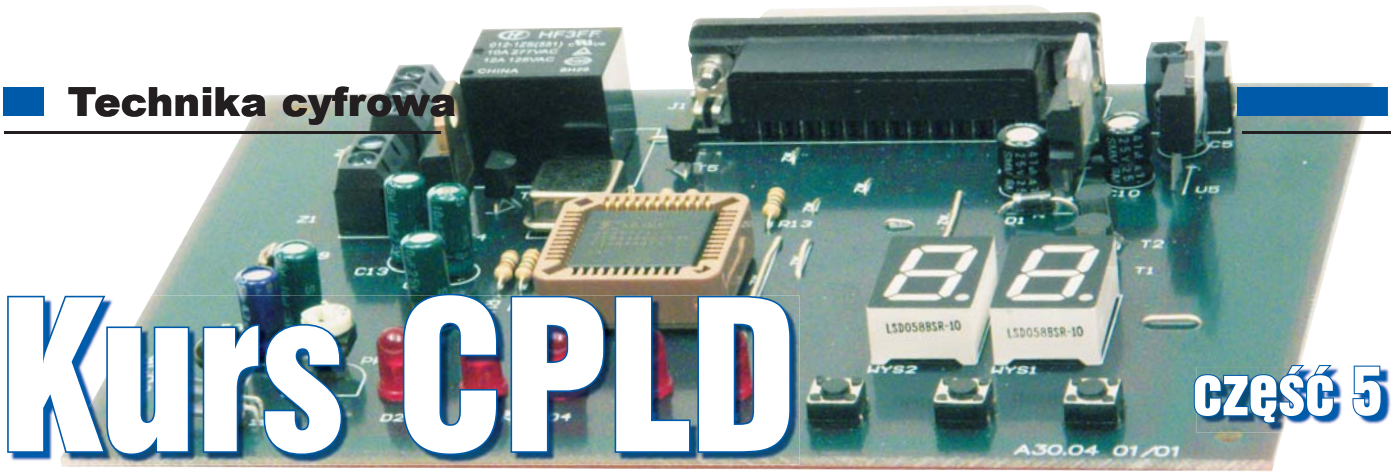


Kurs CPLD

część 5

Trochę przynudzania...

Dzisiejszy odcinek poświęcony jest językowi VHDL. Zacznijmy od rozwinięcia tego tajemniczego skrótu. Oznacza on VHSIC Hardware Description Language, natomiast skrót VHSIC to Very High Speed Integrated Circuits. Całość można przetłumaczyć jako: język opisu sprzętu (dla) bardzo szybkich układów scalonych. Pragnę uświadomić Czytelników, że VHDL NIE JEST językiem programowania, tylko jak wskazuje jego nazwa, językiem opisu sprzętu. Instrukcje klasycznego języka programowania są wykonywane sekwencyjnie, tzn. jedna po drugiej. W VHDL-u wygląda to inaczej, mianowicie instrukcje wykonywane są równolegle, przykładowo polecenia odnoszące się do bramek logicznych są wykonywane natychmiast po pojawieniu się sygnału na ich wejściu. Naśladuje to zachowanie fizycznego sprzętu, podobnie do tego, co pojawiło się w poprzednich częściach kursu – po wstawieniu do projektu kilku bramek ich stan zmieniał się w momencie zmiany stanu wejść. Stan kilku bramek mógł zmienić się w tym samym czasie.

Przyjrzymy się obecnie modelowaniu behawioralnemu, co oznacza, że będziemy opisywać, jak dany układ ma się zachować (co ma robić). Sprowadza się to do ustalenia relacji pomiędzy wejściami układu a jego wyjściami.

Na omówienie języka VHDL poświęcimy tylko tę i następną część kursu, więc będzie to na pewno skrót najważniejszych wiadomości. Niektóre kwestie zostaną z braku miejsca przemilczane i pozostawione Czytelnikom do doczytania w Internecie. Mam nadzieję, że przedstawione tu podstawowe informacje będą dobrym punktem wyjścia do dalszych eksperymentów i dociekań, gdyby komuś ten sposób projektowania sprzętu przypadł do gustu. Płytka kursu CPLD powinna być dobrą platformą do prowadzenia własnych prób.

Podstawy języka VHDL.

Jak wspomniano przed chwilą, nie mamy do czynienia z językiem programowania, jednakże język opisu sprzętu również ma zastrzeżone słowa kluczowe i struktury, których należy używać, aby osiągnąć założony cel. Warto wspomnieć w tym miejscu, że wielkość liter w słowach kluczowych i nazwach nie ma znaczenia.

Podczas tworzenia wszelkiego rodzaju oprogramowania, do dobrej praktyki należy pisanie komentarzy, aby potem łatwiej było rozeznać się jak dany fragment kodu pracuje. Język VHDL również daje taką możliwość ignorując wszystko, co znajduje się za znakami „--" (dwa myślniki).

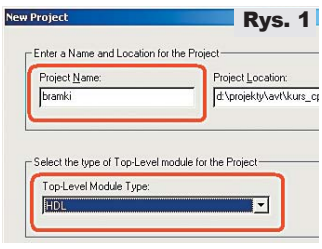
Utwórzmy zatem nowy projekt, w którym będziemy mogli zapisać pierwszy program. Zasadniczo proces ten jest bardzo zbliżony do tego, co poznaliśmy w pierwszej części kursu. Wybieramy *File->New Project* i w otwartym okienku wpisujemy nazwę projektu, pozostawiając *Top-Level Module Type* z domyślną opcją HDL (**rysunek 1**). Po kliknięciu *Dalej* ukazuje się znane już okienko, w którym określamy rodzaj układu programowalnego, w kolejnym oknie klikamy *New Source*, aby dodać nowy plik źródłowy i okienko wypełniamy jak na **rysunku 2** (nazwa oczywiście jest dowolna). Po kliknięciu *Dalej* ukazuje się zupełnie nowe okno – **rysunek 3**. Kreator umożliwia nam łatwe zdefiniowanie wejść i wyjść projektowanego układu. Jak już zapewne część Czytelników się domyśliła, patrząc na rysunek 3, pierwszym projektem będzie bardzo prosta bramka AND, aby łatwiej było się skupić na poznawaniu języka VHDL, a nie zastanawianiu, jak układ funkcjonuje. Wróćmy jednak do kreatora – potrzebujemy dwóch wejść i jednego wyjścia. W kolumnie *Port Name* wpisujemy ich nazwy (UWAGA!!! Nie mogą to być słowa kluczowe!!! Wszystkie zarezerwowane słowa pokazano na **rysunku 4**. Nie wolno ich używać jako nazw, gdyż spotka się to z błędem kompilacji ,), a w sąsiedniej kolumnie *Direction* określamy, co jest wejściem, a co wyjściem. Znaczenie pola *Entity Name* oraz *Architecture Name* wyjaśnię za chwilę, w każdym razie może to być dowolna nazwa niebędąca słowem kluczowym. Przykład wypełnienia tego okienka można zobaczyć na wspomnianym już rysunku 3. Po kliknięciu dalej wyświetli się podsumowanie, które zamykamy przyciskiem *Zakończ*. Później klikamy dwa razy *Dalej* i na końcu *Zakończ*.

Efektem naszych dotychczasowych działań jest plik z kodem źródłowym (**rysunek 5**) otwarty w oknie programu *Project Navigator*. Przeanalizujmy, co się tam znajduje. Pierwszy fragment kodu, oznaczony cyfrą jeden, odpowiada za włączenie standardowych bibliotek do naszego programu. Druga część kodu (cyfra dwa) jest przykładem komentarza, który dołączył kreator. Widać przy okazji, że edytor zapewnia kolorowanie składni i komentarze mają kolor zielony. Następnie znajdujemy blok kodu zaczynający się od słowa ***entity***, co oznacza wyodrębnioną jednostkę. Ze słowem ***entity*** stowarzyszone jest drugie słowo kluczowe, jakim jest ***is***, a całość kończy się słowem ***end***. Zatem struktura takiego bloku ma postać:

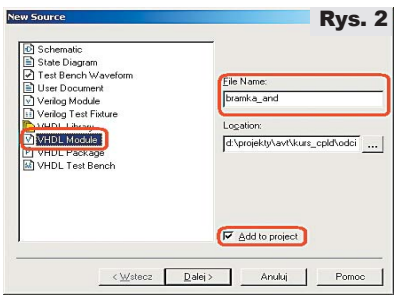
entity NASZA_NAZWA ***is***
(...)
end (NASZA_NAZWA) ; --średnik na końcu!

Zwracam uwagę, że druga nazwa jednostki (ujęta w nawiasy) jest opcjonalna. Zatem wiemy, jak ma wyglądać struktura ***entity***, ale nie wiemy jeszcze, czym ona jest. Można przyjąć, że blok ***entity*** stanowi taką czarną skrzynkę, deklarujemy, że chcemy mieć coś, co nazywa się NASZA_NAZWA i jest blokiem funkcjonalnym w przygotowywanym właśnie projekcie. W naszym przykładowym kodzie widocznym na rysunku 5 (przy cyfrze trzy) jednostka ta nazywa się *bramka_and*, co daje do zrozumienia kompilatorowi, że tworzymy nowy element o takiej nazwie. Wewnątrz bloku ***entity*** znajduje się kolejne słówko kluczowe, jakim jest ***port***. Jego rola sprowadza się do określenia, jakie wejścia i wyjścia będzie miał nasz element. Składnia tego polecenia nie jest przesadnie skomplikowana i sprowadza się do zapisania słowa klu-

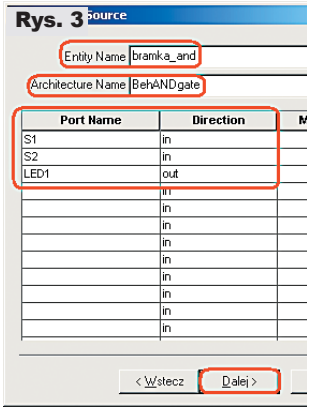
Rys. 1



Rys. 2



Rys. 3



Rys. 4 Zarezerwowane słowa języka VHDL

abs, access, after, alias, all, and, architecture, array, assert, attribute, begin, block, body, buffer, bus, case, component, configuration, contant, disconnect, downto, else, elsif, end, entily, exit, file, for, function, generate, generic, group, guarded, if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod, nand, new, next, nor, not, null, of, on, open, or, others, out, package, port, postponed, procedure, process, pure, range, record, register, reject, return, rol, ror, select, severity, signal, shared, sla, sli, sra, srl, subtype, then, to, transport, type, unaffected, units, until, use, variable, wait, when, while, with, xnor, xor.

czowego *port* pomiędzy słowami kluczowymi *is* oraz *end*. Za słowem *port* najlepiej od razu otworzyć i zamknąć okrągły nawias i postawić na końcu średnik, co ustrzeże nas przed zapomnieniem o tym. Zatem zapisujemy:

port() ;

Wewnątrz nawiasu zdefiniujemy wszystkie porty, jakie ma posiadać tworzony przez nas element według wzoru:

nazwa: *tryb typ ;*

Należy zwrócić uwagę, że po nazwie występuje dwukropek. UWAGA! Ostatni wpis nie kończy się już średnikiem, gdyż ten znajduje się za nawiasem. Można zapisywać sygnały również w skróconej formie:

nazwa1, nazwa2, ... , nazwaN: *tryb typ*;

Parametr *nazwa* określa nazwę sygnału wejściowego lub wyjściowego, który w niektórych wypadkach może być tożsamy z markerem I/O poznanym w poprzednich odcinkach kursu.

Za parametr *tryb* podstawiamy jedno ze słów kluczowych:

– *in* oznaczające, że dany sygnał jest wejściem
– *out* oznaczające, że dany sygnał jest wyjściem
– *inout* oznaczające, że sygnał jest dwukierunkowy
– *buffer* oznaczające, że sygnał jest wyjściem bloku *entity*, którego wartość może być odczytywana wewnątrz architektury tej jednostki.

Natomiast parametr *typ* zastępujemy własnym lub wbudowanym typem określającym jakiego rodzaju ma być sygnał. Dostępne są następujące opcje:

– *bit* – może przyjmować wartość 0 lub 1
– *bit_vector* – jest to wektor (tablica jednowymiarowa) bitów
– *boolean* – może przyjmować wartość TRUE (prawda) lub FALSE (fałsz)
– *integer* – może przyjmować wartości całkowite
– *real* – może przyjmować wartości rzeczywiste (ułamki)
– *character* – reprezentuje pojedynczy znak ASCII
– *time* – reprezentuje czas
– *std_logic*, *std_ulogic*, *std_logic_vector*, *std_ulogic_vector* – typy te mogą przyjmować 9 wartości reprezentujących wartość oraz poziom sygnału (np. poziom wysoki, niski, nieokreślony, słaby poziom niski, etc.). Literatura zaleca używanie typ *std_logic* zamiast *bit*, o czym można się przekonać ponownie, patrząc na rysunek 5, gdyż właśnie ten typ zastosował kreator.

Podsumujmy: stworzyliśmy jednostkę (blok funkcjonalny), który nazwaliśmy *bramka_and* i zdefiniowaliśmy, że posiada on dwa sygnały wejściowe (*S1* oraz S2) i jeden wyjściowy (*LED1*).

Jest to jednak za mało, aby mógł on pracować zgodnie z naszymi oczekiwaniami. Konieczne jest jeszcze opisanie, jak ma działać taka jednostka, czyli należy przedstawić jej „zachowanie" za pomocą języka opisu sprzętu (Hardware Description Language). Dokonuje się tego za pomocą następującej struktury:

architecture NAZWA_ARCHITEKTURY ***of*** NAZWA_OPISYWANEJ_JEDNOSTKI ***is***
--deklaracje
-- komponentów
-- sygnałów
-- stałych
-- funkcji
-- procedur
-- typów
begin
-- instrukcje...
end (NAZWA_ARCHITEKTURY)***;*** *--tu też średnik na końcu.*

Blok deklaracji jest opcjonalny i chwilowo go pominiemy. Opis architektury rozpoczyna się słowem kluczowym ***architecture***, po którym występuje dowolna nazwa (ale nie słowa kluczowe języka VHDL), niech to będzie *BehANDgate*. Ten parametr został wprowadzony w kreatorze i został uwzględniony w szablonie (rysunek 5, kod oznaczony cyfrą cztery). Następnie wpisujemy słowo kluczowe *of*, za którym podajemy jednostkę, której sposób pracy zamierzamy właśnie opisać. W tym momencie nazwa ta (NAZWA_OPISYWANEJ_JEDNOSTKI) jest już ściśle określona i musi to być *bramka_and*, gdyż tak nazwaliśmy nasz blok *entity* i zamierzamy właśnie ten blok opisać. Następnie zapisujemy słowo kluczowe *begin*, za którym umieszczamy opis sprzętu i całość kończymy słowem *end* ze średnikiem na końcu.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
entity nand3_ver1 is
    Port ( a, b, c : in std_logic;
           y : out std_logic);
end;
architecture NAND3_behav of nand3_ver1 is
begin
    y <= not (a and b and c) ;
end;
```

Listing 1

Teraz przyszła pora określić, co umieścić pomiędzy słowami *begin* oraz *end*. Możemy w tym miejscu wykorzystać zdefiniowane uprzednio sygnały *S1, S2* oraz *LED*, gdyż zadeklarowaliśmy pracę z jednostką *bramka_and*. Właściwy wpis powinien wyglądać następująco:

LED1 <= S1 ***and*** S2

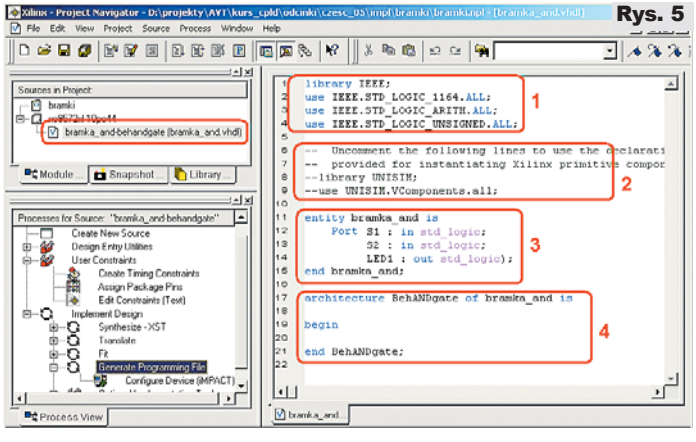
Wyrażenia zapisane pogrubioną czcionka są słowami kluczowymi języka VHDL. Powyższy zapis można przeczytać następująco: pod sygnał *LED1* podstaw iloczyn logiczny sygnałów *S1* oraz *S2*. Operator <= wymusza podstawienie wartości znajdującej się po prawej stronie do sygnału podanego po stronie lewej. Zamiast ***and*** można użyć również następujących słów kluczowych: ***or***, ***nand***, ***nor***, ***xor***, ***not.*** Otrzymamy wtedy odpowiednią bramkę logiczną.

Warto w tym miejscu upewnić się, że wszystko pracuje zgodnie z naszymi założeniami. Zaznaczamy plik z programem w oknie *Project Navigator* i klikamy *Assign Package Pins* (**rysunek 6**). Uruchomi to znane już narzędzie, w którym przypiszemy sygnały do obudowy. Po tej operacji klikamy *Generate Programming File* i uzyskujemy plik wynikowy, którym następnie programujemy układ CPLD.

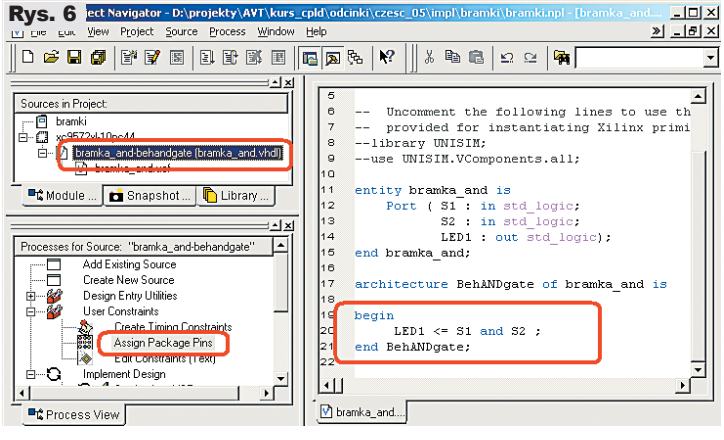
3-wejściowa bramka NAND

Spróbujmy teraz odrobinę podnieść poprzeczkę i stworzyć bramkę z większą liczbą wejść. Pierwsze rozwiązanie jest dość oczywiste i możliwe do zrealizowania w oparciu o przedstawione powyżej informacje – **listing 1**. Ten jakże prosty przykład miał za zadanie uświadomić, że do wyjścia można przypisać kilka sygnałów i łącznie stosować operatory takie jak *and* i *not*. Co więcej, okrągłe nawiasy pozwalają „ręcznie" ustalić kolejność wykonywania poszczególnych operacji logicznych.

Rys. 5



Rys. 6



Listing 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity nand3_ver2 is
  port(
    x : in std_logic_vector (2 downto 0) ;
    y : out std_logic
  );
end nand3_ver2;
architecture nand3Behav of nand3_ver2 is
  signal temp : std_logic_vector (2 downto 0) ;
begin
  --temp(0) <= not x(0) ;
  --temp(1) <= not x(1) ;
  --temp(2) <= not x(2) ;
  temp(2 downto 1) <= not x(2 downto 1) ;
  temp(0) <= not x(0) ;
  y <= temp(0) or temp(1) or temp(2) ;
end nand3Behav;
```

Warto także zauważyć, że wszystkie sygnały wejściowe zostały zadeklarowane po przecinku w jednej linii.

Drugi sposób został przedstawiony na **listi-ngu 2**. Pojawiły się tutaj dwie nowe rzeczy. Po pierwsze, nie utworzyliśmy trzech zmiennych wejściowych, tylko wektor *x* (tablica jednowymiarowa) za pomocą instrukcji:

x : in std_logic_vector (2 ***downto*** *0) ;*

Polecenie to różni się od deklaracji pojedynczej zmiennej tylko obecnością nawiasów, w których zapisujemy liczbę pozycji i typem – zamiast *std_logic* jest *std_logic_vector* (typ ten informuje, że chcemy pracować z wektorem, a nie pojedynczym sygnałem). Dwójka oznacza, że w naszym wektorze najstarsza pozycja ma numer dwa, natomiast zero oznacza, że najmłodsza ma adres zero. W związku z tym mamy trzy pozycje o numerach: 2, 1, 0. Znajduje się tu jeszcze słowo kluczowe ***downto***, które określa, że definiujemy zakres.

W dalszej części programu pojawia się linia: *signal temp : std_logic_vector (2* ***downto*** *0) ;* która definiuje na potrzeby opisywanej architektury sygnał pomocniczy *temp*. Warto zauważyć, że nie ma tu słówka ***port***, gdyż sygnały te będą wykorzystywane na wewnętrzne potrzeby bez wyprowadzania ich na zewnątrz. Zamiast tego mamy słowo ***signal***, które określa, że mamy do czynienia z wewnętrznym sygnałem. Z tego samego powodu nie ma już parametru *in* ani *out*. Pojawia się natomiast identyczny zapis *(2* ***downto*** *0)* informujący, że mamy do czynienia z wektorem zawierającego trzy pozycje.

W sekcji ***begin*** pokazano, w jaki sposób można wykorzystać sygnały pomocnicze i jak obchodzić się z wektorem. Do wektora pomocniczego *temp* możemy przypisać poszczególne pozycje z wektora *x,* podając w nawiasach indeksy, np.:

temp(0) <= x(0) ;

Spowoduje to skopiowanie wartości z zerowej pozycji wektora *x* do zerowej pozycji wektora *temp*. W programie z listingu 2 wykonywana jest jeszcze dodatkowo operacja negowania przed przypisaniem, za co odpowiada słowo kluczowe ***not***.

Okazuje się, że można od razu przypisać cały wektor instrukcją:

temp <= x;

co zastępuje ciąg instrukcji:

temp(0) <= x(0) ;
temp(1) <= x(1) ;
temp(2) <= x(2) ;

co więcej, można wymusić również negację wszystkich składników w czasie kopiowania, dodając słówko *not*:

temp <= ***not*** *x ;*

Warto jeszcze przy okazji wspomnieć, że istnieje możliwość skopiowania tylko części wektora, podając zakresy pozycji do skopiowania:

temp (2 ***downto*** *1) <= x (2* ***downto*** *1) ;*

co będzie odpowiadać operacjom:

temp(2) <= x(2) ;
temp(1) <= x(1) ;

Wstawienie operatora ***not*** ponownie spowoduje zanegowanie wszystkich kopiowanych sygnałów:

temp (2 ***downto*** *1) <= not x (2* ***downto*** *1) ;*

Przypominam: ! (a*b*c) = !a + !b + !c => NAND3

„Sterownik" diod LED

Mając już jakieś pojęcie o zarządzaniu sygnałami i przypisywaniu im wartości, spróbujmy zrealizować następujące zadanie: przyciskami wprowadzamy liczbę binarną i chcemy zaświecić odpowiednią liczbę diod LED. Zapewne dałoby się wyznaczyć stosowne funkcje boolowskie i za pomocą samych operacji przypisania zrealizować to zadanie. Czytelnicy mogą poeksperymentować w tym kierunku we własnym zakresie. Chciałbym teraz pokazać dwie nowe rzeczy: podstawianie sygnałów stałych do portów oraz przybliżyć strukturę ***when***.

Na początku zdefiniujmy dwa wektory:
- wektor *x*, który będzie reprezentował stan przycisków (3 pozycje)
- wektor *y*, który będzie reprezentował stan diod LED (5 pozycji)

Można tego dokonać znanymi już poleceniami:

x : in std_logic_vector (2 ***downto*** *0) ;*
y : out std_logic_vector (4 ***downto*** *0) ;*

Oba polecenia mają bardzo podobną składnię: podajmy nazwę wektora (odpowiednio *x* lub *y*), następnie pojawia się dwukropek, potem podajemy typ, czyli *std_logic_vector* i na końcu definiujemy zakres podając w nawiasach indeks najbardziej znaczącej pozycji oraz indeks najmniej znaczącej pozycji rozdzielając je słowem kluczowym ***downto***.

Po zdefiniowaniu jednostki ***entity*** przychodzi kolej na stworzenie opisu architektury. Wykorzystamy do tego celu następujące polecenie:

przypisanie ***when*** *warunek* ***else***

Dla czytelności zdefiniujemy sobie jeszcze sygnał pomocniczy temp:

signal temp : std_logic_vector (2 ***downto*** *0) ;*

który będzie zawierał zanegowane stany przycisków, aby po ich puszczeniu żadna dioda się nie świeciła. Konieczne będzie zatem znane nam już przypisanie:

temp <= ***not*** *x ;*

Spójrzmy na następującą instrukcję:

y <= „11110" ***when*** *temp = „001"* ***else***
„11100" ***when*** *temp = „010"* ***else***
„11000" ***when*** *temp = „011"* ***else***
„10000" ***when*** *temp = „100"* ***else***
„00000" ***when*** *temp = „101"* ***else***
„11111" ;

Pierwszą interesującą rzeczą jest przypisanie: *y <= „11110"*. Okazuje się, że można od razu przypisać określone wartości wszystkim pozycjom wektora. Wystarczy zapisać ciąg bitów w cudzysłowach (długość tego ciągu musi być zgodna z rozmiarem wektora). Dalej znajduje się słowo kluczowe ***when***, które ściśle określa, w jakiej sytuacji takie przypisanie może być wykonane, tzn. określony jest warunek, pod jakim można wystawić na wyjścia ciąg bitów podany przed ***when***. Następnie znajduje się ***else***, określające co zrobić, gdy poprzedni(e) warunek(i) nie został(y) spełniony(e). Za nim znajduje się inna sekwencja bitów i ponownie warunek po słowie ***when***, którego spełnienie pozwala podstawić określony ciąg bitów. Gdyby i ten warunek nie został spełniony, to za nim jest następny, itd. Po ostatnim ***else*** znajduje się ciąg samych jedynek, który zostanie przypisany do wektora *y* w momencie, gdy wektor *temp* będzie miał wartość inną niż te wyszczególnione.

Całość można uogólnić do postaci

sygnał <=
wartość1 ***when*** *warunek1* ***else***
(...)
wartośćN ***when*** *warunekN* ***else***
wartość_domyślna ;

W oparciu o ten szablon widać, że na początku zapisujemy nazwę wektora, do którego chcemy coś przypisać. Następnie podajemy wartość, jaką chcemy przypisać, a za nią umieszczamy słowo ***when*** i podajemy warunek jaki musi zostać spełniony, aby takie przypisanie mogło mieć miejsce (w przykładzie jest to porównanie wektora *temp* z wartością stałą). Następnie umieszczamy słowo ***else*** i podajemy kolejną parę: wartość oraz warunek. Po ostatnim słowie ***else*** zapisujemy domyślną wartość, która zostanie przypisana w sytuacji, gdy żaden warunek nie został spełniony. Pełny kod źródłowy został przedstawiony na **listingu 3**. Jego działanie sprowadza się do podstawienia odpowiedniego słowa wyjściowego do wektora *y* przypisanego do wyprowadzeń układu CPLD. Po puszczeniu wszystkich klawiszy zmienna *temp* ma wartość „000", zatem nie spełnia żadnego warunku i diody są gaszone. Przed kompilacją programu należy, oczywiście, skorzystać z narzędzia *Assign Package Pins*. Wektory są reprezentowane w nim jako:

nazwa<indeks>

Czyli do przypisania dostaniemy porty: *x<2>, x<1>, x<0>, y<4>, y<3>, y<2>, y<1>, y<0>.*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity sterownikLED is
  port(
  x : in std_ulogic_vector  (2 downto 0) ;
  y : out std_ulogic_vector (4 downto 0) );
end sterownikLED;
architecture SterLEDbehav of sterownikLED is
  signal temp  : std_ulogic_vector (2 downto 0) ;
begin
  temp <= not x ;
  y <= "11110" when temp = "001" else
    "11100" when temp = "010" else
    "11000" when temp = "011" else
    "10000" when temp = "100" else
    "00000" when temp = "101" else
    "11111" ;
end SterLEDbehav;
```

Listing 3

Dekoder 7-segmentowy

Okazuje się, że istnieje mechanizm podobny do ***when***... ***else***... konkretnie struktura ***with***... ***select***. Pierwsza z nich analizuje kolejno wszystkie warunki i gdy nie zostanie znaleziony warunek spełniający przyjęte założenia (w naszym przykładzie zmienna *temp* miałaby wartość np. „000"), to wybierana jest ostatnia klauzula ***else***. W przypadku struktury ***with***... ***else*** sprawa wygląda inaczej, gdyż wszystkie alternatywy są sprawdzane równocześnie. Stąd konieczność, aby wyszczególnić wszystkie możliwe przypadki. Istnieje jednakże zapis ***when others***, którego można użyć do podstawienia wartości domyślnej, gdy dany przypadek nie został wyszczególniony.

Przejdźmy do konkretów. Zaprojektujmy dekoder kodu binarnego na kod 7-segmentowy, do którego podłączymy trzy przyciski. W zależności od wybranej kombinacji włączyć ma się odpowiednia cyfra na wyświetlaczu.

Struktura ***with***... ***else*** ma następujący szablon:

with *analizowany_sygnał* ***select***
wyjście <=
wartość_przypisania1 ***when*** *wartość_porównywana1*
wartość_przypisania2 ***when*** *wartość_porównywana2*
(...)
wartość_przypisaniaN ***when*** *wartość_porównywanaN*
wartość_przypisania_domyślna ***when others;***

Na potrzeby dekodera zdefiniujemy wektor *x* reprezentujący przyciski, wektor *y* reprezentujący sygnały wyświetlacza 7-segmentowego oraz sygnał T1 odpowiadający za włączenie portu tranzystora. Umówmy się, że najmniej znacząca pozycja wektora *y* odpowiada segmentowi *a*, a najbardziej znacząca – segmentowi *g*.

Stosowny kod przedstawiono na **listingu 4**. W przykładzie tym zaprezentowano sposób użycia struktury ***with... select***. Pomiędzy tymi dwoma słowami kluczowymi umieszczamy wektor *x*, który decyduje o tym, co znajdzie się na wyjściach (czyli co zostanie przypisane do *y*). Następnie umieszczamy kody dla wyświetlacza 7-segmentowego i instrukcją ***when*** jednoznacznie określamy, dla jakiej kombinacji bitów wejściowych wektora *x* (bo on jest między poleceniami ***with*** oraz ***select***) ma się dany kod binarny pojawić. Ostatni zapis (same jedynki – wyświetlacz wyłączony) rezerwujemy dla stanów innych niż wymienione, co w tym wypadku raczej nie wystąpi. Całość zamykamy średnikiem, a poszczególne przyporządkowania słów wejściowych do słów wyjściowych rozdzielamy przecinkiem. Jak widać, niepotrzebne było wprowadzanie negacji wejść, gdyż możemy przyjąć, że trzy jedynki na wejściu oznaczają zero i przypisać do nich kod wyświetlacza, który skutkuje wyświetleniem liczby zero.

W ostatniej linii opisu architektury włączamy jeszcze tranzystor. Zwracam uwagę, że zero znajduje się w apostrofach, a nie cudzysłowach, gdyż jest to pojedynczy bit, a nie wektor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dekoder7segm is
  port(
    x : in std_logic_vector (2 downto 0) ;
    y : out std_logic_vector (6 downto  0) ;
    T1 : out std_logic
  );
end dekoder7segm;
architecture dekoder7segmBehav of dekoder7segm is
begin
  with x select
    y <= "1000000" when "111", --0 (000)
      "1111001" when "110", --1 (001)
      "0100100" when "101", --2 (010)
      "0110000" when "100", --3 (011)
      "0011001" when "011", --4 (100)
      "0010010" when "010", --5 (101)
      "0000010" when "001", --6 (110)
      "1111000" when "000", --7 (111)
      "1111111" when others  ;
  T1 <=  '0'  ;
end dekoder7segmBehav;
```

Listing 4

Przerzutnik D

Do tej pory przyglądaliśmy się układom kombinacyjnym. Pora odpowiedzieć sobie na pytanie, jak zrealizować układ sekwencyjny. Najprostszym zadaniem jest zrealizowanie przerzutnika D, który jest wyzwalany zboczem. Oznacza to, że trzeba poszukać narzędzia pozwalającego wychwycić moment wystąpienia zmiany sygnału niskiego na wysoki (lub odwrotnie). Jest nim instrukcja ***process***, której składnia jest następująca:

nazwa_procesu: ***process*** *(lista_czułości)* ***is begin***
--kod...
end process *nazwa_procesu* ***;***

Na liście czułości umieszczamy pojedyncze sygnały, wektory lub dowolne ich kombinacje, rozdzielając poszczególne składniki przecinkiem, np.:

proces1: ***process*** *(a, b, c, x)* ***is begin***
--kod...
end process *proces1* ***;***

gdzie *x* jest wektorem. Jeżeli któryś z sygnałów zawartych na liście czułości zmieni wartość, to zostanie wykonany kod wewnątrz procesu. Sygnał może reprezentować np. port wejściowy, który po zmianie swojego stanu uruchomi proces. W przypadku wektorów zawierających wiele pozycji, zmiana choćby jednej również rozpocznie wykonanie kodu wewnątrz procesu.

W tym miejscu można wykorzystać strukturę *if* o następującej postaci:

if *warunek* ***then***
--instrukcje;
elsif *warunek* ***then***
-- instrukcje;
elsif *warunek* ***then***
(...)
elsif *warunek* ***then***
--instrukcje;
else
--instrukcje ;
end if*;*

Należy mieć na uwadze dwie rzeczy:
– bloki ***else*** oraz ***elsif*** są opcjonalne i nie muszą wystąpić w programie
– struktury ***if***... ***then***... ***elsif***... ***else***... ***end if*** wolno używać tylko WEWNĄTRZ procesu. Nie moglibyśmy zrealizować np. dekodera 7-segmentowego w ten sposób, tzn. podmieniając blok ***with***... ***select*** na strukturę ***if***.

Zanim zajmiemy się realizacją przerzutnika, chciałbym pokazać jeszcze dwie interesujące instrukcje, mianowicie:
– ***rising_edge***, która pozwala wykryć zbocze narastające sygnału
– ***falling_edge***, która pozwala wykryć zbocze opadające sygnału.

W oparciu o zebrane dane możemy przystąpić do implementacji przerzutnika D – **listing 5**. Interesuje nasz szczególnie fragment kodu pomiędzy znakami #. Instrukcje:

zbocze: ***process*** *(CLK)* ***is*** *begin*
(...)
end process *zbocze;*

tworzą nowy proces, który został nazwany *zbocze*. Na liście czułości znajduje się sygnał *CLK*, więc gdy zmieni on swoją wartość, zostanie wywołany ten właśnie proces. W jego wnętrzu znajduje się prosta instrukcja *if* sprawdzająca jeden warunek – czy wystąpiło zbocze narastające. Jeżeli taka

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
entity przerzutnikD is
    Port ( D : in std_logic;
           Q : out std_logic;
           CLK : in std_logic);
end przerzutnikD;
architecture przerzutnikDbehav of przerzutnikD is
begin
  -- ##############################
  zbocze: process (CLK) is begin
    if rising_edge(CLK) then
      Q <= D ;
    end if;
  end process zbocze;
  -- ##############################
end przerzutnikDbehav;
```

Listing 5

sytuacja miała miejsce, to następuje przypisanie do wyjścia *Q* wartości wejścia *D*.

Licznik 5-bitowy

Wiedząc, w jaki sposób wykrywać zbocza, spróbujmy przygotować prosty licznik, który będzie liczył w naturalnym kodzie binarnym i będzie wyświetlał wynik za pomocą diod LED. Do tego celu będziemy potrzebować wejścia zegarowego *CLK* oraz wektora wyjściowego *led*, reprezentującego poszczególne diody LED. Utworzymy te elementy w znany nam już sposób. Dalej niezbędny będzie proces z sygnałem *CLK* znajdującym się na liście czułości. Niezbędna będzie również zmienna do przechowywania stanu licznika oraz sygnał resetu.

Kod licznika spełniającego powyższe kryteria przedstawiono na **listingu 6**. Utworzono tu jeden proces nazwany *licznik*, na jego liście czułości znajdują się sygnały *reset* oraz *CLK*. Oznacza to, że zostanie on uruchomiony w momencie, gdy pojawi się zmiana sygnału *CLK* (zwiększenie zawartości licznika) lub *reset* (wyzerowanie licznika). Wewnątrz procesu (pomiędzy słowami kluczowymi ***is*** oraz ***begin***) utworzono zmienną pomocniczą *temp* przeznaczoną do przechowywania stanu licznika. Jest ona deklarowana przy użyciu słowa kluczowego ***variable***, potem podaje się jej nazwę, umieszcza dwukropek i podaje typ. W przykładzie zastosowano ***unsigned*** umożliwiający korzystanie z operatora + i podano w nawiasie, że posiada ona 5 pozycji, czyli jest 5-bitowa.

Wewnątrz procesu znajduje się omówiona przed chwilą struktura ***if***. W pierwszym kroku sprawdzane jest, czy sygnał *reset* ma wartość zero, jeżeli tak – następuje zerowanie licznika. Dzięki temu, że zerowanie odbywa się poziomem niskim, sygnał ten może być bezpośrednio podłączony do przycisku, który ma stan niski dopiero po naciśnięciu. Po słowie kluczowym ***elsif*** testowany jest drugi warunek, mianowicie wystąpienie zbocza narastającego na wejściu *CLK*, które podłączone jest do przestrajanego generatora. Gdy zostanie on spełniony, następuje zwiększenie wartości zmiennej *temp*, czyli zliczenie kolejnego zbocza.

Po syntezie (kompilacji) i załadowaniu tego kodu najprawdopodobniej okaże się, że nie pracuje on do końca zgodnie z naszymi założeniami. Zakłócenia powodują, że licznik zmienia swoją wartość niekiedy o więcej niż jeden. W związku z tym należy zmodyfikować program tak, aby wyeliminować te zakłócenia. Do tej pory używaliśmy do tego celu przerzutnika D i podłączaliśmy do jego wejścia zegarowego sygnał z generatora kwarcowego. Do wejścia D podłączany był sygnał CLK, a z wyjścia Q pobieraliśmy sygnał, który był następnie podłączony do wejścia licznika.

Działanie to można powtórzyć również w tym przypadku. Potrzebny będzie nam pośredni sygnał, dajmy na to *CLKprim*, który zadeklarujemy wewnątrz bloku opisującego architekturę układu. Wykorzystamy do tego prostą deklarację:

signal *CLKprim :* ***std_ulogic*** *;*

Przerzutnik D utworzymy jako osobny proces i nazwiemy go *synchr*. Na liście czułości umieścimy sygnał *SYNC*, który będzie pochodził z generatora (należy go zadeklarować w bloku ***entity***). Umieścimy tu instrukcję ***if*** sprawdzającą, czy wystąpiło zbocze narastające. Po jego wystąpieniu, zgodnie z zasadą pracy przerzutnika D, skopiujemy wartość sygnału wejściowego (*CLK*) na wyjście (*CLKprim*). W tym momencie sygnałem z przestrajanego generatora jest dla nas sygnał *CLKprim*, więc należy zmodyfikować odpowiednio proces *licznik* – zmienić wpis na liście czułości oraz testować zbocze sygnału *CLKprim*, zamiast *CLK*. Kod zawierający omówione zmiany jest widoczny na **listingu 7**.

Licznik modulo 100

W ramach dalszych ćwiczeń zaprojektujmy bardziej złożony układ, który jest podobny do tego, co robiliśmy wcześniej. Mianowicie zaprojektujmy licznik modulo 100 z możliwością resetowania. Stosowny kod został pokazany na **listingu 8**.

Pierwszym krokiem jest utworzenie jednostki *entity*, w której definiujemy potrzebne nam sygnały, takie jak wejście zegarowe (*CLK)*, wejście generatora kwarcowego (*SYNC*), wejście resetujące (*reset)* oraz wyjścia *wysw* (sterowanie poszczególnymi segmentami wyświetlacza), *T1* i *T2* (tranzystory włączające wyświetlacze).

Zdefiniujmy jeszcze sygnały pomocnicze wewnątrz bloku opisującego architekturę. Na wzór poprzedniego przykładu dodamy sygnał *CLKprim* reprezentujący sygnał zegarowy z przestrajanego generatora po „przepuszczeniu" przez przerzutnik D. Potrzebny jest również sygnał odpowiedzialny za przełączanie wyświetlaczy i tę rolę będzie pełnić *sel*. Dwie ostatnie pozycje (*jednosci*, *dziesiatki*) będą odpowiedzialne za przechowywanie wskazań licznika, odpowiednio, jedności oraz dziesiątek i posłużą do określenia, jaki kod do wyświetlacza zapisać.

Przyjrzyjmy się najpierw pierwszemu procesowi: *synchro*. Między słowami kluczowymi ***is*** oraz ***begin*** umieszczono zmienną *preskaler*, która będzie wykorzystywana w tym procesie. Na liście czułości procesu (czyli w nawiasach za słowem kluczowym *process*) wymieniony został sygnał *SYNC*, tak więc proces ten zostanie wykonany w sytuacjach, gdy stan portu *SYNC* ulegnie zmianie. Wtedy wykonamy dwie czynności – po pierwsze, przepiszemy sygnał *CLK* do *CLKprim,* realizując w ten sposób prosty przerzutnik D wyzwalany generatorem kwarcowym. Po drugie, wysterujemy sygnał *sel*, tak, aby cyfry na wyświetlaczach były wyświetlane naprzemiennie (multipleksowanie). Jak mieliśmy okazję się przekonać, zbyt szybkiemu przełączaniu wyświetlaczy towarzyszy obecność poświaty, co skutecznie zmniejsza ich estetykę i czytelność. Tu właśnie wykorzystamy zmienną *preskaler*, zwiększając ją o jeden przy każdym zboczu narastającym generatora kwarcowego. W momencie, gdy osiągnie ona maksymalną wartość i zwiększymy jej zawartość znowu o jeden, ulegnie wyzerowaniu. Otrzymujemy zatem licznik modulo 2^{11} (bo tyle pozycji ma ta zmienna => 10 downto 0 => 10, 9, 8, 7, 6, 5, 4,

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--  Uncomment the following lines to use ¬
                          the declarations that are
--  provided for instantiating Xilinx ¬
                            primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
  entity licznik5bitSYNC is
    Port ( led : out ¬
                  std_ulogic_vector(4 downto 0);
         CLK : in std_ulogic;
     SYNC :  in std_ulogic ;
           reset : in std_ulogic);
end licznik5bitSYNC;
architecture licznik5bitSYNCBehavioral ¬
                              of licznik5bitSYNC is
  signal CLKprim : std_ulogic ;
begin
  synchr: process (SYNC) is begin
    if rising_edge(SYNC) then
      CLKprim <= CLK ;
    end if ;
  end process synchr ;
  licznik: process (CLKprim, reset) is
    variable temp : unsigned (4 downto 0) ;
  begin
    if reset = '0' then
      temp := "00000" ;
    elsif rising_edge(CLKprim) then
      temp := temp + 1 ;
    end if;

    --zapisanie do wektora led
    led <= not std_ulogic_vector(temp) ;
  end process licznik ;
end licznik5bitSYNCBehavioral;
```

Listing 7

Listing 6

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;
entity licznik5bit is
    Port ( led : out std_ulogic_vector(4 downto 0);
         CLK : in std_ulogic;
           reset : in std_ulogic);
end licznik5bit;
architecture Behavioral of licznik5bit is
begin
  licznik: process (CLK, reset) is
    variable temp : unsigned (4 downto 0) ;
  begin
    if reset = '0' then
      temp := "00000" ;
    elsif rising_edge(CLK) then
      temp := temp + 1 ;
    end if;

    --zapisanie do wektora led
    led <= not std_ulogic_vector(temp) ;
  end process licznik ;
end Behavioral;
```

3, 2, 1, 0, jak łatwo policzyć jest tych pozycji 11). Dalej znajduje się znana już struktura ***if***, która sprawdza wartość zmiennej, gdy wynosi ona zero, co zdarza się raz na 2^{11} taktów sygnału generatora kwarcowego, negowany jest sygnał *sel,* wymuszając tym samym zmianę aktywnego wyświetlacza.

Procesy *liczJedn* oraz *liczDzies* są bardzo podobne w budowie i odpowiadają za zliczanie, odpowiednio, jedności oraz dziesiątek. Każdy z procesów posiada zadeklarowaną zmienną 4--bitową służącą do zliczania zboczy, która w ostatnim etapie procesu jest kopiowana do sygnału *jednosci* lub *dziesiatki* (zależnie od tego, z którym procesem mamy do czynienia). Na liście czułości znajdują się sygnały niezbędne do uruchomienia procesu i tym samym zwiększenia o jeden odpowiedniej zmiennej. Dla licznika jedności jest to sygnał *CLKprim*, który pochodzi z przestrajanego generatora i został „przepuszczony" przez przerzutnik D. Dla licznika dziesiątek sygnałem takim jest najstarszy bit licznika jedności (sygnał *jednosci(3)*), co jest analogiczne dla rozwiązań tworzonych w edytorze schematów. Tam również taktowaliśmy licznik dziesiątek sygnałem pochodzącym z licznika jedności. Na liście czułości obu procesów umieszczono jeszcze sygnał *reset*, co pozwala wywołać je w momencie naciśnięcia przycisku i wyzerować stosowne zmienne.

Oba procesy posiadają także bardzo podobną strukturę ***if***, która sprawdza, jakie zdarzenie wywołało proces. Może to być sygnał resetowania (warunek *reset = '0'*) powodujący wyzerowanie zawartości zmiennych *temp1*, *temp2* odpowiedzialnych za, odpowiednio, zliczanie jedności oraz dziesiątek. Innym zdarzeniem jest wystąpienie zbocza narastającego/opadającego sygnału taktującego licznik jedności/dziesiątek (odpowiednio: *CLKprim* oraz *jednosci(3)*). W takiej sytuacji zwiększana jest wartość stosownej zmiennej, ale jest tu pewne zastrzeżenie – zliczamy modulo 10, więc licznik musi się wyzerować w momencie, gdy znajdzie się w nim wartość 10. Stąd wypływa konieczność zastosowania jeszcze jednej instrukcji ***if*** sprawdzającej, czy zmienna nie osiągnęła czasem wartości 10. Przypisaniu zmiennych do sygnałów towarzyszy operacja rzutowania:

jednosci <= std_logic_vector(temp1) ;

Oznacza ona, że życzymy sobie, aby wartość zmiennej *temp1* skonwertować do postaci sygnału *std_logic_vector* i następnie przypisać do zmiennej *jednosci*. Rzutowanie sprowadza się do podania typu i umieszczeniu w nawiasie zmiennej, która ma być poddana konwersji.

Dalsza część programu nie powinna wzbudzić większych kontrowersji. Wykorzystujemy tutaj strukturę ***when***... ***else*** do wysterowania tranzystorów. Są one włączane naprzemiennie, zależnie od utworzonego wcześniej sygnału *sel*. W ostatniej części programu wykorzystujemy ponownie strukturę ***when***... ***else***, ale do przesłania do wyświetlacza stosownego kodu reprezentującego liczbę. Warunek jest w tym przypadku trochę bardziej złożony, gdyż kod, jaki zapiszemy do wyświetlacza, zależy od dwóch rzeczy: po pierwsze od cyfry, gdyż kod ten będzie inny dla '2' i inny dla '7'. Drugi warunek musi uwzględniać, który wyświetlacz w danym momencie obsługujemy – jeżeli wyświetlana jest liczba 19, to inny kod wyślemy do wyświetlacza, gdy obsługujemy wskazanie dziesiątek, a inny kod obsługując wskazanie jedności. Po instrukcji ***when*** należy podać warunek, może być ich więcej, gdy zastosujemy operator ***and*** (wszystkie warunki muszą być spełnione, aby wysłać dany kod) lub ***or*** (gdy wystarczy spełnienie tylko jednego warunku). Przypominam, że wszystkie eksperymenty z torem podczerwieni można przeprowadzać tylko po uprzednim umieszczeniu na diodzie IR czarnej rurki termokurczliwej. W innym przypadku odbiornik będzie nieustannie włączony.

Bariera podczerwieni

Na zakończenie spróbujmy przygotować prostą barierę podczerwieni. Załóżmy, że stan jej pracy będzie sygnalizowany przez dwie diody LED: pierwsza z nich (LED1) będzie na bieżąco wskazywać, czy wiązka podczerwieni ulega odbiciu od jakiegoś przedmiotu, czy nie. Druga dioda (LED5) będzie „pamiętała" czy bariera została przekroczona, tzn. po pierwszym odbiciu się podczerwieni będzie ona włączona do momentu resetu przyciskiem S1.

Idea pracy nie powinna wzbudzać wątpliwości – po wyemitowaniu wiązki podczerwieni (a w zasadzie w trakcie jej emisji) sprawdzamy stan odbiornika SFH5110 i gdy wystąpi na jego wyjściu stan niski, to znaczy, że wysłana wiązka uległa odbiciu od jakiegoś przedmiotu.

Użyty odbiornik podczerwieni reaguje na częstotliwość 36kHz, zatem nie możemy włączyć diody IRED na stałe, gdyż wtedy emitowane przez nią światło nie zmieni stanu wyjściowego układu SFH5110. Częstotliwość pracy odbiornika nie jest krytyczna i może się wahać w pewnych granicach, ale im mniej jest ona zbliżona do 36kHz, tym mniej wrażliwy jest odbiornik. Z tych rozważań wynika, że musimy kluczować stan diody – włączać i wyłączać

Listing 8

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity modulo100 is
    Port ( CLK : in std_logic;
           SYNC : in std_logic;
     reset : in std_logic ;
           wysw : out std_logic_vector(6 downto 0);
           T1 : out std_logic;
           T2 : out std_logic);
end modulo100;
architecture modulo100Behav of modulo100 is
  signal CLKprim : std_logic ;
  signal jednosci  : std_logic_vector (3 downto 0) ;
  signal sel : std_logic ;
  signal dziesiatki  : std_logic_vector (3 downto 0) ;
begin
  --usuwanie zaklocen z sygnalu CLK
  synchro: process (SYNC) is
    variable preskaler : unsigned (10 downto 0) ;
  begin
    if rising_edge(SYNC) then
      --przerzutnik D
      CLKprim <= CLK ;
      --sterowanie przelaczaniem wyswietlaczy
      preskaler := preskaler + 1;
      if preskaler = 0 then
        sel <= not sel ;
      end if  ;
    end if;
  end process synchro ;
  --zliczanie jednosci
  liczJedn: process (CLKprim, reset)  is
    variable temp1 : unsigned (3 downto 0) ;
  begin
    --zliczenie zbocza
    if reset = '0' then
      temp1 := "0000" ;
    elsif rising_edge(CLKprim) then
      temp1 := temp1 + 1;
      --operacja modulo 10
      if temp1 = 10 then
        temp1 := "0000" ;
      end if;
    end if;
    --zapisanie do sygnalu
    jednosci <= std_logic_vector(temp1) ;
  end process liczJedn ;
  --zliczanie dziesiatek
  liczDzies: process (jednosci(3), reset) is
    variable temp2 : unsigned (3 downto 0) ;
  begin
    if reset = '0'  then
      temp2  := "0000";
    elsif falling_edge(jednosci(3)) then
      temp2 := temp2 + 1;
      --operacja modulo 10
      if temp2 = 10 then
        temp2  := "0000" ;
      end if ;
    end  if ;
    dziesiatki <= std_logic_vector(temp2) ;
  end process liczDzies;
  --obsluga tranzystorow
  T1 <= '0' when sel='0' else '1' ;
  T2 <= '0' when sel='1' else '1' ;
  --wyswietlenie prawdidlowej wartosci na wyswietlaczu
  wysw <=
    --wyswietlenie jednosci
    "1000000" when (jednosci = "0000") and (sel='0') else
    "1111001" when (jednosci = "0001") and (sel='0') else
    "0100100" when (jednosci = "0010") and (sel='0') else
    "0110000" when (jednosci = "0011") and (sel='0') else
    "0011001" when (jednosci = "0100") and (sel='0') else
    "0010010" when (jednosci = "0101") and (sel='0') else
    "0000010" when (jednosci = "0110") and (sel='0') else
    "1111000" when (jednosci = "0111") and (sel='0') else
    "0000000" when (jednosci = "1000") and (sel='0') else
    "0010000" when (jednosci = "1001") and (sel='0') else

    --wyswietlenie dziesiatek
    "1000000" when (dziesiatki = "0000") and (sel='1') else
    "1111001" when (dziesiatki = "0001") and (sel='1') else
    "0100100" when (dziesiatki = "0010") and (sel='1') else
    "0110000" when (dziesiatki = "0011") and (sel='1') else
    "0011001" when (dziesiatki = "0100") and (sel='1') else
    "0010010" when (dziesiatki = "0101") and (sel='1') else
    "0000010" when (dziesiatki = "0110") and (sel='1') else
    "1111000" when (dziesiatki = "0111") and (sel='1') else
    "0000000" when (dziesiatki = "1000") and (sel='1') else
    "0010000" when (dziesiatki = "1001") and (sel='1') else
    "1111111";
end modulo100Behav;
```

z zadaną częstotliwością. Taki sposób pracy zapewni nam generator – daje on na wyjściu sygnał prostokątny, który można podłączyć do diody, otrzymując w ten sposób jej sekwencyjne przełączanie. Na płytce nie ma generatora 36kHz, więc musimy wykorzystać generator kwarcowy, przygotowując stosowny dzielnik (preskaler).

Niestety złe wiadomości nie kończą się na tym, trzeba wykonać jeszcze jedną czynność, aby cały tor podczerwieni pracował zgodnie z naszymi oczekiwaniami. Mianowicie nie można podłączyć do diody bezpośrednio przebiegu 36KHz, gdyż odbiornik też przestanie po krótkiej chwili reagować – stanie się niewrażliwy na ten sygnał. W związku z tym potrzebne jest kolejne kluczowanie, zapewniające na wejściu diody sygnał o żądanej częstotliwości, ale włączany okresowo.

Rozwiązanie spełniające wszystkie te kryteria widoczne jest na **listingu 9**. W pierwszej fazie deklarujemy sygnały wejściowe i wyjściowe, a następnie przystępujemy do opisu architektury. Tutaj wprowadzimy sobie dwa pomocnicze sygnały: *fout36kHz* oraz *fout500Hz,* i tak zorganizujemy kod, aby pierwszy z nich zmieniał swój stan z częstotliwością 36kHz, a drugi 500Hz (wybór tej częstotliwości jest arbitralny). Jest to możliwe po zastosowaniu odpowiedniego dzielnika dla generatora kwarcowego. Jako preskaler pracuje proces *generator*, który ma dwie zmienne pomocnicze: *temp1*, *temp2*. Proces ten posiada na swojej liście czułości sygnał SYNC (24MHz) i w momencie wystąpienia zbocza narastającego wartość obu zmiennych jest zwiększana. *Temp2* odpowiada za dostarczenie częstotliwości 500Hz, żeby taką uzyskać, należy podzielić 24MHz z generatora przez 48000. Oznacza to, że musimy mieć dwa półokresy (zero i jeden) trwające po 48000/2=24000 taktów zegara 24MHz. Stąd po każdym zliczeniu do 24000 zerujemy zmienną *temp2*, aby odliczać półokres od początku i zmieniamy stan *fout500Hz* na przeciwny. Uzyskujemy w ten sposób cykliczną zmianę co 24000 okresów generatora kwarcowego, zatem okres tego przebiegu będzie miał 48000 cykli i w efekcie żądaną częstotliwość 500Hz. Podobnie postąpimy ze zmienną *temp1*, z tym że zliczanie odbywa się do 333, co da w efekcie 36kHz.

Sygnał właściwy dla diody IR można uzyskać za pomocą prostej instrukcji:

IRED <= fout500Hz and fout36kHz;

Jest to bardzo prosta bramka AND, której pracę w podobnej roli mogliśmy już prześledzić. Kiedy *fout500Hz* ma stan wysoki, to sygnał *fout36kHz* jest przenoszony na wyjście, w przeciwnym wypadku nie jest. W efekcie częstotliwość 36kHz pojawia się na wyjściu tej bramki (i na wejściu diody IR) okresowo, a o to nam chodziło.

Ostatnie instrukcje dotyczą obsługi diod LED oraz przycisku S1. Dioda LED1 ma taki sam stan jak wyjście odbiornika podczerwieni (w obu wypadkach zero jest sygnałem aktywnym). Dioda LED5 jest natomiast włączana pierwszym zerem, jakie pojawi się na wejściu SFH, ale może zostać wyłączona tylko wtedy, gdy spełnione są dwa warunki: przycisk jest naciśnięty oraz do odbiornika nie dociera sygnał podczerwieni (wyjście w stanie wysokim).

Uwagi końcowe

Chciałbym zwrócić jeszcze uwagę na dwa aspekty jakie się wiążą z opisywaniem sprzętu w języku VHDL. Mianowicie należy unikać podstawiania wartości do tej samej zmiennej w różnych miejscach programu, gdyż prowadzi to najczęściej do błędów syntezy projektu. Drugą istotną kwestią jest synchronizacja sygnałów wewnątrz procesów. Okazuje się, że sprawdzanie sygnałów i manipulacja zmiennymi nie może odbywać się w dowolnej kolejności, gdyż może powodować błędy. Przykładowo, gdy umieszcza na liście czułości dwa sygnały, ważne jest, który najpierw opracujemy z użyciem instrukcji ***if***. Co więcej, następujący fragment kodu NIE JEST prawidłowy:

```
--generator czestotliwosci 36kHz i 500Hz
generator: process (SYNC) is
  variable temp1 : unsigned (8 downto 0) ;
  variable temp2 : unsigned (15 downto 0);
begin
  if rising_edge(SYNC) then
    temp1 := temp1 + 1 ;
    temp2 := temp2 + 1 ;
  end if ;
  if temp2 = 24000 then
    temp2 := „0000000000000000” ;
    fout500Hz <= not fout500Hz ;
  end if ;
  if temp1 = 333 then
    temp1 := „000000000” ;
    fout36kHz <= not fout36kHz ;
  end if ;
end process generator ;
```

W tym przypadku operacje na zmiennych *temp1* oraz *temp2* odbywają się poza blokiem ***if*** sprawdzającym zbocze *SYNC*, co prowadzi do błędu:

Signal $n0000 cannot be synthesized, bad synchronous description.

W takich wypadkach należy trochę poeksperymentować, aby dobrać właściwą kolejność sprawdzania sygnałów (gdyby na liście czułości były dwa lub więcej). Umieszczenie dwóch ostatnich instrukcji ***if*** wewnątrz pierwszej rozwiązuje problem.

Warto także zwrócić uwagą na składnię przypisań. Podstawienie do zmiennej odbywa się z użyciem operatora **:=**, natomiast dla sygnałów wykorzystujemy **<=**. Różny jest także sposób zapisywania wartości: wartość pojedynczego sygnału umieszcza się między apostrofami (np. *‘1’*), ale do wektorów odnosimy się z użyciem cudzysłowów (np. „000” – liczba cyfr musi się zgadzać z liczbą bitów wektora).

Dość ciekawą właściwością języka VHDL jest brak tablic Karnauga – minimalizację funkcji boole'owskich przeprowadza oprogramowanie. Zwalnia nas to ze żmudnego wyliczania poszczególnych funkcji na rzecz prostego zapisania ciągu bitów, jakie chcemy otrzymać w zależności od tego, co znajduje się na wejściu.

Na zakończenie niniejszego odcinka jeszcze mała ciekawostka. Okazuje się, że można podejrzeć sposób implementacji naszego układu w postaci modelu RTL. Wystarczy tylko zaznaczyć plik z kodem VHDL i kliknąć dwukrotnie *View RTL Schematic* (**rysunek 7**). Klikając na wybranym elemencie, przechodzimy „poziom niżej” i mamy możliwość zobaczenia jak jest zbudowany. Powrót do „poziomu wyżej” odbywa się przez dwukrotne kliknięcie w dowolnym, pustym miejscu.

Jakub Borzdyński
jakub.borzdynski@elportal.pl

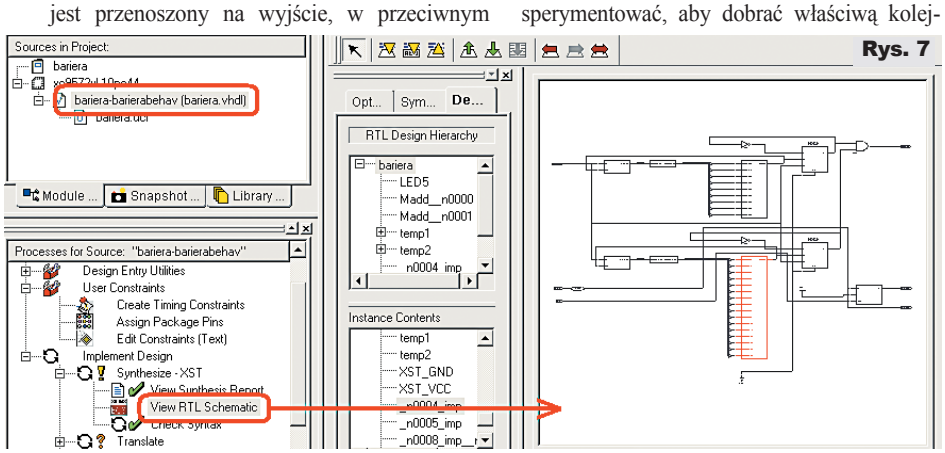
LITERATURA

[1] http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html

[2] Mark Zwoliński, „Projektowanie układów cyfrowych z wykorzystaniem języka VHDL”, WKŁ, Warszawa 2002.

Listing 9

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity bariera is
  port(
    SYNC : in std_logic ;
    SFH  : in std_logic ;
    S1   : in std_logic ;
    IRED : out std_logic ;
    LED1 : out std_logic ;
    LED5 : out std_logic
  );
end bariera;
architecture barieraBehav of bariera is
  signal fout36kHz : std_logic ;
  signal fout500Hz : std_logic ;
begin
  --generator czestotliwosci 36kHz i 500Hz
  generator: process (SYNC) is
    variable temp1 : unsigned (8 downto 0) ;
    variable temp2 : unsigned (15 downto 0);
  begin
    if rising_edge(SYNC) then
      temp1 := temp1 + 1 ;
      temp2 := temp2 + 1 ;
      if temp2 = 24000 then
        temp2 := “0000000000000000” ;
        fout500Hz <= not fout500Hz ;
      end if ;
      if temp1 = 333 then
        temp1 := “000000000” ;
        fout36kHz <= not fout36kHz ;
      end if ;
    end if ;
  end process generator ;
  --generacja modulowanej fali 36kHz
  IRED <=  fout500Hz and fout36kHz;
  --odbior sygnalu podczerwonego
  LED1 <= SFH ;
  LED5 <= ‘0’ when (SFH = ‘0’) else
          ‘1’ when (SFH = ‘1’) and (S1 = ‘0’) ;
end barieraBehav;
```



Rys. 7