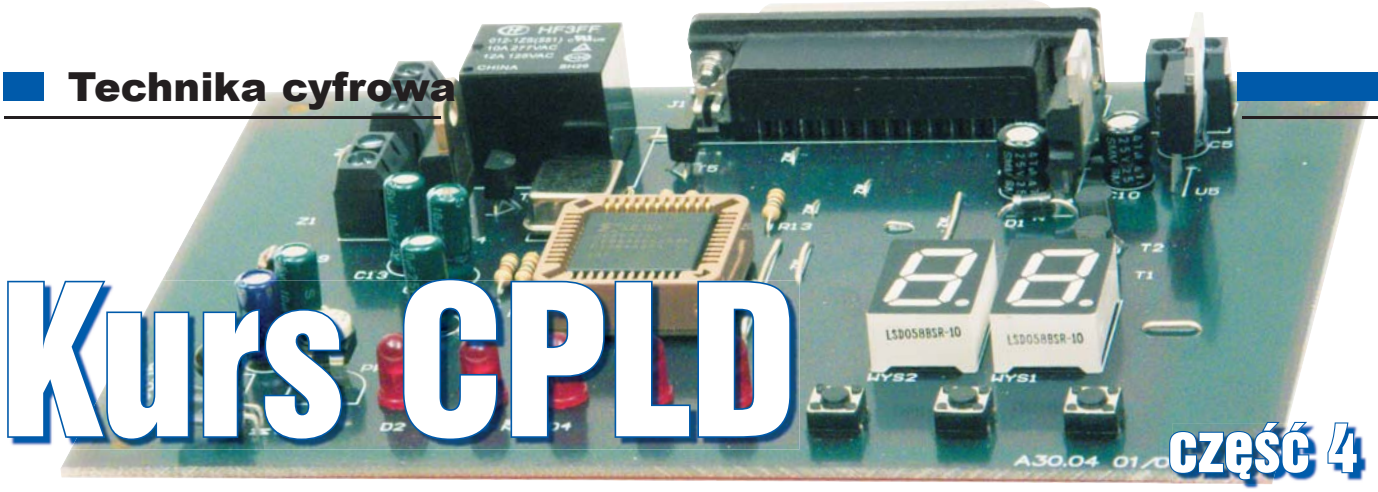


Kurs CPLD

część 4

W niniejszym odcinku mieliśmy skupić się na doskonaleniu umiejętności z poprzednich części. Można powiedzieć, że tak właśnie będzie, ale z drugiej strony szkoda byłoby zmarnować okazję i nie nauczyć się niczego nowego. W związku z tym materiał został rozszerzony o ważne dodatkowe informacje.

Co na dobry początek? Zaczniemy od zrealizowania kostki do gry, czyli prostego projektu przypominającego zagadnienia z poprzednich części kursu. Następnie zajmiemy się układami iteracyjnymi. Przedstawię definicję takich układów i pokażę, w jaki sposób zabrać się do ich tworzenia. Przy okazji w artykule przedstawionych zostanie kilka ciekawych rozwiązań, jakie można napotkać, zagłębiając tajniki techniki cyfrowej.

Jeżeli tablica Karnaugha z poprzedniej części kursu nie przypadła Ci do gustu, to mam dobrą wiadomość, dziś zostanie przedstawiona alternatywna metoda. W literaturze spotykana jest ona pod nazwą metody Quine'a–McCluskeya. Jest ona znacznie łatwiejsza do implementacji w formie programu komputerowego.

Coś na rozgrzewkę – kostka do gry

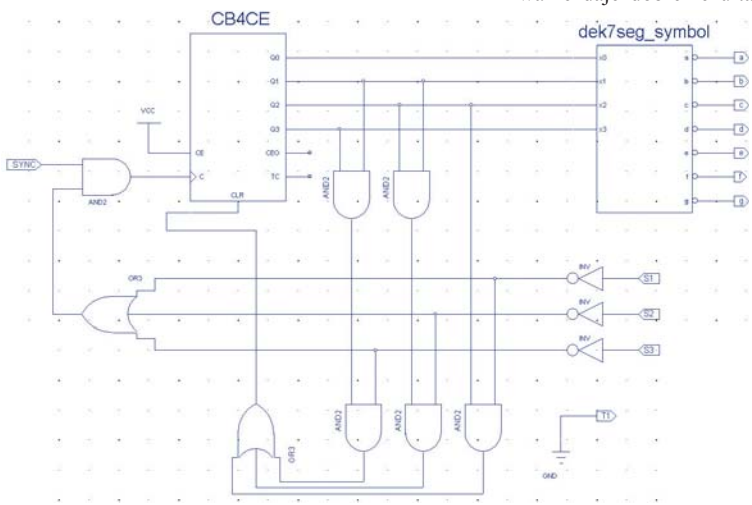
Pierwszym krokiem jest jak zwykle określenie koncepcji. W jaki sposób zrealizować generator losowy? Idea leżąca u podstaw generowania liczb losowych zakłada w dużym uproszczeniu, że podczas losowania nie da się przewidzieć wyniku. Założenie to można zrealizować m.in. bardzo szybką zmianą kolejnych stanów. Wymagałoby to zastosowania licznika taktowanego bardzo dużą częstotliwością, która może pochodzić z generatora kwarcowego. Możemy pokusić się o zaimplementowanie więcej niż jednej kostki w projekcie, dajmy na to kostki 4-, 6- oraz 10-ściennej. Wybór zakresu losowanej liczby będzie odbywał się przez naciśnięcie przycisku S1, S2 bądź S3. Wiemy, z poprzednich odcinków kursu, że takie niestandardowe liczniki można zrealizować za pomocą licznika modulo 16 (4-bitowego) i odpowiednio przygotowanego układu resetującego. Mając generator i sterowany licznik, należy zastanowić się, jak ma wyglądać uruchamianie zliczania.

Teoretycznie można by przygotować układ kombinacyjny z jednym wyjściem (reset dla licznika), trzema wejściami przeznaczonymi na doprowadzenie sygnałów z przycisków oraz czterema wejściami, do których podłączylibyśmy wyjścia licznika modulo 16. Stwarza to jednak pewien problem – daje 7 zmiennych na wejściu. Wyznaczenie stosownych funkcji boole'owskich nie będzie sprawą trywialną, a tym bardziej szybką. Spróbujmy znaleźć inne rozwiązanie. Przygotowanie odpowiedniego układu resetującego tak, aby otrzymać zliczanie modulo 4, 6 czy 10 jest sprawą trywialną. Co więcej, układy te pracują niejako niezależnie. Potrzebny jest zatem element, który ma trzy wejścia dla przycisków, trzy wejścia dla sygnałów pochodzących z naszych trzech układów resetujacych oraz jedno wyjście, z którego będziemy odbierać sygnał zerowania licznika. W efekcie daje nam to 6 wejść i jedno wyjście. Na tablice Karnaugha jest to dużo. Można to oczywiście wyznaczyć, np. omówioną za chwilę alternatywną metodą, ale czy nie ma prostszego rozwiązania? Metodyczne postępowanie daje dobre rezultaty, ale czasami dobry pomysł połączony z intuicją daje jeszcze lepsze wyniki. Tym bardziej że w poprzednich częściach kursu wspomniano o pewnym elemencie, który realizuje funkcję przełączającą. Nie chodzi tu o multiplekser, bo on ma wejścia adresowe i konieczne jest zadanie liczby w kodzie binarnym, aby określić interesujące nas wejście. Mowa tu o bramce AND, która posiada dwa wejścia – gdy na jednym z nich jest zero, na wyjściu jest zawsze zero, a gdy na pierwszym wejściu jest jeden, to na wyjściu jest wartość obecna na drugim wejściu. Wypływa stąd prosty wniosek – dołączając do jednego wejścia bramki przycisk, a do drugiego sygnał resetujacy, możemy go zablokować lub umożliwić jego propagację na wyjście. Tym samym trzy bramki AND pozwolą niezależnie blokować poszczególne sygnały. Ze względu na fakt, że po puszczeniu przycisku nic nie powinno się dziać, czyli poszczególne bramki powinny być „zablokowane", trzeba wymusić tam obecność zera logicznego. Wymagane więc będzie dołączenie przycisków przez negatory, aby jedynka pojawiała się po wciśnięciu przycisku, a nie po jego puszczeniu. Nie można jednak dołączyć do wejścia licznika trzech różnych sygnałów, więc dodamy jeszcze jedną bramę OR z trzema wejściami. Pojawienie się choć jednej jedynki sprawi, że licznik zostanie wyzerowany. Wyjścia licznika dołączymy oczywiście do zaprojektowanego wcześniej dekodera 7-segmentowego, aby zaprezentować wynik w czytelnej formie.

Pozostał ostatni problem, mianowicie jak i kiedy włączyć generator kwarcowy? Odpowiedź na pytanie „kiedy" jest oczywista – gdy zostanie wciśnięty jeden z przycisków. A jak zorganizować włączanie i wyłączanie generatora? Również za pomocą bramki AND! Oprócz tego zastosujemy jeszcze jedną bramkę OR z trzema wejściami i dołączymy do niej przyciski (przez negatory). Wciśnięcie dowolnego przycisku da na wyjściu tej bramki sygnał wysoki i uruchomi zliczanie.

Po tych rozważaniach można przystąpić do implementacji, której wynik widoczny jest na **rysunku 1**.

Rys. 1



Lepsza kostka do gry

Implementacja kostki przedstawiona na rysunku 1 nie jest, niestety „ładna". Podczas losowania, cyfry na wyświetlaczu zmieniają się bardzo szybko, co stwarza nieciekawy efekt wizualny. Poza tym, czy nie lepiej byłoby losować liczby w zakresie 1...4, 1...6 oraz 1...10 zamiast od zera? Spróbujmy poprawić te dwie wady, zaczynając od migotania wyświetlacza. W jednej z poprzednich części kursu badaliśmy dekodery i przy tej okazji stworzyliśmy efekt kreski „biegającej" po wyświetlaczu. Wyglądałby on znacznie lepiej niż obecnie obserwowane migotanie. Ponownie zadajmy sobie pytanie: co chcemy osiągnąć? Na czas losowania chcieli-

byśmy odłączyć licznik od wyświetlacza i dołączyć do niego sygnały sterujące, które dadzą w efekcie „biegającą kreskę". Podobne zadanie już zostało wykonane podczas budowania licznika modulo 100. Zostały tam wykorzystane multipleksery do przełączania sygnałów sterujących wyświetlaczem. Dodanie jednakże do tego projektu siedmiu takich elementów bardzo mocno zaciemni schemat. Ta część kursu poświęcona jest m.in. utrwalaniu umiejętności zdobytych w poprzednich częściach, zatem powrócimy do tworzenia własnych elementów. Przygotujmy komponent, który posiada dwa razy po siedem wejść: dla dekodera 7-segmentowego i dla dekodera *1 z n* realizującego animacje. Potrzebne będzie 7 wyjść dla poszczególnych segmentów wyświetlacza oraz sygnał sterujący. Można postawić pytanie, czy nie dałoby się zaszyć w takim układzie dekodera 7-segmentowego oraz dekodera na kod *1 z n*? Zmniejszyłoby to liczbę wejść do dwóch par po cztery. Odpowiedź brzmi: oczywiście, że można i pozostawiam to jako dodatkowe zadanie dla Czytelników.

Wiemy, jak funkcjonuje multiplekser oraz w jaki sposób go przełączać, zatem nie powinno sprawić problemu przygotowanie schematu omawianego elementu. Został on przedstawiony na **rysunku 2**. Mając już 7-bitowy przełącznik, trzeba sobie odpowiedzieć na pytanie, jak go wykorzystać. Z poczynionych przed chwilą założeń wiemy, że ma być on przełączany podczas losowania. Proces losowania jest uruchamiany przez przyciski S1, S2, S3, które są podłączone do bramki OR. Nadaje się ona idealnie do generowania sygnału przełączającego – gdy żaden przycisk nie jest wciśnięty, na wyjściu bramki jest zero i nasz przełącznik będzie podawał sygnały z licznika, natomiast w czasie losowania jest tam jedynka, która wymusi podawanie sygnałów z dekodera kodu *1 z n*.

Budowa układu generującego animację nie powinna przysporzyć nikomu trudności. Wykorzystamy osobny licznik, gdyż musi on zawsze liczyć modulo 6, aby odpowiednio wysterować dekoder *1 z n*. Licznik ten może być taktowany z generatora kwarcowego po zastosowaniu odpowiednio dużego dzielnika częstotliwości (kolejne liczniki) lub przestrajanego generatora. Wykorzystam ten drugi sposób, aby dodatkowo nie zaciemniać schematu. Na **rysunku 3** widoczny jest schemat kostki z animacją podczas losowania. W czasie testowania pojawiły się zakłócenia, które powodowały, że kreska nie przemieszczała się w sposób płynny. Można temu przeciwdziałać poprzez zmianę parametru *Output Slawe Rate* na wartość *Slow*, co zmniejszy liczbę generowanych zakłóceń. Dokonuje się tego poprzez zaznaczenie schematu z projektem, kliknięcie prawym przyciskiem myszy na *Implement Design* i wybranie *Properties*. W otwartym okienku należy przejść do zakładki *Fitting* i zmienić wartość pola *Output Slew Rate* (**rysunek 4**).

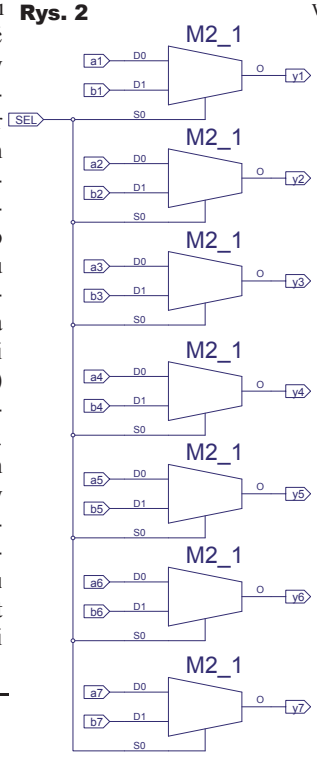
Powróćmy teraz do problemu losowania w zakresie od 1, a nie od zera. Zmuszenie licznika, aby liczył od jeden, może być trudne. Możemy przygotować stosowny układ kombinacyjny dokonujący odpowiedniego przekształcenia. Co miałby on robić? Zamieniać 0 na 1, 1 na 2, 2 na 3, etc. Wniosek – zwiększać zawartość licznika o jeden. Po co jednakże tracić czas na projektowanie takiego rozwiązania, skoro bez problemu potrafimy dodawać wartość stałą? Wykorzystamy tu sumator jednobitowy opracowany w poprzednim odcinku i połączymy go w kaskadę umożliwiającą pracę z liczbami 4-bitowymi.

Ponownie uprościmy sobie sprawę i przygotujemy kolejny element – 4-bitowy układ dodający jeden do słowa wejściowego. Czy dostrzegasz pułapkę w takim rozumowaniu? W jaki sposób będzie wyświetlana liczba 10? Na pewno nie może zostać wyświetlona na jednym wyświetlaczu, więc co się stanie, gdy zostanie ona podana na dekoder 7-segmentowy? Określenie, co znajdzie się na wyjściu, jest możliwe na dwa sposoby: można podać liczbę 10 na ten dekoder i zobaczyć, jaki będzie rezultat, albo przeanalizować tablice Karnaugha z części 3 kursu. W ten sposób uda się ustalić, że liczba dziesięć zostaje wyświetlona jako 2.

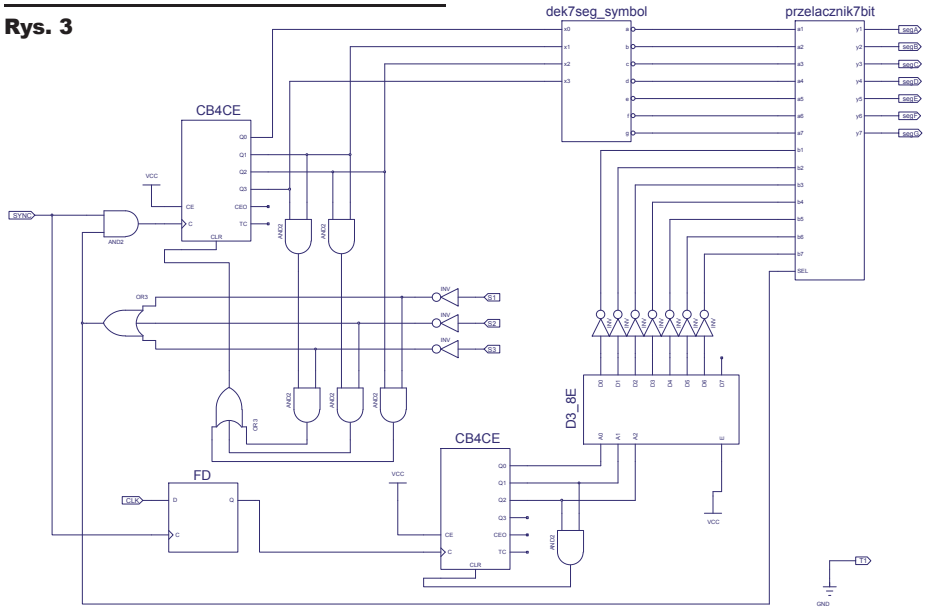
Zaprojektujmy zatem bardziej inteligentnie element dodający jeden do słowa wejściowego. Mianowicie zdefiniujemy jeszcze jedno wejście (ADD), które pozwoli określić, czy dodawanie ma być wykonywane, czy nie. Losując liczbę kostką 4- bądź 6-ścienną wynik będziemy zwiększać o jeden, a w przypadku kostki 10-ściennej już nie. Na **rysunku 5** widoczny jest schemat omawianego elementu. Dodając ten element do projektu, pamiętaj, aby dodać również symbol sumatora 1-bitowego, na którym on bazuje (przebiega to analogicznie do dodawania każdego innego symbolu).

W tym miejscu nasza rola jest już bardzo prosta – wystarczy element z rysunku 5 dodać do projektu i włączyć pomiędzy licznik i dekoder 7-segmentowy, w wyniku czego otrzymamy schemat z **rysunku 6**. Jest na nim widoczna kostka, która spełnia wcześniejsze postulaty – umożliwia losowanie liczb w zakresie 1...4, 1...6 oraz 0...9.

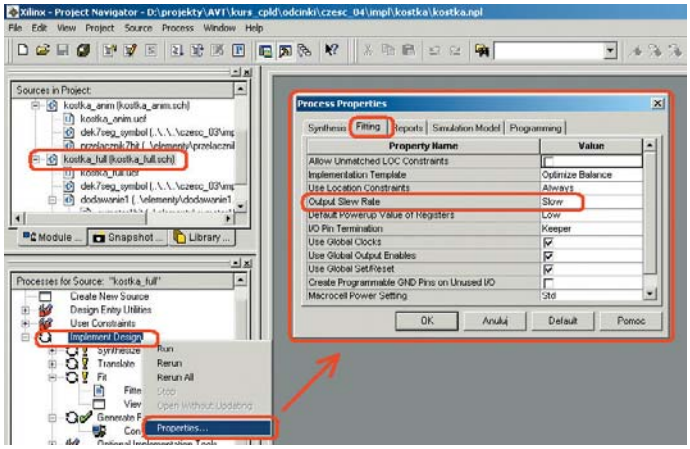
Rys. 2



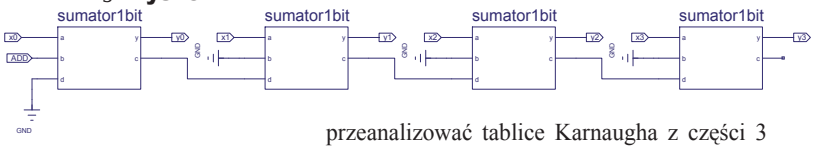
Rys. 3

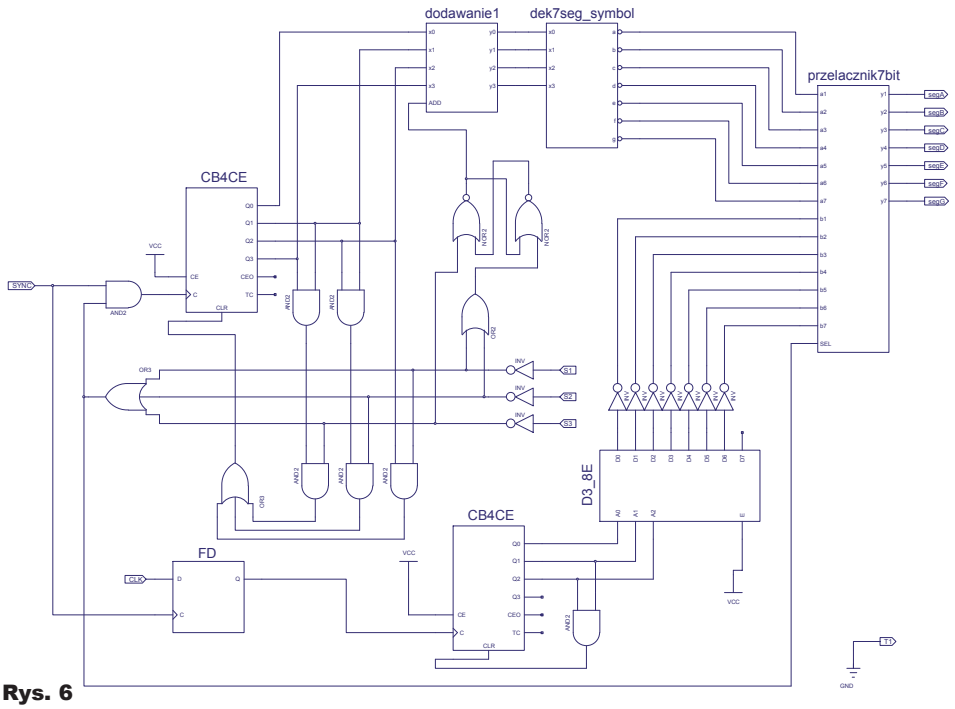


Rys. 4



Rys. 5



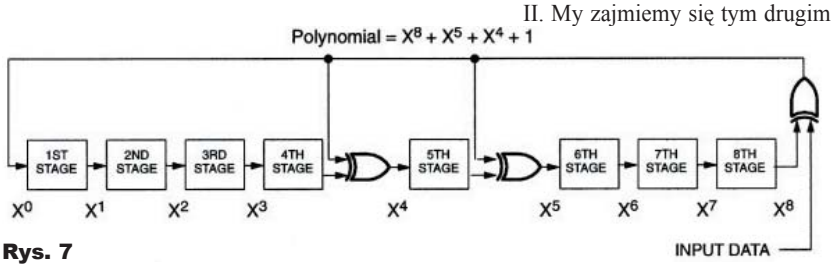


Rys. 6

Warto zwrócić uwagę, że oprócz elementu dodającego pojawił się przerzutnik RS złożony z dwóch bramek NOR. Po naciśnięciu S1 lub S2 jedynka trafi na 2-wejściową bramkę OR, która da na wyjściu jedynkę i wymusi ustawienie na wyjściu przerzutnika RS stanu wysokiego. Spowoduje to, że na wejście ADD trafi jedynka i wartość licznika zostanie zwiększona o jeden. Naciśnięcie S3 wymusi wyzerowanie przerzutnika RS, w związku z czym nie będzie dodawania i losowanie zacznie się od zera.

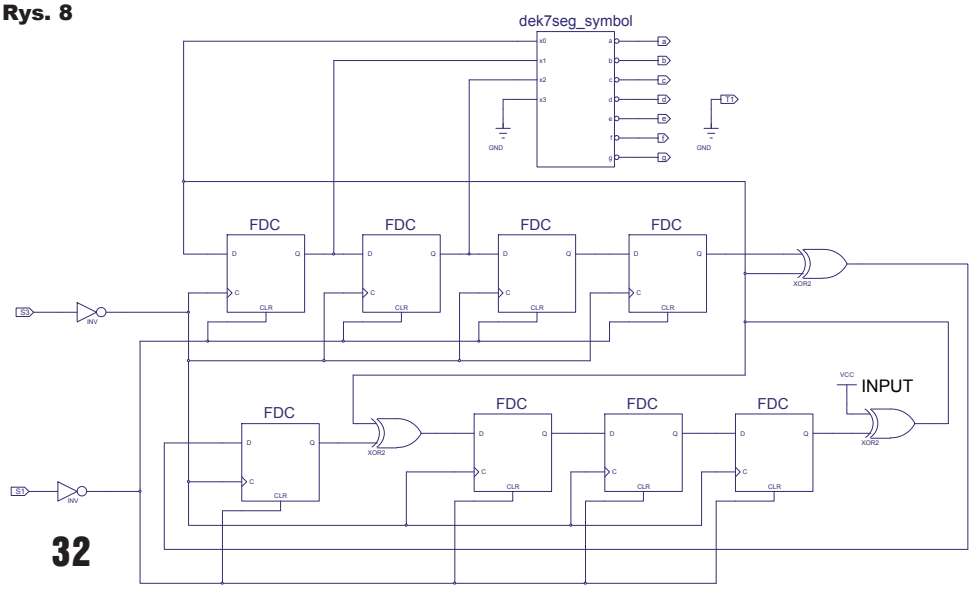
Generator pseudolosowy – rejestry LFSR

Co zrobić w sytuacji, gdy nie mamy dostępu do generatora „wysokiej" częstotliwości? Pozostaje wykorzystanie ciągów pseudolosowych. Jednym ze sposobów pozyskania takiego ciągu może być rejestr przesuwający ze sprzężeniem zwrotnym (LFSR). Układy te pełnią jeszcze jedną, bardzo ważną rolę – generują sumy kontrolne do zabezpieczania transmisji (CRC). Choćby z tego względu warto je omówić. Spotyka się dwa rodzaje takich rejestrów: typu I oraz typu II. My zajmiemy się tym drugim przypadkiem, który bywa także nazywany LFSR-em Galois (od nazwiska francuskiego matematyka).



Rys. 7

Rys. 8



Rejestry takie opisuje się najczęściej wielomianem, np. $h(x) = x^8 + x^5 + x^4 + 1$.

Należy zaznaczyć, że nie może to być dowolny wielomian, gdyż wtedy rejestr taki nie będzie spełniał swojej roli. W Internecie można znaleźć wiele przykładów odpowiednich wielomianów (różnego stopnia) i należy z nich korzystać. Dzięki temu rejestr taki nigdy nie ulegnie wyzerowaniu, pod warunkiem że nie ma wartości zerowej na początku. Dodatkowo przechodzi on przez wszystkie możliwe stany, których jest 2^n-1, gdzie n jest stopniem użytego wielomianu.

Uzbrojeni w wiedzę teoretyczną, zobaczmy, jak skonstruować LFSR, mając podany jakiś wielomian. Najwyższa potęga informuje o długości bitowej rejestru przesuwającego, a pozostałe potęgi określają, które z bitów są poddawane operacji XOR. Wykorzystajmy przytoczony powyżej wielomian:

$h(x) = x^8 + x^5 + x^4 + 1$

Najwyższą potęgą jest 8, więc rejestr jest 8-bitowy. Wykładniki potęg informują, gdzie wstawić operacje XOR (przypominam: $1 = x^0$). W związku z tym rysujemy poszczególne bity w formie kwadratów i numerujemy je. Od której strony zacząć numerowanie? Spotkałem się z dwoma sposobami, niektórzy robią to od lewej strony, a niektórzy od prawej. W tym miejscu przyjmijmy, że będziemy to robić od lewej i bramki XOR wstawimy ZA danym bitem. Ta konwencja pochodzi od firmy Dallas, tak samo jak wielomian h(x) używany do wyznaczania sum kontrolnych np. w cyfrowych termometrach DS18B20. Najwyższa i najniższa (zerowa) potęga wielomianu ma wspólną bramkę XOR.

Rysujemy zatem 8 kwadratów reprezentujących poszczególne bity i wstawiamy operacje XOR za pozycją czwartą i piątą. Pierwszy i ostatni kwadrat łączymy przez bramkę XOR, do której dołączamy jeszcze sygnał wejściowy. Powinniśmy otrzymać LFSR na wzór tego z **rysunku 7** (rysunek pochodzi z materiałów firmy Dallas). Jak wspomniano wcześniej, okres takiego rejestru wynosi 2^n-1, gdzie n jest największą potęga wielomianu.

Wiedząc jak ma wyglądać LFSR, można przystąpić do jego implementacji. Jak już wspomniano, rejestr ten jest typu przesuwającego i zrealizujemy go w znany nam już sposób – używając szeregowo połączonych przerzutników D. Do tego wykorzystamy również standardowe bramki XOR. Po zaimplementowaniu LFSR-a dodamy do schematu jeszcze dekoder 7-segmentowy, do którego dołączymy trzy najmłodsze bity. Dla celów eksperymentalnych dodany został przycisk S1 umożliwiający wyzerowanie rejestru. W ten sposób powstanie pseudolosowa kostka 8-ścienna. Widoczna jest ona na **rysunku 8**. Na wejście podamy jedynkę logiczną, natomiast naciśnięcie przycisku S3 spowoduje podanie zbocza na wszystkie przerzutniki i w efekcie rejestr zmieni swój stan. Należy zauważyć, że wprowadzanie za każdym razem jedynki do rejestru nie spowoduje wypełnienia go samymi jedynkami ze względu na obecność bramek XOR, które dwie jedynki zamieniają na zero. Jak wspomniano, generator jest

pseudolosowy, zatem otrzymywany na wyświetlaczu ciąg jest w pełni deterministyczny (przewidywalny). Czy jednak rzeczywiście tak jest? Czy po zresetowaniu LFSR-a otrzymamy identyczną sekwencję następnym razem? Prawdopodobnie nie. To „magiczne" zachowanie układu pozwoli przedyskutować kolejny, bardzo ważny aspekt układów cyfrowych – drganie zestyków. Okazuje się, że naciśnięcie przycisku nie powoduje wyłącznie zmiany stanu logicznego, ale podaje wręcz całą serię impulsów przed ustabilizowaniem wyjścia! Poglądowy sygnał wejściowy trafiający na port układu CPLD po naciśnięciu przycisku widoczny jest na **rysunku 9**. Liczba impulsów jest losowa i należy je wyeliminować, aby zagwarantować pewne, powtarzalne działanie urządzenia. Jedną z metod jest eliminacja sprzętowa polegająca na zastosowaniu filtru złożonego z kondensatora i rezystora oraz bramki z wejściem Schmitta. Chcąc uprościć urządzenie i dysponując zapasem zasobów można to osiągnąć również za pomocą odpowiedniej implementacji układu. Na **rysunku 10** widoczny jest układ eliminacji drgań zestyków. Zastosowano dzielnik częstotliwości dla generatora kwarcowego i na wejście zegarowe przerzutnika D trafia przebieg o częstotliwości około 90Hz. Sprawia to, że przerzutnik może zmienić swój stan 90 razy na sekundę i przez to bardzo krótkie impulsy pochodzące od drgających zestyków są po prostu „gubione". Otrzymywany na wyjściu LFSR-a ciąg będzie już w pełni deterministyczny (po wstawieniu układu z rysunku 10). Może pojawić się pytanie, jak dobrać częstotliwość dla przerzutnika eliminującego drgania zestyków? Odpowiedź jest prosta: tak, aby nie było na wyjściu niepożądanych oscylacji i tak, aby użytkownik nie zauważył opóźnienia towarzyszącego naciskaniu przycisku. Optymalna wydaje się wartość w granicach 50...100Hz.

Układ iteracyjny – komparator

Odpowiedzmy sobie na pytanie: czym są układy iteracyjne? Najprostszą definicję tego typu układów można ograniczyć do jednego zdania: jest to układ, który realizuje swoją funkcję za pomocą określonej liczby identycznych bloków funkcjonalnych. Z układami iteracyjnymi spotkaliśmy się już, projektując sumator 1-bitowy. Jest to bez wątpienia prosty blok funkcjonalny, który dodaje dwie liczby 1-bitowe. Łącząc kilka takich bloków kaskadowo, otrzymujemy układ iteracyjny – suma n-bitowa jest realizowana za pomocą n podstawowych bloków. Cechą charakterystyczną tego typu układów są linie pomocnicze (sygnały przeniesienia), które również wystąpiły w sumatorze – chodzi tu konkretnie o wejścia i wyjścia przeniesienia. Należy jednak pamiętać, że sygnały te mogą również „wędrować" w drugą stronę, niekoniecznie w jedną, jak miało to miejsce w przypadku sumatora.

Projektowanie układu iteracyjnego sprowadza się do przygotowania pojedynczego bloku funkcjonalnego. Trudność polega na dokładnym zdefiniowaniu jego funkcji, wejść, wyjść oraz sygnałów przeniesienia. Trzeba również zapewnić możliwość łączenia ich w sposób kaskadowy. Budowa takich układów ma tą zaletę, że wymaga opracowania małego elementu, który łączony następnie w większą grupę pozwala realizować znacznie poważniejsze zadania. Tak samo było w przypadku sumatora – zaprojektowanie bloku dodającego liczby 1-bitowe było rzeczą prostą. Czy można w porównywalnym czasie zaprojektować sumator 16-bitowy? Zadanie jest bardzo trudne, gdyż potrzebujemy dwa razy po 16 wejść i 17 wyjść. Siedemnaście tablic Karnaugha dla 32-zmiennych? Desperaci prawdopodobnie daliby radę, ale gdyby nagle się okazało, że trzeba dodawać liczby 18-bitowe ich zdrowie psychiczne byłoby poważnie zagrożone. Nie da się bowiem w prosty sposób rozszerzyć takiego układu o dwa dodatkowe bity w przeciwieństwie do układów iteracyjnych. Czy w takim razie układ iteracyjny jest rozwiązaniem idealnym? Niestety, nie jest, bo dodając dwie liczby 32-bitowe, urządzenie musi poczekać, aż sygnał przepropaguje przez wszystkie 32 bloki i dopiero wtedy można pobrać wynik – pojawiają się opóźnienia. Z naszego punktu widzenia nie ma to znaczenia, ale w przypadku prowadzenia intensywnych obliczeń na dużych liczbach (cyfrowe przetwarzanie sygnałów) jest to poważny problem.

W ramach przykładu spróbujmy opracować komparator n-bitowy. W zasadzie w bibliotece są wersje 2-, 4-, 8- i 16-bitowe, jednakże jest to dobry materiał do ćwiczeń. Poza tym za chwilę przyda nam się wersja 5-bitowa.

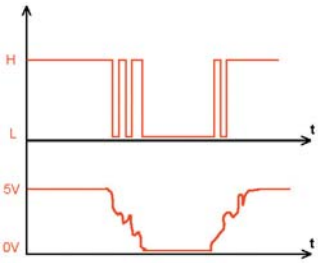
Projektowanie jak zwykle rozpoczniemy od określenia, CO chcemy uzyskać. Podobnie jak w przypadku sumatora będziemy dążyć do uzyskania bloku porównującego pojedyncze bity. Z tego wynika, że będą potrzebne dwa wejścia do wprowadzania liczb 1-bitowych. Co więcej, chcielibyśmy mieć informację, czy porównywana liczba jest mniejsza, równa czy może większa, co daje trzy wyjścia z takiego bloku. Konieczne jest również określenie sposobu komunikacji pomiędzy poszczególnymi blokami – jak mają one przekazywać informacje między sobą. Chcąc odpowiedzieć na to pytanie, zastanówmy się, co to znaczy, że dana liczba jest większa – kiedy możemy stwierdzić, że rzeczywiście tak jest? Przytoczę prosty przykład liczb 321 oraz 123, która jest większa? Oczywiście pierwsza, bo liczba setek jest większa. Wypływa stąd pierwszy wniosek: porównywanie zaczynamy „od lewej" (pozycja najbardziej znacząca). Drugie spostrzeżenie też nie powinno wywołać zdziwienia: po stwierdzeniu, że na danej pozycji (w tym wypadku na pozycji setek) liczba jest większa, dalsze porównywanie nie ma sensu. A co w przypadku liczby 3209 i 3211? W oparciu o sformułowane przed chwilą spostrzeżenia zaczynamy porównywanie „od lewej" strony. Okazuje się, że cyfry są identyczne, więc idziemy dalej. Na pozycji setek cyfry również są jednakowe. Dopiero na pozycji dziesiątek występuje różnica i stwierdzamy, że liczba 3211 jest większa i zaniedbujemy dalsze sprawdzanie. W systemie binarnym wygląda to identycznie.

Sformułujmy zatem wnioski już pod kątem układu iteracyjnego. Musi on posiadać dwa wejścia dla porównywanych bitów. Na wyjściu, jak już wspomniano, wystawiamy trzy sygnały: mniejsza, większa równa. Skoro mamy łączyć takie bloki w kaskady, to trzeba uwzględnić te informacje z bloku nadrzędnego, czyli są potrzebne jeszcze trzy wejścia określające, czy w nadrzędnym bloku stwierdzono, która z liczba jest większa. Tu trzeba poczynić jeszcze jedno bardzo ważne założenie – po pojawieniu się informacji, że któraś z liczb jest większa, kolejne bloki NIE wykonują porównywania, a tylko przenoszą tę informację do samego końca.

Przejdźmy do konkretów. Mamy w sumie pięć sygnałów wejściowych – dwa bity do porównania i trzy sygnały z poprzedniego bloku („bardziej znaczącego"): mniejsza, większa, równa. Zasadniczo potrzebnych jest pięć wejść, ale informacje o tym, czy poprzednia cyfra była większa, mniejsza, czy równa da się zapisać na dwóch bitach. Ograniczamy w ten sposób liczbę sygnałów przeniesienia do dwóch. Mamy tym samym cztery zamiast pięciu zmiennych. Znacznie upraszcza to proces projektowania. Należy jeszcze ustalić, co oznaczają poszczególne wartości liczbowe. Przyjmijmy, że:

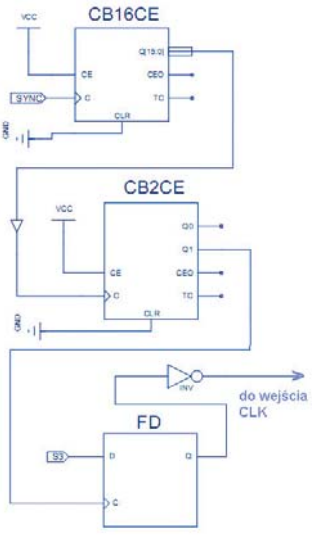
00 → a>b
01 → b>a
10 → a=b
11 → stan niedozwolony

Pozostaje zatem przygotowanie stosowanej tabeli prawdy, którą pokazano na **rysunku 11**. Zmienne *i1*, *i0* oznaczają wejścia, natomiast zmienne *o1*, *o0* – wyjścia. Porównywane bity wprowadzane są przez wejścia oznaczone *a* oraz *b*. Jak stworzyć taką tabelkę? Jeżeli wejścia *i1*, *i0* mają wartość równą 00 lub 01, to przepisujemy ten kod na wyjście bez żadnej modyfikacji, gdyż określono już, która liczba jest większa i dalsze porównywanie nie ma sensu. Gdy wejścia *i1*, *i0* mają stan 10, to sprawa, czy zmienna *a* jest większa od zmiennej *b*, czy odwrotnie, nie została jeszcze rozstrzygnięta. Sprawdzamy teraz, czy wejście *a*=1 i *b=0* – jeżeli tak, to



Rys. 9

Rys. 10



zapisujemy kod 00 oznaczający, że liczba *a* jest większa. Gdyby się okazało, że na wejściach *i1*, *i0* jest 10 (czyli a = b) i wejścia *a* oraz *b* są sobie równe, to nie jesteśmy w stanie określić, która liczba jest większa i zapisujemy na wyjściach *o1*, *o0* wartość 10, pozostawiając rozstrzygnięcie, czy a>b, czy może b>a kolejnym blokom.

Jeszcze kilka słów komentarza. Po pierwsze stan niedozwolony. Nie może być sytuacji, że na wejściach pojawia się liczba 11. Należy tak zaprojektować blok funkcjonalny, aby nigdy to nie miało miejsca. Teoretycznie można zmienić założenia i przyjąć, że wartość 11 będzie także oznaczać równość, ale wtedy nie będzie znaków don't care w tabeli, a jak wiadomo, pozwalają one lepiej minimalizować funkcje boole'owskie.

Odpowiedzmy sobie przy okazji na pytanie, co z wejściami pierwszego bloku? W końcu przed najstarszymi bitami porównywanych liczb nic nie ma. Czy aby na pewno? Liczbę 321 można zapisać jako 0321 albo 00321, itd. Podobnie sytuacja wygląda z liczbami w systemie binarnym – z przodu można dopisać dowolną liczbę zer. Można je dopisywać w nieskończoność, a to oznacza, że zawsze będzie tam równość i taka informacja powinna trafić na wejścia bloku porównującego najstarsze bity. Wymusimy tam wartość 10, co sprawi, że porównywanie zacznie się właśnie od najstarszego bitu.

Pozostaje jeszcze jedna kwestia – w jaki sposób odczytać z układu iteracyjnego, która z porównywanych liczb jest większa? Nie przewidzieliśmy żadnych wyjść służących do tego celu. Nie były one potrzebne – informację odbierzemy z wyjść *o1* oraz *o0* ostatniego bloku, trzymając się wyżej przyjętej konwencji (00 oznacza, że liczba *a* jest większa, itd.).

Alternatywa dla tablic Karnaugha

Moglibyśmy narysować teraz dwie tabele Karnaugha i określić funkcje boolowskie dla wyjść *o1* oraz *o0*. Czytelnicy mogą wykonać to we własnym zakresie. Teraz spróbujemy dojść do celu inną drogą – stosując metodę Quine'a–McCluskeya. Pozwoli to poszerzyć wiedzę z zakresu układów cyfrowych i dokonać świadomego wyboru metody, która bardziej odpowiada Czytelnikowi.

Algorytm postępowania można streścić w kilku krokach:

1. Wypisz wartości binarne reprezentujące słowa wejściowe, dla których wyjście przyjmuje wartość jeden lub don't care. Zapisz przy nich wartość reprezentowaną w systemie dziesiętnym.
2. Pogrupuj wszystkie słowa wejściowe (mintermy) tak, aby w danej grupie znajdowały się słowa zawierające identyczną liczbę jedynek.
3. Zaczynając od grupy zawierającej najmniej jedynek, weź kolejne słowo wejściowe i porównaj z mintermami z kolejnej grupy. Jeżeli różnią się ona tylko na jednej pozycji, miejsce występowania tej różnicy zastąp kreską i zapisz w następnej kolumnie. Oba słowa zaznacz symbolem # (lub dowolnym innym).
4. Podobnie postępuj z drugą i kolejnymi kolumnami tak długo, aż nie będzie można przeprowadzić operacji „sklejenia".
5. Przygotuj tabelkę, w pierwszej kolumnie zapisz wszystkie słowa, które nie mają znaku #, a w górnym wierszu wszystkie wartości, dla jakich wyjście przyjmuje wartość jeden.
6. Wpisz znak X do każdego pola tabeli, dla którego wartość z pierwszego wiersza może być reprezentowana przez słowo znajdujące się w pierwszej kolumnie.
7. Wypisz wszystkie wartości z kolumny tak, aby pokrywały one wszystkie wartości wypisane w pierwszym wierszu.

Algorytm ten może wydawać się niejasny, dlatego prześledźmy jego wykonanie na przykładzie wyjścia *o0*. W pierwszym kroku wypisujemy słowa wejściowe zgodnie z konwencją przyjętą na rysunku 11, tzn. dla słowa o postaci *a b i1 i0* (pierwszy bit to zmienna *a*, a ostatni to zmienna *i0*). Otrzymujemy zatem: 0001(1), 0101(5), 0110(6), 1001(9), 1101(13), 0011(3), 0111(7), 1011(11), 1111(15).

Następnie dokonujemy grupowania zależnie od liczby jedynek w słowie – **rysunek 12**. Teraz zaczynamy proces sklejania. Pierwsza na liście jest wartość (1)0001, którą porównujemy z kolejnymi wartościami z grupy zawierającej po dwie jedynki. Zauważamy, że jednym bitem różnią się wartości (3)0011, (5)0101 oraz (9)1001. Tak więc miejsce wystąpienia różnicy oznaczamy kreską i otrzymujemy trzy wartości: 00-0, 0-01 i -001. Zapisujemy je w drugiej kolumnie, pamiętając o oznaczeniu wykorzystanych wartości znakiem # i zapisaniu w nawiasie liczb, z których ta wartość powstała, czyli: (1,3)00-1, bo wyrażenie 00-0 powstało ze sklejenia 1 i 3. Na **rysunku 13** pokazano, jak powinna wyglądać tabelka na tym etapie. Następnie bierzemy kolejną wartość z listy, czyli (3)0011, i próbujemy sklejać z wartościami z trzeciej grupy. Widać, że (3)0011 może zostać sklejone z (7)0111, co da (3,7)0-00 oraz z (11)1011 co da (3,11)-011. Odznaczamy 7 i 11 (3 już jest odznaczona) i zapisujemy w drugiej kolumnie – **rysunek 14**. Podobnie postępujemy z pozostałymi wartościami i otrzymujemy dwie kolumny jak na **rysunku 15**. Tutaj również obowiązuje zasada grupowania wyrazów w zależności od liczby jedynek.

Teraz zajmujemy się drugą kolumną i postępujemy analogicznie. Bierzemy pierwszy wyraz (1,3)00-1 i porównujemy go z zawartością drugiej grupy z tej samej kolumny. Widzimy, że różni się on tylko jedną pozycją od wyrazu (5,7)01-1, więc zapisujemy sklejony wyraz w kolejnej, trzeciej kolumnie – **rysunek 16**, zaznaczając jednocześnie oba wyrazy wzięte z drugiej kolumny. Powtarzające się wyrazy, takie które mają identyczną reprezentację binarną, pomijamy (nie zapisujmy ich w kolejnej kolumnie), ale odznaczamy je znakiem #. Podobnie postępujemy z pozostałymi wyrazami, a potem tworzymy czwartą kolumnę. W tym miejscu nie da się już nic skleić i otrzymujemy tabelę jak na **rysunku 17**.

Teraz przechodzimy do piątego kroku algorytmu i tworzymy opisaną tam tabelkę. W pierwszym wierszu wpisujemy liczby 1, 5, 6, 9, 13, gdyż dla tych słów wejściowych na wyjściu ma pojawić się jedynka, a w pierwszej kolumnie zapisujemy implikanty nieoznaczone znakiem #. Na przecięciu danego wiersza i kolumny umieszczamy znak X, gdy w pierwszym wierszu i pierwszej kolumnie znajduje się ta sama liczba. Wynik tego etapu jest widoczny na **rysunku 18**.

Został ostatni, siódmy etap. Należy wybrać takie wartości z pierwszej kolumny, aby zawierały one wszystkie liczby wypisane w pierwszym wierszu. Oczywiście im mniej, tym lepiej, bo oszczędzimy w ten sposób zasoby. Na pierwszy rzut oka widać, że wartość:

(1,5,3,7,9,13,11,15) ---1

jest najlepsza, gdyż automatycznie załatwia nam cztery mintermy z pierwszego wiersza. Brakuje tylko 6, więc wybieramy wyraz (6,7) 011-, gdyż żaden inny nie zawiera

Rys. 11

Lp	wejścia				wyjścia	
	a	b	i1	i0	o1	o0
0	0	0	0	0	0	0
1	0	0	0	1	0	1
2	0	0	1	0	1	0
3	0	0	1	1	x	x
4	0	1	0	0	0	0
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	x	x
8	1	0	0	0	0	0
9	1	0	0	1	0	1
10	1	0	1	0	0	0
11	1	0	1	1	x	x
12	1	1	0	0	0	0
13	1	1	0	1	0	1
14	1	1	1	0	1	0
15	1	1	1	1	x	x

Rys. 12

(1) 0001

(3) 0011
(5) 0101
(6) 0110
(9) 1001

(7) 0111
(11) 1011
(13) 1101

(15) 1111

Rys. 13

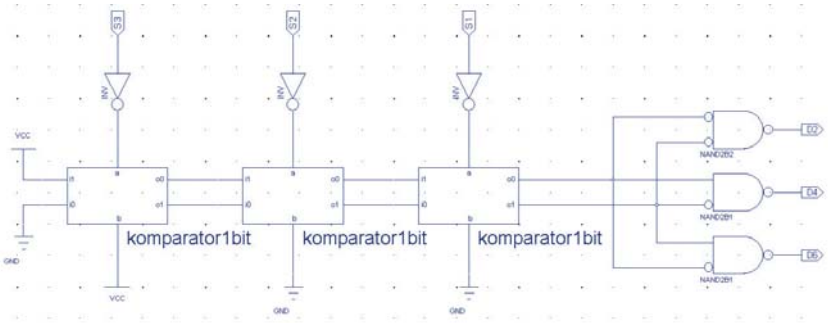
(1) 0001 #	(1,3) 00-1
	(1,5) 0-01
(3) 0011 #	(1,9) -001
(5) 0101 #	
(6) 0110	
(9) 1001 #	
(7) 0111	
(11) 1011	
(13) 1101	
(15) 1111	

Rys. 14

(1) 0001 #	(1,3) 00-1
	(1,5) 0-01
(3) 0011 #	(1,9) -001
(5) 0101 #	
(6) 0110	(3,7) 0-11
(9) 1001 #	(3,11) -011
(7) 0111 #	
(11) 1011 #	
(13) 1101	
(15) 1111	

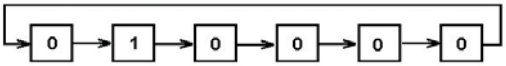
Rys. 15

(1) 0001 #	(1,3) 00-1
	(1,5) 0-01
(3) 0011 #	(1,9) -001
(5) 0101 #	
(6) 0110	(3,7) 0-11
(9) 1001 #	(3,11) -011
	(5, 7) 01-1
(7) 0111 #	(5,13) -101
(11) 1011 #	(6,7) 011-
(13) 1101 #	(6,7) 011-
	(9,11) 10-1
(15) 1111 #	(9,13) 1-01
	(7, 15) -111
	(11,15) 1-11
	(13,15) 11-1



Rys. 21

Rys. 22



liczby 6. W sytuacji, gdy jest wybór, zawsze bierzemy ten implikant, który ma najmniej cyfr, a najwięcej kresek.

Zostało już tylko zaimplementować projektowany z takim trudem komparator. Zawartość nawiasów nie ma dla nas już znaczenia i bierzemy tylko wartości binarne: ---1 oraz 011-, które oznaczają słowo wejściowe *a b i1 i0*. Kreski informują, że dana zmienna nie ma znaczenia, natomiast zero wymusza negację. Otrzymujemy zatem funkcję:

*o0(a, b, i1, i0)=i0 + !a * b * i1*

Pozostała do wyznaczenia jeszcze druga funkcja:

o1(a, b, i1, i0)= ?

odpowiedzialna za drugie wyjście. Tok postępowania jest analogiczny – tabelę prawdy mamy już przygotowaną (rys. 11), więc zostaje rozpisanie poszczególnych mintermów i ich sklejenie – **rysunek 19**. Po tej operacji tworzymy tabelę z punktu 5 algorytmu (**rysunek 20**), wybieramy implikanty (001-, 111-) i zapisujemy funkcję boole'owską:

*o1(a, b, i1, i0) = !a * !b * i1 + a * b * i1*

Implementacja tych prostych funkcji nie powinna sprawić problemu. Ze schematu tworzymy od razu symbol. Sprawdźmy teraz, czy nasz komparator pracuje prawidłowo. Stworzymy układ iteracyjny złożony z trzech bloków, do wejść *a* podłączymy przyciski S1, S2, S3, a do wejść *b* wartość stałą. Do wejść *i1*, *i0* pierwszego bloku doprowadzimy wartość stałą informującą o równości na poprzedniej pozycji. Sygnały *i1*, *i0* z ostatniego bloku wyprowadzimy na prosty dekoder złożony z bramek NAND, którego zadaniem jest włączenie odpowiedniej diody w zależności od wyniku porównania. Pewnie jest to oczywiste, jednakże przypomnę, że przyciski podłączone są przez negatory, aby po ich puszczeniu było zero, a bramka NAND sprawia, że sygnałem aktywnym jest zero potrzebne do zaświecenia diody LED. Dioda D2 włączy się, gdy wartość wprowadzona z „klawiatury" jest większa od czterech (bo taka wartość znajduje się na wejściach *b*), dioda D4, gdy obie wartości są równe, a dioda D6, gdy ta wartość stała jest większa.

Schemat tego układu pokazano na **rysunku 21**.

Licznik pierścieniowy

Zanim przystąpimy do budowy ostatniego w tej części kursu projektu, zaprojektujmy jeszcze jeden interesujący układ. Zadanie jest następujące: zrealizować układ, który na jednym z pięciu wyjść będzie dawał jedynkę przesuwającą się w takt naciskania przycisku (kod *1 z n*). Problem można rozwiązać na kilka sposobów, jednak narzucę konkretne rozwiązanie, spotykane pod nazwą licznika pierścieniowego. Idea pracy takiego układu zilustrowana jest na **rysunku 22** i nie powinna ona wzbudzić większych wątpliwości. Jest to zwykły rejestr przesuwający (omówiony w drugiej części kursu) z tą różnicą, że jest on zapętlony. Znajdująca się w układzie jedynka wędruje przez kolejne przerzutniki, dochodzi do ostatniego i dzięki zapętleniu może rozpocząć kolejny obieg. Pytanie zasadnicze: gdzie w tym rozwiązaniu kryją się pułapki? Są dwie. Pierwsza z nich streszcza się w pytaniu: skąd ta jedynka się tam wzięła? Po uruchomieniu urządzenia mogą znajdować się tam same zera i co wtedy? Równie dobrze mogą trafić się dwie jedynki albo więcej. Druga pułapka jest mniej oczywista – w czasie pracy układu może pojawić się zakłócenie, które sprawi, że w rejestrze będzie więcej niż jedna jedynka. Przypadek ten nie będzie częsty, ale warto byłoby się i przed nim zabezpieczyć.

Tu pojawia się kolejna okazja do zaprojektowania układ kombinacyjnego, który powinien realizować następujące zadanie: usuwać zbędne jedynki i wprowadzać jedynkę, gdy są same zera. Nasuwa się wniosek, że musi mieć on pięć wejść, do których podłączone zostaną wyjścia przerzutników – umożliwi to bieżącą analizę pracy układu. Pozostaje zastanowić się, jak ma przebiegać ingerencja w pracę układu, czyli na jakiej zasadzie usuwać lub wstawiać jedynki. Wprowadzenie czegokolwiek do licznika pierścieniowego jest możliwe wtedy, gdy mamy dostęp do wejścia któregoś z przerzutników. W związku z tym należałoby „przeciąć" licznik w którymś miejscu i dołączyć nasz układ kombinacyjny. Zasadniczo miejsce „cięcia" nie ma znaczenia, ale z powodów czysto estetycznych niech to będzie pierwszy przerzutnik.

Mamy pięć wejść, zatem mogą pojawić się 32 różne stany i dla nich należy przygotować stosowną tabelę prawdy. Po przeanalizowaniu naszych wymagań można dojść do wniosku, że jedynka na wyjściu pojawi się tylko w dwóch sytuacjach – wtedy gdy na wejściach są same zera albo wtedy, gdy na wyjściu ostatniego przerzutnika jest jedynka. Bardzo upraszcza to projektowanie takiego układu – tabelka prawdy jest widoczna na **rysunku 23**, a wyznaczenie funkcji boole'owskiej nie powinno sprawić nikomu problemu. Ma ona postać:

*y(x4, x3, x2, x1, x0) = !x4 * !x3 * !x2 * !x1*

Możemy teraz przystąpić do implementacji licznika pierścieniowego. Wykorzystamy do tego celu pięć przerzutników typu D, które stworzą rejestr przesuwający i między pierwszy i ostatni wstawimy opracowany przed chwilą układ gwarantujący, że

Rys. 16

(1) 0001 #	(1,3) 00-1 #	(1,3,5,7) 0—1
	(1,5) 0-01	
(3) 0011 #	(1,9) -001	
(5) 0101 #		
(6) 0110	(3,7) 0-11	
(9) 1001 #	(3,11) -011	
	(5, 7) 01-1 #	
(7) 0111 #	(5,13) -101	
(11) 1011 #	(6,7) 011-	
(13) 1101 #	(6,7) 011-	
	(9,11) 10-1	
(15) 1111 #	(9,13) 1-01	
	(7, 15) -111	
	(11,15) 1-11	
	(13,15) 11-1	

Rys. 17

(1) 0001 #	(1,3) 00-1 #	(1,3,5,7) 0—1 #	(1,5,3,7,9,13,11,15) —-1
	(1,5) 0-01 #	(1,9,3,11) -0-1	
(3) 0011 #	(1,9) -001#		
(5) 0101 #		(9,13,11,15) 1—1 #	
(6) 0110	(3,7) 0-11 #		
(9) 1001 #	(3,11) -011 #		
	(5, 7) 01-1 #		
(7) 0111 #	(5,13) -101		
(11) 1011 #	(6,7) 011-		
(13) 1101 #	(9,11) 10-1 #		
	(9,13) 1-01#		
(15) 1111 #			
	(7, 15) -111		
	(11,15) 1-11 #		
	(13,15) 11-1 #		

Rys. 18

	1	5	6	9	15
(5,13) -101		X			
(6,7) 011-			X		
(7, 15) -111					X
(1,9,3,11) -0-1	X			X	
(1,5,3,7,9,13,11,15) ---1	X	X		X	X

Rys. 19

(2) 0010 #	(2,3) 001-	(3,7,11,15) –11
(3) 0011 #	(3,7) 0-11 #	
	(3,11) -011 #	
(7) 0111 #		
(11) 1011 #	(7, 15) -111 #	
(14) 1110 #	(11,15)1 -11 #	
	(14,15)111 -	
(15) 1111 #		

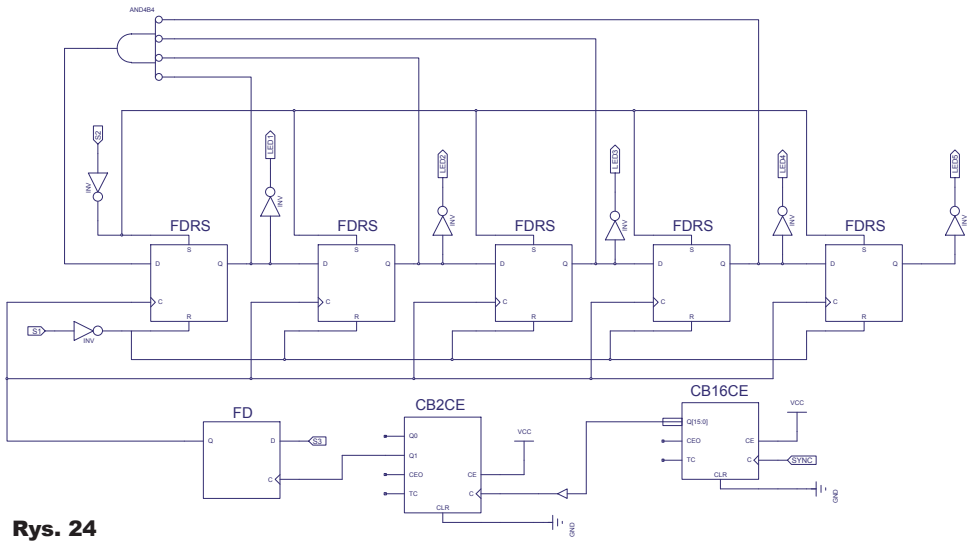
Rys. 20

	2	14
(2,3) 001-	X	
(14,15) 111-		X
(3,7,11,15) –11		

na wyjściach będzie obecna tylko jedna jedynka. Do wyjść przerzutnika podłączone zostały diody LED, aby można było sprawdzić, czy całość pracuje zgodnie z naszymi oczekiwaniami. Do schematu z **rysunku 24** dołączono również przyciski – S1 i S2 umożliwiają wprowadzanie „zakłóceń" polegających na ustawieniu na wszystkich wyjściach, odpowiednio, jedynek lub zer. Przycisk S3 razem z układem eliminacji drgań zestyków pozwala na przesuwanie jedynki obecnej w liczniku oraz umożliwia obserwowanie, jak eliminowane są owe „zakłócenia". Uwaga! Ustawianie i resetowanie przerzutników odbywa się synchronicznie – należy nacisnąć S1/S2, następnie S3 i puścić S3 przed puszczeniem S1/S2.

wejścia					wyjście
x4	x3	x2	x1	x0	y
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	0
		(...)			
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

Rys. 23



Rys. 24

Wadą tego układu jest to, że mogą wystąpić w nim niedozwolone stany, tzn. nie zostaną one usunięte samoistnie, tylko w takt naciskania S3. Spróbujmy coś z tym zrobić. Rozwiązaniem może być podanie kilku przebiegów sygnał zegarowego pochodzącego z generatora tak, aby układ korekcyjny mógł wykonać swoją pracę. Po stwierdzeniu, że występuje błąd, przycisk S3 powinien zostać odłączony i zamiast niego dołączony powinien zostać właśnie generator. Do przełączania można wykorzystać znany już nam multiplekser z dwoma wejściami. Ostatnim niezbędnym elementem jest układ sterujący pracą takiego multipleksera. Należałoby sobie zadać pytanie, jak ma być podejmowana decyzja o tym, czy nasz licznik znajduje się w stanie zabronionym, czy też nie. Myślę, że odpowiedź nie jest trudna – powinien on po prostu liczyć jedynki na wyjściach – gdy ich liczba jest różna od jednego, to dołączany jest generator. Projektowanie układu dla pięciu zmiennych nie będzie już takie proste, więc skorzystamy z innej drogi – układu iteracyjnego. Stworzymy pięć bloków, każdy będzie miał jedno wejście, do którego podłączymy wyjście jednego z przerzutników. Pozostało określić jeszcze wymagane sygnały przeniesienia. Czy wystarczy jeden? Nie bardzo, gdyż chcemy wiedzieć, czy w poprzednich blokach wystąpiła dokładnie jedna jedynka, czy nie było ich wcale, czy może były dwie lub więcej. Teoretycznie wystarczyłaby jedna linia określająca, czy układ pracuje prawidłowo, czy nie. Łatwo jednak wskazać przykład, gdy takie rozwiązanie nie sprawdzi się – po pojawieniu się pierwszej jedynki wystawiany jest informacja, że wszystko jest dobrze. Co, jeżeli pojawi się druga jedynka? Informacja, że układ pracuje prawidłowo, jest nieprecyzyjna – wcześniej mogło nie być w ogóle jedynek albo mogła jedna już się pojawić. Nie da się jednoznacznie określić, która z tych dwóch sytuacji miała miejsce.

Rys. 25

wejścia			wyjścia	
x	i1	i0	o0	o1
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

Z tego względu liczbę sygnałów przeniesienia ustalimy na dwa. Będą one miały następujące znaczenie:
00 → brak jedynek w poprzednich blokach
01 → jedna jedynka w poprzednich blokach
10 → dwie jedynki w poprzednich blokach
11 → trzy lub więcej jedynek w poprzednich blokach.
W oparciu o zebrane dane można przygotować tabelkę prawdy – **rysunek 25**. Wejście do zliczania jedynek zostało oznaczone *x*, wejścia sygnałów przeniesienia: *i1*, *i0*, a wyjścia: *o1*, *o0*. Jak przygotować taką tabelkę? Linie *i1*, *i0* oraz *o1*, *o0* należy potraktować jako liczby binarne – gdy wejście *x* ma stan wysoki, wartość linii *i1*, *i0* zwiększamy o jeden i przekazujemy do linii *o1*, *o0*. Zrealizujemy w ten sposób proste zwiększanie stanu wyjść po wystąpieniu jedynki.
Mając tabelkę prawdy, można przystąpić do wyznaczenia stosownych funkcji boole'owskich. Są w sumie trzy wejścia i dwa wyjścia, więc otrzymamy dwie funkcje, które zależą od trzech zmiennych. Nic nie stoi na przeszkodzie, aby posłużyć się tablicą Karnaugha, jednakże w ramach niniejszej części kursu nadal będziemy ćwiczyć metodę Quine'a–McCluskeya. Sposób minimalizacji za jej pomocą pokazano na **rysunku 26** oraz **27**. Wybieramy stosowne implikanty (wypisane na rysunku 27) i na ich podstawie zapisujemy ostateczne funkcje:
*o0(x, i1, i0) = (!x * i0 + x * !i0) + x * i1*
*o1(x, i1, i0) = i1 + x * i0*
Chciałbym wspomnieć o pewnej „sztuczce", jaką można w tym miejscu zastosować, mianowicie istnieje następująca zależność:
*!a * b + a * !b = a XOR b*
Funkcja *o0* zawiera taki człon, zatem można go zapisać w skróconej formie:
*o0(x, i1, i0) = (x XOR i0) + x * i1*
Obie pozyskane funkcje implementujemy według poznanych wcześniej zasad i tworzymy element. Jego działanie można sprawdzić za pomocą schematu z **rysunku 28**. Diody LED1 i LED2 podają, w systemie binarnym, ile zliczono jedynek – w tym wypadku ile wciśnięto przycisków.
Mając opracowany układ iteracyjny, można przystąpić do implementacji pełnego licznika pierścieniowego (**rysunek 29**). W tym wypadku naciśnięcie przycisku S1 lub S2 nie wywołuje żadnego efektu, gdyż korekcja działa prawidłowo i stany niedozwolone są natychmiast usuwane. Można jednak korzystać z przycisku S3, który powoduje przemieszczenie jedynki na wyjściu licznika. Po sprawdzeniu, czy wszystko pracuje prawidłowo utwórzmy gotowy element wprowadzając, uprzednio parę kosmetycznych zmian – zmieńmy nazwy portów, aby były one bardziej adekwatne do pełnionych funkcji:
– x4..x0 – wyjścia licznika pierścieniowego
– INPUT – zbocze podane na to wejście powoduje,

Rys. 26

o1(x, i1, i0) = ?

(2) 010 #	(2,3) 01- #	(2,3,6,7) - 1-
	(2,6) -10 #	
(3) 011 #	(3,7) -11 #	
(5) 101 #	(5,7) 1-1	
(6) 110 #	(6,7) 11- #	
(7) 111 #		

o0(x, i1, i0) = ?

(1) 001 #	(1,3) 0-1
(4) 100 #	(4,6) 1-0
(3) 011 #	(3,7) -11
(6) 110 #	(6,7) 11-
(7) 111 #	

Rys. 27

o1(x, i1, i0) = ?

	2	3	5	6	7
(5,7) 1-1			X		X
(2,3,6,7) -1-	X	X		X	X

implikanty: 1-1, -1-

o0(x, i1, i0) = ?

	1	3	4	6	7
(1,3) 0-1	X	X			
(4,6) 1-0			X	X	
(3,7) -11		X			X
(6,7) 11-				X	X

implikanty 0-1, 1-0, 11-

że jedynka przechodzi do kolejnego wyjścia – CLK – sygnał zegarowy wymuszający ustawienie prawidłowego stanu
Co więcej usuńmy bramki NOT przez które podłączone były diody LED oraz pozbądźmy się portów dla przycisków wymuszających stany niedozwolone.

Minutnik do jajek

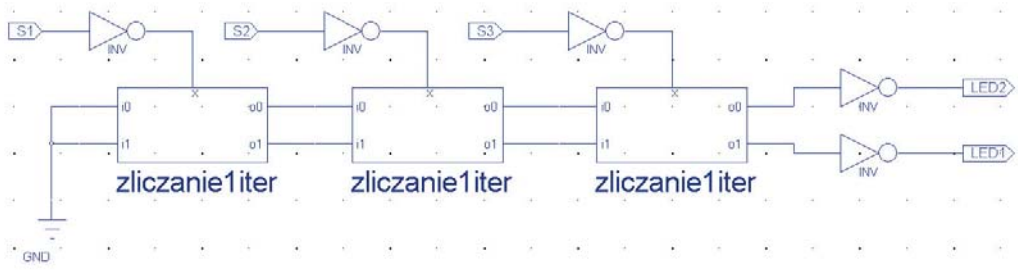
Ostatnie ćwiczenie przygotowane na dzisiaj to prosty minutnik do jajek. Napracowaliśmy się, budując komparator oraz licznik pierścieniowy, więc wykorzystamy je teraz. Zacznijmy od zdefiniowania wymagań – załóżmy, że chcemy mieć możliwość odmierzania czasu w przedziale od jednej do pięciu minut. Do wybierania wykorzystamy przycisk S1, a S3 do resetowania (zliczania od początku). Zakończenie zliczania powinno spowodować włączenie kropki na wyświetlaczu 7-segmentowym. Można również włączyć tranzystor MOSFET i dołączyć do niego buzer, zyskując w ten sposób sygnalizację akustyczną.
Potrzebny jest impuls o okresie 1 minuty. Da się go uzyskać z generatora kwarcowego po zastosowaniu dzielnika przez 24 000 000 * 60. Dzielnik taki można uzyskać przez zastosowanie trzech liczników modulo 200 i jednego modulo 180. Otrzymany w ten sposób impuls powinien trafić na licznik z koderem na kod *1 z n*, którego wyjścia powinny być porównywane z zawartością licznika pierścieniowego. Komparator będzie więc porównywał stan dekodera ze stanem zadanym w liczniku pierścieniowym i po stwierdzeniu równości zasygnalizuje upływ ustawionego czasu.
Schemat tego układu widzimy na **rysunku 30**. Dodano tutaj jeszcze jedną bramkę NOT, przed licznikiem 4-bitowym, aby zagwarantować zliczanie zboczy opadających, a nie narastających.
Czy minutnik można było zrealizować prościej, np. stosując liczniki 4-bitowe i klasyczny, 4-bitowy komparator? Oczywiście! Jednakże wykonując po drodze wszystkie karkołomne operacje, mogliśmy poćwiczyć używanie metody Quine'a–McCluskey'a, projektowanie układów iteracyjnych i przy okazji parę innych rzeczy. Rozwiązanie zawierające „zwykły" licznik i komparator wydaje się przy tym proste i przyjemne, dlatego pozostawiam je Czytelnikom do samodzielnej implementacji. Zwróćcie uwagę na bardzo ważny wniosek: jeden cel można osiągnąć na wiele sposobów. Różnią się one przede wszystkim stopniem złożoności koncepcji i wymaganymi do implementacji zasobami. A może da się to zrobić jeszcze inaczej? Poszukajcie jeszcze innej koncepcji i spróbujcie ją zrealizować, aby dodatkowo poćwiczyć.

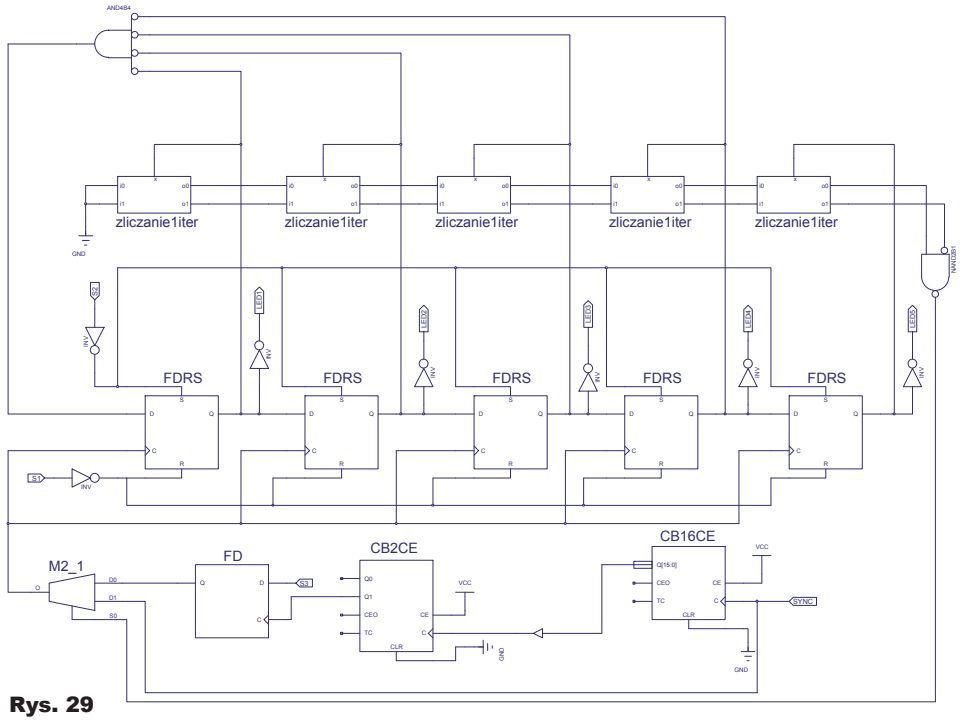
Ćwiczenia

W ramach dodatkowego ćwiczenia proponuję opracowanie kostek o znacznie większej liczbie ścianek – np. kostek mających 12, 20 czy 100 ścianek. Po zrealizowaniu tego celu zachęcam do opracowania jeszcze trudniejszej wersji, np. kostki 20-ściennej liczącej w zakresie 1..20. Gdy to się uda, można pokusić się o zaimplementowanie trzech wielościennych kostek w jednym projekcie.
Kolejnym, trudnym zadaniem jest opracowanie kostki opartej o LFSR, ale mają liczbę ścianek różną od 2^n, np. kostki 6-ściennej.
Pragnę przypomnieć, gdyby nie zostało to wystarczająco mocno podkreślone w artykule, że dodając do projektu samodzielnie utworzony symbol bazujący na innym symbolu należy wstawić schematy OBU tych symboli do projektu (nie dotyczy to symboli pobranych z biblioteki).

Jakub Borzdyński
jakub.borzdynski@elportal.pl

Rys. 28





Rys. 29

Rys. 30