Kurs CPLD

Odbiornik RC5 i automaty część 7

W dzisiejszym odcinku będziemy kontynuować temat odbiornika RC5. Okazuje się, że za pomocą zestawu uruchomieniowego jesteśmy w stanie odbierać i dekodować polecenia przesyłane z pilota pracującego w standardzie RC5. Zachęcam do wnikliwego zapoznania się z tym materiałem, gdyż w ten sposób można zrealizować prosty system automatyki umożliwiający np. włączanie i wyłączanie dowolnego urządzenia.

Po omówieniu dekodera RC5 podane zostaną pierwsze informacje o automatach stanów, które stanowią jeden ze sposobów projektowania układów sekwencyjnych.

Odbiór kodu RC5

Spróbujmy teraz opracować odbiornik kodu RC5, który umożliwi odbieranie poleceń wysłanych z pilota. Do zapoznania się z niniejszym ćwiczeniem, konkretnie do sprawdzenia, czy układ pracuje prawidłowo, niezbędny będzie stosowny pilot. Czytelnicy nieposiadający takowego będą niestety zmuszeni udać się do sklepu lub przeszukać zasoby serwisów aukcyjnych w Internecie.

Mając na uwadze, że niniejszy kurs czytają także osoby mniej doświadczone, omawianie urządzenia rozpocznę od przybliżenia standardu RC5 i podania informacji o sposobie transmisji sygnałów za pomocą podczerwieni.

Pierwszą rzeczą, o której należy wspomnieć, jest modulacja wiązki podczerwieni. Polega ona na przemiennym włączaniu i wyłączaniu diody IR. Częstotliwość takiego kluczowania jest narzucona odgórnie – tutaj częstotliwość ta wynosi 36kHz. Rozwiązanie takie pozwala skuteczniej wyeliminować szum tła, czyli uniewrażliwia układ odbiorczy na różne źródła emisji podczerwieni. Tak zmodulowany sygnał powoduje, że na wyjściu układu SFH pojawia się stan NISKI, a nie sygnał modulowany. Brak wiązki podczerwieni kluczowanej z częstotliwością 36kHz sprawia, że na wyjściu układu występuje stan wysoki.

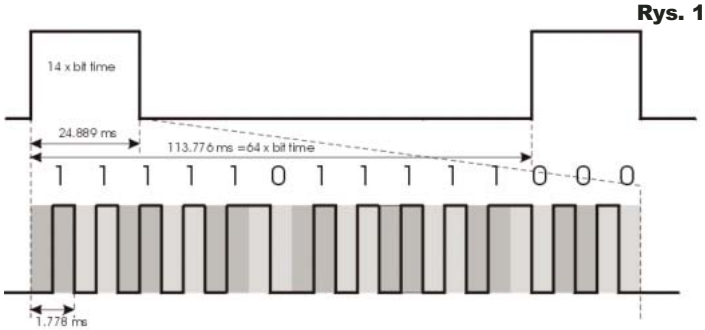
Kolejną rzeczą, o jakiej należy wspomnieć, to użyty sposób kodowania. Nie mamy tutaj oczywiście żadnego wyboru i musimy obsłużyć kodowanie narzucone przez specyfikację RC5, które w literaturze nosi nazwę kodowania Manchester. Cała rzecz opiera się na zapisywaniu pojedynczego bitu za pomocą dwóch stanów – zawsze wysokiego i zawsze niskiego. O tym, z jakim bitem mamy do czynienia decyduje, kolejność wystąpienia stanu niskiego i wysokiego lub inaczej rodzaj zbocza występującego w czasie trwania danego bitu. Pokazuje to **rysunek 1**. Widać na nim, że zbocze narastające, czyli kolejno stan niski i wysoki, informuje, że odbieramy właśnie jedynkę logiczną. Analogicznie zbocze opadające określa, że nadawane jest zero logiczne. Stan niski jest nadawany w postaci zmodulowanej fali IR, natomiast stan wysoki to brak emisji podczerwieni (dioda IR wyłączona). Efekt jest taki, że na wyjściu SFH otrzymujemy przebieg zanegowany, więc będziemy musieli podczas implementacji zastosować negator.

Co oznaczają poszczególne bity? **Rysunek 2** rzuca trochę światła na tę sprawę. Dwa pierwsze bity zawsze są jedynkami i wyznaczają w ten sposób początek ramki. Następny bit informuje o przytrzymaniu przycisku. Po wciśnięciu klawisza na pilocie polecenia są wysyłane cały czas z odstępem 114ms i bit ten jest taki sam. Gdyby jednak puścić przycisk i wcisnąć go ponownie, bit ten będzie miał wartość przeciwną. Dzięki temu zyskujemy możliwość rozróżnienia pomiędzy naciśnięciem przycisku a jego przytrzymaniem. Kolejne pięć bitów stanowi adres, co daje możliwość obsługi 32 urządzeń w sposób niezależny. Ta cecha kodu RC5 nie będzie nam jednak potrzebna, więc ją pominiemy. Dopiero 6 ostatnich bitów stanowi esencję – jest to kod polecenia. Każdy przycisk ma przypisany własny kod i dzięki temu możemy określić, który klawisz został naciśnięty.

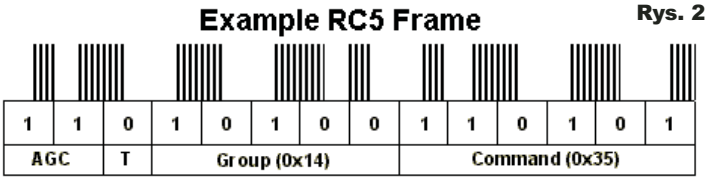
Zastanówmy się teraz, w jaki sposób odbierać taki sygnał. W najprostszym wypadku interesować nas będzie tylko odbiór jednego ze stanów bitu, gdyż jest to informacja wystarczająca, aby określić, jaki bit został nadany i zdekodować kod polecenia. Potrzebna będzie „próbka" sygnału w odpowiednich odstępach czasu, czyli co 1,778ms, bo tyle wynosi czas transmisji pojedynczego znaku. Nie można rozpocząć próbkowania od razu po pojawieniu się zbocza narastającego na wyjściu układu SFH. Uważna lektura karty katalogowej tego czujnika pozwoli stwierdzić, że pojawienie się modulowanej fali podczerwieni lub jej zanik NIE powodują natychmiastowej zmiany stanu wyjścia odbiornika! Pokazuje to **rysunek 3**. W związku z tym dobrze jest „próbkować" sygnał nie na samym jego początku, ale dopiero po pewnym czasie.

Badanie sygnału jest jednak obłożone dwoma warunkami: musi ono trwać 24,889ms, bo tyle trwa cała ramka i musi się rozpocząć po wykryciu zbocza narastającego.

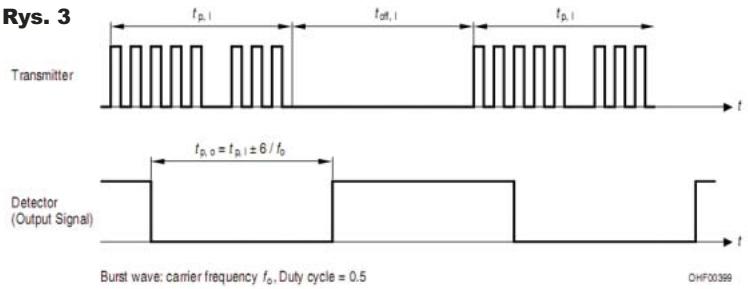
Odstęp czasu wynoszący 1,778ms można uzyskać poprzez zastosowanie preskalera dla generatora kwarcowego. W takim wypadku otrzymamy sygnał prostokątny o okresie 1,778ms i przy każdym zboczu narastającym odczytamy stan wyjścia odbiornika SFH. Co jednak zrobić z tak odebranymi danymi? Bity pojawiają się jeden po drugim: najpierw bity startu, potem przytrzymania, adresu i na końcu numer polecenia. Nam bardziej odpowiadałyby dane w postaci równoległej, gdyż wtedy można je podać na dekoder 7-segmentowy lub jakiś układ decyzyjny. W związku z tym potrze-



Rys. 1



Rys. 2



Rys. 3

bujemy układu konwersji z postaci szeregowej na równoległą, a taką rolę pełni rejestr przesuwający. W tym miejscu uprościmy sobie trochę sprawę i wykorzystamy wersję 4-bitową, co w efekcie pozwoli odebrać cztery najmłodsze bity przesłanego polecenia. W oparciu o zebrane do tej pory przemyślenia można już zaproponować schemat blokowy odbiornika – **rysunek 4**.

Poszczególne bloki widoczne na rysunku 4 (oprócz rejestru) opiszemy w VHDL-u, utworzymy z nich symbole i połączymy na schemacie. Należy najpierw określić, co dokładnie mają realizować poszczególne moduły. Zacznijmy od końca. Do dekodera doprowadzimy magistralę 4-bitową (bo taki zamierzamy zastosować rejestr). Musi on zatem wyświetlać liczby w zakresie 0...9, a przekroczenie tego zakresu będziemy symbolizować kreską.

Zadanie rejestru przesuwającego jest oczywiste, ma on pobierać z wejścia sygnał, gdy na wejściu zegarowym pojawi się zbocze narastające.

Układ taktujący musi spełniać dwie funkcje – wytwarzać przebieg prostokątny o okresie 1,778ms (czyli tyle, ile trwa bit) i pracować tylko wtedy, gdy wejście *EN* ma stan wysoki. Najciekawszą rolę pełni jednak detektor transmisji. Powinien on przede wszystkim wykryć zbocze narastające na wyjściu układu SFH. Kiedy to nastąpi, powinien on ustawić sygnał *EN* w stan wysoki, ale nie od razu! Jak już wspomniałem wcześniej, próbkowanie sygnału powinno odbywać, po określonej chwili zwłoki, którą detektor transmisji musi wytworzyć. Ostatnie zadanie przed nim stojące to odliczenie czasu 24,889ms, po którym ramka się kończy i należy ustawić sygnał *EN* w stan niski. Zatrzyma to generowanie sygnału taktującego i tym samym zapobiegnie pobieraniu danych do rejestru w chwilach, gdy żaden sygnał nie jest odbierany.

Po tych wszystkich rozważaniach przyszła pora na konkretną implementację. Zacznijmy od detektora transmisji. Jego kod został przedstawiony na **listingu 1**. Utworzone tutaj porty mają intuicyjne znaczenie: *rec* jest sygnałem pozwalającym na odbiór (rejestrowanie) sygnału z odbiornika podczerwieni (*sfh*), natomiast wejście *sync* przeznaczono dla generatora kwarcowego (umożliwi on odliczanie czasu). Na opis architektury składa się w zasadzie tylko jeden proces wyzwalany zboczem opadającym generatora kwarcowego (wybór zbocza w tym wypadku nie ma znaczenia). Proces ten może znajdować się w dwóch stanach: oczekiwania na nową ramkę lub oczekiwania na zakończenie bieżącej ramki, o czym decyduje sygnał *odbierz*. Domyślnie ma on wartość zero i testowany jest stan wejścia *sfh*. W momencie pojawienia się na nim jedynki (początek transmisji) sygnałowi *odbierz* przypisywana jest wartość jeden. Powoduje to, że wewnątrz procesu wykonywany jest zupełnie inny fragment kodu – zwiększana jest zawartość zmiennej *licznik*. Gdy osiągnie ona arbitralnie wybraną wartość 5000, co odpowiada ¼ czasu trwania połowy bitu, sygnał *rec* przyjmuje wartość jeden i tym samym rozpoczyna się praca układu taktującego. Zliczanie do 5000 zapobiega próbkowaniu na samym początku, zgodnie z wcześniejszymi założeniami. W momencie, gdy *licznik* osiągnie wartość 610000, czyli po zakończeniu nadawania ramki, sygnał *rec* przyjmuje stan niski. Powrót do stanu oczekiwania na nową ramkę i zerowanie licznika następuje dopiero po upływie 1 000 000 cykli generatora, co sprawia, że przez krótki czas detektor transmisji jest zupełnie niewrażliwy na sygnał wejściowy. Jest to dodatkowa strefa ochronna przed ewentualnymi zakłóceniami, które mogłyby wymusić ponowny, ale niepotrzebny odbiór ramki (np. na skutek zbyt długiego zaniku sygnału na wyjściu układu SFH).

Listing 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity detektor_transmisji is
  Port ( sfh : in std_logic;
    sync : in std_logic ;
    rec : out std_logic);
end detektor_transmisji;
architecture detektor_transmisji_Behavioral ¬
of detektor_transmisji is
  signal odbierz :  std_logic ;
begin
  --detekcja pierwszego  znaku
  receive: process (sfh, sync) is
    variable licznik : unsigned (19 downto 0);
  begin
    if falling_edge(sync) then
      if odbierz='1' then
        licznik := licznik + 1;
        --NOM: if licznik = 10656 then
        if licznik = 5000 then
          rec <= '1' ;
        --NOM: elsif licznik = 597336 then
        elsif licznik = 610000 then
          rec <= '0' ;
        elsif licznik = 1000000 then
          odbierz <= '0' ;
          licznik := „00000000000000000000";
        end if ;
      else
        if sfh  = '1' then
          odbierz <= '1' ;
        end if ;
      end if ;
    end if;
  end process receive ;
end detektor_transmisji_Behavioral;
```

Listing 2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity uklad_taktujacy is
  Port ( enable : in std_logic;
    sync : in std_logic;
    out_clk : inout std_logic);
end uklad_taktujacy;
architecture Behavioral of uklad_taktujacy is begin
  taktowanie: process (sync) is
    --zakres do 21312 (15 bitow)
    variable presk : unsigned (14 downto 0)  ;
  begin
    if falling_edge(sync) then
      if enable = '1' then
        presk := presk + 1 ;
        if presk = 21312 then
          out_clk <= not out_clk ;
          presk := „000000000000000";
        end if ;
      else
        presk := „000000000000000";
        out_clk <= '1' ;
      end if ;
    end if ;
  end process taktowanie ;
end Behavioral;
```

Listing 3

```
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(3 downto 0);
    y : out std_logic_vector(6 downto 0));
end dekoder7segm;
architecture Behavioral of dekoder7segm is
  signal temp : std_logic_vector(3 downto 0) ;
begin
  y <=
  --wyswietlenie jednosci
  „1000000" when (x = „0000") else
  „1111001" when (x = „0001") else
  „0100100" when (x = „0010") else
  „0110000" when (x = „0011") else
  „0011001" when (x = „0100") else
  „0010010" when (x = „0101") else
  „0000010" when (x = „0110") else
  „1111000" when (x = „0111") else
  „0000000" when (x = „1000") else
  „0010000" when (x = „1001") else
  „0111111"  ;
end Behavioral;
```

Kod układu taktującego pokazano na **listingu 2**. Jest on dość prosty, gdyż zawiera jeden proces, wewnątrz którego mieści się instrukcja *if* sprawdzająca stan wejścia *enable*. Gdy jest na nim zero (generowanie przebiegu zabronione), to następuje zerowanie preskalera i ustawianie wyjścia w stan wysoki. W przeciwnym wypadku następuje zwiększanie wartości preskalera i gdy osiągnie on wartość odpowiadającą połowie okresu (1,778ms / 2), zmieniany jest stan wyjścia na przeciwny.

Program dekodera 7-segmentowego pokazano na **listingu 3**. Jest on bardzo prosty – do wektora wyjściowego zapisywana jest kombinacja zależna od stanu wektora wejściowego. Całość odbywa się za pomocą wielokrotnie już stosowanej struktury *when*. Pewną ciekawostką są tutaj dwie ostatnie linijki:

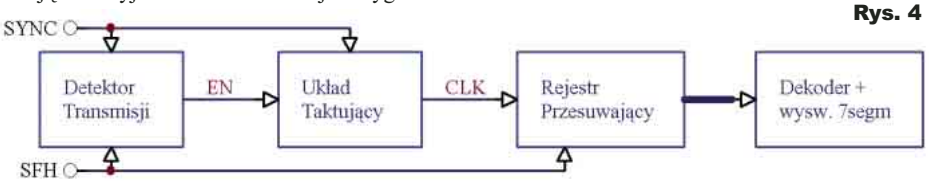
--ostatni warunek
„0010000" when (temp = „1001") else
„0111111" ; --wartosc domyslna

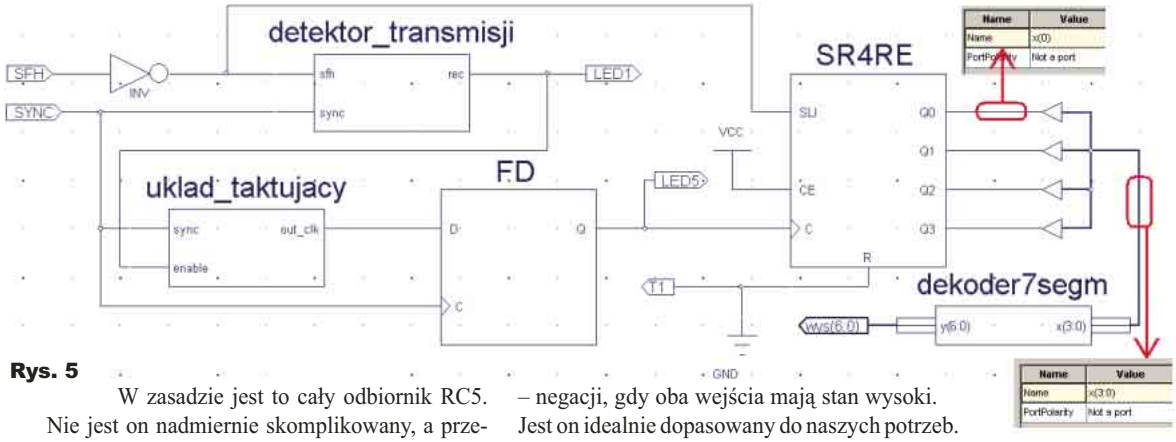
Gdy okaże się, że wektor wejściowy nie mieści się w zakresie 0...9, to po sprawdzeniu ostatniego warunku ładowana jest wartość domyślna wymuszająca zaświecenie jednej kreski znajdującej się na środku wyświetlacza.

Jest to w zasadzie wszystko, co potrzebne jest do odbioru poleceń w kodzie RC5. Można teraz zamienić poszczególne pliki z kodem w VHDL-u na elementy biblioteczne podobnie jak do tej pory. Końcowy schemat pokazano na **rysunku 5**. Nie odbiega on praktycznie w żaden sposób od schematu blokowego z rysunku 4, pomijając sprawy kosmetyczne. Jedną z nich jest przerzutnik D, który ma za zadanie usunąć ewentualne zakłócenia z przebiegu taktującego. Dodane zostały również dwie diody LED:

– LED1 – umożliwia obserwację wyjścia detektora transmisji, co umożliwia sprawdzenie, czy wykrywa on sygnał
– LED5 – dioda wizualizuje stan wyjścia układu taktującego – miga, gdy odbierane są znaki z układu SFH, gdyż pojawia się tam przebieg prostokątny

Rys. 4





Rys. 5

W zasadzie jest to cały odbiornik RC5. Nie jest on nadmiernie skomplikowany, a przeważająca część zasobów jest wykorzystywana jedynie na implementację liczników i rejestrów.

Odbiornik RC5 – wersja rozszerzona

W zasadzie opracowany wyżej odbiornik kodu RC5 jest skończony. Ma on jednak pewne mankamenty. Po pierwsze, wyświetlenie informacji o tym, jaki kod polecenia został odebrany, jest ciekawe, ale ograniczenie się tylko do tego nie jest zbyt praktyczne. Bo po co komu układ pokazujący numer polecenia? Warto go zmodyfikować tak, aby dawał możliwość sterowania jakimś urządzeniem, np. przekaźnikiem. Zadanie to nie jest trudne – wystarczy zastosować prosty dekoder poleceń. Jego praca mogłaby polegać na wyłączeniu przekaźnika, gdy na pilocie naciśnięty zostanie przycisku „0" (kod polecenia to zero). Jego włączenie następowałoby przez wybranie klawisza „1" (pilot wysyła kod polecenia jeden). W zasadzie można by utworzyć stosowny element i opisać go w VHDL-u, jednakże tym razem wykorzystamy standardowe elementy z biblioteki. W najprostszym wypadku wystarczy jedna bramka AND z czterema wejściami i czterema negatorami. W takiej sytuacji pojawienie się zer na wszystkich wyjściach rejestru przesuwającego spowoduje podanie na bramkę czterech jedynek (w wyniku negacji) i pojawienie się jedynki na jej wyjściu. Jest to sygnał odpowiedni do sterowania przekaźnikiem – zero włączy go, a każdy inny kod wyłączy. Wracając jednak do założonego sposobu działania (sterowanie przyciskami „0" oraz „1"), potrzebujemy dwóch 4-wejściowych bramek AND. Jedna da na wyjściu stan wysoki, gdy na wyjściach rejestru pojawi się zero, a druga da stan wysoki, gdy będzie tam jedynka. Potrzebny jest zatem element zdolny wykorzystać te informacje. Warto zauważyć, że musi on mieć pamięć, gdyż konieczne jest przechowywanie ostatniego, poprawnego stanu. W tym miejscu przyda się przerzutnik JK. Uważna jego analiza pozwoli odkryć trzy tryby pracy:

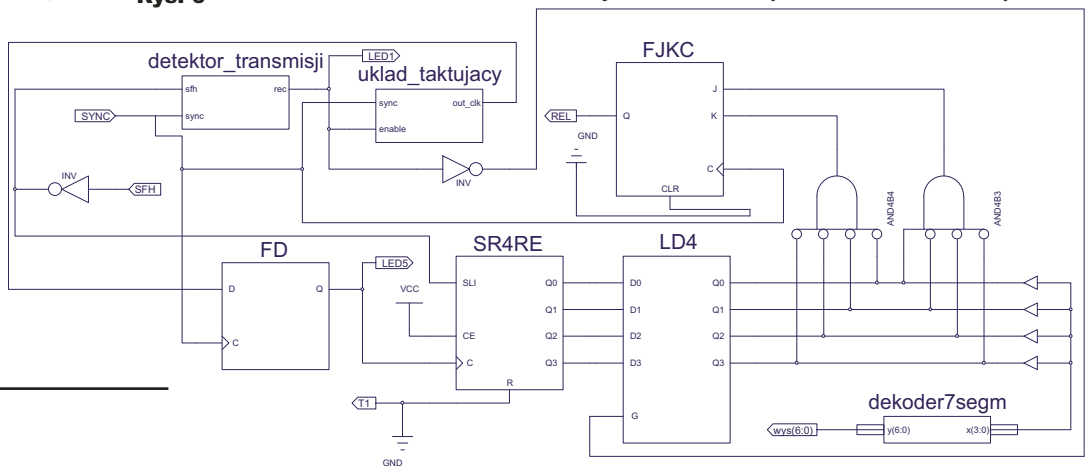
– podtrzymania, gdy na obu wejściach (J oraz K) znajdują się zera,
– przepisania z wejścia J, gdy stan wejścia J różni się od stanu wejścia K,
– negacji, gdy oba wejścia mają stan wysoki.

Jest on idealnie dopasowany do naszych potrzeb. Dwie jedynki na wejściach J oraz K nie wystąpią nigdy. Pojawienie się kodu poprawnego (jeden lub zero) sprawi, że jedna bramka AND będzie miała na wyjściu stan wysoki, a druga stan niski. Pozostałe kody poleceń sprawią, że obie bramki będą miały wyjścia w stanie niskim. Nietrudno zgadnąć, że do wejścia J podłączymy wyjście bramki dekodującej kod „jeden", gdyż po jego wystąpieniu na wyjściu przerzutnika pojawi się stan wysoki włączający przekaźnik (wejście K ma stan niski – następuje przepisanie z J). Po odebraniu zera sytuacja jest analogiczna, wejście J ma stan niski, a wejście K wysoki, więc skoro się różnią, następuje przepisanie, tym razem zera i w efekcie przekaźnik zostanie wyłączony. W pozostałych przypadkach na wejściach będą dwa zera i przerzutnik podtrzyma poprzedni stan.

Jest jeszcze jedna wada odbiornika RC5 omówionego w poprzednim paragrafie. Mianowicie chodzi o migotanie wyświetlacza w trakcie odbierania ramki. Jest to powodowane wprowadzaniem 14 bitów do przesuwającego rejestru 4-bitowego. Rozwiązanie stanowić może omówiony wcześniej zatrzask. Mam też dobrą wiadomość – nie trzeba się specjalnie trudzić z przygotowywaniem sygnału wyzwalającego. Odpowiednie będzie bowiem wyjście *rec* detektora transmisji. Przyjmuje ono jednakże stan wysoki w momencie odbioru ramki, czyli wtedy, gdy zatrzask nie powinien przenosić informacji z wejścia. Stosując negator, otrzymamy jedynkę w chwilach, gdy żadna rama nie jest transportowana i zatrzask będzie „przezroczysty" – przeniesie informację z wejścia na wyjście.

Gotowe rozwiązanie, zawierające wspomniane modyfikacje, przedstawiono na **rysunku 6**.

Rys. 6



Odbiornik RC5 – wersja z detekcją błędów

Prawdopodobnie Czytelnicy zauważyli, że powyższe układy pracują poprawnie tylko wtedy, gdy wiązka podczerwieni jest bezpośrednio skierowana na układ SFH. W innym przypadku na wyświetlaczu mogły pojawić się kody poleceń niezgodne z wysłanymi (np. cztery zamiast dwa). Wynika to z błędów transmisji na skutek np. zaniku sygnału. W zasadzie przeciwdziałanie takim zjawiskom jest trudne i raczej niemożliwe do zaimplementowania w układzie XC9572 (zbyt mało zasobów). Możemy jednak sytuację poprawić poprzez odrzucanie ramek, o których wiemy, że są niepoprawne. Pytanie brzmi: jak stwierdzić, czy ramka jest poprawna, czy nie? Należy tutaj zwrócić uwagę na przyjęty sposób kodowania bitów, które składają się ZAWSZE z dwóch stanów i oba stany NIGDY nie są jednakowe. To właśnie jest klucz. Wniosek może być tylko jeden – musimy odbierać oba stany wchodzące w skład bitu zamiast jednego, jak do tej pory. Po stwierdzeniu, że chociaż jedna para jest jednakowa, będziemy mieli pewność, że odebrany kod polecenia jest nieprawidłowy.

Chcąc rozbudować układ o sprawdzanie poprawności transmisji, należy, po pierwsze wymienić rejestr przesuwający na 8-bitowy, ponieważ odbierać będziemy teraz dwa razy więcej informacji. Po drugie trzeba zmodyfikować układ taktujący i podwoić jego częstotliwość, aby pobieranie danych z wejścia SFH następowało dwa razy częściej. Sprowadza się to jedynie do zmiany linijki:

if presk = 21312 then

na:

if presk = 10656 then

Można przy okazji zaoszczędzić jeden przerzutnik, zmniejszając rozmiar zmiennej *presk* o jeden bit, gdyż przechowuje ona teraz dwa razy mniej stanów. Po wymianie rejestru przesuwającego na każdy odebrany bit przypadają dwa wyjścia tego rejestru (odpowiadające dwóm stanom). W jaki sposób stwierdzić, czy oba stany są różne, czy może równe? Można zaprojektować stosowny układ, ale można też wyko-

rzystać bramkę XOR, która daje na wyjściu jedynkę, gdy liczba jedynek na wejściach jest nieparzysta. W przypadku dwóch wejść musi być zarówno jedynka, jak i zero – tylko wtedy na wyjściu bramki XOR będzie stan wysoki. Tak więc cztery takie bramki rozwiązują problem. Ich wyjścia podłączone do 4-wejściowej bramki AND sprawią, że na wyjściu pojawi się stan wysoki tylko wtedy, gdy wszystkie cztery wejścia będą w stanie wysokim, czyli gdy transmisja przebiegnie prawidłowo.

Ostatnim krokiem jest wymyślenie sposobu na „odrzucanie" nieprawidłowych ramek. W tym miejscu można zaproponować różne rozwiązania, ja zdecydowałem się użyć do tego celu zatrzasku. Jak wspomniano wcześniej, pobiera on dane z wejścia, gdy sygnał *rec* jest w stanie niskim. Dodałem w tym miejscu bramkę AND i dołączyłem sygnał pochodzący z 4-wejściowej bramki AND. W efekcie zatrzask przepisuje wejścia na wyjścia, gdy są spełnione dwa warunki:
– nie jest odbierana ramka (sygnał *rec* ma stan niski, który jest zamieniany w stan wysoki przez negator)
– transmisja przebiegła bezbłędnie (na wyjściu 4-wejściowej bramki AND jest jedynka logiczna).

Spełnienie obu tych warunków jednocześnie spowoduje, że na wyjściu bramki AND (2-wejściowej) pojawia się stan wysoki zezwalający zatrzaskowi na przepisanie sygnału z wejścia na wyjście.

Tym razem dla poćwiczenia został dodany nowy element opisany w VHDL-u. Jest to prosty dekoder poleceń przeznaczony do sterowania pracą przekaźnika. Ma on dogodną dla nas formę – wprowadzamy 4-bitowy wektor i taki sam otrzymujemy na wyjściu. W efekcie jest on przezroczysty, co pozwala na łatwe wstawienie go pomiędzy wyjście zatrzasku a wejście dekodera 7-segmentowego. Wewnątrz układu analizowany jest sygnał – kod tego elementu pokazano na **listingu 4**. W zasadzie całość ogranicza się do przepisania wejścia na wyjście oraz wykorzystania struktury *when* warunkującej zapis jedynki bądź zera na wyjście *rel*, w zależności od wartości wektora wejściowego.

Schemat urządzenia jest widoczny na **rysunku 7**.

AUTOMATY STANU

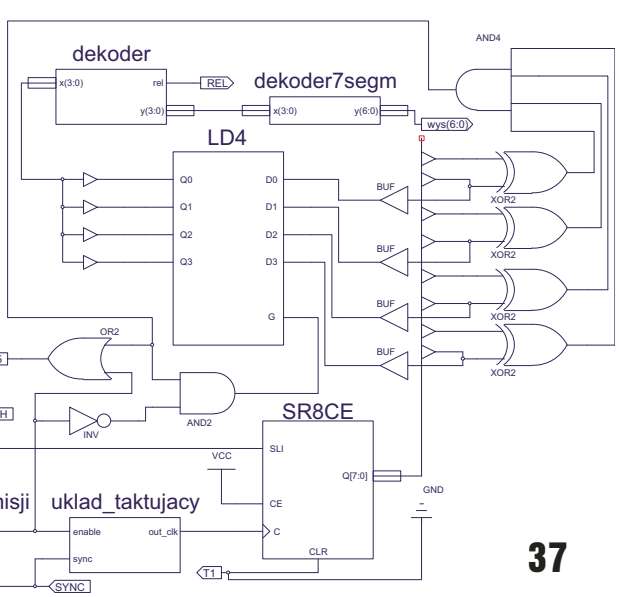
Pod pojęciem automatu stanu rozumiemy, jak sama nazwa wskazuje, układ sekwencyjny mający stan wewnętrzny. Stan ten jest modyfikowany w zależności od sygnału wejściowego (np. stanu przycisków, czujników, etc.) oraz AKTUALNEGO stanu automatu. W życiu codziennym bardzo łatwo znaleźć przykłady takich układów. Może to być choćby sygnalizacja świetlna, która ma cztery stany: zielony, żółty, czerwony oraz czerwono-żółty. Przejścia pomiędzy nimi odbywają się pod wpływem sygnału zewnętrznego (impulsy zegarowe z generatora) i przejścia te są ŚCIŚLE określone oraz ŚCIŚLE zależne od aktualnego stanu. Po zielonym świetle zawsze pojawia się światło żółte, nigdy czerwone. Skąd wiadomo, że ma się pojawić akurat światło żółte? Właśnie na podstawie aktualnego stanu (światło zielone) oraz na podstawie pobudzenia (impuls wymuszający zmianę stanu).

Innym przykładem automatu może być kierowca samochodu (przykład po części żartobliwy, ale oddaje ideę automatu), który ma różne, zdefiniowane stany, np. przyspiesz, hamuj, skręć, uruchom samochód, zgaś samochód. Możemy na ich podstawie stworzyć bardzo zgrabny automat, który w zasadzie będzie funkcjonował samoistnie, np. jako urządzenie elektroniczne kierujące samochodem. Jak taki automat będzie działał? Praca zaczyna się od sygnału o braku gotowości silnika (nie słychać go i/lub świecą się odpowiednie kontrolki). Automat pod wpływem tego sygnału przechodzi w stan *uruchom pojazd*. Po znalezieniu się w tym stanie przebywa w nim dopóty, dopóki nie pojawią się nowe pobudzenia: silnik pracuje i prędkość wynosi zero, co spowoduje przejście do drugiego stanu: przyspiesz. Stan ten zakończy pojawienie się informacji o osiągnięciu zadanej prędkości lub kontroli drogowej (np. fotoradar). Tutaj pojawia się decyzja o przejściu w stan *utrzymaj prędkość* bądź *hamuj* w zależności od rodzaju dochodzącej informacji. Warto zwrócić uwagę na fakt, że automat nie musi i często nie jest liniowy. Co więcej, do wybranych stanów można powrócić, np. po przejechaniu koło fotoradaru znika informacja o „zagrożeniu" i następuje przejście ze stanu *hamuj* ponownie do stanu *przyspiesz*. W taki sposób, krok po kroku, można stworzyć automat pod tytułem „kierowca", który w typowych sytuacjach bezpiecznie zawiózłby nas do pracy.

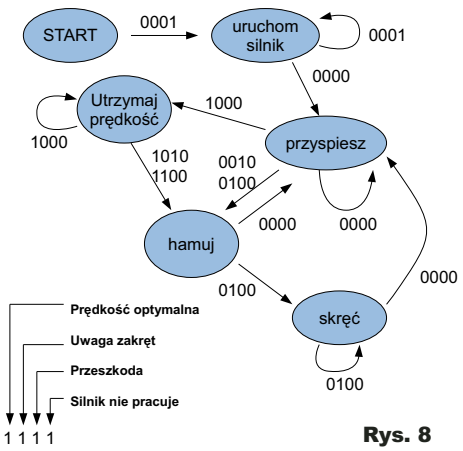
W zasadzie istnieją dwa rodzaje takich automatów: Mealy'ego oraz Moore'a. Istnieją pomiędzy nimi pewne różnice, jednakże aby nie wprowadzać zbyt dużego zamieszania, skupimy się na projektowaniu automatów Moore'a bez wdawania się w szczegóły dotyczące różnic między nimi.

Listing 4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity dekoder is
  port( x : in std_logic_vector(3 downto 0) ;
    y : out std_logic_vector(3 downto 0) ;
    rel : out std_logic );
end dekoder;
architecture dekoderBehav of dekoder is
begin
  --przepisz wejscie na wyjscie
  y <= x ;
  --dekoduj sygnaly (zanegowane!)
  rel<=
  '0' when x="0000" else
  '1' when x="0001" ;
end dekoderBehav;
```



Rys. 7



Rys. 8

Zachęcam jednak Czytelników do poszukania w Internecie dodatkowych informacji i próby implementacji obu typów automatów.

Etapy projektowania automatów stanu

Krótki wstęp do automatów nie wyczerpuje kwestii związanych z tym zagadnieniem. Przyjrzyjmy się zatem, jak powstaje typowy automat, aby nieco rozjaśnić to zagadnienie.

Pierwszą rzeczą jest zdefiniowanie stanów. Musimy określić, jakie stany mogą występować w urządzeniu. Gdybyśmy projektowali sygnalizację, byłyby to stany: zielony, żółty, czerwony i czerwono-żółty. W przypadku kierowcy byłoby to stany: włącz samochód, przyspiesz, utrzymaj prędkości, hamuj, skręć, zatrzymaj, etc.

Drugim etapem jest narysowanie grafu przejść. Jego zadanie polega na wizualnym przedstawieniu drogi, po jakiej może się poruszać automat. Spójrz na **rysunek 8**, na którym pokazano automat „kierowcy", jest on mocno uproszczony, ale oddaje sens takiego grafu przejść. W zasadzie nigdy nie mamy do czynienia z sygnałami typu „przeszkoda" czy „uwaga zakręt", ale z sygnałami cyfrowymi zorganizowanymi od razu w słowo bitowe. Na rysunku 8 mamy do czynienia z takim, 4-bitowym słowem, którego poszczególne bity oznaczają:
– bit 3 (MSB) – prędkość optymalna,
– bit 2 – uwaga zakręt,

– bit 1 – uwaga przeszkoda (np. przechodzień, fotoradar),
– bit 0 (LSB) – silnik wymaga uruchomienia.

Słowa te są zapisywane przy strzałkach, dzięki czemu widać, jaki warunek musi zostać spełniony, aby przejść do danego stanu. Przyjrzyjmy się zatem dokładnie rysunkowi 8. Praca zaczyna się od stanu *START*. W momencie, gdy stwierdzone zostanie, że silnik nie pracuje, nie ma zakrętu, prędkość jest nieoptymalna oraz nie ma przeszkód (sygnał 0001) automat spróbuje uruchomić silnik poprzez przejście do drugiego stanu. Widać, że sygnał 0001 zapętla ten stan, tzn. tak długo, jak słowo wejściowe będzie miało wartość 0001, stan ten nie ulegnie zmianie. Ma to swój sens, gdyż dzięki temu uzyskujemy pewność, że w kolejnych stanach silnik pracuje i będzie uruchamiany do skutku. Nie zawsze udaje się uruchomić samochód od razu, szczególnie w zimie i takie zapętlenie odzwierciedla kolejne próby. Po uruchomieniu silnika słowo wejściowe ma wartość 0000, co oznacza, że silnik pracuje, ale prędkość jest nieoptymalna, nie ma zakrętów ani przeszkód. Tym samym rozpoczyna się proces nabierania prędkości – tu również jest zapętlenie. Zmiana stanu dokona się w trzech przypadkach:
– osiągnięta zostanie prędkość optymalna (1000),
– pojawi się przeszkoda (0010), co rozpocznie hamowanie,
– pojawi się zakręt i konieczne będzie zwolnienie.

W tym miejscu widzimy wyraźnie, że mamy do czynienia z układem decyzyjnym. Podejmowane działanie, czyli przejście do wybranego stanu, odbywa się pod wpływem czynników zewnętrznych oraz tego, co w danej chwili robi automat.

Po osiągnięciu prędkości optymalnej (1000) automat przechodzi do stanu utrzymującego tę prędkość. Zmianę stanu mogą wywołać dwa czynniki (najstarszy bit jest ustawiony, gdyż w momencie pojawienia się tego czynnika mamy prędkość optymalną):
– pojawienie się zakrętu (1100),
– pojawienie się przeszkody (1010).

W tym miejscu pojawia się ciekawy przypadek – stan *hamowanie* jest osiągalny z dwóch różnych stanów (*utrzymaj prędkość* oraz *przyspiesz*)! Oczywiście moglibyśmy dodać kolejny stan, np. *hamowanie_prim*, ale jak widać, jest to niepotrzebne i skomplikowałoby automat. W przypadku stanu *hamowanie* także mamy do czynienia z podejmowaniem decyzji – kiedy prędkość spada i dojeżdżamy do zakrętu, odbywa się skręt i gdy się on kończy, przechodzimy kolejny raz do stanu *przyspiesz*. Podobnie sprawa wygląda z przeszkodą – hamujemy tak długo, aż przeszkoda nie zniknie i wtedy przechodzimy do stanu *przyspiesz* i dalej do utrzymywania optymalnej prędkości. Widać na tym przykładzie, że automat przechodzi przez poszczególne stany wielokrotnie i może podejmować decyzje.

Prosty przykład z rysunku 8 jest niepełny, gdyż automat nie uwzględnia wszystkich sytuacji, np. co się stanie, gdy podczas skrętu pojawi się przeszkoda? Brakuje sygnału 0110 wychodzącego ze stanu *skręć*. Należałoby uzupełnić automat o takie brakujące przejścia. Spowodowałoby to jednak mocne zaciemnienie rysunku 8 i straciłby on charakter poglądowy, dlatego automat ten jest uproszczony. W optymalnym przypadku z każdego stanu powinno wychodzić 16 strzałek (bo tyle sytuacji można przedstawić na 4 bitach), gdyż w tej sytuacji będziemy mieli opracowaną odpowiedź na każdą ewentualność.

Powróćmy jednak do projektowania automatów. Kolejnym etapem jest optymalizacja grafu polegająca na redukcji liczby stanów. Często podczas projektowania grafu zdarza się, że część stanów jest wtórna i może być wyeliminowana, co prowadzi do wymiernych oszczędności zasobów logicznych. Jak już wspomniano, mogły pojawić się niepotrzebnie dwa stany hamowania. Jest to nasze pierwsze spotkanie z automatami, więc pominiemy to zagadnienie.

Następnym etapem jest kodowanie stanów. Operujemy wszakże w kategoriach układów cyfrowych i nie możemy stworzyć stanu „hamowanie". Jedyne, co mamy do dyspozycji, to ciągi zer i jedynek, z których można tworzyć liczby. Właśnie na tym polega ten etap – na przypisaniu liczby do każdego ze stanów automatu. W tym miejscu można już zdradzić, czym jest „stan automatu" – jest to zawartość komórki pamięci, w której przechowywany jest numer aktualnego stanu. Pod wpływem sygnałów zewnętrznych modyfikujemy tę pamięć. Zależałoby nam na oszczędności zasobów, zatem numery stanów nie powinny być większe niż to konieczne – numerujemy je od zera kolejnymi liczbami. Musimy również określić, w jaki sposób przygotować komórkę pamięci do przechowywania numeru aktualnego stanu automatu. Nie będzie chyba dużym zaskoczeniem wykorzystanie do tego celu przerzutników. Okazuje się, że istnieje tu pewna dowolność polegająca na możliwości zastosowania przerzutników typu D, T bądź JK. Pociąga to za sobą jednak konsekwencje – w niektórych zastosowaniach jeden przerzutnik da lepszy rezultat (pod względem optymalności zużycia zasobów) niż inny. Zastosowanie przerzutnika ma jeszcze jedną poważną zaletę – zmiany stanów odbywają się w sposób synchroniczny, tzn. w takt sygnału zegarowego. Unikamy tym samym przypadkowych stanów np. na skutek różnego czasu propagacji sygnałów w urządzeniu.

Po przypisaniu cyfr poszczególnym stanom pozostaje przygotowanie funkcji wzbudzeń przerzutników. Być może nazwa wydaje się dziwna, ale cały proces powinien być już Czytelnikom znany, gdyż polega on na przygotowaniu funkcji boole'owskich za pomocą tablicy Karnaugha bądź metody Quine'a-McCluskeya. Słowo wejściowe składa się w tym wypadku z ciągu bitów określających aktualny stan automatu plus bity sygnałów wejściowych. Słowem wyjściowym jest natomiast bitowa reprezentacja następnego stanu automatu.

Prawdopodobnie wiele rzeczy jest jeszcze niejasnych, dlatego przyjrzyjmy się przykładom automatów i sposobom ich projektowania krok po kroku. Mam nadzieję, że rozwieje to ewentualne wątpliwości.

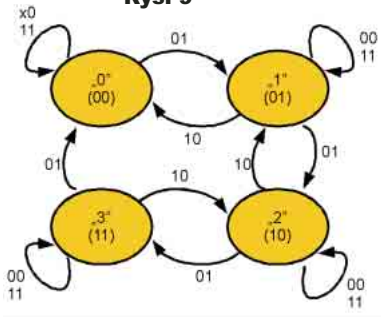
Prosty przykład automatu stanu

Na początek zbudujmy bardzo prosty automat dający możliwość wybierania cyfry w zakresie od 0 do 3. Pierwszym krokiem będzie przygotowanie stosownego grafu. Mamy cztery cyfry do wyświetlenia, zatem będziemy potrzebowali czterech stanów, które będą reprezentować te cyfry. Sterowanie przejściami w automacie będzie się odbywać za pomocą słowa 2-bitowego, na które składać się będzie stan przycisków S1 oraz S2. Naciśnięcie S2 spowoduje przejście do kolejnego stanu, natomiast naciśnięcie S1 przejście do stanu poprzedniego. Istnieje tutaj pewna dowolność, gdyż sami możemy określić, co stanie się po osiągnięciu „ostatniego" stanu – czy nastąpi przejście do stanu pierwszego, czy może automat ulegnie zatrzymaniu i dopuszczalnym działaniem będzie tylko cofnięcie się? Wybierzmy najciekawszą opcję, czyli pośrednią – możliwe jest przejście ze stanu ostatniego (cyfra 3) do pierwszego (cyfra 0), ale w drugą stronę już nie. Stosowny graf został przedstawiony na **rysunku 9** (x=don't care). Przy okazji wykonane zostało kodowanie poszczególnych stanów polegające na przypisaniu liczby binarnej. Mamy cztery stany, zatem do opisania ich wystarczą 2 bity. Można oczywiście wykorzystać trzy lub więcej bitów, ale spowodowałoby to niepotrzebny wzrost komplikacji przykładowego układu.

Należy teraz wybrać typ przerzutnika oraz określić, ile ich ma być. Stany są reprezentowane za pomocą ciągów 2-bitowych, zatem naturalnym jest wybór dwóch przerzutników – jeden przechowuje stan pierwszego bitu, a drugi – drugiego bitu. Najbardziej intuicyjne będzie zastosowanie przerzutników typu D, dlatego na nich właśnie się skupimy. Zanim przystąpimy do pracowitego wyznaczania funkcji boole'owskich niezbędnych do realizacji automatu warto przygotować tabelkę, w której zbierzemy wszystkie informacje. Powinna mieć ona postać jak na **rysunku 10**.

Przyjrzyjmy się jej dokładnie. Pierwsza kolumna oznaczona kolorem zie-

Rys. 9



lonym (*stan-*) to stan poprzedni. Odnosi się to konkretnie do stanu automatu, jaki występuje na moment przed zmianą jego stanu na kolejny (przed pojawieniem się zbocza narastającego sygnału zegarowego). Kolumna czerwona określa stan następny (*stan+*), czyli stan, jaki będzie miał automat po zmianie. Całość można zrozumieć następująco: automat ma jakiś stan, np. 00 (*stan –*), w pewnym momencie przychodzi zbocze narastające, które wymusza zmianę stanu na następny (*stan+*). Będzie on zależny od stanu aktualnego oraz wartości sygnałów wejściowych. Oczywiście możne nastąpić specyficzna sytuacja (często występuje), że stan automatu nie ulegnie zmianie, gdyż pewna kombinacja sygnałów wejściowych „zatrzaskuje" stan obecny. Spójrz jeszcze raz na rysunek 9, na którym wyszczególniono graf przejść automatu. Przykładowo przejście ze stanu 00 do stanu 01 odbywa się w momencie, gdy przycisk S2 daje wartość zero (jest naciśnięty), a przycisk S1 – wartość 1 (słowo wejściowe w takim wypadku to 01). To samo jest w tabeli na rysunku 10 – stan 00 (zielona kolumna) zmienia się na stan 01 (czerwona kolumna) w momencie, gdy S2 = 0 oraz S1 = 1. Jak już wspomniałem, automat wcale nie musi zmienić stanu – np. w wypadku puszczenia obu przycisków ma być pamiętany ostatni ustawiony stan. Jest to widoczne na grafie przejść (rys. 9) – kiedy znajdujemy się w dowolnym stanie i oba przyciski są puszczone (S1=1, S2=1) lub gdy oba są wciśnięte (S1=0, S2=0), to automat jest „zapętlony" i niejako „przechodzi" do tego samego stanu. Zależność ta jest również widoczna w tabeli z rysunku 10 – stan poprzedni (zielona kolumna) jest taki sam, jak stan następny (czerwona kolumna) za każdym razem, gdy oba przyciski są naciśnięte lub puszczone – nic się wtedy nie zmienia. Warto zauważyć również specyficzną sytuację dla stanu 00, który zgodnie z naszymi założeniami nie może ulec zmianie na stan 11 – nawet wymuszenie „cofnięcia się" (S2 = 1, S1 = 0) powoduje, że automat pozostaje w swoim stanie.

Jeszcze raz krótko podsumujmy zasadę komponowania takiej tabelki: wypisujemy wszystkie możliwe stany w kolumnie *stan-* oraz obok wszystkie możliwe wartości słów wejściowych (stany sygnałów wejściowych). Dalej zapisujemy stan, jaki ma następować po wystąpieniu danego słowa wejściowego.

Dwie najważniejsze kolumny tabeli z rysunku 10 zaznaczone są kolorem niebieskim. To właśnie te dwie kolumny są przedmiotem projektowania automatu (może być ich więcej, jeżeli automat ma więcej stanów – liczba kolumn zaznaczonych na niebiesko odpowiada licznie bitów wykorzystanych do zapisu stanu automatu). Czytelnicy na pewno zauważą, że dane zawarte w kolumnie Da i Db odpowiadają wartościom z kolumny czerwonej (*stan+*). Można przyjąć, że będzie tak zawsze, jeżeli zastosujemy przerzutnik D. Wartości Da oraz Db stanowią właśnie sygnały wejściowe tego przerzutnika, czyli bity, jakie należy podać na jego wejście, aby wyjścia miały wartość jak w kolumnie czerwonej. Krótko mówiąc – jeżeli stosujesz, Czytelniku przerzutniki typu D, to zawsze możesz przepisać zawartość czerwonej kolumny do kolumn niebieskich. Oczywiście Da oznacza bit starszym, a Db młodszy.

Możemy teraz przystąpić do wyznaczenia funkcji boole'owskiej: *Da(stan-, S1, S2)* oraz *Db(stan-, S1, S2)*. Wartość pierwszej funkcji zostanie doprowadzona do wejścia pierwszego przerzutnika, natomiast wartość drugiej – do drugiego przerzutnika. Tym samym sprawimy, że wyjście obu przerzutników będzie zmieniać się w sposób zależny od poprzedniego stanu i stanu przycisków, czyli realizowane będą wyznaczone przed momentem funkcje boole'owskie.

Na **rysunku 11** przedstawiono obie tabele Karnaugha oraz stosowne funkcje boole'owskie. Na tej podstawie można zaimplementować automat – **rysunek 12**. Znajduje się tu również znany z poprzednich części kursu dekoder 7-segmentowy pokazujący aktualny stan. Do tej pory w zasadzie nie wspomniałem co daje wejście do danego stanu. Jeżeli stan potraktujemy jak słowo bitowe, to możemy w znany już sposób zaprojektować układ kombinacyjny – dekoder. Jego działanie sprowadzi się np. do, trzymając się analogii automatu kierowcy, włączenia hamowania, gdy automat będzie miał stan o określonej wartości (przypisanej hamowaniu) lub skręcenia kierownicy dla stanu o innym numerze. W tym przypadku mamy dekoder 7-segmentowy, który zamienia numer stanu na świecenie wybranych segmentów wyświetlacza.

Zwracam uwagę, że układ z rysunku 12 zmienia swój stan w takt przestrajanego generatora, co przy przekręceniu potencjometru w skrajne położenie spowoduje bardzo szybką zmianę. Przytrzymanie jednego przycisku skutkuje przejściem do kolejnego stanu, a puszczenie – zatrzymanie w stanie obecnym.

Podsumowanie

W tym odcinku został omówiony dekoder RC5, który nie zmieścił się w poprzednim numerze. Jego implementacja pozwala na zdalne sterowanie urządzeniami za pomocą pilota. Zachęcam do eksperymentów na tym polu.

Warto zapoznać się z przedstawionym materiałem i spróbować wprowadzić własne modyfikacje, aby utrwalić nowe wiadomości. Zachęcam do prób integracji odbiornika RC5 z generatorem PWM. Zadanie to nie będzie trywialne, gdyż zasoby układu XC7295 są mocno ograniczone i konieczne będzie wprowadzenie sensownej optymalizacji. Na pewno warto zrezygnować wtedy ze zbędnych zadań, takich jak obsługa wyświetlacza 7-segmentowego. Drugą kwestią jest usunięcie przycisków, sygnałów pośrednich z nimi stowarzyszonych, układu filtracji drgań zestyków i pierwszego licznika. Wszystkie te moduły zastąpi wtedy bezpośrednie wyjście z odbiornika kodu RC5. Sterowanie odbywać się będzie za pomocą przycisków z cyframi na pilocie.

W przyszłym odcinku będziemy kontynuować temat automatów i wykonamy następne przykłady, które mam nadzieję pozwolą wyrobić sobie znacznie lepsze rozeznanie w tej dziedzinie.

Jakub Borzdyński
jakub.borzdynski@elportal.pl

ŹRÓDŁA:

[1] http://domotica.homeip.net/img/rc5.jpg (rysunek 1)
[2] http://www.geocities.com/digitan000/Hardware/19/rc5_frame_example.gif (rysunek 2)

Rys. 10

stan-	S2	S1	stan+	Da	Db
00	0	0	00	0	0
00	0	1	01	0	1
00	1	0	00	0	0
00	1	1	00	0	0
01	0	0	01	0	1
01	0	1	10	1	0
01	1	0	00	0	0
01	1	1	01	0	1
10	0	0	10	1	0
10	0	1	11	1	1
10	1	0	01	0	1
10	1	1	10	1	0
11	0	0	11	1	1
11	0	1	00	0	0
11	1	0	10	1	0
11	1	1	11	1	1

Rys. 11

Da Db \ S2 S1	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	1	0	1	1
10	1	1	1	0

D_A^+

Da*Db*S2 + Da*!Db*S1 + Da*!S2*!S1 + !Da*Db*!S2*S1

Da Db \ S2 S1	00	01	11	10
00	0	1	0	0
01	1	0	1	0
11	1	0	1	0
10	0	1	0	1

D_B^+

Db*S2*S1 + !Db*!S2*S1 + Db*!S2*!S1 + Da*!Db*S2*!S1

Rys. 12