Kurs CPLD

Ćwiczenia z VHDL

część 6

Słowo na początek

W poprzedniej części naszego kursu poznawaliśmy język VHDL. Nie wiem, czy przypadł on do gustu Czytelnikom, ale pozwolę sobie wymienić kilka powodów, dla których warto polubić język opisu sprzętu. Po pierwsze nie musimy nic projektować ręcznie, obywamy się bez tablic Karnaugha i metody Quinea'a–McCluskeya, co w oczywisty sposób oszczędza nasz czas. Drugą zaletą jest większa „kompresja" treści – listingi są bardziej zwięzłe od schematów. Struktury języka takie jak *when* czy *if* zwiększają poziom abstrakcji projektu i tym samym umożliwiają podejście do tematu projektowania układów cyfrowych od innej strony.

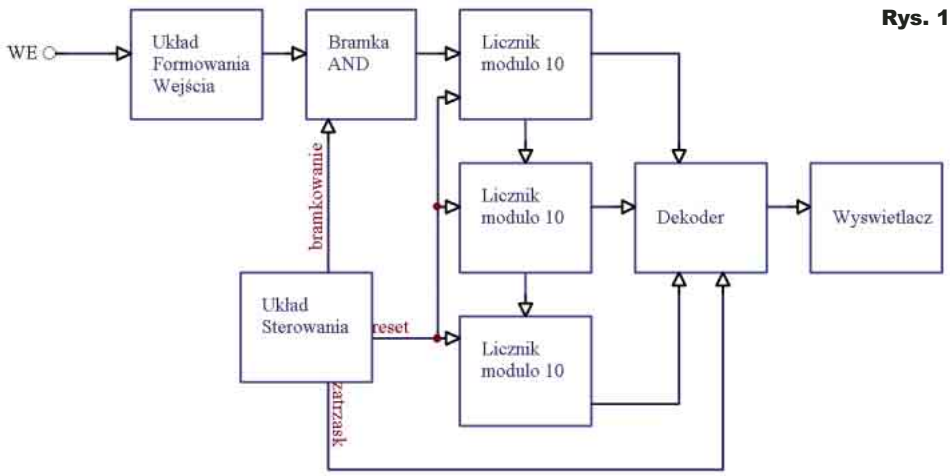
Dzisiaj chciałbym pokazać Czytelnikom kolejny sposób na uproszczenie procesu tworzenia układów cyfrowych – budowanie urządzenia z bloków funkcjonalnych opisanych w języku VHDL. Dlaczego warto postępować w taki sposób ? Odpowiedź jest praktycznie taka sama, jak w przypadku układów iteracyjnych – łatwiej jest przygotować mały, stosunkowo prosty moduł, niż od razu całe, złożone urządzenie. W przypadku bardzo dużych przedsięwzięć, np. systemu operacyjnego dla komputera, praktycznie niemożliwe jest, by jedna osoba stworzyła pełną i spójną wizję systemu i była w stanie zamienić ją w gotowy kod. Nawet gdyby taki przypadek miał miejsce, to osoba taka jest wybitnym geniuszem i prawdopodobnie nie czyta tego kursu :). Mimo to nie będzie to dobre rozwiązanie, gdyż system będzie po prostu nieczytelny i trudny w nanoszeniu poprawek. Przeciwdziałanie nadmiernej złożoności i zwiększanie elastyczności kodu rozwiązuje się właśnie poprzez podział całości na małe bloki, które mogą być realizowane przez różne osoby. Jeżeli to wyjaśnienie nie jest przekonujące, to zróbmy mały test: czy jesteś w stanie szybko opisać, jak działa miernik częstotliwości? Pomijając oczywiście ogólniki typu „liczy i wyświetla". Właśnie zasadnicze pytanie polega na tym, JAK liczy i jakie warunki muszą zostać spełnione? A gdybyś spojrzał na **rysunek 1**? Czy teraz będzie to prostsze do wytłumaczenia? Przyrząd został podzielony na mniejsze kawałki – bloki funkcjonalne.

Sposób jego działania powinien być teraz prostszy do zrozumienia. Na samym początku znajduje się układ formowania sygnału wejściowego. Znajduje on zastosowanie przy pomiarze częstotliwości sygnałów innych niż cyfrowe, które należy przetworzyć tak, aby mogły być „obrabiane" przez układy cyfrowy. Z bloku takiego powinien wychodzić już właściwy sygnał cyfrowy. Następnym elementem jest bramka AND, która spełnia w tym przypadku rolę „klucza" – umożliwia dołączenie bądź odłączenie sygnału od liczników. Za bramką znajdują się liczniki modulo 10. Nietrudno zgadnąć, że schemat z rysunku 1 został zoptymalizowany na nasze potrzeby, w normalnym wypadku może to być również zwyczajny licznik n-bitowy. Dekoder zajmuje się odczytem zawartości liczników i prezentacją ich zawartości na wyświetlaczach.

Najważniejszą rolę pełni jednakże układ sterujący. Ideę jego pracy można streścić w pytaniu: co to znaczy, że sygnał ma częstotliwość *x* Hz ? Oznacza to, że w ciągu jednej sekundy ma on *x* okresów, a okres jest złożony ze stanu niskiego i wysokiego. Wynika z tego, że *x* Hz sprowadza się do pojawienia *x* zboczy narastających/opadających. Konkluzja jest z tego taka, że doprowadzenie tego sygnału do licznika na dokładnie jedną sekundę spowoduje zliczenie *x* zboczy i zawartość licznika będzie stanowiła wynik pomiaru. Zadaniem układu sterującego jest zatem wytworzenie impulsu trwającego dokładnie jedną sekundę, który uaktywni bramkę i tym samym dołączy sygnał do licznika na jedną sekundę, powodując zliczenie *x* zboczy (Hz).

Życie nie jest niestety takie proste i tutaj również pojawiają się dwa „ale". Po pierwsze, nie możemy włączać bramki co jedną sekundę, gdy licznik nie jest wyzerowany. Spowodowałoby to sukcesywne zwiększanie jego zawartości – po dołączeniu do wejścia przebiegu o częstotliwości 10Hz pierwszy pomiar będzie prawidłowy, ale drugi pokaże 20Hz, trzeci 30Hz, itd. Wniosek: przed rozpoczęciem zliczania trzeba licznik wyzerować, stąd obecność sygnału *reset* doprowadzonego do wszystkich liczników. Po drugie, operacje zliczania i zerowania będą widoczne! Najpierw wyświetlacz będzie bardzo szybko zmieniał swoją zawartość, potem wyświetli prawidłową wartość, następnie ulegnie ona wyzerowaniu, a potem znowu zliczanie... Chyba każdy zgodzi się, że nie jest to estetyczne rozwiązanie. Jak temu przeciwdziałać? Potrzebny byłby układ pamiętający ostatni wynik (na czas wykonywania nowego pomiaru) i zapisujący nowy po zakończeniu pomiaru. Omawialiśmy już takie rozwiązanie, stanowi je zatrzask. Na schemacie z rysunku 1 zatrzask jest zintegrowany z dekoderem, ale układ sterujący ma dedykowany sygnał (*zatrzask)*, który jednoznacznie wskazuje, kiedy dane mają być pobrane z wyjść liczników i zapamiętane.

Mam nadzieje, że przykład miernika częstotliwości pokazał, iż podział układu na bloki funkcjonalne upraszcza zarówno budowę, jak i zrozumienie funkcjonowania urządzenia.



Rys. 1

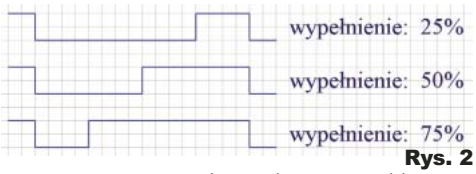
Generator PWM

A teraz spróbujmy przygotować prosty generator PWM opisany w VHDL-u. Najmłodszym Czytelnikom przypomnę, że jest to układ umożliwiający zmianę wypełnienia przebiegu prostokątnego. Oznacza to, że możemy regulować czas trwania poziomu wysokiego w takim sygnale, z tym że jego wydłużenie pociąga za sobą automatycznie skrócenie czasu trwania stanu niskiego. Musi być spełniona zależność:

$t_T = t_{TH} + t_{TL} = const$, gdzie:

t_T – czas trwania całego okresu

t_{TL} – czas trwania poziomu niskiego w okresie



Rys. 2

t_{TH} – czas trwania poziomu wysokiego w okresie

const – oznacza wartość stałą, niezmienną

Niespełnienie tego warunku spowoduje, że zmienimy częstotliwość przebiegu, a nie wypełnienie. Na **rysunku 2** pokazano przykładowe przebiegi o różnym wypełnieniu, które powinny rozwiać ewentualne wątpliwości.

Pojawia się zasadnicze pytanie: jak pracuje generator PWM? Podstawowym zadaniem jest „trzymanie" stałego okresu. Zadanie to może spełniać licznik modulo *n*, który będzie odliczał określoną liczbę sygnałów zegarowych i w takim przypadku okres będzie równy *n*. Załóżmy, że *n* będzie wynosić 10, co pozwoli nam w prosty sposób wizualizować pracę PWM na pojedynczym wyświetlaczu 7-segmentowym. Potrzebny jest również element „zadający", tzn. musimy mieć możliwość ustawienia wartości wypełnienia dla naszego generatora PWM. Dość prostym wyjściem będzie wstawienie drugiego licznika, który opiszemy tak, aby dawał możliwość zliczania w górę i w dół, co pozwoli łatwo ustawiać wypełnienie za pomocą dwóch przycisków.

Pozostaje już jedynie wymyślenie sposobu jak połączyć ze sobą te dwa liczniki, aby dawały generator PWM. Gdybyśmy ustawili wartość 4, to dla licznika modulo 10 (stany 0,1,...,9) jest to połowa wypełnienia. Oznacza to, że na wyjściu panuje stan niski, gdy na pierwszym liczniku jest 0, 1, 2, 3, 4, czyli gdy jego stany są MNIEJSZE LUB RÓWNE od ustawionej wartości. Elementem określającym, czy coś jest mniejsze lub równe, inaczej czy jest większe, jest bez wątpienia komparator. Do jego wejść należy podłączyć wyjścia obu liczników, a wyjście komparatora będzie wyjściem naszego generatora PWM.

Przyjrzyjmy się zatem **listingowi 1** i prześledźmy, w jaki sposób zaimplementowano rozwiązanie omówione przed chwilą. Stworzymy element, który będzie dostępny w bibliotece edytora schematów.

Implementację zaczynamy tradycyjnie od zdefiniowania wejść i wyjść:

x – poprzez ten wektor wyprowadzimy informację do wyświetlacza o aktualnej wartości wypełnienia

pwm – ten port będzie reprezentował sygnał o zmiennym (ustawianym) wypełnieniu

CLK – sygnał zegarowy z generatora kwarcowego

INC, *DEC* – wejścia pozwalające odpowiednio zwiększyć/zmniejszyć wypełnienie

Należy zwrócić uwagę na filtrację drgań zestyków, gdyż docelowo sygnały *INC* oraz *DEC* będą przyłączone do przycisków. Zagadnieniu temu poświęcony jest fragment kodu pod linijką komentarza:

--filtracja drgań zestyków

Znajduje się tam konkretnie proces mający na liście czułości sygnał zegarowy CLK. Wewnątrz procesu tworzona jest zmienna *presk* stanowiąca prosty licznik. Jego zawartość jest zwiększana przy każdym zboczu narastającym, co zapewnia instrukcja:

if falling_edge(CLK) then
(...)
end if;

Podczas każdego zerowania licznika, które występuje po jego przepełnieniu (co 2^{16} taktów zegara, bo licznik jest 16-bitowy), do zmiennych pomocniczych *INCprim* oraz *DECprim* przepisywana jest zawartość sygnałów wejściowych *INC* oraz *DEC*. Rozwiązuje to problem drgań na zestykach przycisków.

Nie opracowałem wprawdzie inteligentnego licznika pozwalającego zmieniać kierunek zliczania, ale sięgnąłem po inne rozwiązanie – dwa niezależne liczniki: *counter_up* i *counter_down*. Są to liczniki programowalne umożliwiające ustawienie wartości za pomocą przycisków. Do poprawnej pracy układu niezbędna jest jeszcze zmienna *temp*, która przechowuje wartość decydującą o aktualnym wypełnieniu. Proces *pwmCounterEdge* ma na liście czułości sygnał zegarowy *CLK*. Każde zbocze opadające powoduje zwiększenie wartości licznika modulo 10 (*pwmCounter*) zrealizowanego następującymi instrukcjami:

pwmCounter := pwmCounter + 1;
if pwmCounter > 9 then
pwmCounter := „0000" ;
end if ;

Mamy tutaj do czynienia z prostym zwiększeniem wartości o jeden i instrukcją

Listing 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity pwm_module is
  port(
    --informuje o ustawionej wartosci
    x : out std_logic_vector(3 downto 0) ;
    --wyjscie z sygnalem PWM
    pwm : out std_logic ;
    --sygnal zegarowy dla PWM
    CLK : in std_logic ;
    --wejscia zwieksz/zmniejsz wypelnienie
    INC, DEC : in std_logic
  );
end pwm_module;
architecture pwm_moduleBehav of pwm_module is
  signal temp : unsigned (3 downto 0) ;
  signal counter_up : unsigned (3 downto 0) ;
  signal counter_down : unsigned (3 downto 0) ;
  signal INCprim, DECprim : std_logic ;
begin
  --filtracja drgan zestykow
  filtr: process (CLK) is
    variable presk : unsigned (16 downto 0)  ;
  begin
    if falling_edge(CLK) then
      presk := presk + 1 ;
      if presk = 0 then
        INCprim <= INC ;
        DECprim <= DEC  ;
      end if ;
    end if ;
  end process filtr ;
  --zliczanie sygnalow zegarowych
  pwmCounterEdge: process (CLK) is
    variable pwmCounter : unsigned (3 downto 0) ;
  begin
    if falling_edge(CLK) then
      pwmCounter := pwmCounter + 1;
      if pwmCounter > 9 then
        pwmCounter := „0000" ;
      end if ;
    end if ;
    --komparator
    if  pwmCounter  <  temp  then
      pwm <= '1' ;
    else
      pwm <= '0' ;
    end if ;
  end process pwmCounterEdge ;
  --licznik programowalny - zwiekszenie zawartosci
  progCounterINC: process (INCprim) is begin
    --wykonanie polecenia
    if falling_edge(INCprim) and (temp < 9) then
      counter_up <= temp + 1;
    end if ;
  end process progCounterINC;
  --licznik programowalny - zwiekszenie zawartosci
  progCounterDEC: process (DECprim) is begin
    --wykonanie polecenia
    if falling_edge(DECprim) and (temp > 0) then
      counter_down <= temp - 1;
    end if ;
  end process progCounterDEC;
  --przypisania
  temp <=
    counter_up   when (INCprim = '0') else
    counter_down when (DECprim = '0') else
    „0000" when temp>9 ;
  --przypisanie do wektora x
  x <= std_logic_vector(temp) ;
end pwm_moduleBehav;
```

Listing 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dekoder7segm is
  port(
      wej : in std_logic_vector(3 downto 0) ;
    wysw : out std_logic_vector(6 downto 0)
  );
end dekoder7segm;
architecture dekoder7segmBehav of dekoder7segm is
begin
  --wyswietlenie prawdidlowej wartosci na wyswietlaczu
  wysw <=
    "1000000" when (wej = "0000") else
    "1111001" when (wej = "0001") else
    "0100100" when (wej = "0010") else
    "0110000" when (wej = "0011") else
    "0011001" when (wej = "0100") else
    "0010010" when (wej = "0101") else
    "0000010" when (wej = "0110") else
    "1111000" when (wej = "0111") else
    "0000000" when (wej = "1000") else
    "0010000" when (wej = "1001") else
    "1111111";
end dekoder7segmBehav;
```

if sprawdzającą, czy nie pojawiła się liczba 10. W naszym liczniku jest ona niedozwolona, więc jest od razu zamieniana na 0. Z procesem została zintegrowana funkcja komparatora porównującego zmienne *temp* oraz *pwmCounter*. Jak widać na listingu 1, komparator zrealizowano za pomocą prostego operatora „>" i im zmienna *temp* ma większą wartość, tym dłużej wyjście *pwm* ma stan wysoki. Jest to proste do wytłumaczenia na przykładzie. Gdy *temp=2*, to warunek:

if pwmCounter < temp then

jest spełniony dla *pwmCounter* równego 0 oraz 1. Gdyby zwiększyć wartość temp do 5, to powyższy warunek spełnia znacznie większa liczba stanów licznika *pwmCounter*: 0, 1, 2, 3, 4. W ten oto sposób wartość zmiennej *temp* decyduje o wypełnieniu sygnału wyjściowego.

Pozostała do wyjaśnienia kwestia, w jaki sposób powiązać ze sobą zmienne *temp*, *counter_up* oraz *counter_down*. Skoro zdecydowaliśmy, że zmienna *temp* przechowuje aktualną wartość wypełnienia PWM, to wszelkie modyfikacje muszą być wykonywane względem niej. Zatem liczniki *counter_up* i *counter_down* są zwiększane/zmniejszane o jeden względem *temp*, co realizują procesy *progCounterINC* oraz *progCounterDEC*. Powstaje jednak pewna niespójność po wykonaniu takiego procesu, bo nowa, aktualna wartość znajduje się teraz w zmiennej *counter_up* lub *counter_down*. Konieczne są więc kolejne instrukcje, które zlikwidują ten problem, konkretnie:

--przypisania
temp <=
counter_up when (INCprim = '0') else
counter_down when (DECprim = '0') else
„0000" when temp>9 ;

Do zmiennej *temp* podstawimy wartość zaktualizowanego licznika, czyli *counter_up* bądź *counter_down*. Określenie, który z nich zmienił swój stan, jest bardzo proste – wystarczy sprawdzić, który z przycisków ma stan niski (jest naciśnięty), co zresztą wykorzystano w powyższych instrukcjach.

Ostatnim etapem jest zapisanie do wektora *x* aktualnej wartości PWM, co nie stanowi problemu i wymaga jedynie przepisania wartości ze zmiennej *temp*.

Zanim przystąpimy do sprawdzenia, jak nasz generator PWM pracuje, opracujmy jeszcze dekoder 7-segmentowy. Dane z układu PWM są wyprowadzane przez wektor *x*, który zostanie automatycznie zamieniony na magistralę. Dla wygody powinniśmy opracować dekoder w taki sposób, aby również miał na wejściu magistralę. Ułatwi to znacznie podłączenie, gdyż unikniemy wstawiania elementów Bus Tap. W zasadzie całość sprowadza się tylko do zadeklarowania wektora w jednostce *entity* zamiast czterech pojedynczych sygnałów. Właściwy kod takiego dekodera jest widoczny na **listingu 2**. Sądzę, że jest on na tyle prosty, że nie wymaga szerszego komentarza.

Zatem mamy już dwa pliki *.vhdl z kodem, który chcielibyśmy zamienić na elementy biblioteczne, możliwe do wstawienia w edytorze schematów. Czeka na nas miłe zaskoczenie, gdyż proces ten jest banalny – wystarczy rzut oka na **rysunek 3**, żeby zrozumieć o co chodzi. Przypomnę tylko krótko, że tworzenie nowego elementu polega na zaznaczeniu pliku zawierającego opis tego elementu (w VHDL-u lub schemat) i kliknięciu pozycji *Create Schematic Symbol*, czyli identycznie jak robiliśmy w poprzednich częściach.

Pozostaje dodanie do projektu nowego schematu (w oknie Project Navigator wybieramy Project->New source) i pobranie z biblioteki utworzonych poprzednio elementów: modułu PWM oraz dekodera. Stosowne połączenie widoczne jest na **rysunku 4**. W zasadzie doprowadziliśmy tylko sygnały przycisków, sygnał zegarowy i podłączyliśmy dekoder. Do magistrali wyjściowej dekodera podłączony jest port wyjściowy. Przypominam o konieczności użycia narzędzia *Assign Package Pin*, które wybieramy po zaznaczeniu schematu, a nie pliku z kodem VHDL (**rysunek 5**). Trzymamy się konwencji, że *wysw<0>* podłączony jest do segment *a*, itd.

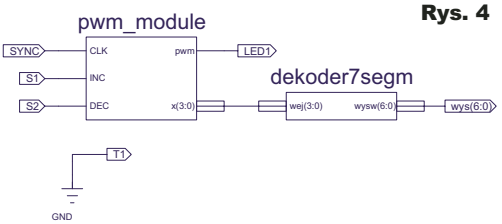
Po skompilowaniu i załadowaniu programu do układu CPLD będziemy mieli możliwość regulowania jasności świecenia diody LED. Zmiany można wprowadzać przyciskami S1 oraz S2. Warto zwrócić uwagę, że dioda świeci najjaśniej, kiedy na wyświetlaczu jest cyfra 0. Wynika to z faktu, że jest ona włączana zerem logicznym. Chcąc sprawić, aby dioda świeciła najmocniej przy cyfrze 9, należy wstawić negator.

Pomiar częstotliwości

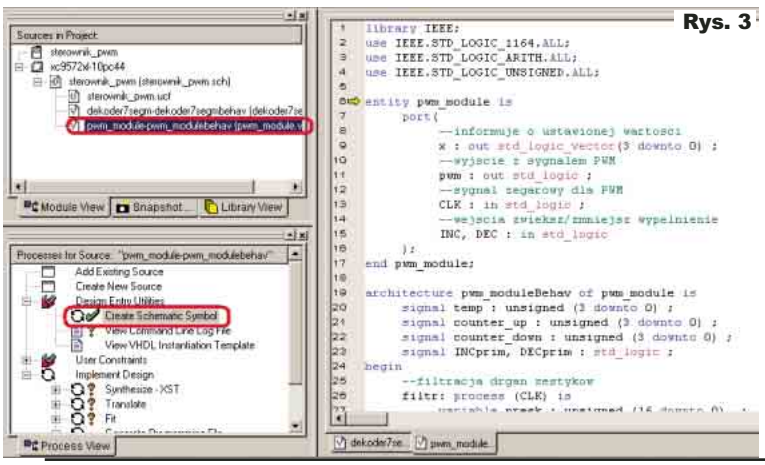
Nieprzypadkowo na początku niniejszego artykułu przybliżyłem budowę miernika częstotliwości. Obecnie zaimplementujemy takie właśnie rozwiązanie. Nasz prosty miernik będzie mierzył częstotliwości w zakresie od 0 do 999Hz. Wykorzystamy go do określenia, jaka jest częstotliwość generatora U7 (sterowanego fotorezystorem). Uzyskane w ten sposób informacje będą pomocne podczas budowy sterownika oświetlenia. Zasadniczo schemat blokowy z rysunku 1 nie wymaga zmian i może być bezpośrednio zaimplementowany. Jednakże z jednym wyjątkiem – nie będzie nam potrzebny układ formowania wejścia, gdyż sygnał pochodzący z układu NE555 jest prostokątny.

Do wyświetlania liczb trzycyfrowych potrzebne będą trzy wyświetlacze 7-segmentowe. Na płytce prototypowej niestety są tylko dwa. Obejdziemy ten problem, wykorzystując diody LED – cyfra setek zostanie wyświetlona w naturalnym kodzie binarnym. W ramach ćwiczenia opiszemy w VHDL-u również liczniki modulo 10, mimo że moglibyśmy wykorzystać elementy z biblioteki edytora schematów.

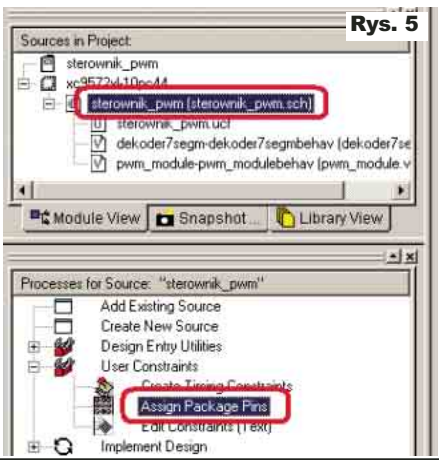
Rys. 4



Rys. 3



Rys. 5



Zacznijmy od tych właśnie liczników. Do zliczania wykorzystamy procesy zawierający na liście czułości dwa sygnały: *reset* oraz *inCLK*. Pierwszy z nich jest potrzebny do zerowania licznika. Znaczenie drugiego sygnału jest również intuicyjne – trzeba gdzieś doprowadzić zliczany sygnał. Potrzebna będzie jeszcze zmienna do pamiętania kolejnych stanów (*temp*). W zasadzie potrzebujemy przechowywać tylko 10 stanów, ale najbliższym odpowiednikiem jest liczba 16 (4-bitowa). W związku z tym musimy dodać instrukcję warunkową *if* do sprawdzania, czy nie nastąpiło przepełnienie naszego licznika (czy nie pojawił się niedozwolony stan 10). W takiej sytuacji niezbędne będzie wyzerowanie zmiennej. Pojawia się jeszcze jeden problem – trzeba wygenerować sygnał wyjściowy, który będzie miał częstotliwość dziesięciokrotnie mniejszą niż sygnał wejściowy. Można to zrobić w sposób podobny do tego, który stosowaliśmy wcześniej, tzn. przypisać do wyjścia najstarszy bit zmiennej. Postąpimy jednak inaczej – dodamy dodatkowy warunek do istniejącej już instrukcji warunkowej *if,* konkretnie dla przypadku, gdy zmienna będzie równa pięć, aby wtedy do wyjścia przypisać zero. Jedynkę na wyjściu umieścimy, gdy *temp* będzie równe dziesięć (jednocześnie zerując *temp*). Otrzymamy tym samym sygnał o dziesięciokrotnie mniejszej częstotliwości, ponieważ okres tego przebiegu wymaga dziesięciu zliczeń sygnału wejściowego. Zbocze narastające pojawi się w momencie przepełnienia licznika, a nie przy zmianie wartości z siedmiu do ośmiu, co miałoby miejsce podczas kopiowania najstarszego bitu na wyjście.

Listing 3

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity modulo10 is
  Port ( inCLK : in std_logic;
    reset : in std_logic;
    outCLK : out std_logic;
    x : out std_logic_vector (3 downto 0)
  );
end modulo10;
architecture modulo10Behavioral of modulo10 is
begin
  --zliczanie i resetowanie licznika
  zliczanie: process (inCLK, reset) is
    variable temp : unsigned (3 downto 0);
  begin
    if reset = '1' then
      temp := "0000" ;
    else
      if rising_edge(inCLK) then
        temp := temp + 1;
        if temp = 5 then
          outCLK <= '0' ;
        elsif temp = 10 then
          outCLK <= '1' ;
          temp := "0000";
        end if;
      end if ;
    end if ;
    --wystawienie sygnalow wyjsciowych
    x <= std_logic_vector(temp) ;
  end process zliczanie;
end modulo10Behavioral;
```

Musimy mieć wgląd w stan licznika, aby określić, jaką wartość on zliczył. Jest to potrzebne do określenia częstotliwość sygnału. Sprowadza się to do wykonania jeszcze jednego przypisania, mianowicie do wektora *x* przypiszemy aktualnie zliczoną wartość znajdującą się w zmiennej *temp*. Kod w VHDL-u realizujący funkcję licznika modulo 10 pokazano na **listingu 3**.

Drugim niezbędnym blokiem jest układ sterujący. Przede wszystkim potrzebujemy na jego wyjściu jedynki logicznej trwającej dokładnie jedną sekundę. Źródłem stabilnego przebiegu zegarowego jest oczywiście generator kwarcowy o częstotliwości 24MHz. W ciągu jednej sekundy dostarcza on 24 mln impulsów zegarowych. Wynika z tego, że konieczne jest podzielenie tej częstotliwości przez 24 mln, aby otrzymać sygnał trwający jedną sekundę. Wykorzystamy do tego zmienną 25-bitową, aby „zmieścić" tak dużą liczbę. Do zliczania utworzymy proces z sygnałem *SYNC* umieszczonym na liście czułości. Przy każdym zboczu narastającym będziemy zwiększać o jeden zawartość zmiennej *licznik* i zerować ją po osiągnięciu wartości 24000000, co utworzy stosowny dzielnik. Podczas zerowania zanegujemy sygnał *bramka*, aby otrzymać przebieg bramkujący.

Listing 4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity uklad_sterujacy is
  Port ( SYNC : in std_logic;
    bramka : inout std_logic;
    zatrzask : out std_logic;
    reset : out std_logic);
end uklad_sterujacy;
architecture uklad_sterujacyBehavioral of uklad_sterujacy is begin
  sterownik: process (SYNC) is
    variable licznik : unsigned (24 downto 0) ;
  begin
    --generacja impulsu 1 sek
    if rising_edge(SYNC) then
      --zwiekszenie zawartosci licznika
      licznik := licznik + 1 ;
      --wysterowanie syganlow
      if (licznik=1000000) then
        zatrzask <= '1' and (not bramka);
      elsif (licznik=1500000) then
        zatrzask <= '0' ;
      elsif (licznik=2000000) then
        reset <= '1' and (not bramka);
      elsif (licznik=2500000) then
        reset <= '0' ;
      end if;
      --odliczanie dlugosci impulsu
      if licznik = 24000000 then
        licznik := "000000000000000000000000" ;
        bramka <= not bramka;
      end if ;
    end if;
  end process sterownik;
  --obsluga zakresu
end uklad_sterujacyBehavioral;
```

Warto zauważyć, że w tym wypadku *bramka* będzie portem typu *inout*, co pozwala zarówno na odczyt jak i zapis. Gdybyśmy wybrali typ *out*, to operacja:

bramka <= not bramka

nie byłaby możliwa, gdyż sygnał *bramka* byłby wyjściem i nie można by go odczytywać. Ktoś mógłby powiedzieć, że przecież nie chcemy go odczytywać, tylko zapisywać! No niby tak, ale aby zapisać wartość zanegowaną, musimy odczytać najpierw obecną wartość i dopiero wtedy możemy ją zanegować i zapisać.

Pozostaje jeszcze wygenerowanie sygnałów sterujących: *zatrzask* (określającego moment pobrania danych z liczników) oraz *reset* (wymuszającego wyzerowanie liczników). Można od razu zauważyć, że w czasie zliczania (*bramka = 1*) oba sygnały są nieaktywne - mają wartość zero. Natomiast aktywne stają się w momencie, gdy liczniki nie pracują (*bramka = 0*). Trzeba jeszcze mieć na uwadze, że najpierw pojawia się impuls na wyjściu *zatrzask*, a dopiero później na wyjściu *reset*. Gdybyśmy zrobili to odwrotnie, to wyświetlacz zawsze wskazywałby zero, gdyż liczniki byłyby najpierw resetowane, a dopiero później odczytywane. Taką organizację impulsów możemy uzyskać poprzez zastosowanie bloku instrukcji *if... elsif... end if*. Zmienna *licznik* sukcesywnie zwiększa się w takt sygnału zegarowego z wejścia *SYNC*, więc możemy za pomocą instrukcji *if* określić momenty wystawienia jedynek i zer na sygnałach *zatrzask* oraz *reset*. Instrukcje odpowiedzialne za wystawienie jedynek logicznych są minimalnie bardziej skomplikowane, gdyż ich wykonanie trzeba uzależnić od stanu wyjścia bramkującego. W tym miejscu wykorzystamy prostą operację *and*, która sprawi, że jedynka zostanie wystawiona tylko wtedy, gdy liczniki modulo 10 nie liczą. Okazuje się, że poziomem aktywnym sygnału *bramka* jest jedynka, więc potrzebna jest jeszcze negacja za pomocą instrukcji *not*. Gotowy kod sterownika pokazano na **listingu 4**.

Do opracowania pozostał jeszcze dekoder. Zasadniczo jego rola została już omówiona wcześniej, więc skupimy się na konkretnych wymaganiach. Zacznijmy od zdefiniowania potrzebnych sygnałów. Po pierwsze, potrzebne są wejścia do wprowadzania wartości pochodzących z liczników modulo 10, więc w programie do tego celu utworzymy trzy wektory 4-bitowe: *s*, *d*, *j*. Niezbędne jest także wejście *zatrzask*, które będzie podłączone do korespondencyjnego wyjścia w sterowniku. Dzięki niemu będziemy mieli informację, kiedy należy pobrać dane z wyświetlaczy, a kiedy tego nie robić. Skoro jesteśmy przy wyświetlaczach, to dodamy od razu wyjścia *T1*, *T2* do sterowania tranzystorami, co umożliwi nam sterowanie multipleksowane. Przygotujemy także wektor *wysw*, aby umożliwić sobie sterowanie poszczególnymi segmentami. Układ do pomiaru częstotliwości pochłonął większość dostępnych zasobów układu CPLD, więc nie udało się zaimplementować dzielnika częstotliwości do przełączania wyświetlaczy

Listing 5

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dekoder is
  Port ( s : in std_logic_vector (3 downto 0);
    d : in std_logic_vector (3 downto 0);
    j : in std_logic_vector (3 downto 0);
    zatrzask : in std_logic;
    sel : in std_logic  ;
    T1 : out std_logic  ;
    T2 : out std_logic  ;
    wysw : out std_logic_vector (6 downto 0);
    led : out std_logic_vector (3 downto 0));
end dekoder;
architecture dekoderBehavioral of dekoder is
  signal setki, dzies, jedn  : std_logic_vector(3 downto 0);
begin
  --zarzaskiwanie informacji
  setki <= s when zatrzask='1' ;
  dzies <= d when zatrzask='1' ;
  jedn  <= j when zatrzask='1' ;
  --obsluga tranzystorow
  T1 <= '0' when sel='0' else '1' ;
  T2 <= '0' when sel='1' else '1' ;
  --wyswietlenie prawdidlowej wartosci na wyswietlaczu
  led <= not setki  ;
  wysw <=
    --wyswietlenie jednosci
    "1000000" when (jedn = "0000") and (sel='0') else
    "1111001" when (jedn = "0001") and (sel='0') else
    "0100100" when (jedn = "0010") and (sel='0') else
    "0110000" when (jedn = "0011") and (sel='0') else
    "0011001" when (jedn = "0100") and (sel='0') else
    "0010010" when (jedn = "0101") and (sel='0') else
    "0000010" when (jedn = "0110") and (sel='0') else
    "1111000" when (jedn = "0111") and (sel='0') else
    "0000000" when (jedn = "1000") and (sel='0') else
    "0010000" when (jedn = "1001") and (sel='0') else
    --wyswietlenie dziesiatek
    "1000000" when (dzies = "0000") and (sel='1') else
    "1111001" when (dzies = "0001") and (sel='1') else
    "0100100" when (dzies = "0010") and (sel='1') else
    "0110000" when (dzies = "0011") and (sel='1') else
    "0011001" when (dzies = "0100") and (sel='1') else
    "0010010" when (dzies = "0101") and (sel='1') else
    "0000010" when (dzies = "0110") and (sel='1') else
    "1111000" when (dzies = "0111") and (sel='1') else
    "0000000" when (dzies = "1000") and (sel='1') else
    "0010000" when (dzies = "1001") and (sel='1') else
    "XXXXXXX"  ;
end dekoderBehavioral;
```

wymaganego w sterowaniu multipleksowanym. Na szczęście problem ten można obejść, dodając specjalnie do tego celu sygnał *sel*, do którego przyłączymy zewnętrzny generator. Ma to jeszcze jedną dużą zaletę, w postaci interesującego eksperymentu, ale o tym za moment. Pozostał ostatni wektor *led*, do którego podłączymy diody LED zastępujące trzeci, nieobecny na płytce wyświetlacz 7-segmentowy. W bloku opisu architektury należy dodać jeszcze trzy inne wektory – *setki*, *dzies*, *jedn*, które będą stanowiły naszą pamięć (zatrzask). W momencie, gdy sygnał *zatrzask* przyjmie wartość jeden, skopiujemy tam wyjścia liczników i w ten sposób zapamiętamy ich stan. Kiedy układ sterujący wymusi nowe zliczanie, będzie można dzięki temu pokazywać ostatni pomiar. Kod realizujący tę operację jest zadziwiająco prosty:

```
--zatrzaskiwanie informacji
setki <= s when zatrzask='1' ;
dzies <= d when zatrzask='1' ;
jedn <= j when zatrzask='1';
```

Omówiona w poprzedniej części kursu struktura *when* wymusza podstawienie do „pamięci" stanu wejść, gdy sygnał *zatrzask* przyjmuje wartość jeden.

Ta sama struktura rozwiązuje problem sterowania tranzystorami:

```
--obsluga tranzystorow
T1 <= '0' when sel='0' else '1';
T2 <= '0' when sel='1' else '1';
```

W zależności od stanu wejścia *sel* tranzystory *T1* oraz *T2* przyjmują określoną wartość. Całość można zrealizować w inny, być może prostszy sposób:

```
T1 <= sel ;
T2 <= not sel ;
```

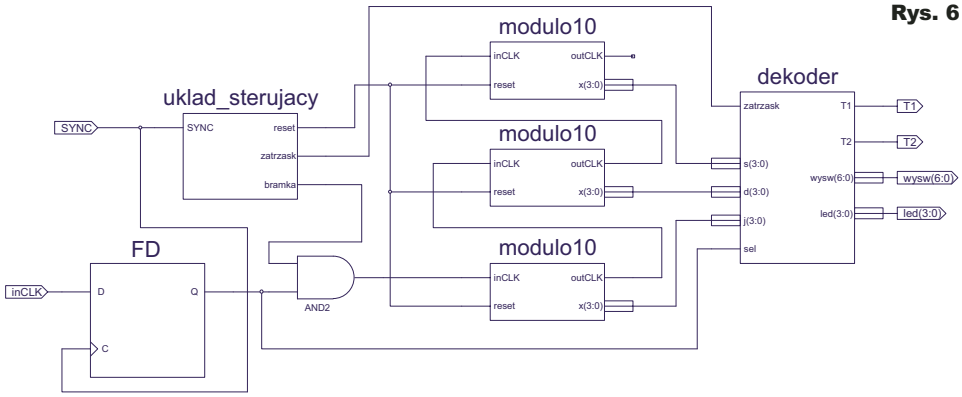
Do wysterowania segmentów wyświetlaczy wykorzystać można strukturę *when* w sposób podobny do omówionego już w poprzedniej części kursu. Przypomnę tylko, że wyrażenie warunkowe może być uzależnione od tego jaką cyfrę zamierzamy wyświetlić i na którym wyświetlaczu (o czym decyduje sygnał *sel*).

Ostatnim zadaniem stawianym przed dekoderem jest obsługa diod LED. Jedyna trudność polega na tym, że dla wyjść licznika modulo 10 sygnałem aktywnym są jedynki logiczne, a dla diod LED – zera. Rozwiązanie nie jest skomplikowane – wymaga negacji za pomocą instrukcji *not*. Właściwe przypisanie będzie miało postać:

```
led <= not setki ;
```

Pełny kod źródłowy tego modułu jest widoczny na **listingu 5**.

Rys. 6



Ostatnim etapem jest utworzenie symboli dostępnych w bibliotece (poprzez kliknięcie *Create Schematic Symbol*) i połączenie wszystkich elementów w jedno urządzenie. Przy oparciu się na schemacie blokowym z rysunku 1 nie powinno to sprawić problemu. Dodajemy do projektu nowy plik źródłowy – schemat. W bibliotece powinny pojawić się utworzone elementy, które możemy połączyć podobnie jak na **rysunku 6**. Osobnego komentarza może wymagać marker *inCLK*, do niego doprowadzimy sygnał, którego częstotliwość chcemy zmierzyć. Może to być *CLK* (przestrajany generator) lub *CLK_FR* (generator przestrajany fotorezystorem). Przerzutnik D obecny na schemacie usuwa zakłócenia z tych sygnałów. Warto zwrócić uwagę na nietypową rzecz, mianowicie do wejścia *sel* dekodera doprowadzony jest mierzony sygnał! Jeżeli ta częstotliwość będzie zbyt niska, wyświetlacz będzie zauważalnie migotał. W naszym przykładowym projekcie ma to jednak dwie zalety: oszczędza zasoby logiczne CPLD oraz pozwala samodzielnie zbadać, jak częstotliwość odświeżania wpływa na jakość pracy wyświetlacza. Kręcąc potencjometrem lub zasłaniając fotorezystor ręką (lub oświetlając lampką), można samodzielnie określić częstotliwość, dla jakiej wskazanie wyświetlacza jest stabilne i „pozbawione" migotania. Innymi słowy: jaka musi być częstotliwość odświeżania, aby ludzkie oko dało się oszukać i „uwierzyło", że obraz jest stabilny. Czy jest to rzeczywiście 30Hz? A może wystarczy mniej? A może nie? Na to pytanie już nie odpowiem, pozostawiając Czytelnikom samodzielne wyznaczenie optymalnej częstotliwości odświeżania „ekranu". Mam nadzieję, że ćwiczenie to okaże się podwójnie interesujące i kształcące.

Podsumowanie

W tym odcinku planowany był jeszcze odbiornik RC5, jednakże brak miejsca sprawił, że zostanie on zaprezentowany za miesiąc, razem z pierwszymi informacjami o automatach.

Na koniec chciałbym zaproponować Czytelnikom modyfikację miernika częstot-liwości w taki sposób, aby mierzył znacznie większe wartości, np. w zakresie 0..999kHz. Jego działanie można wtedy sprawdzić poprzez dołączenie sygnału zegarowego z generatora kwarcowego po zastosowaniu jakiegokolwiek preskalera. Pozwoli to sprawdzić, czy wyliczona częstotliwość zgadza się z tą zmierzoną, co umożliwi łatwe skontrolowanie poprawności projektu. Głównym problemem jest określenie, JAK zmodyfikować układ, aby mierzył częstotliwości w podanym zakresie, ale to już pozostawiam jako zadanie dla Czytelników.

Osoby czujące się na siłach mogą również pokusić się o zbudowanie układu mierzącego okres sygnału. Pomiary tego typu wykonuje się w przypadku „powolnych" sygnałów, gdyż dają one bardziej dokładne wyniki. Układ taki powinien wtedy wyświetlać np. w milisekundach, ile trwa okres danego przebiegu.

Jakub Borzdyński
jakub.borzdynski@elportal.pl