Kurs CPLD

Automaty stanu

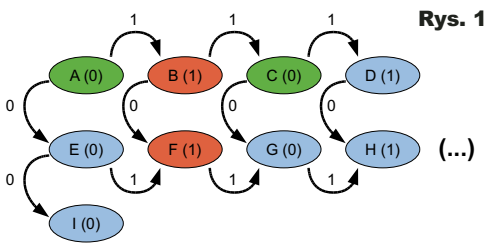
część 8

Dzisiejsza, ostatnia część kursu CPLD będzie poświęcona automatom stanu. Opracujemy kilka przykładów, aby utrwalić zdobytą poprzednio wiedzę. Powinno to także ułatwić zrozumienie zasad związanych z projektowaniem automatów.

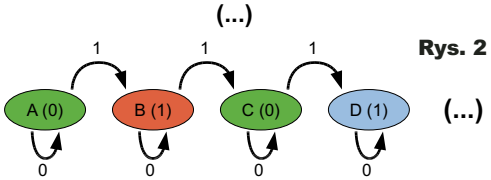
Układ kontroli parzystości

Na początek prosty przykład – układ do zliczania jedynek. Znajduje on zastosowanie np. w szeregowej transmisji danych (RS232) do wykrywania błędów. Idea polega na zapisaniu na jednym, dodatkowym bicie, czy przesyłana liczba jedynek jest parzysta (dopisujemy zero), czy nieparzysta (dopisujemy jeden). Wysyłając następujący ciąg bitów: 11110000, dodamy do niego zero, gdyż liczba jedynek jest parzysta. Skuteczność tego mechanizmu jest dyskusyjna, gdyż przekłamanie np. dwóch bitów nie zostanie zauważone. Pomimo tej wady układ kontroli parzystości stanowi dobry materiał do ćwiczeń. Automat zrealizujemy na przerzutnikach JK, a nie D jak poprzednim razem. Dzięki temu przyjrzymy się innemu sposobowi implementacji.

Na początku należy przygotować graf przejść. Zmiany poszczególnych stanów będą uzależnione od wartości bitu odczytanego z portu szeregowego. Sprawa wydaje się prosta, tworzymy pierwszy stan, dajmy na to *A* i przypisujemy do wyjścia zero, bo liczba jedynek jest parzysta (zero jest parzyste). Podanie na wejście jedynki spowoduje przejście do kolejnego stanu, powiedzmy *B* dającego na wyjściu jedynkę, bo liczba jedynek jest już nieparzysta. Gdybyśmy podali zero, będąc w stanie *A*, to przeszlibyśmy do innego stanu, powiedzmy *E*, i na wyjściu nadal byłoby zero, bo liczba jedynek byłaby wciąż parzysta. Kontynuując ten tok rozumowania, można przygotować graf jak na **rysunku 1**. Szybko okaże się, że dla ośmiu bitów rozrośnie się on dość mocno i będzie trudny do zaprojektowania.

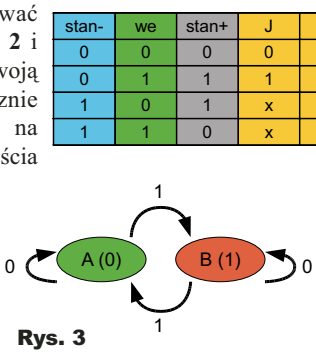


Rys. 1



Rys. 2

W taki właśnie sposób można natknąć się na konieczność zredukowania grafu. Dokładne przyjrzenie się stanom *B* oraz *F* (zaznaczone na pomarańczowo) pozwoli stwierdzić, że są one tożsame. Podanie zera podczas przebywania w stanie B powoduje przejście do stanu F, który jest identyczny ze stanem B. W obu stanach podanie zera powoduje pozostawienie jedynki na wyjściu, a podanie jedynki sprawia, że na wyjściu pojawia się zero. Co by się stało, gdybyśmy nie opuszczali stanu B, tylko się w nim „zatrzasnęli", gdy na wejście podawane jest zero? W zasadzie nic, bo dopóki pojawia się zero, to wyjście automatu się nie zmienia. Pojawienie się jedynki powoduje zmianę stanu wyjścia na przeciwny, co odzwierciedla poruszanie się w kierunku poziomym po grafie. Możemy zatem zredukować graf do postaci z **rysunku 2** i nadal będzie on pełnił swoją funkcję, a jest już znacznie prostszy. Każda jedynka na wejściu zmienia stan wyjścia na przeciwny, natomiast zero nic nie zmienia. Być może nie brzmi to do końca przekonująco, dlatego zachęcam do przetestowania obu grafów wybranym ciągiem bitów, np. 1100011. Powinny one dać na wyjściu 0, bo liczba jedynek jest parzysta.



Rys. 3

Dobra wiadomość jest taka, że to nie jest koniec. Całość można uprościć jeszcze bardziej. Czy stany A oraz C (zielone) nie wydają się tożsame? Zarówno A, jak i C daje na wyjściu jedynkę, gdy wejście ma stan wysoki, a w przypadku zera następuje zatrzaśnięcie w stanie aktualnym. Możliwa jest dalsza redukcja – ostateczny graf pokazano na **rysunku 3**. Można pokusić się o podanie dowolnego ciągu testowego i sprawdzenie, czy całość pracuje zgodnie z naszymi oczekiwaniami.

Przyszła kolej na kodowanie stanów – aby ułatwić sobie pracę, stan A będzie reprezentowany zerem, a stan B – jedynką. W ten sposób obędziemy się bez dekodera, który w innym przypadku musiałby być dołączony do wyjścia automatu. Przygotujmy zatem funkcje wzbudzeń przerzutnika. Mamy tylko dwa stany i możemy je zapisać na jednym bicie, czyli za pomocą pojedynczego przerzutnika (tym razem JK).

Spójrz teraz na **rysunek 4**. Widoczna tam tabela powstała według wcześniej poznanych zasad. Na początku wypisujemy numery wszystkich stanów (kolumna niebieska). Będą to stany początkowe, czyli stany przed podaniem sygnału zegarowego na przerzutnik. Następnie (obok) wypisujemy wszystkie wartości, jakie może przyjąć słowo wejściowe – w tym wypadku jest ono 1-bitowe.

Teraz wypełniamy kolumnę szarą, patrząc na graf z rysunku 3. Zaczynamy od sprawdzenia wartości w pierwszej kolumnie i w pierwszym wierszu. Jest tam 0, czyli stan zakodowany zerem, na rysunku 3 widzimy, że jest to stan A. W kolumnie obok (kolor zielony) znajduje się zero, czyli patrzymy, gdzie przejdziemy po podaniu zera na wejście, gdy przebywamy w stanie zerowym (A). Strzałka pokazuje, że zostajemy w stanie A który ma kod zero, zatem zero zapisujemy do kolumny szarej. Podobnie postępujemy z pozostałymi wierszami. W ostatnim wierszu mamy stan zakodowany jedynką, czyli stan B. Patrzymy na graf (rys. 3) – widać, że gdy jesteśmy w stanie B i na wejściu pojawi się jedynka, to przechodzimy do stanu A, który ma kod 0, zatem do szarej kolumny wpisujemy zero.

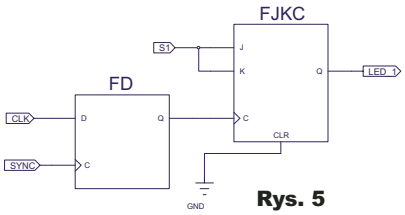
Rys. 4

stan-	we	stan+	J	K
0	0	0	0	x
0	1	1	1	x
1	0	1	x	0
1	1	0	x	1

JK
00 01
10 11
00 10
01 11

Zasada wypełnienia kolumn żółtych nie odbiega od zasad poznanych w poprzednim odcinku kursu. Przerzutnik JK ma dwa wejścia, musimy podać na nie takie sygnały, aby uzyskać wartość na wyjściu zgodną z zawartością kolumny *stan+*. Przykładowo, jakie wymuszenia podać na wejścia J oraz K, aby ze stanu 0 (kolumna *stan-*) przejść do stanu 0 (*stan+*)? Możemy podać dwa zera (J = 0, K = 0), co będzie odpowiadać podtrzymaniu stanu poprzedniego albo podać zero i jeden (J = 0, K = 1) w wyniku czego nastąpi przepisanie z J, bo J jest różne od K. Wniosek jest prosty: wejście J musi mieć stan niski, a wartość

wejścia K nie ma znaczenia. Podobnie wypełnia się pozostałe pola. W niebieskich „dymkach" podano możliwe stany wejść przerzutnika, które gwarantują osiągnięcie pożądanej wartości na wyjściu.



Rys. 5

Ostatni etap to wyznaczenie funkcji boole'owskich. Szczerze powiedziawszy nie ma nawet po co rysować tablicy Karnaugha, bo od razu widać, że wejście J i K można połączyć bezpośrednio z wejściem danych.

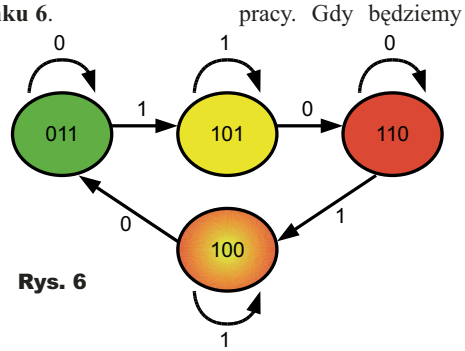
Mogliśmy zastosować także przerzutnik D, ale czy będzie on lepszy? Czy układ byłby aż tak prosty? Pozostawię to Czytelnikom do sprawdzenia. Implementacja tego prostego automatu (z przerzutnikiem JK) została przedstawiona na **rysunku 5**. Sygnał zegarowy jest pobierany z przestrajanego generatora, natomiast przyciskiem S1 możemy wprowadzać dane, czyli symulować port szeregowy. Przerzutnik D odpowiada za usuwanie ewentualnych zakłóceń.

Sygnalizacja świetlna

Spróbujmy teraz przygotować model sygnalizacji świetlnej. Znane z polskich dróg sygnalizacje mają cztery stany: *zielony*, *żółty*, *czerwony* oraz *żółto-czerwono*. Założymy, że sygnałem przełączającym będą impulsy zegarowe pochodzące z przestrajanego generatora. Pracę rozpoczniemy od narysowani grafu. W tym wypadku jest on bardzo prosty, gdyż nie są podejmowane żadne decyzje, a jedynie jest realizowana prosta ścieżka przejścia przez wszystkie stany. Należy jednak zwrócić uwagę na zapewnienie stabilnych warunków pracy.

Nie będzie dobrym pomysłem założenie, że jedynka powoduje zmianę stanu, a zero pozostanie w stanie obecnym, gdyż jakość pracy jest wtedy uzależniona od sygnału zegarowego doprowadzonego do przerzutników. Lepszym wyjściem będzie sterowanie sygnalizacją za pomocą poziomów logicznych sygnału generatora. Pierwsza zmiana stanu automatu odbywać się będzie po zmianie poziomu logicznego. Przykładowo – ze stanu *zielony* przejdziemy do stanu *żółty* w momencie, gdy wejście zmieni poziom z niskiego na wysoki. W stanie *żółty* będziemy przebywać do czasu kolejnej zmiany stanu wejścia, tym razem na poziom niski. Taki sposób pracy automatu przedstawiono na **rysunku 6**.

Dokładność sygnału taktującego przerzutniki nie ma znaczenia, musi mieć on jedynie większą częstotliwość, aby nie wprowadzał widocznego opóźnienia w przełączaniu.
Rysunku 6 kryje w sobie jeszcze jedną sztuczkę,



Rys. 6

bo stany są zakodowane w sposób nieoptymalny. Dzięki temu nie będzie potrzebny wyjściowy dekoder i diody będzie można podłączyć bezpośrednio do wyjść przerzutników. Wynika to z przypisania do każdego koloru zera na jednej z pozycji kodu reprezentującego stany automatu. Przykładowo światło żółte ma stany zakodowane w taki sposób, iż środkowy bit ma zero. Po przyłączeniu diody LED do środkowego przerzutnika będzie ona zaświecona za każdym razem, gdy automat wejdzie do stanu *żółty* bądź *żółto-czerwony*.

Straty zasobów nie są duże (jeden przerzutnik), a oszczędność naszego czasu jest spora, gdyż nie trzeba będzie tworzyć trzech tablic Karnaugha (bo tyle diod sterujemy) dla dwóch zmiennych (bo na tylu bitach można zapisać poszczególne stany).

Mając opracowany graf, możemy przygotować tablicę wzbudzeń przerzutników. Przyjmijmy, że będą to przerzutniki D. W zasadzie nie ma tu nic nowego poza tym, że pojawiły się stany niewykorzystane. Co z nimi zrobić? Są dwie opcje. Po pierwsze można założyć, że nigdy nie wystąpią i wypełnić je znakami *don't care*. Analogiczna sytuacja miała miejsce przy budowie dekodera 7-segmentowego, tam również część stanów (od 10 do 15) nie występowała. Takie podejście uproszcza projektowanie układu, ale ewentualne błędy lub zakłócenia spowodują nieprzewidywane zachowanie automatu. Łatwo podać konkretny przykład – na skutek zakłócenia sygnalizacja wchodzi w stan 000. Nie został on przez nas uwzględniony, a jego skutek jest oczywisty – zapalą się natychmiast wszystkie światła. Co będzie dalej? To zależy od naszego szczęścia, w najlepszym przypadku nastąpi szybkie przejście do poprawnego stanu i powrót do normalnej pracy. Gdy będziemy

	stan-	we	stan+	Da	Db	Dc
	000	0	100	1	0	0
	000	1	100	1	0	0
	001	0	101	1	0	1
	001	1	101	1	0	1
	010	0	011	0	1	1
	010	1	011	0	1	1
OK	011	0	011	0	1	1
OK	011	1	101	1	0	1
OK	100	0	011	0	1	1
OK	100	1	100	1	0	0
OK	101	0	110	1	1	0
OK	101	1	101	1	0	1
OK	110	0	110	1	1	0
OK	110	1	100	1	0	0
	111	0	110	1	1	0
	111	1	110	1	1	0

Rys. 7

mieli tego szczęścia mniej, automat nigdy nie wróci na prawidłową ścieżkę i np. zatrzaśnie się w którymś z zabronionych stanów. Błąd wystąpi również po uruchomieniu układu – przerzutniki będą wyzerowane, a stan 000 jest niedopuszczalny. Czy można temu przeciwdziałać? Oczywiście! Należy jasno zdefiniować przejścia ze stanów niedozwolonych do stanów dozwolonych. Skutkiem tego będzie wzrost niezawodności urządzenia, ale kosztem zwiększenia wymaganych zasobów logicznych. Podobna sytuacja występowała przy liczniku pierścieniowym, tam również zaimplementowaliśmy mechanizm zapobiegający występowaniu dwóch albo więcej jedynek oraz samych zera.

Kolejne pytanie, jakie należy sobie postawić, to do jakiego stanu przejść po wystąpieniu stanu niedozwolonego? Istnieje tu pewna dowolność uzależniona od rodzaju implementowanego urządzenia. Można np. założyć, że po wystąpieniu błędu należy zatrzymać cały ruch na drodze i włączyć światło czerwone. Można również przyjąć, że przechodzimy do najbliższego, dozwolonego stanu. Przyjąłem tę drugą opcję, która zaowocowała powstaniem tabeli wzbudzeń przerzutników widocznej na **rysunku 7** (słowo *OK* oznacza stan poprawny).

Na tej podstawie możemy przygotować tablice Karnaugha (**rysunek 8**), wyznaczyć funkcje boole'owskie (Da, Db, Dc) i przystąpić do implementacji układu (**rysunek 9**). Oznaczenia Qa, Qb, Qc oznaczają wyjścia przerzutników, odpowiednio Da (MSB), Db, Dc (LSB).

Działanie układu nie odbiega od tego, czego się spodziewaliśmy – sygnalizacja włącza kolejno poszczególne światła na wzór sygnalizacji spotykanej na polskich drogach.

Qa Qb \ Qc we	00	01	11	10
00	1	1	1	1
01	0	0	1	0
11	1	1	1	1
10	0	1	1	1

Da= Qc*we + !Qb*Qc + !Qb*we + + Qa*Qb + !Qa*!Qb

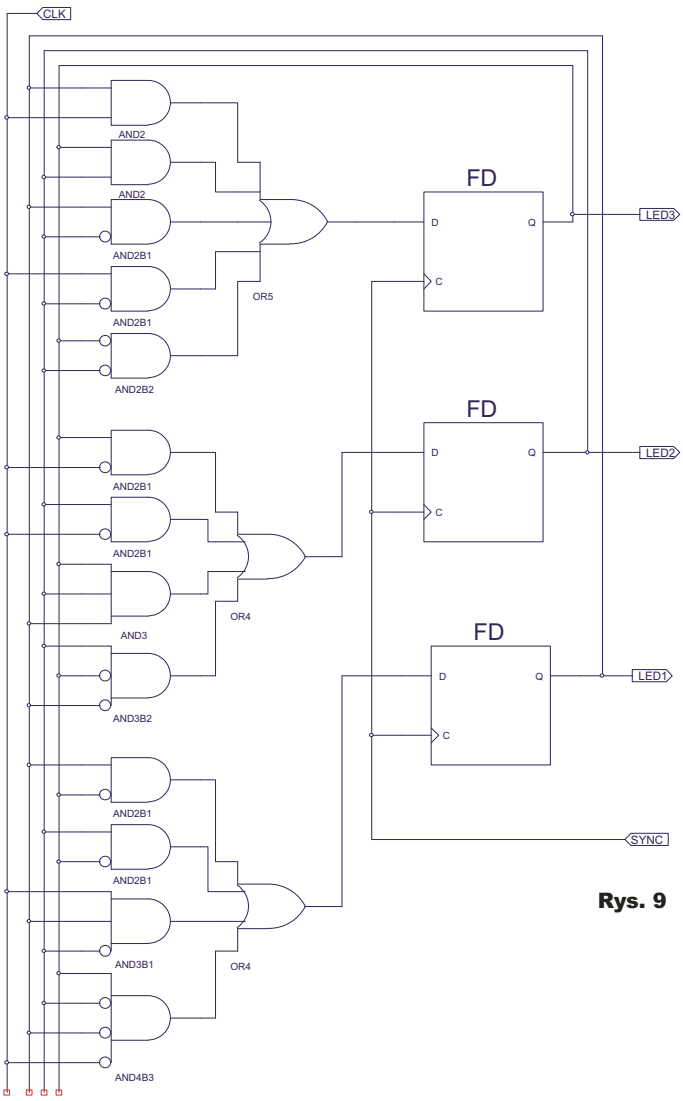
Qa Qb \ Qc we	00	01	11	10
00	0	0	0	0
01	1	1	0	1
11	1	0	1	1
10	1	0	0	1

Db= Qa*!we + Qb*!we + Qa*Qb*Qc + + !Qa*Qb*!Qc

Qa Qb \ Qc we	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	0	0	0	0
10	1	0	1	0

Dc= !Qa*Qc + !Qa*Qb + !Qb*Qc*we + + Qa*!Qb*!Qc*!we

Rys. 8



Rys. 9

Mankamentem jest brak kolorowych diod, ale gdyby ktoś chciał, może podmienić:).

Sygnalizacja świetlna ze stanem awaryjnym

Spróbujmy nieco rozbudować poprzedni przykład. Dodamy konkretnie stan awaryjny, czyli pulsowanie żółtego światła od momentu naciśnięciu S1 do chwili naciśnięcia S2.

Na początek przygotujmy graf przejść. Jest on bardzo podobny do grafu z rysunku 6. Potrzebny będzie jednak, oprócz sygnału wymuszającego zmianę świateł w trybie normalnym, dodatkowy sygnał zegarowe odpowiedzialny za pulsowanie światła w trybie awaryjnym. Właściwy graf przedstawiono na **rysunku 10**. Dwa najstarsze bity sygnału wejściowego reprezentują stan przycisków S1 oraz S2, a dwa najmłodsze sygnały zegarowe: *CLK_PULS* (pulsowania żółtego światła w trybie awaryjnym) oraz *CLK_ZM* (czas świecenia świateł w trybie normalnym). Stany *Y_ON* oraz *Y_OFF* odpowiadają za pracę w trybie awaryjnym. W pierwszym z nich światło żółte jest włączone, a w drugim – wyłączone. Wejście do tych stanów odbywa się po naciśnięciu przycisku S1, co odpowiada ręcznemu „wyłączeniu" sygnalizacji (spotykane w miastach najczęściej w nocy i/lub przy małym ruch). Uważni Czytelnicy zapewne zauważyli przejście ze stanu *G* do *Y_OFF* zamiast do *Y_ON* jak w pozostałych przypadkach. Wynika to tylko i wyłącznie z chęci uproszczenia rysunku. Naciśnięciu S1 towarzyszy włączenie trybu „awaryjnego" (znaki *x* odzwierciedlają ignorowanie pozostałych bitów w słowie wejściowym – don't care) i cykliczne przechodzenie pomiędzy stanami Y_ON oraz Y_OFF w takt zegara *CLK_PULS*. Powrót do normalnej pracy odbywa się po zwolnieniu S1 i naciśnięciu S2.

Chcąc przygotować tabelkę wzbudzeń przerzutników, napotkamy pewien problem: potrzebne będą trzy rejestry do przechowywania stanu automatu i cztery wejścia. Daje to w efekcie siedem zmiennych. Minimalizacja za pomocą tablicy Karnaugha będzie raczej trudna, natomiast metoda Quine'a-McCluskeya będzie długa i żmudna. Możemy oczywiście poczynić pewne uproszczenia, aby ułatwić sobie pracę, ale można cały automat przygotować w VHDL-u. Z jego implementacją Czytelnicy powinni sobie poradzić, wykorzystując poznane dotychczas metody. Stan obecny może być reprezentowany przez zmienną 3-bitową, wyzwalanie sygnałem zegarowym realizuje proces z sygnałem *SYNC* na liście czułości, a poszczególne przejścia można zdefiniować za pomocą warunków *if*. Okazuje się, że całość można jeszcze bardziej uprościć – spójrz na **listing 1**. Na samym początku bloku opisującego architekturę pojawiają się dwie dziwne linie:

type moje_stany is (R, Y, G, YR, Y_ON, Y_OFF);

signal stan: moje_stany ;

W pierwszej linii definiujemy nowy typ, nazwany tutaj *moje_stany*. Definicja zaczyna się słowem kluczowym *type*, następnie podajemy dowolną nazwę, wstawiamy słowo kluczowe *is* i w nawiasie wymieniamy wszystkie stany jakie mieć będzie nasz automat. Jak widać, wymieniono wszystkie stany występujące w grafie z rysunku 10.

Linijkę niżej definiujemy sygnał przeznaczony do przechowywania aktualnego stanu automatu. Jest to znana już składnia, z tym że zamiast typu *std_logic* podstawiamy zdefiniowany przez nas typ. Co nam to daje? Kodowanie poszczególnych stanów spoczywa teraz na barkach środowiska WebPACK ISE – ponownie oszczędzamy trochę czasu. Zmiana aktualnego stanu sprowadzać się będzie do podstawienia pod zmienną *stan* innego, wybranego stanu za pomocą operatora <= (np. *stan <= Y_ON*).

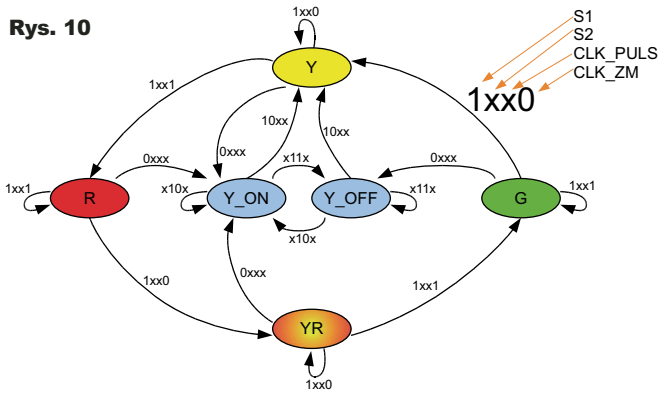
Można teraz utworzyć proces z sygnał SYNC na liście czułości. W jego wnętrzu opiszemy zachowanie automatu. Tutaj czeka kolejna niespodzianka w postaci nowej struktury – ***case***.

Jej szablon jest następujący:

etykieta: ***case*** *wyrazenie* ***is***
when *opcja1 =>*
instrukcje;
when *opcja2 =>*
instrukcje;
(...)
when others *=>*
instrukcje;
end case *etykieta ;*

Etykieta jest opcjonalna, podobnie jak wyrażenie *when others* i następujące po nim instrukcje (ale POD WARUNKIEM, że wyczerpaliśmy wszystkie możliwe opcje). Przełóżmy to na naszą sygnalizację. Pod *wyrazenie* podstawiamy sygnał *stan* określający aktualny stan automat. Za poszczególne opcje (tj. *opcja1, opcja2*, etc) podstawiamy wszystkie stany zdefiniowane w utworzonym wcześniej typie *moj_typ*. Warto zauważyć, że nie stosujemy przy nazwach stanów żadnych apostrofów czy cudzysłowów. Na listingu 1 nie pojawiły się słowa kluczowe *when others*, gdyż zostały wyczerpane wszystkie możliwość – nie istnieje stan automatu, który nie zostałby wyszczególniony w ramach instrukcji *when*. Można jednak rozbudować strukturę *case* o to opcjonal-

Rys. 10



Listing 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity sygnalizacja_pelna is
  Port ( S1 : in std_logic;
         S2 : in std_logic;
         CLK_ZM : in std_logic;
         CLK_PULS : in std_logic;
  SYNC : in std_logic ;
  LED_R : out std_logic ;
  LED_Y : out std_logic ;
  LED_G : out std_logic );
end sygnalizacja_pelna;
architecture Behavioral of sygnalizacja_pelna is
  type moje_stany is (R, Y, G, YR, Y_ON, Y_OFF);
  signal stan : moje_stany ;
begin
  sygnalizacja: process (SYNC) is begin
    if rising_edge(SYNC) then
      case stan is
        when R =>
          if S1 = '0' then
            stan <= Y_ON  ;
          elsif CLK_ZM = '0' then
            stan <= YR ;
          end if;
        when YR =>
          if S1 = '0' then
            stan <= Y_ON  ;
          elsif CLK_ZM = '1' then
            stan <= G ;
          end if;
        when G =>
          if S1 = '0' then
            stan <= Y_OFF  ;
          elsif CLK_ZM = '0' then
            stan <= Y ;
          end if;
        when Y =>
          if S1 = '0' then
            stan <= Y_ON  ;
          elsif CLK_ZM = '1' then
            stan <= R ;
          end if;
        when Y_ON =>
          if S2 = '0'then
            stan <= Y  ;
          elsif CLK_PULS = '1' then
            stan <= Y_OFF ;
          end if;
        when Y_OFF =>
          if S2 = '0'then
            stan <= Y  ;
          elsif CLK_PULS = '0' then
            stan <= Y_ON ;
          end if;
        end case ;
    end if ;
  end process sygnalizacja ;
  --obsluga diod wyjsciowych
  LED_Y <= '0' when (stan=Y) or (stan=YR) or
                              (stan=Y_ON) else '1' ;
  LED_R <= '0' when (stan=R) or (stan=YR)
                                        else '1' ;
  LED_G <= '0' when (stan=G) else '1' ;
end Behavioral;
```

ne wyrażenie, aby zapisać tam domyślny stan na wypadek błędu.

Teraz posłużymy się instrukcjami *if*, aby opisać zachowanie automatu. Poniższy fragment kodu:

when R =>
if S1 = '0' then
stan <= Y_ON ;
elsif CLK_ZM = '0' then
stan <= YR ;
end if;

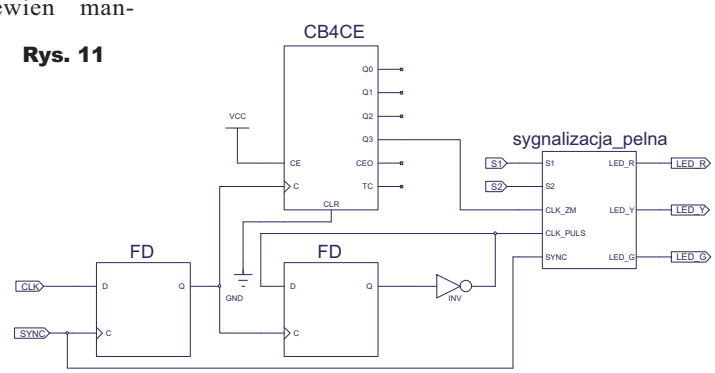
odpowiada za stan, w którym włączona jest żarówka czerwona sygnalizacji. Pierwsza instrukcja sprawdza, czy sygnał S1 ma wartość zero, co zgodnie z grafem przejść spowoduje skok do stanu *Y_ON* odpowiadającego za pracę w trybie awaryjnym. Warto zauważyć, że zapis 0xxx daje priorytet wejściu S1 – bez względu na to, co jest na pozostałych wejściach, automat ma przejść do stany *Y_ON*. Drugi warunek (*elsif CLK_ZM = '0' then*) sprawdza, czy zmienił się poziom sygnału *CLK_ZM* i jeżeli tak, następuje przejście do stanu odpowiedzialnego za równoczesne włączenia światła czerwonego i żółtego. A co z pozostałymi wartościami sygnałów wejściowych? W zasadzie nic, jeżeli czegoś nie przewidzieliśmy, to automat nie zmieni swojego stanu i będzie oczekiwał na pojawienie się zera na wejściu S1 bądź CLK_ZM.

Analogicznie zrealizowano obsługę pozostałych stanów. Potrzebny jest jeszcze dekoder do sterowania diodami LED uzależniony od obecnego stanu automatu. Do tego celu nadaje się struktura *when*, którą poznaliśmy już wcześniej. Jedyną nowością jest to, że nie operujemy na liczbach binarnych, tylko na nazwach stanów. Reszta pozostaje taka sama – możemy sygnał *stan* porównywać ze stanami określonymi w typie *moj_stan* i podejmować decyzje w zależności od wyniku tych porównań.

Po zaprojektowaniu automatu najwygodniej będzie utworzyć element – zaznaczamy plik *.vhdl i klikamy *Create Schematic Symbol*. Następnie dodajemy go do schematu i kilka elementów dodatkowych – **rysunek 11**. Pierwszy przerzutnik ma za zadanie, standardowo, usunąć ewentualne zakłócenia. Rola drugiego przerzutnika jest już mniej oczywista i wynika ze sposobu wykorzystania sygnału zegarowego. Tym razem nie wykrywamy pojedynczego zbocza, ale badamy OBA poziomy logiczne, więc w sytuacji, gdy będą miały one różny czas trwania (wypełnienie inne niż 50%, co jest bardzo prawdopodobne w układzie NE555), diody będą świeciły nierównomiernie. Drugi przerzutnik z bramką NOT rozwiązuje ten problem, dokonując symetryzacji przebiegu CLK. Licznik 4-bitowy dzieli przebieg zegarowy, aby światła w normalnym trybie pracy nie zmieniały się tak szybko.

Układ posiada pewien mankament, mianowicie światło żółte w trybie pracy normalnej świeci się tak samo długo jak pozostałe światła, co nie ma miejsca w rzeczywistej sygnalizacji. Opracowanie poprawki pozostawiam jednak Czytelnikom w ramach ćwiczenia.

Rys. 11



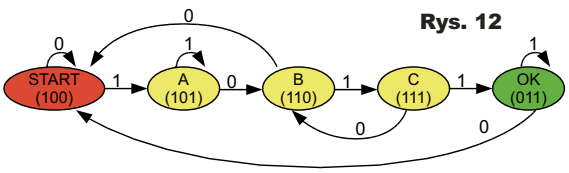
Zamek szyfrowy

Budowa zamka szyfrowego nie jest trudniejsza niż budowa dotychczasowych automatów, więc dla urozmaicenia zaimplementujemy go na przerzutnikach typu T. Podniesie to walory edukacyjne tego przykładu. Zamek szyfrowy można zrealizować na dwa sposoby. Po pierwsze, można go zaprojektować tak, że przy pierwszej pomyłce wróci on do stanu początkowego i trzeba będzie wprowadzać ciąg w całości jeszcze raz. Drugie podejście daje możliwość wprowadzenia ciągu w dowolnym momencie, tzn. sekwencje bitów mogą się na siebie nakładać. Dla przykładu załóżmy, że tajny kod to 1011. W pierwszym przypadku po wprowadzeniu ciągu 11 musimy od nowa wpisać cały kod, czyli 1011, a w drugim przypadku po wpisaniu 11 możemy dopisać tylko 011 i automat przejdzie w stan *otwarte*, ponieważ wystąpi pożądana sekwencja: 11011. Zamek z nakładającymi się sekwencjami jest, moim zdaniem, ciekawszy i oferuje większą niezawodność i przewidywalność pracy. Bez nakładających się sekwencji może być trudno określić, kiedy układ jest gotowy przyjąć kod i w jakim stanie obecnie się znajduje. Przyjmijmy, że hasłem będzie 1011. Nic nie stoi na przeszkodzie, aby przygotować automat akceptujący dłuższe hasła, ale pozostawiam to Czytelnikom. Ten przykład zostanie ograniczony do 4 bitów, aby był łatwiejszy do zrozumienia.

Tok postępowania jest standardowy – zaczynamy od narysowania grafu przejść. Dla uproszczenia przyjęto, że kod będzie wprowadzany w takt sygnału zegarowego pochodzącego z przestrajanego generatora.

Graf projektowanego zamka jest widoczny na **rysunku 12**. Warto zauważyć, że pomyłka przy wprowadzeniu kodu nie zawsze powoduje powrót do stanu początkowego (*START*). Przykładowo, ze stanu C przechodzi się do stanu B w sytuacji podania ciągu 1010, ponieważ stan B reprezentuje sytuację, w której wprowadzono poprawnie część kodu – w tym przypadku 10. Po wprowadzeniu dalszej części (11) zamek otworzy się, bo wprowadzona została sekwencja 101011, a cztery ostatnie bity stanowią prawidłowy ciąg bitów.

Rys. 12



stan-	we	stan+	Ta	Tb	Tc
000	0	100	1	0	0
000	1	100	1	0	0
001	0	100	1	0	1
001	1	100	1	0	1
010	0	100	1	1	0
010	1	100	1	1	0
011	0	100	1	1	1
011	1	011	0	0	0
100	0	100	0	0	0
100	1	101	0	0	1
101	0	110	0	1	1
101	1	101	0	0	0
110	0	100	0	1	0
110	1	111	0	0	1
111	0	110	0	0	1
111	1	011	1	0	0

Rys. 13

Na rysunku 12 pokazano także propozycje kodowania poszczególnych stanów. Jest ich pięć, zatem dwa bity nie wystarczą i konieczne będą trzy. Ponownie pojawia się trik przy kodowaniu stanów – ostatni stan (OK) odpowiadający za otwarcie drzwi jako jedyny ma zero na najbardziej znaczącej pozycji. Możemy zatem do odpowiedniego przerzutnika podłączyć bezpośrednio diodę symulującą zamek. Dzięki temu nie będzie potrzebny dekoder, co jak zwykle oszczędzi nam trochę pracy...

Tabelę wzbudzeń przerzutników przedstawiono na **rysunku 13**. Stany automatu kodowane są na trzech bitach, więc będą potrzebne trzy przerzutniki, w tym wypadku będą one typu T (zaznaczę dla jasności, że równie dobrze może być D, jak i JK).

Aby rozwiać ewentualne wątpliwości, prześledźmy, jak powstał ostatni wiersz w tabeli z rysunku 13. Znajduje się w nim stan 111 przechodzący w 011, gdy na wejściu pojawi się jedynka. Jakie zatem wpisać wartości do kolumn Ta, Tb, Tc?

Kolumna Ta reprezentuje najstarszy bit, który zmienia się z 1 na 0, zatem zgodnie z zasadą pracy przerzutnika T, musimy wpisać jedynkę do kolumny Ta. Spowoduje ona negację i jeden „zamieni się" na zero, zgodnie z naszymi oczekiwaniami. W przypadku dwóch młodszych bitów mamy do czynienia z dwoma jedynkami – przed zmianą stanu (stan-) i po zmianie stanu (stan+), czyli nic się nie zmienia. Takie zachowanie gwarantuje podanie zera na wejścia przerzutników Tb i Tc, zatem wpisujemy tam zera. Na podobnej zasadzie wypełniona została reszta tabeli. Nie ma w niej znaków *don't care* ze względu na nieprzewidywalność pracy takiego automatu.

Teraz można wyznaczyć funkcje boole'owskie dla poszczególnych przerzutników. Do tego celu przydadzą się tablice Karnaugha – **rysunek 14**. Mając funkcje Ta, Tb oraz Tc, można zaimplementować automat – **rysunek 15**.

Obsługa zamka nie jest trudna. W momencie zaświecenia LED1 naciskamy S1 celem podania zera logicznego i puszczamy po zgaszeniu diody. Chcąc podać jedynkę logiczną, nie robimy nic.

Qa Qb \ Qc we	00	01	11	10
00	1	1	1	1
01	1	1	0	1
11	0	0	0	0
10	0	0	0	0

Ta=!Qa*!we + !Qa*!Qb + !Qa*!Qc

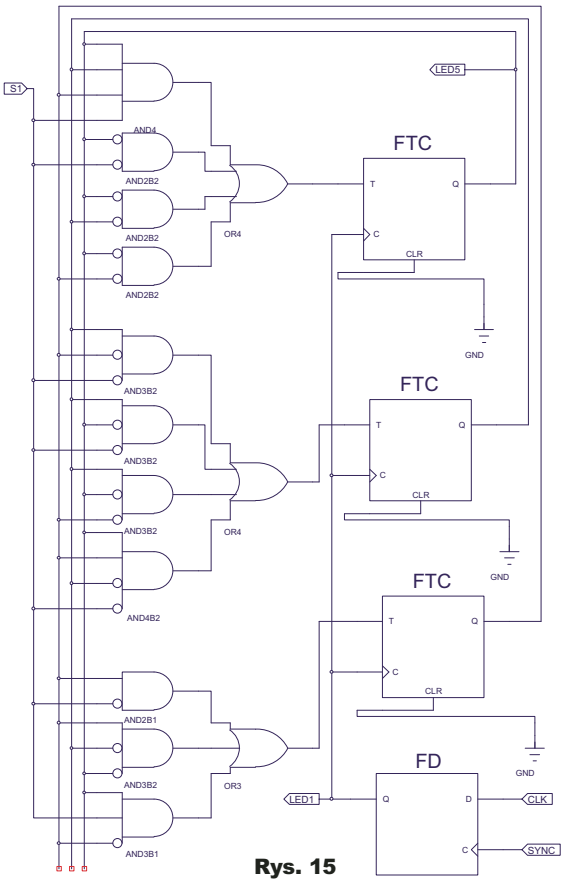
Qa Qb \ Qc we	00	01	11	10
00	0	0	0	0
01	1	1	0	1
11	1	0	1	0
10	0	0	0	1

Tb= Qb*!Qc*!we + !Qa*Qb*!we +
+ !Qa*Qb*!Qc + Qa*Qb*Qc*we +
+ Qa*!Qb*Qc*!we

Qa Qb \ Qc we	00	01	11	10
00	0	0	1	1
01	0	0	0	1
11	0	1	1	1
10	0	1	0	1

Tc= Qc*!we + Qa*Qb*Qc +
+ !Qa*!Qb*Qc + Qa*!Qc*we

Rys. 14



Rys. 15

Mam nadzieję, że ten przykład pozwolił zrozumieć zasadę pracy zamka szyfrowego i że... pozostawił uczucie pewnego niedosytu. W zasadzie kod jest bardzo łatwy do złamania i wprowadzany w niezbyt wygodny sposób. Zachęcam do samodzielnych eksperymentów i prób budowy lepszych wersji zamka. Warto pokusić się o rozdzielenie sterowania na dwa przyciski – pierwszy będzie podawał zero, a drugi jedynkę. Pozwoli to uniezależnić się od sygnału zegarowego. Potrzebne będą jednak stany pośrednie oczekujące na zwolnienie obu przycisków. Należałoby też zwrócić uwagę na problem filtracji drgań zestyków, gdyż może być on przyczyną nieprawidłowej pracy układu.

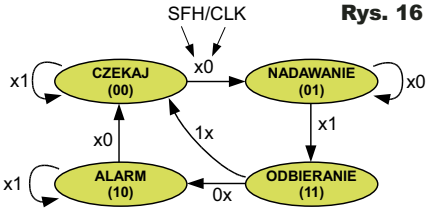
Zamek szyfrowy z barierą podczerwieni

Nic nie stoi na przeszkodzie, aby do schematu wstawić więcej niż jeden automat. Pierwszym może być zamek szyfrowy opracowany przed chwilą, a drugim bariera podczerwieni. W ten sposób może powstać trzeci automat bazujący na dwóch poprzednich. Hierarchiczna budowa ułatwia pracę i pozwala tworzyć bardziej zaawansowane rozwiązania. Obecnie nie dysponujemy automatem do sterowania barierą podczerwieni, zatem musimy go opracować. Potrzebne będą następujące stany pracy:

– czekaj, określający stan przerwy w nadawaniu wiązki IR (patrz piata część kursu),
– nadawanie, czyli emisja wiązki podczerwieni, przyjmijmy, że ten stan będzie włączał generator 36kHz dla diody IR,
– odbieranie, czyli sprawdzenie, czy światło podczerwone uległo odbiciu, czy nie,
– alarm, stan występujący, gdy zostanie stwierdzone odbicie podczerwieni (ktoś przeszedł przez barierę).

Wyjście stanowić będzie sygnał informujący o naruszeniu chronionego obszaru.

Stosowny graf można zobaczyć na **rysunku 16**. Praca automatu jest oparta o sygnał zegarowy CLK (młodszy bit). Określa on bowiem moment rozpoczęcia nadawania, odbierania i czas przebywania w stanie *czekaj*. Pewną ciekawostką jest stan *odbieranie*, gdyż nie jest on zapętlany – automat przebywa w nim tylko chwilę. Po zakończeniu nadawania (stan CLK zmienia się z niskiego na wysoki) sprawdzany jest stan odbiornika podczerwieni SFH (starszy bit) i podejmowana jest decyzja o przejściu do stanu alarmowego (gdy bariera została naruszona) lub dalszym oczekiwaniu (gdy bariera nie została naruszona). Warto zauważyć, że automat w stanie *alarm* przebywa tylko określony czas (konkretnie tyle, ile trwa stan wysoki sygnału CLK), potem



Rys. 16

stan-	SFH/CLK	stan+	Da	Db
00	00	01	0	1
00	01	00	0	0
00	10	01	0	1
00	11	00	0	0
01	00	01	0	1
01	01	11	1	1
01	10	01	0	1
01	11	11	1	1
10	00	00	0	0
10	01	10	1	0
10	10	00	0	0
10	11	10	1	0
11	00	10	1	0
11	01	10	1	0
11	10	00	0	0
11	11	00	0	0

Rys. 17

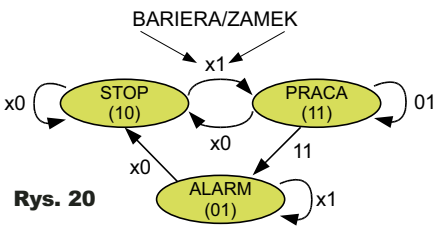
Qa Qb \ SFH CLK	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	1	1	0	0
10	0	1	1	0

Da= Qa*!Qb*CLK + !Qa*Qb*CLK + + Qa*Qb*!SFH

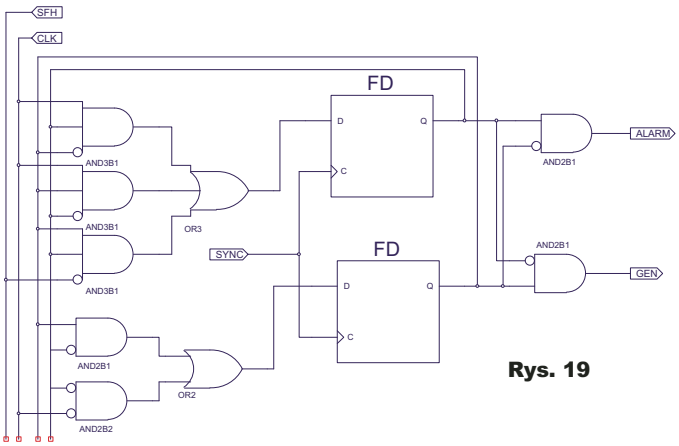
Qa Qb \ SFH CLK	00	01	11	10
00	1	0	0	1
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

Db=!Qa*Qb + !Qa*!CLK

Rys. 18



Rys. 20



Rys. 19

stan-	bar/zam	stan+	Da	Db
00	00	10	1	0
00	01	10	1	0
00	10	10	1	0
00	11	10	1	0
01	00	10	1	0
01	01	01	0	1
01	10	10	1	0
01	11	01	0	1
10	00	10	1	0
10	01	11	1	1
10	10	10	1	0
10	11	11	1	1
11	00	10	1	0
11	01	11	1	1
11	10	10	1	0
11	11	01	0	1

Rys. 21

Qa Qb \ bar zam	00	01	11	10
00	1	1	1	1
01	1	0	0	1
11	1	1	0	1
10	1	1	1	1

Da= !zam + !Qb + Qa*!bar

Qa Qb \ bar zam	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

Db= Qa*zam + Qb*zam

Rys. 22

jego praca zaczyna się od początku. Zapobiegamy w ten sposób „zacięciu" się automatu. Sygnał CLK powinien mieć częstotliwość paruset herców (patrz część 5 kursu).

Po przygotowaniu grafu (na rysunku 16 zaproponowano także sposób kodowania stanów) można przygotować tabelkę wzbudzeń przerzutników (**rysunek 17**). Użyliśmy dwóch przerzutników typu D do pamiętania poszczególnych stanów. Stosowne tablice Karnaugha widoczne są na **rysunku 18**. Następnym krokiem jest implementacja tego automatu (**rysunek 19**) i utworzenie z niego symbolu bibliotecznego. Należy zauważyć, że posiada on DEKODERY złożone z bramek AND. Jeden z nich dekoduje stan *alarm* reprezentowany kodem 10, a drugi stan *nadawanie* reprezentowany przez 01.

Do wejść CLK, SFH oraz SYNC doprowadzamy sygnały z odpowiednich podzespołów umieszczonych na płytce testowej. Mniej oczywiste może być znaczenie wyjść ALARM oraz GEN. Pierwsze z nich sygnalizuje jedynką naruszenie bariery podczerwieni, natomiast drugie wyjście służy do kluczowania generatora 36kHz dla diody IR.

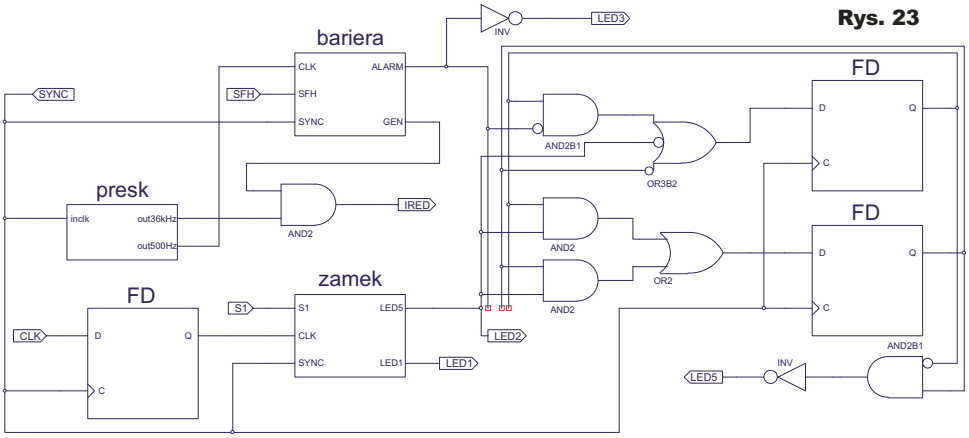
Ostatnim etapem jest utworzenie docelowego automatu wykorzystującego informacje dostarczane przez barierę i zamek szyfrowy, co pozwoli stworzyć alarm zabezpieczony kodem dostępu.

Zaczynamy od narysowania grafu – **rysunek 20**. Jego działanie jest proste – do momentu, kiedy zamek jest „otwarty" (na wyjściu przerzutnika reprezentującym najstarszy bit jest zero) stan bariery podczerwieni nie ma znaczenia. Można w takiej sytuacji dowolnie ją przekraczać. Po uzbrojeniu alarmu młodszy bit słowa wejściowego z rysunku 20 przyjmuje wartość 1 i od tego momentu przekroczenie bariery włączy alarm. Po wykryciu wtargnięcia automat przechodzi w stan *alarm* i przebywa w nim do chwili wprowadzenia poprawnego kodu (wtedy najmłodszy bit przyjmie wartość zero). Warto zauważyć, że zamek może zostać rozbrojony bez włączania alarmu – wprowadzenie poprawnego kodu w stanie *pracuj* spowoduje przejście do stanu początkowego (*stop*), w którym bariera jest nieaktywna.

Po opracowaniu grafu tworzymy tabelę (**rysunek 21**), tablice Karnaugha i wyznaczamy funkcje boolowskie (**rysunek 22**). Sygnał *zam* reprezentuje wyjście z automatu zamka szyfrowego, natomiast sygnał *bar* – automat bariery podczerwieni. Na końcu implementujemy całe urządzenie – **rysunek 23**. Pierwszy od lewej przerzutnik D usuwa ewentualne zakłócenia z przebiegu CLK, a dwa pozostałe przechowują obecny stan automatu. Symbol nazwany *bariera* jest automatem z rysunku 16, a element *zamek* automatem z rysunku 12. Oba te automaty zostały zamienione na symbole i wstawione do schematu, co podniosło czytelność całego projektu. Układ *presk* został opisany w VHDL-u (**listing 2**) i dokonuje on podziału sygnału generatora kwarcowego w taki sposób, aby uzyskać częstotliwości 36kHz (kluczowanie diody IR) oraz 500Hz (odstępy kluczowania diody IR).

Stan automatu wizualizują ponadto cztery diody LED:

– LED1 – reprezentuje przebieg zegarowy CLK, w momencie, gdy świeci, należy nacisnąć S1,



Rys. 23

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity presk is
  Port ( inclk : in std_logic;
  out36kHz : out std_logic;
  out500Hz : out std_logic);
end presk;
architecture Behavioral of presk is
begin
  prc: process (inclk) is
    variable temp : std_logic_vector
                    (9 downto 0) ;
    variable temp2 : std_logic_vector
                     (15 downto 0) ;
  begin
    if  rising_edge(inclk) then
      --preskaler dla 36kHz
      temp := temp + 1 ;
      if temp=333 then
        out36kHz <= '1' ;
      elsif temp=666 then
        out36kHz <= '0';
        temp := "0000000000" ;
      end if ;
      --preskaler dla 500Hz
      temp2 := temp2 + 1 ;
      if temp2=24000 then
        out500Hz <= '1' ;
      elsif temp2=48000 then
        out500Hz <= '0';
        temp2 := "0000000000000000" ;
      end if ;
    end if;
  end process ;
end Behavioral;
```

Listing 2

aby wprowadzić do zamka szyfrowego zero logiczne (analogicznie jak wcześniej),
– LED2 – jest zaświecana, gdy zamek jest otwarty,
– LED3 – zaświeca się, gdy zostaje naruszona bariera,
– LED4 – zostaje włączona, gdy pojawi się alarm (naruszenie bariery przy zamkniętym zamku).

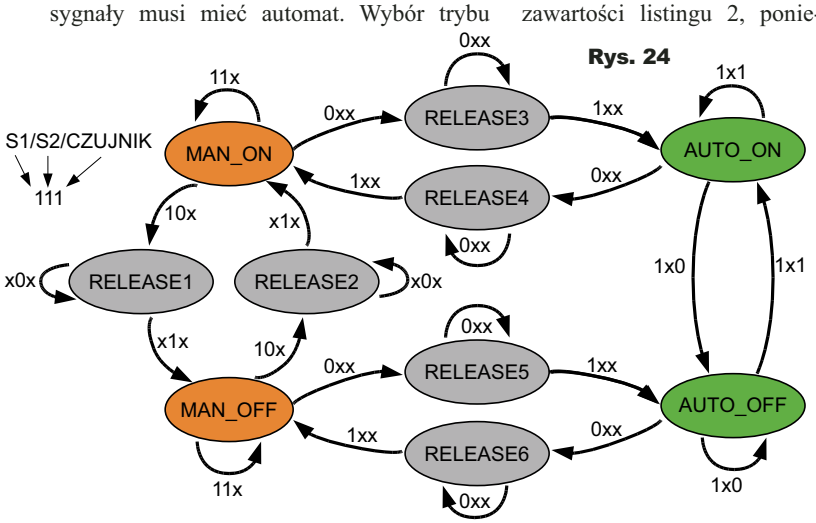
Przypominam o konieczności nasunięcia czarnej rurki na diodę IR. W przeciwnym przypadku dioda LED3 będzie świeciła przez cały czas. Pomóc może także „odstrojenie" preskalera z 36kHz na np. 30...33kHz.

Automat oświetleniowy

Ostatnim projektem w ramach niniejszego kursu będzie automat do kontrolowania oświetlenia, umożliwiający wybór pomiędzy sterowaniem manualnym a automatycznym. Po pierwsze, należy zdefiniować, jakie sygnały musi mieć automat. Wybór trybu pracy (manualny czy automatyczny) będzie dokonywany przyciskiem S1, a przycisk S2 posłuży do włączania i wyłączania oświetlenia w trybie manualnym. Tryb automatyczny wymaga sygnału z czujnika oświetlenia, najlepiej sygnału binarnego: ciemno/jasno. Załóżmy, że umiemy pozyskać taki sygnał i zaprojektujmy stosowny automat (konkretną realizacją czujnika oświetlenia zajmiemy się za chwilę).

Jakie będą potrzebne stany? Przede wszystkim *włączony* i *wyłączony* do pracy manualnej. Podobnie będzie jednak z pracą automatyczną, gdyż w takim przypadku światło również może być włączone bądź wyłączone. Rola przycisku S1 będzie sprowadzać się do zmiany trybu pracy z manualnej na automatyczną i odwrotnie. Przejście pomiędzy stanami *włączony* i *wyłączony*, w zależności od trybu pracy, będzie odbywało się przyciskiem S2 bądź sygnałem dochodzącym z czujnika. Pojawia się tu jednak pewien problem – mianowicie przycisk S2 nie może zostać wykorzystany bezpośrednio: np. stan niski – światło włączone, a stan wysoki – wyłączone. Nikt nie będzie trzymał przycisku przez cały czas, gdy światło ma być włączone. Potrzebne będą zatem stany pośrednie, oczekujące na zwolnienie S1. Graf takiego automatu pokazano na **rysunku 24**.

Tabelka wzbudzeń przerzutników złożona byłaby z sześciu zmiennych (3 bity do przechowywania stanu automatu plus 3-bitowe słowo wejściowe). Łatwiej będzie przygotować automat w VHDL-u, jego kod przedstawiono na **listingu 3**. Jest on bardzo zbliżony do zawartości listingu 2, ponieważ w podobny sposób tworzymy własny typ do przechowywania wszystkich stanów automatu. Tworzymy proces z sygnałem *SYNC* na liście czułości i wykorzystujemy strukturę *case* do opisania zachowania automatu (zgodnie z grafem z rysunku 24). Na samym końcu występuje bardzo prosty dekoder, który włącza diodę LED po wejściu do stanu *MAN_ON* lub *AUTO_ON*.



Rys. 24

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity oswietlenie is
  port(
    S1 : in std_logic ;
    S2 : in std_logic ;
    SYNC : in std_logic ;
    czujnik : in std_logic;
    swiatlo : out std_logic);
end oswietlenie;
architecture oswietlenieBehavioral
                of oswietlenie is
  type typy_osw is (MAN_OFF, MAN_ON,
                AUTO_ON, AUTO_OFF,
  RELEASE1, RELEASE2, RELEASE3,
    RELEASE4, RELEASE5, RELEASE6);
  signal stan : typy_osw ;
begin
  --automat
  process (SYNC) is begin
    if rising_edge(SYNC) then
      case stan is
        when MAN_OFF=>
          if S1='0' then
            stan<=RELEASE5 ;
          elsif S2='0' then
            stan<=RELEASE2  ;
          end if  ;
        when MAN_ON=>
          if S1='0' then
            stan<=RELEASE3 ;
          elsif S2='0' then
            stan<=RELEASE1 ;
          end if ;
        when AUTO_ON=>
          if S1='0' then
            stan<=RELEASE4 ;
          elsif czujnik='0' then
            stan<=AUTO_OFF;
          end  if;
        when AUTO_OFF=>
          if S1='0' then
            stan<=RELEASE6 ;
          elsif czujnik='1' then
            stan<=AUTO_ON;
          end  if;
        when RELEASE1=>
          if S2='1'then
            stan <= MAN_OFF;
          end if ;
        when RELEASE2=>
          if S2='1'then
            stan <= MAN_ON;
          end if ;
        when RELEASE3=>
          if S1='1'then
            stan <= AUTO_ON;
          end if ;
        when RELEASE4=>
          if S1='1'then
            stan <= MAN_ON;
          end if ;
        when RELEASE5=>
          if S1='1'then
            stan <= AUTO_OFF;
          end if ;
        when RELEASE6=>
          if S1='1'then
            stan <= MAN_OFF;
          end if ;
      end case ;
    end  if ;
  end process ;
  --dekoder
  swiatlo <= '0' when (stan=MAN_ON)¬
       or (stan=AUTO_ON) else '1'  ;
end oswietlenieBehavioral;
```

Listing 3

Do prawidłowej pracy układu niezbędny jest czujnik oświetlenia, który również opisany został w VHDL-u (**listing 4**). Idea pomiaru oświetlenia sprowadza się do zliczania zboczy sygnału CLK_FR (z generatora przestrajanego fotorezystorem) w odcinkach czasu wyznaczanych przez sygnał CLK. Na pierwszy rzut oka nie wygląda to dobrze, bo nie mierzymy rzeczywistej częstotliwości – nie mamy przecież dokładnego impulsu 1-sekundowego. Rozwiązanie to ma jednak tę zaletę, że umożliwia „zmianę" częstot-liwości, a co za tym idzie, także progu oświetlenia, przy którym nastąpi włączenie światła. Na listingu 4 zastosowano pewną karkołomną operację wykrywania zboczy sygnałów CLK oraz CLK_FR. Wynikało to z problemów z synchronizacją, gdy sygnały te były umieszczone na liście czułości procesu. Obecnie znajduje się tam tylko *SYNC*, co likwiduje takie problemy. Wprowadzone zostały sygnały pośrednie (CLK_prim oraz CLK_FR_prim) i sprawdzane jest, czy są

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity czujnik_osw is
  port(
    CLK : in std_logic ;
    CLK_FR : in std_logic ;
    SYNC : in std_logic ;
    wyjscie : out std_logic);
end czujnik_osw;
architecture czujnik_oswBehavioral of czujnik_osw is
  signal CLK_prim : std_logic ;
  signal CLK_FR_prim : std_logic ;
begin
  process (SYNC) is
    variable temp : std_logic_vector (8 downto 0)
;
  begin
    if rising_edge(SYNC) then
      if (CLK_FR='1') and (CLK_FR_prim='0') then
        temp := temp + 1;
        CLK_FR_prim <= CLK_FR ;
      else
        CLK_FR_prim <= CLK_FR ;
      end if ;
      if (CLK = '1') and (CLK_prim='0') then
        if temp > 30 then
          wyjscie <= '0' ;
        else
          wyjscie <= '1' ;
        end if ;
        temp := "000000000" ;
        CLK_prim <= CLK ;
      else
        CLK_prim <= CLK ;
      end if ;
    end if;
  end process ;
end czujnik_oswBehavioral;
```

Listing 4

one równe wejściom CLK oraz CLK_FR. Gdy takiej równości nie ma, to mamy do czynienia ze zboczem. Wtedy podejmowana jest odpowiednia akcja: zliczenie zbocza CLK_FR lub sprawdzenie, czy „częstotliwość" jest większa od pewnego progu. W zależności od tego, czy spełniony jest ten drugi warunek, wyjście przyjmuje stan niski bądź wysoki.

Oba te moduły opisane w języku VHDL możemy zamienić na elementy i narysować schemat całego urządzenia – **rysunek 25** (licznik i przerzutniki D odpowiadają za filtrację drgań zestyków).

Obsługa układu jest zgodna z wcześniejszymi założeniami. Przycisk S1 pozwala zamienić tryb automatyczny na manualny (i odwrotnie), a S2 włączyć bądź wyłączyć oświetlenie w przypadku ręcznej kontroli. Regulując potencjometrem częstotliwość CLK, wpływamy na próg oświetlenia. Można go także zmienić poprzez zastąpienie wartości 30 z listingu 4 inną liczbą.

Podsumowanie

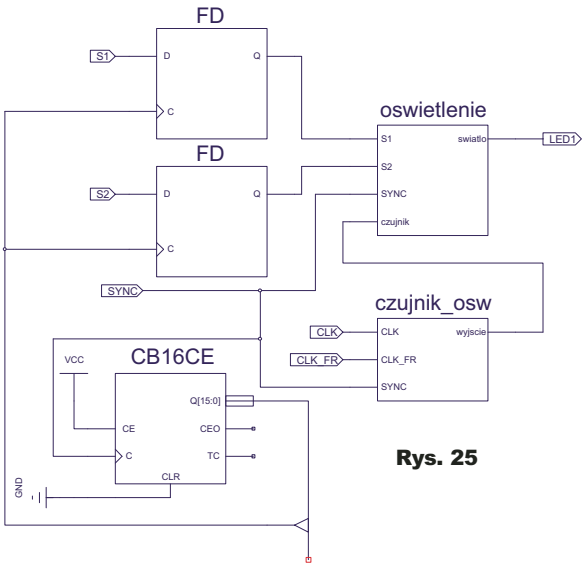
Niniejszy odcinek kończy kurs poświęcony układom CPLD. Mam nadzieję, ze okazał się on kształcący, ciekawy i napisany w sposób przystępny.

Ja jako autor będę wdzięczny za wszelkie opinie o kursie CPLD, przesłane na mój adres mailowy. Pozwoli mi to poprawić jakość materiałów przesyłanych do publikacji. Szczególnie cenne będą uwagi dotyczące jakości materiału, sposobu jego prezentacji (przystępny, nieprzystępny, ciekawy, nudny, łatwy, trudny, etc.) i poziomu merytorycznego. Z góry dziękuję za wszelkie uwagi!

A teraz poważniejsza sprawa: układy CPLD stanowią dobry wstęp do poznawania układów FPGA, oferujących znacznie potężniejsze możliwości i posiadających nieporównywalnie bogatsze zasoby logiczne.

Istnieje możliwość przygotowania i przedstawienia na łamach EdW kursu FPGA. Kurs taki mógłby obejmować zarówno podstawowe informacje o FPGA, jak i bardziej zaawansowane, np. podstawy przetwarzania dźwięku. Jednak obecność tego kursu na łamach EdW jest zależna od zainteresowania nim Czytelników. Dlatego osoby chętne do dalszych eksperymentów z układami programowalnymi powinny wyrazić swoje zainteresowanie poprzez kontakt z Redakcją.

Można przesyłać maile na adres *edw@elportal.pl*, lub wykorzystać formularz Miniankiety (strona 81).



Rys. 25

Jakub Borzdyński
jakub.borzdynski@elportal.pl

KONKURS CPLD

Zakończenie kursu jest dobrą okazją do ogłoszenia konkursu na urządzenia bazujące na układzie CPLD. Do dnia 30 czerwca 2009 można nadsyłać własne, praktyczne realizacje urządzeń bazujących na wiedzy pozyskanej z kursu. Nie ma obowiązku ograniczania się do XC9572, można wykorzystać zarówno „mniejszą" jak i „większą" wersję. Programator dla tych układów można zbudować na podstawie schematu z EdW 8/2008 (część zaznaczona linią przerywaną), zakupić lub wykorzystać płytkę kursu CPLD (np. przylutować przewody po portów TDI, TMS, TCK, TDO oraz GND – po WYJĘCIU XC9572 z podstawki).

Oceniany będzie pomysł, staranność wykonania i użyteczność urządzenia (interesujące „zabawki" i gadgety również mają szansę na nagrody). Urządzenie może być zaimplementowane przy wykorzystaniu edytora schematów, języka VHDL lub w sposób mieszany. Najciekawsze projekty mają ponadto szansę na publikację w EdW.