znaczący bit przyjmuje wartość 1 to wymagana jest konwersja do kodu NKB. W przypadku gdy najbardziej znaczący bit jest różny od 1 to sumator ADD4 dodaje do liczby w kodzie Aikena liczbę równą 0000 co nie zmienia jest wartości.

Synteza podukładu wykrywającego, czy wynik dodawania jest większy niż 9

W momencie wystąpienia wartości większej bądź równej 10 występuje przepełnienie.

Wartość	x3	x2	x1	x0	y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Minimalizacja siatką Karnaugh

x3x2/x1x0	00	01	11	10
01	0	0	1	0
11	0	0	1	0
11	0	0	1	1
10	0	0	1	1

Y podłączyliśmy do bramki OR2, której wyjściem jest wyjście CO(generujące standardowe przesunięcie dla NKB) poprzednio użytego sumatora. Taki prosty układ pozwala uwzględnić wszystkie przypadki przepełnienia.

Sprawdzanie czy wartość należy do przedziału <5;9>

Wartość	x3	x2	x1	x0	y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

x3x2/x1x0	00	01	11	10
00	0	0	0	1
01	0	1	0	1
11	0	1	0	0
10	0	1	0	0

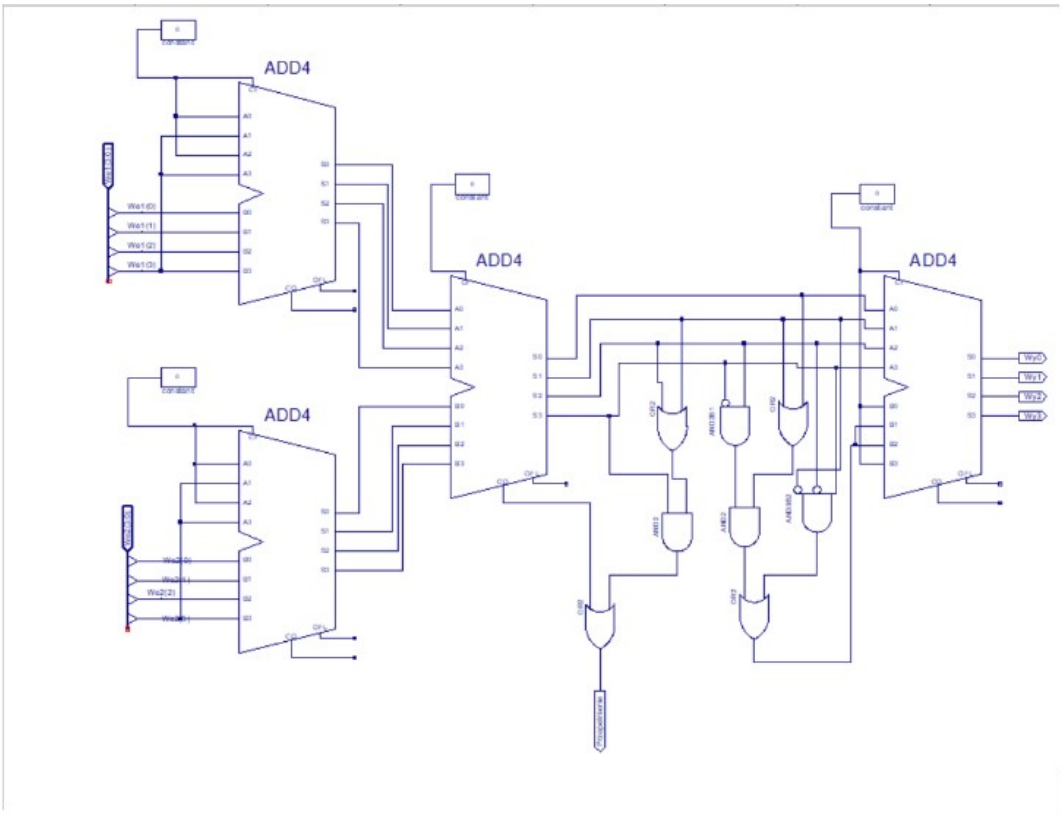
Schemat

We1 - Pierwsze wejście na liczbę w kodzie Aikena

We2 - Drugie wejście na liczbę w kodzie Aikena

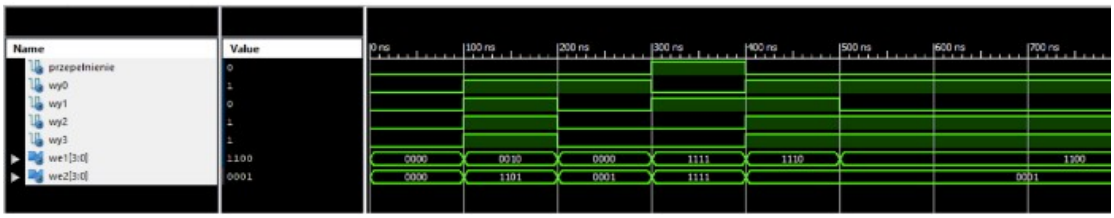
Wy0,Wy1,Wy2,Wy3 -Kolejne bity wyniku dodawania

Przepełnienie - symbolizuje wystąpienie przepełnienie



Symulacja

Symulacja na podstawie poniższego kodu



Kod w języku VHDL

Kod w większości został wygenerowany ze schematu, który znajduje się powyżej. Dopisana przez nas została jedynie 55 i 56 linijka odpowiadająca za zmianę wejść magistrali w odstępach czasowych potrzebna do symulacji.

```
15  LIBRARY ieee;
16  USE ieee.std_logic_1164.ALL;
17  USE ieee.numeric_std.ALL;
18  LIBRARY UNISIM;
19  USE UNISIM.Vcomponents.ALL;
20  ENTITY sumator_sumator_sch_tb IS
21  END sumator_sumator_sch_tb;
22  ARCHITECTURE behavioral OF sumator_sumator_sch_tb IS
23
24     COMPONENT sumator
25     PORT( Przepelnienie  :  OUT   STD_LOGIC;
26           Wy0  :  OUT   STD_LOGIC;
27           Wy1  :  OUT   STD_LOGIC;
28           Wy2  :  OUT   STD_LOGIC;
29           Wy3  :  OUT   STD_LOGIC;
30           We1  :  IN STD_LOGIC_VECTOR (3 DOWNTO 0);
31           We2  :  IN STD_LOGIC_VECTOR (3 DOWNTO 0));
32     END COMPONENT;
33
34     SIGNAL Przepelnienie :  STD_LOGIC;
35     SIGNAL Wy0  :  STD_LOGIC;
36     SIGNAL Wy1  :  STD_LOGIC;
37     SIGNAL Wy2  :  STD_LOGIC;
38     SIGNAL Wy3  :  STD_LOGIC;
39     SIGNAL We1  :  STD_LOGIC_VECTOR (3 DOWNTO 0);
40     SIGNAL We2  :  STD_LOGIC_VECTOR (3 DOWNTO 0);
41
42  BEGIN
43
44     UUT: sumator PORT MAP(
45        Przepelnienie => Przepelnienie,
46        Wy0 => Wy0,
47        Wy1 => Wy1,
48        Wy2 => Wy2,
49        Wy3 => Wy3,
50        We1 => We1,
51        We2 => We2
52     );
53
54     We1 <="0000", "0010" after 100ns, "0000" after 200ns, "1111" after 300ns, "1110" after 400ns, "1100" after 500ns;
55     We2 <= "0000", "1101" after 100ns, "0001" after 200ns, "1111" after 300ns, "0001" after 400ns, "0001" after 500ns;
56
57  -- *** Test Bench - User Defined Section ***
58     tb : PROCESS
59     BEGIN
60        WAIT; -- will wait forever
61     END PROCESS;
62  -- *** End Test Bench - User Defined Section ***
63
64  END;
```

Plick UCF

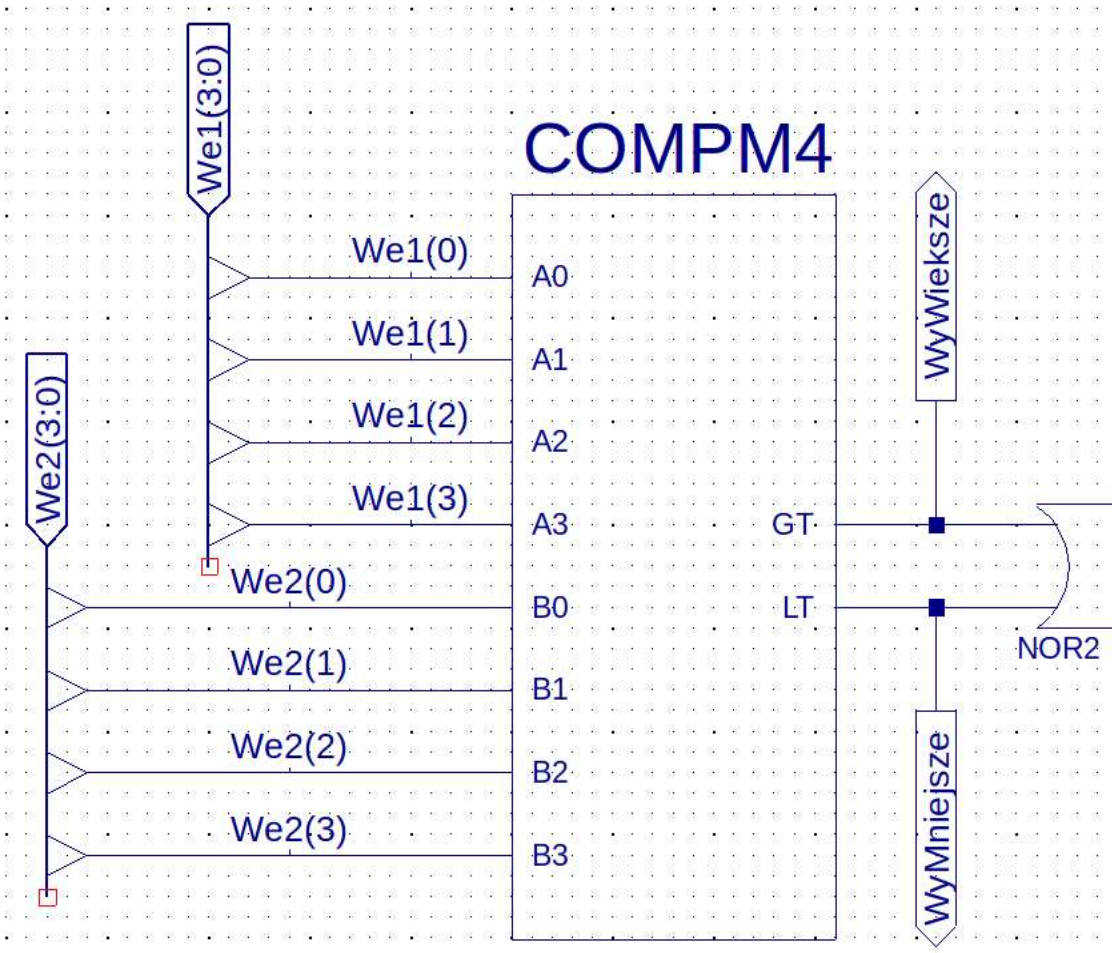
```
#NET "Clk_XT" LOC = "P5" | BUFG = CLK | PERIOD = 500ns HIGH 50%;

# Keys
NET "We1(0)" LOC = "P42";
NET "We1(1)" LOC = "P40";
NET "We1(2)" LOC = "P43";
NET "We1(3)" LOC = "P38";
NET "We2(0)" LOC = "P37";
NET "We2(1)" LOC = "P36";  # shared with ROT_A
NET "We2(2)" LOC = "P24";  # shared with ROT_B
NET "We2(3)" LOC = "P39";  # GSR

# LEDS
NET "Wy0"  LOC = "P35";
NET "Wy1"  LOC = "P29";
NET "Wy2"  LOC = "P33";
NET "Wy3"  LOC = "P34";
#NET "LED<4>"  LOC = "P28";
#NET "LED<5>"  LOC = "P27";
#NET "LED<6>"  LOC = "P26";
NET "Przepelnienie"  LOC = "P25";

#NET "LED<8>"  LOC = "P13"; # shared with seg. B
#NET "LED<9>"  LOC = "P11"; # shared with seg. F
#NET "LED<10>" LOC = "P12"; # shared with seg. A
#NET "LED<11>" LOC = "P18"; # shared with seg. DP
#NET "LED<12>" LOC = "P22"; # shared with seg. C
#NET "LED<13>" LOC = "P20"; # shared with seg. G
#NET "LED<14>" LOC = "P19"; # shared with seg. D
#NET "LED<15>" LOC = "P14"; # shared with seg. E

# DISPL. 7-SEG
#NET "D7S_D<0>" LOC = "P8" | SLEW = "SLOW";
#NET "D7S_D<1>" LOC = "P6" | SLEW = "SLOW";
#NET "D7S_D<2>" LOC = "P4" | SLEW = "SLOW";
#NET "D7S_D<3>" LOC = "P9" | SLEW = "SLOW";
#NET "D7S_S<0>" LOC = "P12";    # Seg. A; shared with LED<10>
#NET "D7S_S<1>" LOC = "P13";    # Seg. B; shared with LED<8>
#NET "D7S_S<2>" LOC = "P22";    # Seg. C; shared with LED<12>
#NET "D7S_S<3>" LOC = "P19";    # Seg. D; shared with LED<14>
#NET "D7S_S<4>" LOC = "P14";    # Seg. E; shared with LED<15>
#NET "D7S_S<5>" LOC = "P11";    # Seg. F; shared with LED<9>
#NET "D7S_S<6>" LOC = "P20";    # Seg. G; shared with LED<13>
#NET "D7S_S<7>" LOC = "P18";    # Seg. DP;shared with LED<11>

# Rotary encoder
#NET "ROT_A" LOC = "P36";   # shared with Key<5>
#NET "ROT_B" LOC = "P24";   # shared with Key<6>

# PS/2
#NET "PS2_Clk"   LOC = "P3";
#NET "PS2_Data"  LOC = "P2";

# RS-232
#NET "RS_RX" LOC = "P1";
#NET "RS_TX" LOC = "P44";
```

Zadanie 4

Komparator dwóch 4-bitowych liczb: 2 wejścia po 4 bity, 3 wyjścia 1-bitowe: mniejszy, większy, równy

Schemat

Do tego zadania użyliśmy COMPM4, który porównuje dwie czterobitowe liczby oraz na wyjściu GT będzie stan wysoki, jeśli pierwsza liczba (wejścia A0 - A3) jest większa, natomiast na wyjściu LT będzie stan wysoki, gdy pierwsza liczba będzie mniejsza. W przypadku, gdy liczby są równe wyjścia GT i LT są stanem niskim, więc aby WyRowne, mówiące o tym, że liczby są równe, musimy użyć bramki NOR2.



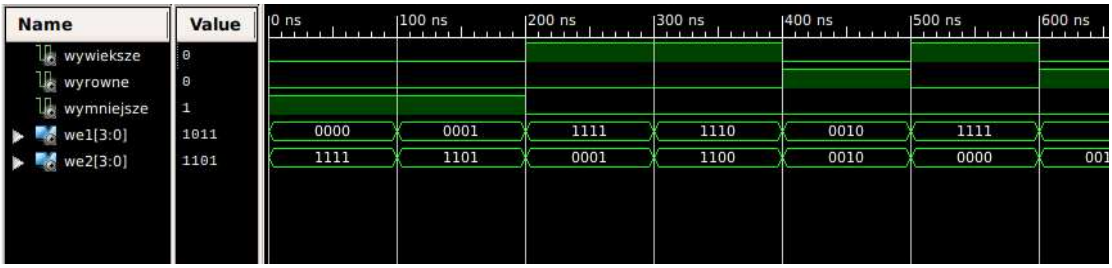
Kod w języku VHDL

Kod w większości został wygenerowany ze schematu, który znajduje się powyżej. Dla wejść We1 oraz We2 przypisujemy czterobitowe słowa.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;
ENTITY schemat_schemat_sch_tb IS
END schemat_schemat_sch_tb;
ARCHITECTURE behavioral OF schemat_schemat_sch_tb IS

   COMPONENT schemat
   PORT( WyWieksze   :  OUT   STD_LOGIC;
         WyMniejsze :  OUT   STD_LOGIC;
         WyRowne :  OUT   STD_LOGIC;
         We1   :  IN STD_LOGIC_VECTOR (3 DOWNTO 0);
         We2   :  IN STD_LOGIC_VECTOR (3 DOWNTO 0));
   END COMPONENT;

   SIGNAL WyWieksze  :  STD_LOGIC;
   SIGNAL WyMniejsze :  STD_LOGIC;
   SIGNAL WyRowne :  STD_LOGIC;
   SIGNAL We1   :  STD_LOGIC_VECTOR (3 DOWNTO 0);
   SIGNAL We2   :  STD_LOGIC_VECTOR (3 DOWNTO 0);

BEGIN

   UUT: schemat PORT MAP(
      WyWieksze => WyWieksze,
      WyMniejsze => WyMniejsze,
      WyRowne => WyRowne,
      We1 => We1,
      We2 => We2
   );

   We1 <= "0000", "0001" after 100ns, "1111" after 200ns, "1110" after 300ns, '
          "1111" after 500ns, "0011" after 600ns, "0011" after 700ns, "0001" af

   We2 <= "1111", "1101" after 100ns, "0001" after 200ns, "1100" after 300ns, '
          "0000" after 500ns, "0011" after 600ns, "0100" after 700ns, "0100" af

   tb : PROCESS
   BEGIN
      WAIT; -- will wait forever
   END PROCESS;

END;
```

Symulacja

Na symulacji widać po kolei zmieniające się liczby czterobitowe oraz wyjścia WyWieksze, WyRowne i WyMniejsze. Dla liczb We1 = 0000 i We2 = 1111 stan wysoki jest dla wyjścia WyMniejsze, ponieważ We1 jest mniejsze od We2. Dla liczb We1 = 1110 oraz We2 = 1100 stan wysoki jest dla WyWieksze, ponieważ We1 > We2. Natomiast stan wysoki dla WyRowne jest dla dwóch takich samych liczb, czyli przykładowo We1 = 0010 oraz We2 = 0010.



Testowanie na płytce

W pierwszej kolejności musieliśmy skonfigurować plik UCF. KEY od 0 do 3 jest ustawione dla We1, a KEY od 4 do 7 dla We2. WyMniejsze, WyRowne oraz WyWieksze są odpowiednio przypisane dla diod led 0, 4 oraz 7. Ustawianie zegara nie było potrzebne w tym zadaniu. Po uruchomieniu układu na płytce otrzymaliśmy oczekiwane wyniki. Przykładowo – dla słów We1 - 0000 i We2 - 0001 zaświeciła się dioda odpowiadająca WyMniejsze, dla We1 = 0011 i We2 = 0011 zaświeciła się dioda odpowiadająca WyRowne oraz dla We1 = 1100 i We2 = 0001 zaświeciła się dioda odpowiadające WyWieksze.

```
#-------------------------------------------------
#  ZL-9572 CPLD board,  J.Sugier 2009
#-------------------------------------------------

# Clocks
#NET "Clk_LF" LOC = "P7" | BUFG = CLK | PERIOD

#NET "Clk_XT" LOC = "P5" | BUFG = CLK | PERIOD

# Keys
NET "We1(0)" LOC = "P42";
NET "We1(1)" LOC = "P40";
NET "We1(2)" LOC = "P43";
NET "We1(3)" LOC = "P38";
NET "We2(0)" LOC = "P37";
NET "We2(1)" LOC = "P36";  # shared with ROT_A
NET "We2(2)" LOC = "P24";  # shared with ROT_B
NET "We2(3)" LOC = "P39";  # GSR

# LEDS
NET "WyMniejsze"  LOC = "P35";
#NET "LED<1>"  LOC = "P29";
#NET "LEC<2>"  LOC = "P33";
#NET "LED<3>"  LOC = "P34";
NET "WyRowne"  LOC = "P28";
#NET "LED<5>"  LOC = "P27";
#NET "LED<6>"  LOC = "P26";
NET "WyWieksze"  LOC = "P25";
```

Wnioski

Aby wykonać te zadania musieliśmy poszukać o wiele więcej informacji niż wcześniej, aby móc wybrać odpowiednie rozwiązanie, które będą spełniać potrzeby zadań. Wymagały one także więcej pomyślunku i dogłębnego zrozumienia, przy tym laboratorium nie wystarczyły grafy, tabele czy siatki Karnaugh - musieliśmy poszukać nowych sposobów i bardziej się zagłębić w gotowe rozwiązania zawarte w Xilinxie. Szczególnie problematyczne okazało się zadanie numer 2 w którym dokładna analiza kodu Aikena była niezbędna do znalezienia odpowiedniego rozwiązania wyżej wymienionego problemu.